GRAPHCORE

PopRT User Guide

Version 1.5.0

Graphcore Ltd

Oct 17, 2023

CONTENTS


GRAPHCORE

CHAPTER

ONE

INTRODUCTION

PopRT is a high-performance inference engine for the Graphcore IPU. It compiles and optimises models that have been trained and exported for inference, generating an executable program that can be run on IPUs.

PopRT generates the executable program as a PopEF model. The Poplar Exchange Format is a Graphcore file format that is mainly used for exporting and importing models. PopRT provides a flexible runtime to perform low-latency and high-throughput inference on PopEF models.

1.1 Background

The Graphcore Poplar SDK includes tools and libraries to support programming the IPU for various industry-standard machine learning frameworks such as TensorFlow, MXNET, ONNX, Keras, and PyTorch. The SDK packages need to balance functionality for both training and inference when performing optimisation. PopRT, on the other hand, is a specialised engine for inference that allows more targeted optimisation for inference scenarios.

The main functions of PopRT are to:

- Convert an ONNX model to a lower precision (float16/float8)
- Convert the opset version of the ONNX model
- Apply various optimisations to the model, including general and IPU-specific optimisations
- Provide a flexible runtime for inference using PopEF models.

1.2 Architecture

Fig. 1.1 shows the architecture of PopRT.

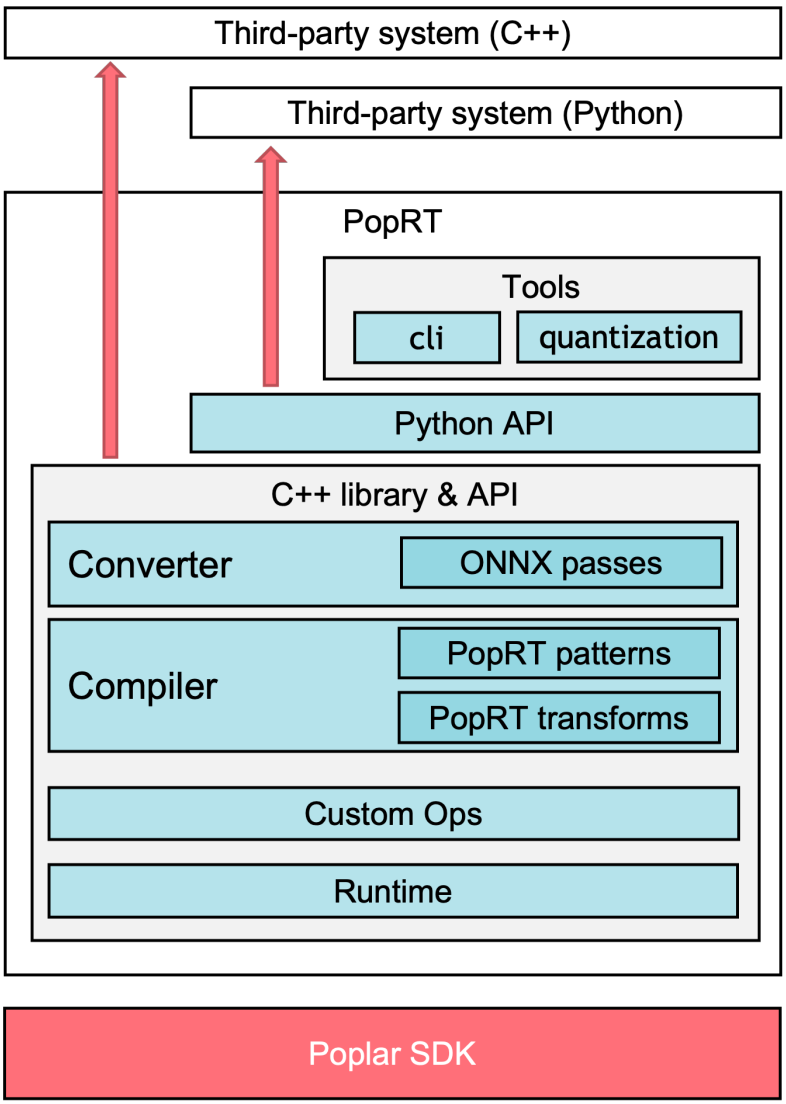


Fig. 1.1: PopRT architecture

The architecture of PopRT consists of three main parts:

- Converter: A converter is responsible for optimising models exported from other frameworks. Converters are framework-specific and only the ONNX converter is currently supported.
  - Pass: Passes perform graph optimisation of the framework-specific model IR. Only the ONNX pass is currently supported.
- Compiler: Optimises the PopART IR for inference and includes patterns and transforms. The serialisable binary PopEF model is generated after compilation.
  - Pattern: Matches and replaces operators.
  - Transform: Optimises overall graph.
  - Custom Ops: Additional custom operator libraries for inference scenarios.
- Runtime: Loads PopEF model and performs low-latency and high-throughput inference.
- Tools: Provides general tools, such as FP8 quantisation tools and command-line tools.

1.3 Workflow

The workflow of PopRT for converting, compiling and running models is shown in Fig. 1.2. PopRT provides *command-line tools*, a *Python API* and a *C++ API*.

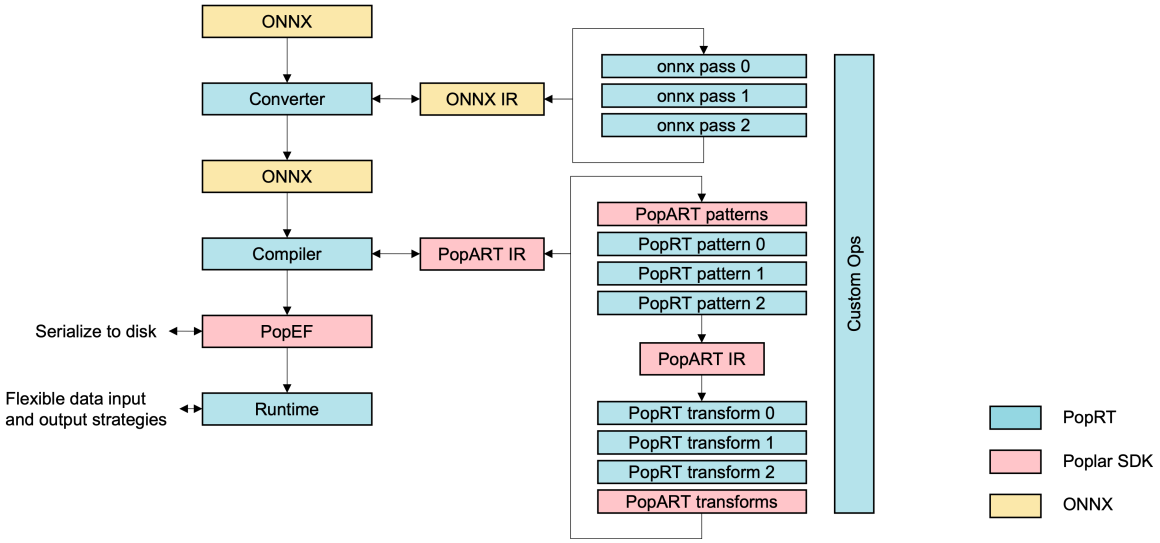


Fig. 1.2: PopRT workflow

CHAPTER

TWO

INSTALLATION

Note: The parameters of device mapping will be omitted in the examples in this chapter. Please add them as needed or use the gc-docker tool for automatic processing.

For the C600 card, each IPU device needs to map three device files. Taking ipu0 as an example, each IPU device needs to map /dev/ipu0, /dev/ipu0_ex and /dev/ipu0_mem.

The following is an example of mapping devices ipu0 and ipu1 to the Docker container:

```
docker run \
    --device=/dev/ipu0:/dev/ipu0 --device=/dev/ipu0_ex:/dev/ipu0_ex --device=/dev/ipu0_mem:/dev/ipu0_mem \
    --device=/dev/ipu1:/dev/ipu1 --device=/dev/ipu1_ex:/dev/ipu1_ex --device=/dev/ipu1_mem:/dev/ipu1_mem \
    --ipc=host \
    ...
```

For IPU-M2000 and Bow-2000 systems, the following parameters will need to be added:

```
docker run \
    --ulimit memlock=-1:-1 --net=host --cap-add=IPC_LOCK --device=/dev/infiniband/ --ipc=host\
    -v /path/to/ipu.conf:/etc/ipuof.conf -e IPUOF_CONFIG_PATH=/etc/ipuof.conf \
    ...
```

More information about IPUOF_CONFIG_PATH can be found in Using IPU Pod partitions .

2.1 Compatibility of PopRT with the Poplar SDK

The PopRT version must correspond to the Poplar SDK version to ensure the compatibility of PopRT functions with the Poplar SDK.

Table 2.1: Compatible versions of PopRT and the Poplar SDK

PopRT version	Poplar SDK version	Note
v1.5.0	v3.4.0	None
v1.4.0	v3.3.0	None
v1.3.0	v3.3.0	None
v1.2.0	v3.2.0	None
v1.1.0	v3.2.0	None
v1.0.2	v3.1.0	None

2.2 Quick start with a Docker image

You can use Docker images that contain PopRT. Details are available on the DockerHub page.

For example to pull the Docker image with the latest version of PopRT:

```
docker pull graphcorecn/poprt:latest
docker run --rm graphcorecn/poprt:latest -v
```

2.3 Install PopRT on host server

To install PopRT on a host server, you must:

- Download the Poplar SDK (from the Software Downloads page) for your OS. You can check which OS you are running by using the command:

  ```
  $ lsb_release -a
  ```

- Download the PopRT wheel file from the PopRT Github Releases page. Make sure that the version you download corresponds to the version of the Poplar SDK you have already downloaded. See Table 2.1 for compatible versions.

2.3.1 For Ubuntu 20.04

Note: PopRT only supports Python 3.8 on Ubuntu 20.04.

Install Poplar SDK

The Poplar SDK tarball is named as follows:

```
poplar_sdk-[os]-[poplar_ver]-[build].tar.gz
```

where [os] is the host OS, [poplar_ver] is the SDK software version number and [build] identifies a specific build.

An example of the Poplar SDK tarball for Ubuntu 20.04 and version 3.2 of the SDK is:

```
poplar_sdk-ubuntu_20_04-3.2.0-7cd8ade3cd.tar.gz
```

Install the SDK by unpacking the tarball as follows:

```
$ tar -xvzf poplar_sdk-[os]-[ver].tar.gz
```

You can install the SDK to any directory that you have write access to.

For details of the components included in the Poplar SDK, refer to Contents of the SDK .

Enable the SDK

After you have installed the SDK, you need to enable it. First check whether you already have a version of the SDK enabled:

```
echo $POPLAR_SDK_ENABLED
```

This will return the path of the enabled SDK. If a path is already set, then run first:

```
unset POPLAR_SDK_ENABLED
```

If there is no path set, then run:

```
source [path_to_SDK]/enable
```

where [path_to_SDK] is the location of the Poplar SDK on your system.

Install PopRT

Note: If you do not wish to install PopRT in the default Python environment, then you can create a Python virtual environment with:

```
python -m venv [name-of-virtual-environment]
```

Install the PopRT wheel file with:

```
pip install poprt-*.whl
```

Verify the version of the installed wheel file with:

```
poprt -v
```

This will return the version, for example, for PopRT version 1.0.2, the expected output is:

```
PopRT version: 1.0.2
Build with SDK: 3.1.0+6824 (8387f79d96)
Runtime SDK: 3.1.0+6824 (8387f79d96)
```

CHAPTER

THREE

QUICK START

This chapter will use an ONNX model as the example to explain how to perform model conversion and rapid deployment with PopRT.

Note:

- PopRT is required to run the examples in this chapter. If PopRT is not installed, refer to the *Installation* chapter.
- PopRT supports the conversion and running of ONNX models, while exporting to PopEF files for model deployment. PopEF is a unified file format provided by Poplar SDK for importing and exporting models. The PopEF file contains the binary code compiled by the Poplar compiler that can run on IPU, the input and output of the model as well as the hardware information required for the running of models. For more information, please refer to to the PopEF: User Guide .
- The examples in this chapter use the PopRT command-line interface.

3.1 CLI parameters

PopRT is easy to use from the command line. The following are some of the CLI parameters:

- --input_model (Required) Path to original model (for example, ONNX) and model name.
- --show (Optional) Print only the input and output information of the model.
- --input_shape (Optional) Specify the model input shape.
- --output_model (Optional) The name of the converted model. The default storage path is the current directory. If this parameter is not specified, the converted model will be saved with the default name.
- --precision (Optional) Specify the accuracy of the converted model. If this parameter is not specified, the original model accuracy will be used by default.
- --run (Optional) Run the converted model using random input.
- --export_popef (Optional) Export the PopEF model generated by compilation. The file is saved with a default name of executable.popef.

For full details of all CLI parameters, refer to the *Command line interface* chapter.

Note:

- For --input_shape, only the variable dimensions in the specified input shape are supported, and the size of known dimensions cannot be changed.
- The IPU only supports static graphs. If the input shape of the model is variable, --input_shape must be used to specify the input shape.
- For more configurations, please refer to the *Command line interface* chapter.

3.2 Convert and run model

This section will use the BERT Squad model in the ONNX model zoo as the example to explain how to perform model conversion.

3.2.1 Download ONNX model

Download the ONNX model with:

```
wget https://github.com/onnx/models/raw/main/text/machine_comprehension/bert-squad/model/bertsquad-12.onnx
```

3.2.2 Obtain input and output information for ONNX model

Obtain the input and output information for the ONNX model with:

```
poprt \
  --input_model bertsquad-12.onnx \
  --show

# The Input/Output Information of the Model
2023-01-06 02:59:32,897 INFO cli.py:327] Input unique_ids_raw_output____9:0: dtype - int64, shape - [0]
2023-01-06 02:59:32,897 INFO cli.py:327] Input segment_ids:0: dtype - int64, shape - [0, 256]
2023-01-06 02:59:32,897 INFO cli.py:327] Input input_mask:0: dtype - int64, shape - [0, 256]
2023-01-06 02:59:32,898 INFO cli.py:327] Input input_ids:0: dtype - int64, shape - [0, 256]
2023-01-06 02:59:32,898 INFO cli.py:334] Output unstack:1: dtype - float32, shape - [0, 256]
2023-01-06 02:59:32,898 INFO cli.py:334] Output unstack:0: dtype - float32, shape - [0, 256]
2023-01-06 02:59:32,898 INFO cli.py:334] Output unique_ids:0: dtype - int64, shape - [0]
```

3.2.3 Specify input shape

According to the input and output information of the original model, the input shape is variable. This means that --input_shape must be used to specify the input shape. In this example, batch size = 2 is used.

Note:

- The original model input is int64. Since the IPU does not support int64, the converted model input will be int32.

```
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --output_model bertsquad-12_fp32_bs2.onnx
```

3.2.4 Specify model accuracy

According to the input and output information of the original model, the accuracy is fp32. In this example, we will use an accuracy of fp16.

```
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --output_model bertsquad-12_fp16_bs2.onnx \
    --precision fp16
```

3.2.5 Run model

Note:

- --run can be used to quickly verify whether the converted model can be compiled properly and run on an IPU.
- The data shown in this example does not represent optimal performance and only demonstrates the default conversion process.

```
# Convert, compile and run the ONNX model of fp32
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --output_model bertsquad-12_fp32_bs2.onnx \
    --run

# Random number run results
2023-01-06 05:58:14,209 INFO cli.py:452] Bs: 2
```

(continues on next page)

(continued from previous page)

```
2023-01-06 05:58:14,209 INFO cli.py:455] Latency: 4.58ms
2023-01-06 05:58:14,210 INFO cli.py:456] Tput: 436

# Convert, compile and run the ONNX model of fp16
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --output_model bertsquad-12_fp16_bs2.onnx \
    --precision fp16 \
    --run

# Random number run results
2023-01-06 06:00:59,283 INFO cli.py:452] Bs: 2
2023-01-06 06:00:59,283 INFO cli.py:455] Latency: 2.23ms
2023-01-06 06:00:59,283 INFO cli.py:456] Tput: 896
```

3.2.6 Export PopEF model

The exported PopEF model can be used for offline deployment.

Note:

- The --export_popef parameter uses the default name executable.popef to save the PopEF model to a file. Note that multiple exports will overwrite the executable.popef file.

```
# Convert, compile and export the PopEF model of fp32
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --export_popef

# Convert, compile and export the PopEF model of fp16
poprt \
    --input_model bertsquad-12.onnx \
    --input_shape unique_ids_raw_output___9:0=2 segment_ids:0=2,256 input_mask:0=2,256 input_ids:0=2,256 \
    --precision fp16 \
    --export_popef
```

3.3 Quick deployment

This section will take the ONNX model and the PopEF model converted in the previous section as an example to explain how to use the PopRT Python API for quick deployment.

3.3.1 Run exported PopEF model

To run the exported PopEF model:

```
from poprt import runtime
import numpy as np

# Create the ModelRunner instance, and load the PopEF file
runner = runtime.Runner('executable.popef')

# Obtain the model output information
outputs = runner.get_execute_outputs()

# Create the input and output data
input_dict = {
    "unique_ids_raw_output____9:0": np.ones([2]).astype(np.int32),
    "segment_ids:0": np.ones([2, 256]).astype(np.int32),
    "input_mask:0": np.ones([2, 256]).astype(np.int32),
    "input_ids:0": np.ones([2, 256]).astype(np.int32),
}
output_dict = {x.name: np.zeros(x.shape).astype(x.numpy_data_type()) for x in outputs}

# Run PopEF
runner.execute(input_dict, output_dict)
print(output_dict)
```

3.3.2 Run converted ONNX model

You can directly compile the converted ONNX model with the PopRT `Compiler` class. Then, you can use the `ModelRunner` class to run the PopEF instance generated by compilation.

```
import onnx
import numpy as np
from poprt import runtime
from poprt.compiler import Compiler

# Import the ONNX model
model = onnx.load("bertsquad-12_fp32_bs2.onnx")
model_bytes = model.SerializeToString()
output_names = [o.name for o in model.graph.output]

# Compile ONNX, and generate the PopEF instance
executable = Compiler.compile(model_bytes, output_names)

# Create the ModelRunner instance, and load the PopEF instance
runner = runtime.Runner(executable)

# Obtain the model output information
outputs = runner.get_execute_outputs()

# Create the input and output data
input_dict = {
    "unique_ids_raw_output____9:0": np.ones([2]).astype(np.int32),
```

(continues on next page)

(continued from previous page)

```
    "segment_ids:0": np.ones([2, 256]).astype(np.int32),
    "input_mask:0": np.ones([2, 256]).astype(np.int32),
    "input_ids:0": np.ones([2, 256]).astype(np.int32),
}
output_dict = {x.name: np.zeros(x.shape).astype(x.numpy_data_type()) for x in outputs}

# Run PopEF
runner.execute(input_dict, output_dict)
print(output_dict)
```

3.4 Python API example

The following is an example of converting, compiling and running the ONNX model entirely with the Python API:

Listing 3.1: Example showing how to convert, compile and run an ONNX model using the PopRT Python API

```
# Copyright (c) 2022 Graphcore Ltd. All rights reserved.
import argparse
import time

from typing import Dict

import numpy as np
import onnx

from onnx import helper

from poprt import Pass, runtime
from poprt.compiler import Compiler, CompilerOptions
from poprt.converter import Converter

RuntimeInput = Dict[str, np.ndarray]


def convert(model_proto: onnx.ModelProto, args) -> onnx.ModelProto:
    """Convert ONNX model to a new optimized ONNX model."""
    converter = Converter(convert_version=11, precision='fp16')
    converted_model = converter.convert(model_proto)
    # Add other passes here
    converted_model = Pass.get_pass('int64_to_int32')(converted_model)
    converted_model = Pass.get_pass('gelu_pattern')(converted_model)

    return converted_model


def compile(model: onnx.ModelProto, args):
    """Compile ONNX to PopEF."""
    model_bytes = model.SerializeToString()
    outputs = [o.name for o in model.graph.output]

    options = CompilerOptions()
```

(continues on next page)

(continued from previous page)

```
    options.num_ipus = 1
    options.ipu_version = runtime.DeviceManager().ipu_hardware_version()
    options.batches_per_step = args.batches_per_step
    options.partials_type = 'half'
    executable = Compiler.compile(model_bytes, outputs, options)

    return executable


def run_synchronous(
    model_runner: runtime.Runner,
    input: RuntimeInput,
    output: RuntimeInput,
    iterations: int,
) -> None:
    deltas = []
    sess_start = time.time()
    for _ in range(iterations):
        start = time.time()
        model_runner.execute(input, output)
        end = time.time()
        deltas.append(end - start)
    sess_end = time.time()

    latency = sum(deltas) / len(deltas) * 1000
    print(f'Latency : {latency:.3f}ms')
    avg_sess_time = (sess_end - sess_start) / iterations * 1000
    print(f'Synchorous avg Session Time : {avg_sess_time:.3f}ms')


def run_asynchronous(
    model_runner: runtime.Runner,
    input: RuntimeInput,
    output: RuntimeInput,
    iterations: int,
) -> None:
    # precreate multiple numbers of outputs
    async_inputs = [input] * iterations
    async_outputs = [output] * iterations
    futures = []

    sess_start = time.time()
    for i in range(iterations):
        f = model_runner.execute_async(async_inputs[i], async_outputs[i])
        futures.append(f)

    # waits all execution ends
    for i, future in enumerate(futures):
        future.wait()
    sess_end = time.time()

    avg_sess_time = (sess_end - sess_start) / iterations * 1000
    print(f'Asyncronous avg Session Time : {avg_sess_time:.3f}ms')

```

(continues on next page)

(continued from previous page)

```
90
91  def run(executable, args):
92      """Run PopEF."""
93      # Create model runner
94      model_runner = runtime.Runner(executable)
95
96      # fix random number generation
97      np.random.seed(2022)
98
99      # Prepare inputs and outpus
100     inputs = {}
101     inputs_info = model_runner.get_execute_inputs()
102     for input in inputs_info:
103         inputs[input.name] = np.random.uniform(0, 1, input.shape).astype(
104             input.numpy_data_type()
105         )
106
107     outputs = {}
108     outputs_info = model_runner.get_execute_outputs()
109     for output in outputs_info:
110         outputs[output.name] = np.zeros(output.shape, dtype=output.numpy_data_type())
111
112     # Run
113     # To correctly generate the popvision report, iteration must be a
114     # multiple of batches_per_step and greater than 2 * batches_per_step
115     iteration = args.batches_per_step * 10
116
117     # warm up device
118     for _ in range(10):
119         model_runner.execute(inputs, outputs)
120
121     run_synchronous(model_runner, inputs, outputs, iteration)
122     run_asynchronous(model_runner, inputs, outputs, iteration)
123
124
125 def default_model():
126     TensorProto = onnx.TensorProto
127     matmul = helper.make_node("MatMul", ["X", "Y"], ["Z"])
128     add = helper.make_node("Add", ["Z", "A"], ["B"])
129     graph = helper.make_graph(
130         [matmul, add],
131         "test",
132         [
133             helper.make_tensor_value_info("X", TensorProto.FLOAT, (4, 4, 8, 16)),
134             helper.make_tensor_value_info("Y", TensorProto.FLOAT, (4, 4, 16, 8)),
135             helper.make_tensor_value_info("A", TensorProto.FLOAT, (4, 4, 8, 8)),
136         ],
137         [helper.make_tensor_value_info("B", TensorProto.FLOAT, (4, 4, 8, 8))],
138     )
139     opset_imports = [helper.make_opsetid("", 11)]
140     original_model = helper.make_model(graph, opset_imports=opset_imports)
141     return original_model
142
143
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Convert onnx model and run it on IPU.'
    )
    parser.add_argument('--onnx_model', type=str, help="Full path of the onnx model.")
    parser.add_argument(
        '--batches_per_step',
        type=int,
        default=100,
        help="The number of on-chip loop count.",
    )
    parser.add_argument('--popef', type=str, help="Full path of the popef file")
    args = parser.parse_args()

    if args.popef:
        run(args.popef, args)
    else:
        if not args.onnx_model:
            print("No onnx model provided, run default model.")
            model = default_model()
        else:
            print(f"Run onnx model {args.onnx_model}")
            model = onnx.load(args.onnx_model)

        converted_model = convert(model, args)
        executable = compile(converted_model, args)
        run(executable, args)
```

CHAPTER

FOUR

COMMAND LINE INTERFACE

poprt is a tool to help quickly deploy ONNX models on IPUs.

```
usage: poprt
       [-h]
       [--apply_hcsr APPLY_HCSR [APPLY_HCSR ...]]
       [--available_memory_proportion AVAILABLE_MEMORY_PROPORTION]
       [--batch_size BATCH_SIZE]
       [--batch_axis BATCH_AXIS]
       [--batches_per_step BATCHES_PER_STEP]
       [--calibration_with_data]
       [--calibration_loss_type {mse,mae,snr,kld,cos_dist,gptq}]
       [--check]
       [--checkpoints CHECKPOINTS]
       [--compiler_options KEY=VAL [KEY=VAL ...]]
       [--config_yaml CONFIG_YAML]
       [--convert_version CONVERT_VERSION]
       [--custom_library_so_paths CUSTOM_LIBRARY_SO_PATHS [CUSTOM_LIBRARY_SO_PATHS ...]]
       [--custom_pass_config CUSTOM_PASS_CONFIG]
       [--custom_shape_inference CUSTOM_SHAPE_INFERENCE]
       [--data_preprocess DATA_PREPROCESS]
       [--disable_compilation_progress_bar]
       [--disable_fast_norm]
       [--eightbitsio]
       [--merge_if_with_same_cond]
       [--enable_compress_pattern]
       [--enable_erf_gelu]
       [--enable_insert_remap]
       [--export_popef]
       [--fold_periodic_initializer]
       [--fp16_skip_op_types FP16_SKIP_OP_TYPES]
       [--enable_avoid_overflow_patterns]
       [--fp8_skip_op_names FP8_SKIP_OP_NAMES]
       [--fp8_params FP8_PARAMS]
       [--framework FRAMEWORK]
       [--infer_shape_ahead]
       [-i INPUT_MODEL]
       [--input_shape INPUT_TENSOR_NAME = INPUT_SHAPE [INPUT_TENSOR_NAME = INPUT_SHAPE ...]]
       [--ipu_version {ipu2,ipu21}]
       [--list_all_passes]
       [--logging_level {DEBUG,INFO,WARNING,ERROR,CRITICAL}]
```

(continues on next page)

(continued from previous page)

```
[--manual_sharding_config MANUAL_SHARDING_CONFIG]
[--max_tensor_size MAX_TENSOR_SIZE]
[--num_io_tiles NUM_IO_TILES]
[--num_of_layers_keep_fp16 NUM_OF_LAYERS_KEEP_FP16]
[--only_manual_sharding]
[--optimize_internal_exchange_code]
[--output_dir OUTPUT_DIR]
[--output_model OUTPUT_MODEL]
[--pack_args KEY=VAL [KEY=VAL ...]]
[--passes PASSES]
[--perf_tuner]
[--popart_options KEY=VAL [KEY=VAL ...]]
[--precision {fp32,fp16,fp8,fp8_weight}]
[--precision_compare]
[--print_completion {bash,zsh}]
[--remap_mode REMAP_MODE]
[--remove_outputs REMOVE_OUTPUTS]
[--run]
[--serialize_matmul KEY=VAL [KEY=VAL ...]]
[--serialize_matmul_add KEY=VAL [KEY=VAL ...]]
[--merge_matmul MERGE_MATMUL]
[--merge_matmul_add MERGE_MATMUL_ADD]
[--merge_moe MERGE_MOE]
[--show]
[--show_io_stat]
[--skip_passes SKIP_PASSES]
[-v]
{tf2onnx}
...
```

4.1 Named Arguments

--apply_hcsr Apply apply_host_concat_split pass, input format: dtype,shape,[num(optional)].For example: --apply_hcsr fp32,[512,1],50 fp32,[512,5]. Default None.

--available_memory_proportion Set the available memory proportion for MatMul, Conv and Gemm Ops. Range (0, 1]. Default None.

--batch_size Set the batch size for all inputs. Works with the batch_axis parameter.

--batch_axis Specify the batch axis for all inputs. Works with the batch_size parameter.

--batches_per_step Set the number of mini-batches to perform on the device before returning to the host. Default: 1.

Default: 1

--calibration_with_data Calibrate the FP8 model using the calibration data. Note that this option only applies when precisionis set to fp8 or fp8_weight.

Default: False

--calibration_loss_type Possible choices: mse, mae, snr, kld, cos_dist, gptq

Choose the calibration method, note that gptq can only be used for calibration of fp8_weight. Default is kld.

Default: ''kld''

--check Use made-up data to check that the model runs.

Default: False

--checkpoints Add intermediate tensor into outputs of graph in order to debug the precision. Default None.

--compiler_options Set PopRT Compiler Options.

--config_yaml Set the path of the yaml config file. Default None.

--convert_version Convert the opset version of ONNX model to CONVERT_VERSION. Default 11.

Default: 11

--custom_library_so_paths Paths of the custom shared library with custom ops/patterns/transforms.

--custom_pass_config Path of the custom pass config file.

--custom_shape_inference Paths of the custom shape inference scripts.For example: *–custom_shape_inference ''./custom_shape_inference_1.py,../ops/custom_shape_inference_2.py''* .

--data_preprocess Path of pickle format file for data preprocessing.

--disable_compilation_progress_bar Do not show compilation progress bar.

Default: False

--disable_fast_norm Do not transfer layer_norm Ops to fast_norm Ops.

Default: False

--eightbitsio Enable 8-bit input/output.

Default: False

--merge_if_with_same_cond enable merge of if Ops that use the same conditional input.

Default: False

--enable_compress_pattern Enable replace Compress patterns with MaskCompress Ops.

Default: False

--enable_erf_gelu Enable replace Erf Gelu patterns with Gelu op.

Default: False

--enable_insert_remap Enable insert remap automatically to improve tensor layout.

Default: False

--export_popef Enable the generation of PopEF model files in the conversion process.

Default: False

--fold_periodic_initializer Fold periodic initializer to save Always-Live memory.

Default: False

--fp16_skip_op_types Set the list of op types which will keep *float32* operands in *float16* mode. Default None.

--enable_avoid_overflow_patterns Enable to keep *float32* for several specific patterns in a *float16* model.

Default: False

--fp8_skip_op_names The names of ops which will remain as *float32* or *float16* in *fp8* mode. For example, ''Conv_1, Conv_2'' . Default None

--fp8_params Set parameters to fp8 model, the format is ''input_format,weight_format,input_scale,weight_scale''

Default: ''F143,F143,-1,-1''

--framework Specify frontend to load input model.

Default: ''onnx''

--infer_shape_ahead Fix input shape and infer shapes at beginning.

Default: False

-i, --input_model Set the path of the original ONNX model.

--input_shape Set the input shape of the model. If the model input is variable, we recommend setting the model input shape. For example: *–input_shape input_ids=1,512 attention_mask=1,512*.

--ipu_version Possible choices: ipu2, ipu21

Set the IPU version: use ipu21 for C600 systems and ipu2 for IPU-M2000 and Bow-2000 systems. Default ipu2.

Default: ''ipu2''

--list_all_passes List all passes. Refer to *–passes*.

Default: False

--logging_level Possible choices: DEBUG, INFO, WARNING, ERROR, CRITICAL

Set the logging level. Default WARNING.

Default: ''WARNING''

--manual_sharding_config Set the path of the yaml config file of sharding and pipelining. Default None.

--max_tensor_size Set max tensor size(bytes) generated by constant_folding. -1 means do not set max_tensor_size by default.For example: *–max_tensor_size 41943040* means constant_folding can only generate tensors smaller than 40MB.

Default: -1

--num_io_tiles Set the number of IPU tiles dedicated to IO. Default 0. IPU run in OverlapIO mode if this number > 0. For more information about OverlapIO see the PopART user guide: https://docs.graphcore.ai/projects/popart-user-guide/en/latest/overlap_io.html.

Default: 0

--num_of_layers_keep_fp16 Set the layer whose loss is topk to fp16 in fp8 quantization.

Default: 0

--only_manual_sharding Only shard the graph in the cli. If enable only_manual_sharding, the cli only supports *–input_model*, *–output_model*, *–output_dir* and *–manual_sharding_config*. *–output_model* and ‘–output_dir are optional.

Default: False

--optimize_internal_exchange_code Enable to optimize the memory usage of internal exchange code.

Default: False

--output_dir Set the output directory where the converted model files and PopEF files are saved. Default current directory.

Default: ‘‘./’’

--output_model Set the name of the converted ONNX model. This will be placed in the *–output_dir* directory.

--pack_args Set the pack args, for example: –pack_args max_valid_num=50 enable_double_batch_unpack=false segment_max_size=13+51

--passes Set the passes to be used during conversion. Default None. For example: *–passes ‘‘pre_scale,fuse_attention’’* . Refer to *–list_all_passes* which is able to show all available passes.

--perf_tuner Enable the performance tuner, unimplemented now.

Default: False

--popart_options Set PopART Session Options. For more information: https://docs.graphcore.ai/projects/popart-python-api/en/latest/api-python.html?highlight=POPART#session-options.

--precision Possible choices: fp32, fp16, fp8, fp8_weight

Quantize the model to the specfied precision. Default fp32.

Default: ‘‘fp32’’

--precision_compare Compare the output precision of conv/matmul/gemm between the origin and the converted model,note that it only take effect when precision is set to fp8 or fp8_weight

Default: False

--print_completion Possible choices: bash, zsh

print shell completion script.

--remap_mode Set the insert position of remap, valid only if enable_insert_remap is set.Must be templated with ‘before/after’ +’ _’ +’ op_type’ (such as after_matmul,before_softmax,after_concat).

Default: ‘‘after_matmul’’

--remove_outputs Remove the specific outputs and useless structures from the graph.

--run Run PopEF with random data.

Default: False

--serialize_matmul Enable to serialize MatMul op to save memory on chip. –serialize_matmul ${OP_NAME}=${FACTOR}/${MODE}/${KEEP_PRECISION} or –serialize_matmul ${OP_NAME}=${FACTOR}/${MODE} or –serialize_matmul ${OP_NAME}=${FACTOR}. ${MODE} choices [input_channels, output_channels, reducing_dim, none]. Default is output_channels. ${KEEP_PRECISION} choices [True, False]. Default is False. For example, –serialize_matmul MatMul_1=4/input_channels/True MatMul_2=4/input_channels MatMul_3=4

--serialize_matmul_add Enable to serialize MatMul weights and Add bias with weights last dim to save memory on chip. –serialize_matmul_add ${MATMUL_OP_NAME}/${ADD_OP_NAME}=${FACTOR} For example, –serialize_matmul_add MatMul_1/Add_2=4

--merge_matmul Enable to merge MatMul operations to save cycles. –merge_matmul ${MATMUL_OP_NAME1},${MATMUL_OP_NAME2} For example, –merge_matmul MatMul_1,MatMul_2

--merge_matmul_add Enable to merge MatMul/Add operations to save cycles. –merge_matmul_add ${MATMUL_OP_NAME1},${ADD_OP_NAME1},${MATMUL_OP_NAME2},${ADD_OP_NAME2} For example, –merge_matmul_add MatMul_1,Add_1,MatMul_2,Add_2

--merge_moe Enable to merge Mixture-of-Experts structure to save cycles. –merge_moe ${EXPERT_BEGIN_OP_NAME1},${EXPERT_END_OP_NAME1},${EXPERT_BEGIN_OP_NAME2},$ For example, –merge_moe MatMul_1,Add_1,MatMul_2,Add_2

--show Show the input and output information of the model.

Default: False

--show_io_stat Show the input/output information statistics of the model.

	Default: False
--skip_passes	Set the list of passes that will be skipped. Default None.
-v, --version	Version of the output tool.
	Default: False

4.2 Sub-commands:

4.2.1 tf2onnx

convert tensorflow model to onnx.

```
poprt tf2onnx [-h] [--saved_model] [--signature_def SIGNATURE_DEF] [--tag TAG]
              [--inputs INPUTS] [--outputs OUTPUTS] [--opset OPSET]
              [--inputs_as_nchw INPUTS_AS_NCHW]
              [--outputs_as_nchw OUTPUTS_AS_NCHW]
```

Named Arguments

--saved_model	Specify if is saved_model. Use input_model to specify model path.
	Default: False
--signature_def	signature_def from saved_model to use.
--tag	tag to use for saved_model.
--inputs	model input_names (optional for saved_model).
--outputs	model output_names (optional for saved_model).
--opset	opset version to use for onnx domain in tf frontend.
	Default: 11
--inputs_as_nchw	transpose inputs as from nhwc to nchw.
--outputs_as_nchw	transpose outputs as from nhwc to nchw.

CHAPTER

FIVE

FEATURES

5.1 Passes

A pass is a component used to optimise an ONNX model in PopRT. To create a custom pass, see the *Custom passes* section.

5.1.1 Pass abstract

Passes in PopRT are all inherited from poprt.Pass.

Passes are auto-registered. All registered passes can be listed with the following CLI command:

```
poprt --list_all_passes
```

or in Python with:

```
import poprt

passes = poprt.get_registered_passes()
print(passes.keys())
```

Use the Python poprt.Pass.get_pass() function to get a pass by its registered name (see *Built-in passes* for the registered names of built-in passes). You can also use poprt.PassManager to apply multiple passes.

Note: Since the structure of the poprt.passes module may change, the following instructions are not recommended:

```
from poprt.passes.float_to_half import Float2Half

Float2Half()(onnx_model)
```

5.2 FP8

5.2.1 IPU FP8 type

FP8, as its name implies, is an 8-bit float data type, with a memory footprint a quarter of that of FP32, half that of FP16 and the same as INT 8. The IPU supports two FP8 data types: F8E4M3 and F8E5M2.

FP8 binary encoding is shown in Table 5.1.

Table 5.1: FP8 Binary Encoding

	E4M3	E5M2
Exponent bias	8	16
Max normal	S.1111.110 = 1.75 * 2^8	S.11111.10 = 1.10 * 2 ^ 16
Min normal	S.0001.000 = 1.0 * 2 ^ -6	S.00001.00 = 1.0 * 2 ^ -14
Max subnorm	S.0000.111 = 0.875 * 2 ^ -6	S.00000.11 = 0.75 * 2 ^ -14
Min subnorm	S.0000.001 = 0.125 * 2 ^ -6	S.00000.01 = 0.25 * 2 ^ -14

As can be seen from Table 5.1, the representation range of E4M3 is smaller, but the precision is slightly higher than E5M2. In addition, compared with FP32/FP16, the dynamic range of FP8 is much smaller. In order to expand the representation range, FP8 also supports adjusting the representation range using an additional scale parameter. When the scale is larger, the representation range is larger, and the precision is lower. When the scale is smaller, the representation range is smaller, and the precision is higher.

For more information about FP8, please refer to 8-bit Numerical Formats for Deep Neural Networks.

5.2.2 FP8 quantisation

Model quantisation is very common in practical applications, among which INT8 quantisation is widely applied. It is a technique that converts all inputs and parameters of the model into INT8 format with minimal loss of precision and accelerates inference speed while reducing memory footprint, mainly including post-training quantisation and quantisation training. The quantisation objects are weights and inputs, both of which introduce additional data reduction layers and increase computational complexity, unlike the direct conversion from FP32 to FP16.

The purpose of FP8 quantisation is also to minimise the loss of precision after conversion. The conversion process is as direct as that of FP32 to FP16. If the model is all FP8, there is no need to modify the model structure; simply convert the inputs and parameters to FP8. If it is a mixed precision model with FP8 and FP16, simply add Cast in the corresponding place. FP8 quantisation will determine a set of scale and format parameters to meet the requirement of minimising precision loss. At present, the IPU does not support the quantisation training of FP8, so only post-training quantisation is discussed here, including the quantisation of weights and inputs.

Regarding weight quantisation, the FP8 operators supported by IPU temporarily only include Conv, MatMul and Cast, so the weights of corresponding operators of the FP16/FP32 model will be taken out and converted into

FP8. During the conversion, different scale and format parameters will be set for each set of weights. The converted weights will then be compared with the weights of FP16/FP32 to compute the loss. The corresponding scale and format of the set with the least loss are the best FP8 parameters for the weights of that layer. The subsequent set of weights will be processed in the same way. The entire conversion process is performed on the CPU, and the losses include the mean square error, mean absolute error, signal-to-noise ratio, kld and cosine distance. Simply specify one during the quantisation.

Regarding input quantisation, the quantisation principle is the same as that of weight quantisation, but the processing method is slightly different. Since the input of each FP8 operator is unknown, it is impossible to directly quantify the input. Therefore, it is necessary to provide some validation data in advance for inference, and then record the input of each FP8 operator. The next step is to find the set of scale and formats with the least loss between the input of FP16/FP32 and the input of FP8 through the traversal method, just like weight quantisation.

5.2.3 Converting an FP32 model to FP8

For operators with high computational density such as MatMul and Conv, FP8 has higher computing power than FP16. Therefore, converting the model from FP32 or FP16 to FP8 can improve model throughput and reduce model latency. In addition, since the IPU runs the entire model on the chip, and FP8 tensors require less storage space than the FP32 or FP16 tensors, the IPU can run larger scale models or support larger batch sizes.

To convert a model from FP32 or FP16 to FP8 means replacing the operator that supports FP32 or FP16 in the model with FP8 and converting the weight tensor corresponding to this operator to an FP8 tensor. Since currently in the IPU only the Conv and MatMul operators support the computation of the FP8 type, the FP8 conversion in PopRT is actually the conversion of the mixed precision model. This means, only the Conv and MatMul operators are converted to FP8, while other operators remain as the original type.

Fig. 5.1 shows an example of converting an ONNX model containing two MatMul operations to FP8. Before the MatMul operation, the output of other ops was in FP16 format, so you need to add a Cast node after these ops to convert them to FP8. Similarly, another weight input of MatMul also needed to be converted to FP8. However, the FP8 conversion of weights is not performed in the model through Cast. Instead, the FP8 conversion of weights in the ONNX model is performed on the CPU in advance, so as to save the Cast of the weight part and improve the inference efficiency. After the conversion is completed, the first FP8 MatMul is executed, and the output type is FP16. Then, when the second MatMul is executed, the input and weights will undergo the same FP8 conversion, and so on, until the entire inference process is completed.

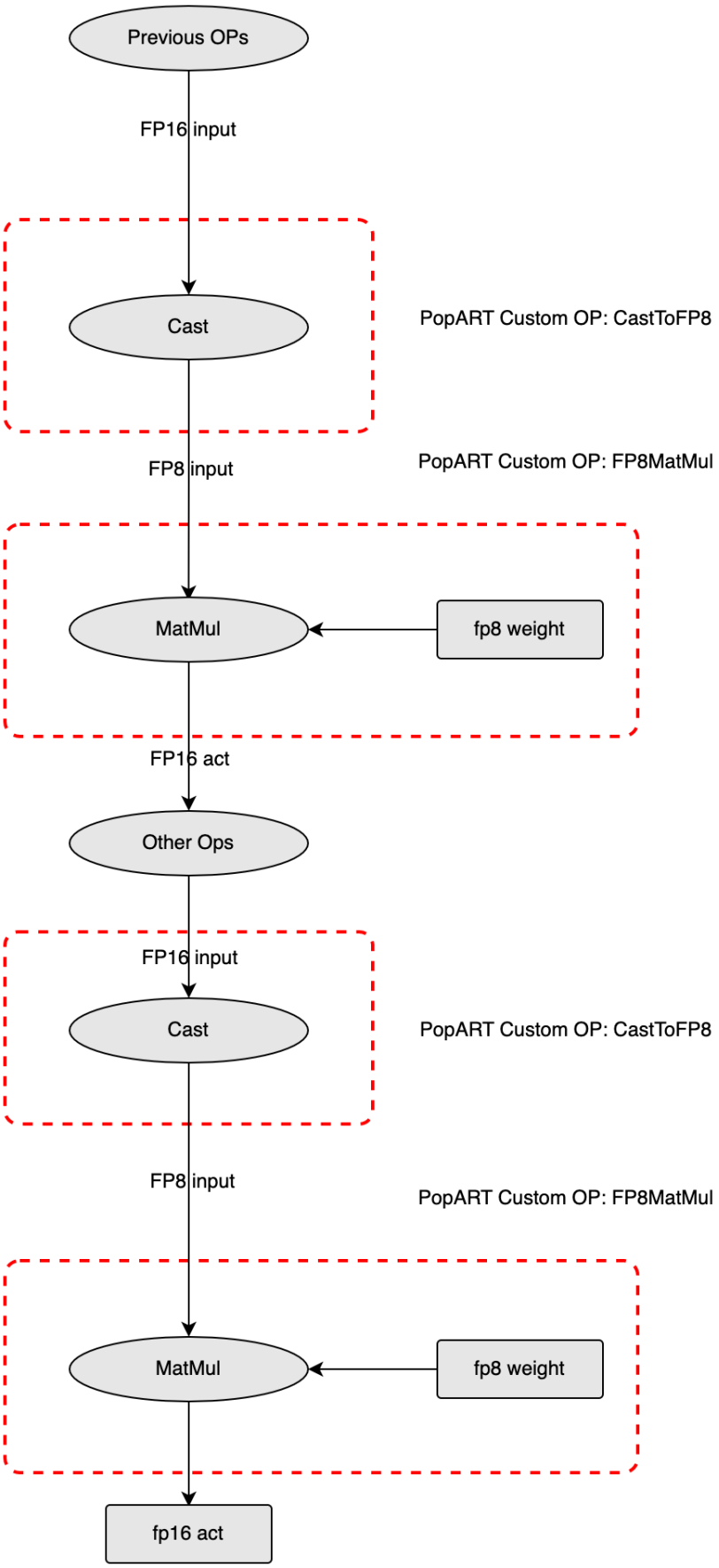


Fig. 5.1: Example showing conversion of an ONNX model to FP8

5.2.4 FP8 model conversion tool

The FP8 model conversion tool currently includes two methods of usage:

1. Use the default scale and format for quick conversion, with values of 0 and F143, respectively. This method can quickly verify whether the model can be successfully converted to FP8 and verify the speed of the FP8 model more conveniently.
2. Enable the quantisation tool of FP8 for conversion. This method will take some time to compute the best scale and format, so as to lower the precision loss of the model and to, obtain the error condition of each layer under different parameters, which facilitates further precision debugging.

We use ResNet50 as an example to demonstrate both methods.

Install the dependencies:

```
pip install torch==1.10.0
pip instal torchvision==0.11.1
```

Download the ResNet50 ONNX model and convert:

```
dummy_input = torch.randn((1, 3, 224, 224))
model = torchvision.models.resnet.resnet50(pretrained=True).eval()
torch.onnx.export(
    model,
    dummy_input,
    'ResNet50.onnx',
    input_names=["input"],
    output_names=["output"],
    opset_version=11,
)
```

The command for quick conversion is as follows:

```
poprt \
    --input_model ResNet50.onnx \
    --output_model ResNet50_FP8.onnx \
    --input_shape input=1,3,224,224 \
    --precision fp8 \
    --fp8_skip_op_names Conv_0,Conv_8,Gemm_121 \
    --fp8_params F143,F143,-3,-3 \
    --convert_version 11
```

`fp8_skip_op_names` and `fp8_params` are used to adjust precision. `fp8_skip_op_names` can specify which layers are retained as FP16, while `fp8_params` is used to specify the format and scale of inputs and weights. If these are not set, the tool will convert all `Conv` `MatMul` and `Gemm` operations in the model to FP8, and the format and scale will use default values.

In addition, the tool also supports using FP8 to store weights and FP16 to perform inference. Compared with the FP16 model, this method can significantly reduce memory footprint with a small increase in latency. The conversion method is basically the same as the above mentioned FP. Simply change the precision parameter to `fp8_weight`.

The command for quantisation conversion is:

```
poprt \
    --input_model ResNet50.onnx \
    --output_model ResNet50_FP8.onnx \
    --input_shape input=1,3,224,224 \
    --precision fp8 \
    --quantize \
    --quantize_loss_type kld \
    --num_of_layers_keep_fp16 3 \
    --data_preprocess calibration_dict.pickle \
    --precision_compare \
    --convert_version 11
```

The quantize parameter indicates that quantisation conversion is enabled, and additional input data is needed for numerical validation. num_of_layers_keep_fp16 is used to keep layers with the loss of topk as FP16 to further improve model precision.

For example, the input name of ResNet50 is input, and the input data is ImageNet. Randomly select approximately 50 images from the ImageNet test set to create a validation set, with a shape of 50*3*224*224, and then save as a pickle file in the form of a dictionary {input:ndarray}. It should be noted that if the model has preprocessing that is not included in the network, the validation data should be saved after preprocessing to ensure that it matches the input of the model.

After quantization is completed, you will get a quantize.log file which records the losses under different scales and formats of the inputs and weights of the FP8 operators. Meanwhile, the set of scales and formats with the least loss will also be given, and this set of parameters is the set of parameters used in the quantised FP8 model. In addition, when quantize_loss_type is not kld, additional information such as mean, variance, skewness or kurtosis before and after quantisation and quantisation noise will be recorded.

If precision_compare is set, the output losses of the middle layers of the FP16 model and FP8 model will be computed, and the results are saved in the precision_compare.log file, which can be used for further precision debugging. The meaning of each indicator and debugging suggestions are as follows:

- Mean square error: The mean square error is used to compute the mean of the sum of squared errors between the original data and the quantised data. The closer it is to 0, the better. The value is affected by the size of the data itself and the volume of the data, so it should be considered together with the condition of the data itself.
- Mean absolute error: The mean absolute error is used to compute the mean of the absolute errors between the original data and the quantised data. The closer it is to 0, the better. The value is affected by the size of the data itself and the volume of the data, so it should be considered together with the condition of the data itself.
- Signal-to-noise ratio: The signal-to-noise ratio is used to compute the ratio of the sum of squared errors between the original data and the quantised data to the sum of squares of the original data. The closer it is to 0, the better. The impact of the size of the data itself and the volume of the data is eliminated, so the error condition can be judged by appropriately comparing this value for each layer.
- Cosine Distance: The cosine distance is used to compute the angle between the original data and the quantised data after expansion into one dimension. The closer it is to 0, the better. It is not affected by the size of the data itself and the volume of the data, and the range is between 0 and 1. These characteristics make it a very intuitive and important evaluation indicator. This error condition can be judged by appropriately

comparing this value for each layer. There may be an order of magnitude difference between some layers.

- Noise skewness: Noise skewness is used to compute the skewness of the difference between the original data and the quantised data to measure the asymmetry of the distribution. If the skewness is greater than 0, the distribution is skewed to the right, and if the skewness is less than 0, the distribution is skewed to the left. Meanwhile, the larger the absolute value of skewness, the more severe the skewness of the distribution.
- Noise kurtosis: Noise kurtosis is used to compute the kurtosis of the difference between the original data and the quantised data to measure the steepness or smoothness of the distribution. If the kurtosis is close to 0, the distribution kurtosis follows the normal distribution. If the kurtosis is greater than 0, the distribution is steeper, and if the kurtosis is less than 0, the distribution is shorter and fatter.
- Noise histogram: The noise histogram is used to compute the difference between the original data and the quantised data. The histogram is divided into 32 bins to measure the overall distribution of the data.

Precision_compare.log provides the mean, standard deviation, minimum, maximum, skewness, kurtosis and histogram indicators of the original data, the quantised data and the error between the two. These indicators are used to characterise the overall distribution shape of the data and help users to compare the data differences before and after quantisation from a statistical perspective.

5.2.5 Debugging FP8 model conversion problems

If the precision is still poor, try the following:

1. For quick conversion, if it is a CV type model, it is recommended to keep the first Conv and last MatMul as FP16; if it is a NLP type model, it is recommended to keep the last MatMul as FP16. Use the F143 format as the precision of the FP8 format of E4M3 is higher in the inference process. Select an integer value between -5 and 0 for scale, and you can try to select the one with the highest precision one by one.
2. For quantisation conversion, the validation set should be selected from the real data, and the volume of the data should not be too large; otherwise, the quantisation process will be very long. For example, for ResNet50, approximately 20 images can be selected as the validation set. The actual situation needs to be adjusted according to the size of the model; if the quantisation speed is too slow, the volume of the data can be appropriately reduced.

5.3 Overlap I/O

The IPU execution of an inference model is generally divided into three stages:

1. Load: Copy input data from the host to the IPU
2. Compute: Model computing
3. Store: Copy result data from the IPU to the host

These three stages are executed serially, which means that the computing resources of the IPU are idle while data is transferred in the Load and Store stages. In some models, with large input and output data, the performance of the whole model is limited by I/O. For this kind of model, enabling overlap I/O can overlap the computing stage and the I/O stage. This improves the utilisation rate of the computing resources of the IPU.

5.3.1 Principle

The principle of overlap I/O is to divide all the tiles on the IPU into two groups, namely compute tiles and I/O tiles. Compute tiles perform all computation, while I/O tiles only handle transferring data with the host. In this way, the stages of Load, Compute and Store form a three-level pipeline in a computational flow, which overlaps compute and I/O to improve the utilisation rate of the computing resources of the IPU.

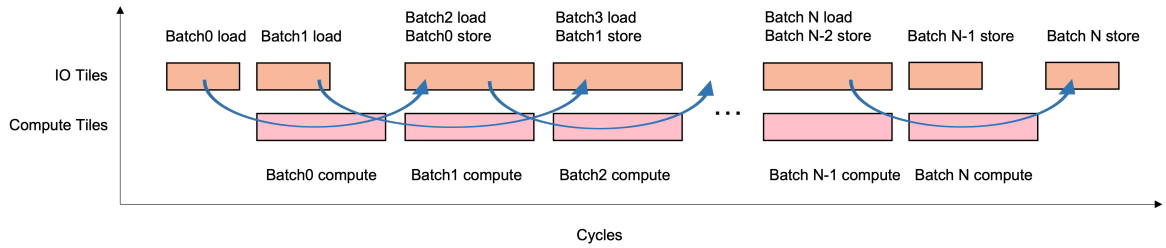


Fig. 5.2: Pipeline formed by Load/Compute/Store

5.3.2 Configuring I/O tiles

To enable overlap I/O, you only need to set one parameter, the number of I/O tiles. The number of I/O tiles can be adjusted to optimise the throughput of the transmission. To calculate the number of I/O tiles, you can divide the sum of the tensor sizes of all input and output by the SRAM size available for each tile and then round to the next power of 2.

- Configuring I/O tiles with the PopRT CLI `--num_io_tiles` parameter:

```
poprt \
    --input_model model.onnx \
    --export_popef \
    --output_dir model \
    --num_io_tiles 128
```

- Configuring I/O tiles with the `poprt.compiler.CompilerOptions` API:

```
opts = poprt.compiler.CompilerOptions()
opts.num_io_tiles = 128
```

5.3.3 Debugging

You can use the PopVision Graph Analyser to display the overlap between I/O and Compute stages to determine whether overlap I/O would improve your throughput. Fig. 5.3 shows an example of the output of the PopVision Graph Analyser.

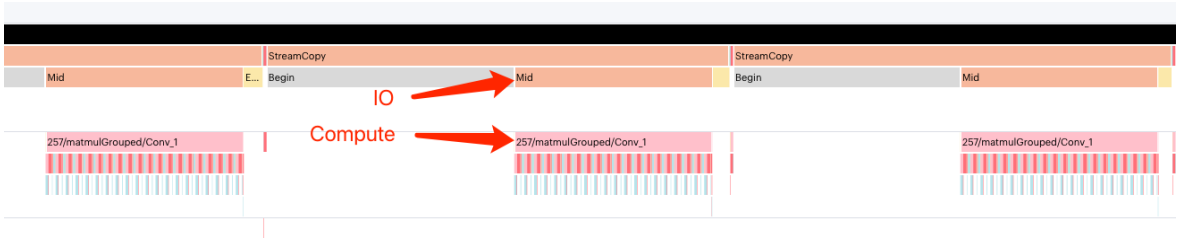


Fig. 5.3: PopVision Graph Analyser shows overlap between I/O and Compute

5.3.4 Concurrent requests

Since the three stages of inference form a three-stage pipeline through overlap I/O, sufficient concurrent data must be fed to the IPU in order to keep the pipeline full. At least three threads are required to feed the concurrent data to the IPU through the multi-threading mode.

5.3.5 Example

The following is a simple example using overlap I/O:

Listing 5.1: simple_overlapio.py

```
# Copyright (c) 2022 Graphcore Ltd. All rights reserved.
import argparse
import threading

import numpy as np
import onnx

from onnx import helper

from poprt import runtime
from poprt.compiler import Compiler, CompilerOptions
from poprt.runtime import RuntimeConfig

'''
PopRT use OverlapInnerLoop strategy as default exchange strategy.
There are two loops in the main program: outer loop and inner loop.
Each batch data needs to be processed in three pipeline stages: load/compute/store.
Therefore, in order to enable the pipeline to run normally, at least three threads
are required to feed data to the pipeline at the same time.
================================================================
OverlapInnerLoop:
- Boxes denote subgraphs / subgraph Ops / loops
- Inputs/outputs are loop carried in order

.- outer loop ---------------------------------------.
|                  .- inner loop -.                   |
| load - compute - | - store      |                  |
|           load - | - compute -- | - store          |
|                  |   load ----- | - compute - store |
```

(continues on next page)

(continued from previous page)

```
|               '--------------'                |
'-----------------------------------------------'
        ^^^^^^^^      ^^^^^^^^      ^^^^^^^^
        overlap       overlap       overlap

================================================================================
'''


def compile(model: onnx.ModelProto, args):
    """Compile ONNX to PopEF."""
    model_bytes = model.SerializeToString()
    outputs = [o.name for o in model.graph.output]

    options = CompilerOptions()
    options.batches_per_step = args.batches_per_step
    options.num_io_tiles = args.num_io_tiles

    executable = Compiler.compile(model_bytes, outputs, options)
    return executable


def run(executable, args):
    """Run PopEF."""
    # Create model runner
    config = RuntimeConfig()
    config.timeout_ns = 0
    # Create model runner
    model_runner = runtime.Runner(executable, config)

    inputs_info = model_runner.get_execute_inputs()
    outputs_info = model_runner.get_execute_outputs()

    # Run in multiple threads
    def execute(bps, inputs_info, outputs_info):
        inputs = {}
        outputs = {}

        for input in inputs_info:
            inputs[input.name] = np.random.uniform(0, 1, input.shape).astype(
                input.numpy_data_type()
            )
        for output in outputs_info:
            outputs[output.name] = np.zeros(
                output.shape, dtype=output.numpy_data_type()
            )

        # To correctly generate the popvision report, iteration must be a
        # multiple of batches_per_step and greater than 2 * batches_per_step
        # There are 3 threads, so the total number feed into IPU is 3 * iteration
        iteration = bps
        for _ in range(iteration):
            model_runner.execute(inputs, outputs)

```

(continues on next page)

(continued from previous page)

```
    threads = []
    num_threads = 3
    print(f"Run PopEF with {num_threads} threads.")
    for _ in range(num_threads):
        threads.append(
            threading.Thread(
                target=execute, args=(args.batches_per_step, inputs_info, outputs_info)
            )
        )

    for t in threads:
        t.start()

    for t in threads:
        t.join()
    print(f"Complete.")


def default_model():
    TensorProto = onnx.TensorProto

    nodes = []
    num_matmuls = 4
    nodes.append(helper.make_node("Expand", ["input", "shape"], ["Act0"]))
    for i in range(num_matmuls):
        nodes.append(helper.make_node("MatMul", [f"Act{i}", "Weight"], [f"Act{i+1}"]))
    nodes.append(
        helper.make_node("ReduceMean", [f"Act{num_matmuls}"], ["output"], axes=[0, 1])
    )

    graph = helper.make_graph(
        nodes,
        "matmul_test",
        [
            helper.make_tensor_value_info("input", TensorProto.FLOAT, (256, 256)),
        ],
        [helper.make_tensor_value_info("output", TensorProto.FLOAT, (256, 256))],
        [
            helper.make_tensor(
                "shape",
                TensorProto.INT64,
                [4],
                np.array([4, 4, 256, 256], dtype=np.int64),
            ),
            helper.make_tensor(
                "Weight",
                TensorProto.FLOAT,
                (4, 4, 256, 256),
                np.random.randn(4, 4, 256, 256),
            ),
        ],
    )
    opset_imports = [helper.make_opsetid("", 11)]
    original_model = helper.make_model(graph, opset_imports=opset_imports)
```

(continues on next page)

(continued from previous page)

```
    return original_model


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Convert onnx model and run it on IPU.'
    )
    parser.add_argument(
        '--batches_per_step',
        type=int,
        default=16,
        help="The number of on-chip loop count.",
    )
    parser.add_argument(
        '--num_io_tiles',
        type=int,
        default=192,
        help="The number of IO tiles.",
    )
    args = parser.parse_args()
    model = default_model()
    exec = compile(model, args)
    run(exec, args)
```

5.4 Dynamic batch size

5.4.1 Background

Since the IPU only supports static graphs, the fixed batch size needs to be specified during the model compilation stage. However, in the actual inference process, the batch size of the input data is usually not fixed. PopRT supports input data of any batch size by setting the following:

- $\mathrm{timeout\_ns}$: The timeout period for PopRT to wait for data to form the batch size required by the PopEF model. The default setting is 5 ms.
- $\mathrm{batching\_dim}$: The dimension that contains the value for the batch size. The default value is $\mathrm{0xFFFFFFFF}$, indicating that dynamic batch sizing is disabled.

Each batch of inference data processed on the IPU is based on the batch size of the loaded model. For example, the shape of the model in Fig. 5.4 is [4,2], and the dimension of the batch size is 0. This means that data with a batch size of 4 is processed each time. When the batch size of the input data is N (where N is an integer greater than or equal to 1) times the batch size of the model, the inference result is returned after N times of inference on the IPU. In this case, there is no need to wait for the $\mathrm{timeout\_ns}$ timeout.

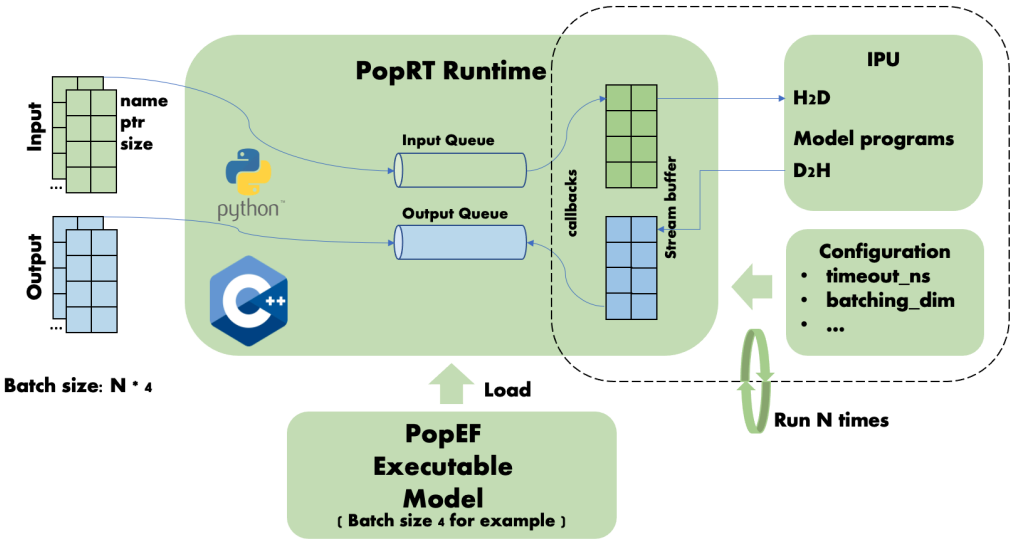


Fig. 5.4: Dynamic batch size

If the batch size of the input data is N (where N is an integer greater than or equal to 1) times the batch size of the model plus M (where M is an integer smaller than or equal to the batch size), the N times data processing is the same as the processing in Fig. 5.4, data M being smaller than the batch size of the model must wait for the timeout set by timeout_ns. If one or more subsequent sets of data are merged with the data M within this time and the batch size is reached, then an inference will be performed to obtain the result. If the data fails to reach the batch size before the timeout, the missing part will be set to 0 and an inference will be performed to obtain the result. As shown in Fig. 5.5, after two requests, batch sizes 1 and 2 are merged, the timeout is reached, and the remaining data with batch size 1 will be set to 0.

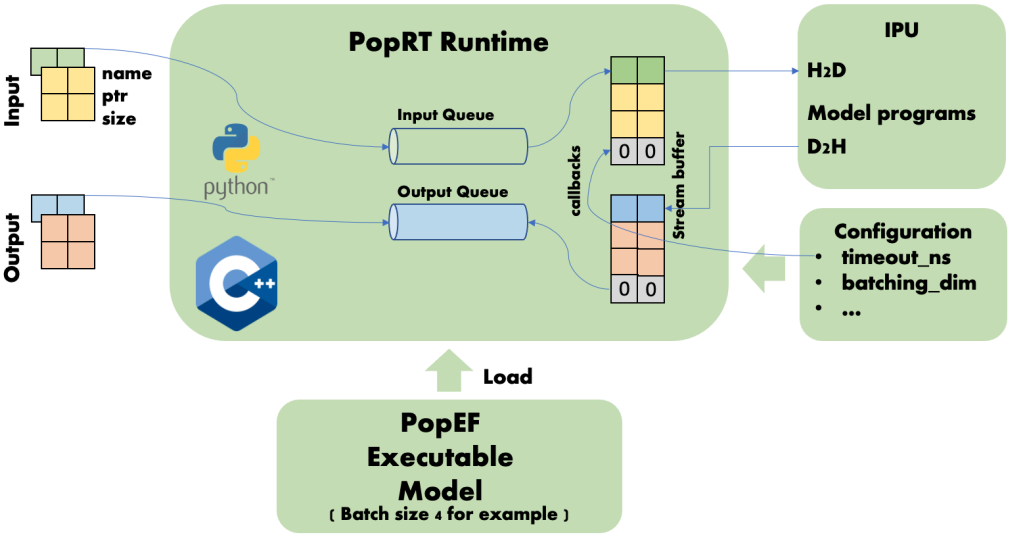


Fig. 5.5: Dynamic batch size timeout

The function of dynamic batch sizing is transparent to the user program. Users do not need to worry about the batch size of the loaded model in the current IPU. Just send the inference request data according to the requirements of the application.

5.4.2 Example

Listing 5.2 shows an example using dynamic batch sizing. In the example, we create a model with an input shape of [4, 2] and a batch size of 4. The application uses data with batch sizes of 1, 4 and 7, respectively, for inference, without considering the batch size of the loaded model.

Listing 5.2: dynamic_batch_size.py

```
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import numpy as np
import numpy.testing as npt
import onnx

from onnx import helper

from poprt import runtime
from poprt.compiler import Compiler
from poprt.runtime import RuntimeConfig


def default_model():
    """Create a test model."""
    TensorProto = onnx.TensorProto
    add = helper.make_node("Add", ["X", "Y"], ["O"])
    graph = helper.make_graph(
        [add],
        "test",
        [
            helper.make_tensor_value_info("X", TensorProto.FLOAT, (4, 2)),
            helper.make_tensor_value_info("Y", TensorProto.FLOAT, (4, 2)),
        ],
        [helper.make_tensor_value_info("O", TensorProto.FLOAT, (4, 2))],
    )
    opset_imports = [helper.make_opsetid("", 11)]
    original_model = helper.make_model(graph, opset_imports=opset_imports)
    return original_model


def compile(model: onnx.ModelProto):
    """Compile ONNX to PopEF."""
    model_bytes = model.SerializeToString()
    outputs = [o.name for o in model.graph.output]
    executable = Compiler.compile(model_bytes, outputs)
    return executable


def run(executable):
    """Run PopEF."""
    config = RuntimeConfig()
    config.timeout_ns = 300 * 1000  # 300us
    config.batching_dim = 0
    model_runner = runtime.Runner(executable, config)
    batch_sizes = [1, 4, 7]
    for batch_size in batch_sizes:
```

(continues on next page)

(continued from previous page)

```
        inputs = {}
        inputs['X'] = np.random.uniform(0, 1, [batch_size, 2]).astype(np.float32)
        inputs['Y'] = np.random.uniform(0, 1, [batch_size, 2]).astype(np.float32)

        outputs = {}
        outputs['O'] = np.zeros([batch_size, 2], dtype=np.float32)
        model_runner.execute(inputs, outputs)
        expected = inputs['X'] + inputs['Y']
        npt.assert_array_equal(
            outputs['O'],
            expected,
            f"Result: outputs['O'] not equal with expected: {expected}",
        )
        print(f'Successfully run with input data in batch size {batch_size}')


if __name__ == '__main__':
    model = default_model()
    executable = compile(model)
    run(executable)
```

When the example has run, you should see the following output:

```
Successfully run with input data in batch size 1
Successfully run with input data in batch size 4
Successfully run with input data in batch size 7
```

5.5 Packing

5.5.1 Background

Currently, the IPU only supports static graphs. The input shape of the model needs to be fixed, since the dynamic shape will cause the model to recompile. However, in practical applications, especially those in natural language processing, the input sequence length of the model is often dynamic. In this case, the conventional processing method is to pad the variable length data to the max sequence length and then input it into the model. However, this method will lead to a lot of invalid computing, resulting in a low utilisation rate of the IPU. In this case, you can use packing to support dynamic sequence length and improve IPU utilisation.

5.5.2 Packing and unpacking

Fig. 5.6 shows an example to illustrate packing and unpacking. Assume that the maximum input length of the model is 8 and the batch size is 4. Currently, there are 7 requests of batch size 1 of different lengths. The length ranges from 1 to 7, and 0 indicates that the pad has invalid data.

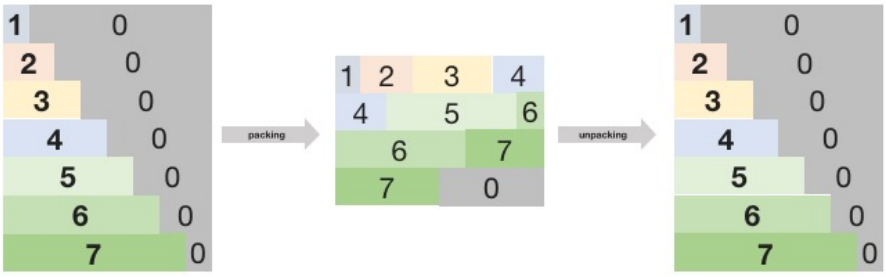


Fig. 5.6: Packing and unpacking

5.5.3 Transformer-based NLP models

Since its introduction in 2017, the application fields of transformer structures have grown considerably, from NLP in the beginning to ASR, CV, DLRM and other fields today. Transformers contain encoders and decoders. This section only focuses on encoders. The structure of a transformer encoder is shown in Fig. 5.7.

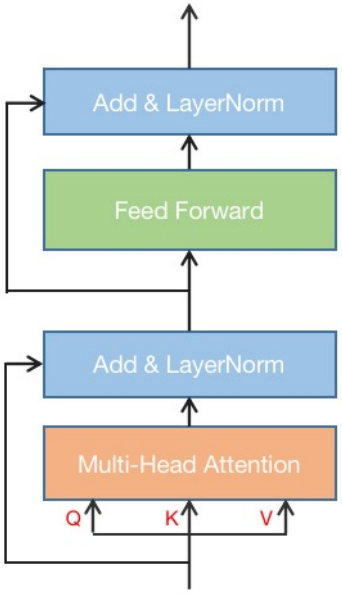


Fig. 5.7: Transformer encoder

Taking BERT as an example, the input shape of the encoder is usually [batch_size, seq_len, hidden_size]. In the encoder, except for the Multi-head Attention module, the computing of other modules is only performed in the last dimension. Therefore, for these modules, packing can be used to reduce invalid computing. For the Multi-head Attention module, since the correlation between tokens needs to be computed, the computing must be performed after unpacking without modifying the mask. Then, re-packing after the computing of Multi-head Attention is completed. The computing process can be represented by the following pseudocode:

```
packed_input from host
activation = packed_input
for encoer in encoders:
    Unpacking
```

(continues on next page)

(continued from previous page)

```
    Attention
    Packing
    Add & LayerNorm
    Feed-Forward
    Add & LayerNorm
    Update activation
Unpacking
unpacked_output to host
```

5.5.4 How to use packing

This section takes the Bert-Base-Squad model as an example. To run this example, you will need Ubuntu 20.04 and Python 3.8.15.

The complete code for this example is:

Listing 5.3: packed_bert_example.py

```
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import argparse
import csv
import os
import queue
import random
import sys
import tempfile
import threading
import time

from multiprocessing.pool import ThreadPool
from queue import Queue

import numpy as np
import packing_utils

from sklearn.metrics import mean_absolute_error

from poprt import runtime
from poprt.backend import get_session

np.random.seed(2023)
INPUT_IDS = "input_ids"
POSITION_IDS = "position_ids"
ATTENTION_MASK = "attention_mask"
TOKEN_TYPE_IDS = "token_type_ids"
UNPACK_INFO = "unpack_info"
OUTPUT2 = "start_logits"
OUTPUT1 = "end_logits"


class BertInputs(object):
    def __init__(
```

(continues on next page)

(continued from previous page)

```
        self,
        input_ids,
        attention_mask,
        token_type_ids,
        position_ids,
        unpack_info,
        input_len,
    ):
        self.input_ids = input_ids
        self.attention_mask = attention_mask
        self.token_type_ids = token_type_ids
        self.position_ids = position_ids
        self.input_len = input_len
        self.unpack_info = unpack_info


def get_synthetic_data(args):
    input_len = np.random.normal(
        args.avg_seq_len, args.avg_seq_len, size=args.dataset_size
    ).astype(np.int32)
    input_len = np.clip(input_len, 1, args.max_seq_len)

    datasets = []
    for s_len in input_len:
        input_ids = np.random.randint(0, args.emb_size, (s_len)).astype(np.int32)

        attention_mask = np.ones(s_len).astype(np.int32)
        token_type_ids = np.random.randint(0, 2, (s_len)).astype(np.int32)

        position_ids = np.arange(s_len).astype(np.int32)
        unpack_info = np.zeros(args.max_valid_num).astype(np.int32)

        feature = BertInputs(
            input_ids, attention_mask, token_type_ids, position_ids, unpack_info, s_len
        )
        datasets.append(feature)

    return datasets


def dump_results(model_name, results):
    fieldnames = [OUTPUT1, OUTPUT2]
    filename = os.path.basename(model_name)[:-4] + 'csv'
    with open(filename, 'w') as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        for result in results:
            dict_name2list = {
                OUTPUT1: result[OUTPUT1],
                OUTPUT2: result[OUTPUT2],
            }
            writer.writerow(dict_name2list)


## create batched inputs and pad samples to max_seq_len
```

(continues on next page)

(continued from previous page)

```
def padding_data(datasets, index, args):
    feed_dicts = {}
    feed_dicts[INPUT_IDS] = np.zeros(
        (args.batch_size, args.max_seq_len), dtype=np.int32
    )
    feed_dicts[ATTENTION_MASK] = np.zeros(
        (args.batch_size, args.max_seq_len), dtype=np.int32
    )
    feed_dicts[POSITION_IDS] = np.zeros(
        (args.batch_size, args.max_seq_len), dtype=np.int32
    )
    feed_dicts[TOKEN_TYPE_IDS] = np.zeros(
        (args.batch_size, args.max_seq_len), dtype=np.int32
    )

    for i in range(args.batch_size):
        input_len = datasets[index].input_len
        feed_dicts[INPUT_IDS][i][:input_len] = datasets[index].input_ids
        feed_dicts[ATTENTION_MASK][i][:input_len] = datasets[index].attention_mask
        feed_dicts[POSITION_IDS][i][:input_len] = datasets[index].position_ids
        feed_dicts[TOKEN_TYPE_IDS][i][:input_len] = datasets[index].token_type_ids
        index = index + 1
    return feed_dicts


# online pack, samples feeded to IPU can reach to maximum num of batches in each running turn
def run_packing_model_with_pack_runner_unpack_repack(args, datasets):
    tmpdir = tempfile.TemporaryDirectory()
    # export popef for PackRunner
    get_session(
        args.model_with_packing_unpack_repack,
        1,
        "poprt",
        output_dir=tmpdir.name,
        export_popef=True,
    ).load()
    config = runtime.PackRunnerConfig(
        timeout_microseconds=args.timeout_microseconds,
        # max_valid_num=args.max_valid_num,
        # dynamic_input_name=args.dynamic_input_name,
    )

    popef_path = tmpdir.name + '/executable.popef'
    # popef_path = "/popconverter/examples/packed_bert_example/executable.popef"
    pack_runner = runtime.Runner(popef_path, config)

    result_queue = queue.Queue()
    results = []
    start_time = time.time()
    for i in range(args.dataset_size):
        feed_dicts = {
            INPUT_IDS: datasets[i].input_ids,
            ATTENTION_MASK: datasets[i].attention_mask,
            TOKEN_TYPE_IDS: datasets[i].token_type_ids,
```

(continues on next page)

(continued from previous page)

```
            POSITION_IDS: datasets[i].position_ids,
            # unpack_info should be hidden from user in the future
            UNPACK_INFO: np.zeros(args.max_valid_num).astype(np.int32),
        }
        out_dict = {
            OUTPUT1: np.zeros([args.max_seq_len]).astype(np.float16),
            OUTPUT2: np.zeros([args.max_seq_len]).astype(np.float16),
        }
        future = pack_runner.execute_async(feed_dicts, out_dict)
        result_queue.put((future, out_dict))
    result_queue.put((None, None))
    while True:
        future, out_dict = result_queue.get()
        if future == None:
            break
        future.wait()
        results.append(out_dict)
    end_time = time.time()

    tput = args.dataset_size / (end_time - start_time)
    latency_ms = (end_time - start_time) / args.dataset_size
    print(
        f"[Pack Online Unpack Repack] Throughput: {tput} samples/s, Latency : {latency_ms * 1000} ms"
    )

    if args.dump_results:
        dump_results(
            "online_unpack_repack" + args.model_with_packing_unpack_repack, results
        )

    tmpdir.cleanup()
    return results


# offline pack, samples feeded to IPU can reach to maximum num of batches in each running turn
# model with pack / unpack ops
def run_packing_model_with_model_runner(args, datasets, model_path, across_rows):
    run_queue = queue.Queue()
    start_time = time.time()
    index = 0
    for i in range(0, args.dataset_size):
        transfer = packing_utils.pack_data(
            datasets,
            index,
            args.batch_size,
            seq_len=256,
            max_valid_num=args.max_valid_num,
            segment_num=1,
            across_rows=across_rows,
        )

        run_queue.put(transfer)
        index = transfer.count
        if index == args.dataset_size:
```

(continues on next page)

(continued from previous page)

```
        break
run_queue.put(None)
duration_of_packing = time.time() - start_time
mean_latency_of_padding_us = duration_of_packing * 1e6 / args.dataset_size

print(f"Mean latency of packing data: {mean_latency_of_padding_us} us/sam")
print(f"Total latency of packing data: {duration_of_packing} s")

sess = get_session(model_path, 1, "poprt").load()

pool = ThreadPool(processes=1)

def execute(feed_dicts, valid_num):
    outputs = sess.run([OUTPUT1, OUTPUT2], feed_dicts)
    res = []
    if across_rows:
        for i in range(valid_num):
            res1 = outputs[0][i].copy().tolist()
            res2 = outputs[1][i].copy().tolist()
            res.append({OUTPUT1: res1, OUTPUT2: res2})
    else:
        outlen = len(outputs[0][0])
        for index in range(len(feed_dicts[ATTENTION_MASK])):
            start = 0
            arr = np.array(feed_dicts[ATTENTION_MASK][index])
            while start < outlen and arr[start] > 0:
                arr = arr - 1
                zero_num = len(arr) - np.count_nonzero(arr)
                out1 = [0] * outlen
                out2 = [0] * outlen
                out1[:zero_num] = outputs[0][index][start : start + zero_num]
                out2[:zero_num] = outputs[1][index][start : start + zero_num]
                res.append({OUTPUT1: out1, OUTPUT2: out2})
                start += zero_num
    return res

asy_results = []

total_start_time = time.time()
while True:
    input_data = run_queue.get()
    if input_data is None:
        break

    feed_dicts = {
        INPUT_IDS: input_data.data[INPUT_IDS],
        ATTENTION_MASK: input_data.data[ATTENTION_MASK],
        TOKEN_TYPE_IDS: input_data.data[TOKEN_TYPE_IDS],
        POSITION_IDS: input_data.data[POSITION_IDS],
        # unpack_info should be hidden from user in the future
        UNPACK_INFO: input_data.unpack_info,
    }
    if not across_rows:
        feed_dicts.pop(UNPACK_INFO)
```

(continues on next page)

(continued from previous page)

```
        valid_num = len(input_data.specs)
        async_result = pool.apply_async(execute, (feed_dicts, valid_num))
        asy_results.append(async_result)

    results = []
    for asy in asy_results:
        for res in asy.get():
            results.append(res)
    total_end_time = time.time()

    tput = len(results) / (total_end_time - total_start_time)
    latency = (total_end_time - total_start_time) / len(results)
    if across_rows:
        print(
            f"[Pack Offline Unpack Repack] Throughput: {tput} samples/s, Latency: {latency*1000} ms"
        )
    else:
        print(
            f"[Pack Offline AttentionMask] Throughput: {tput} samples/s, Latency: {latency*1000} ms"
        )

    if args.dump_results:
        dump_results("offline_" + model_path, results)

    return results


# online pack, samples feeded to IPU can reach to maximum num of batches in each running turn
# model only add AttentionMask op in this mode
def run_packing_model_with_pack_runner_attention_mask(args, datasets, algo):
    tmpdir = tempfile.TemporaryDirectory()
    # export popef for PackRunner
    get_session(
        args.model_with_packing_attention_mask,
        1,
        "poprt",
        output_dir=tmpdir.name,
        export_popef=True,
    ).load()
    config = runtime.PackRunnerConfig(
        timeout_microseconds=args.timeout_microseconds,
        max_valid_num=args.max_valid_num,
        dynamic_input_name=args.dynamic_input_name,
    )

    if algo == "next_fit":
        config.algorithm = runtime.PackAlgorithm.next_fit
    else:
        config.algorithm = runtime.PackAlgorithm.first_fit

    config.enable_input_single_row_mode("attention_mask")
    popef_path = tmpdir.name + '/executable.popef'
    # popef_path = "/popconverter/examples/packed_bert_example/executable.popef"
```

(continues on next page)

(continued from previous page)

```
    pack_runner = runtime.Runner(popef_path, config)

    result_queue = queue.Queue()
    results = []
    start_time = time.time()
    for i in range(args.dataset_size):
        feed_dicts = {
            INPUT_IDS: datasets[i].input_ids,
            ATTENTION_MASK: datasets[i].attention_mask,
            TOKEN_TYPE_IDS: datasets[i].token_type_ids,
            POSITION_IDS: datasets[i].position_ids,
        }
        out_dict = {
            OUTPUT1: np.zeros([args.max_seq_len]).astype(np.float16),
            OUTPUT2: np.zeros([args.max_seq_len]).astype(np.float16),
        }
        future = pack_runner.execute_async(feed_dicts, out_dict)
        result_queue.put((future, out_dict))
    result_queue.put((None, None))
    while True:
        future, out_dict = result_queue.get()
        if future == None:
            break
        future.wait()
        results.append(out_dict)
    end_time = time.time()

    tput = args.dataset_size / (end_time - start_time)
    latency_ms = (end_time - start_time) / args.dataset_size
    print(
        f"[Pack Online AttentionMask({algo})] Throughput: {tput} samples/s, Latency : {latency_ms * 1000} ms"
    )

    if args.dump_results:
        dump_results(
            "online_attention_mask_"
            + algo
            + "_"
            + args.model_with_packing_attention_mask,
            results,
        )

    tmpdir.cleanup()
    return results


def latency_distribuion_with_pack_runner_attention_mask(args, datasets, algo):
    tmpdir = tempfile.TemporaryDirectory()
    # export popef for PackRunner
    get_session(
        args.model_with_packing_attention_mask,
        1,
        "poprt",
        output_dir=tmpdir.name,
```

(continues on next page)

(continued from previous page)

```
    export_popef=True,
).load()
config = runtime.PackRunnerConfig(
    timeout_microseconds=args.timeout_microseconds,
    max_valid_num=args.max_valid_num,
    dynamic_input_name=args.dynamic_input_name,
)

if algo == "next_fit":
    config.algorithm = runtime.PackAlgorithm.next_fit
else:
    config.algorithm = runtime.PackAlgorithm.first_fit

config.enable_input_single_row_mode("attention_mask")
popef_path = tmpdir.name + '/executable.popef'
# popef_path = "/popconverter/examples/packed_bert_example/executable.popef"
pack_runner = runtime.Runner(popef_path, config)

sample_num = args.batch_size * args.iterations
clients = int(args.batch_size * 3.5)
count_percent = 0.6

q = Queue()

def perf_count(model_runner, iteration):
    durations = []
    for i in range(sample_num):
        start_time = time.time()
        random.randint(0, sample_num)
        feed_dicts = {
            INPUT_IDS: datasets[i].input_ids,
            ATTENTION_MASK: datasets[i].attention_mask,
            TOKEN_TYPE_IDS: datasets[i].token_type_ids,
        }
        out_dict = {
            OUTPUT1: np.zeros([args.max_seq_len]).astype(np.float16),
            OUTPUT2: np.zeros([args.max_seq_len]).astype(np.float16),
        }
        pack_runner.execute(feed_dicts, out_dict)
        end_time = time.time()
        durations.append((start_time, end_time))
    # remove first and last example's time counter
    ignored_samples = int(sample_num * (1 - count_percent) / 2)
    durations = durations[ignored_samples:-ignored_samples]
    q.put(durations, timeout=10)

thp = [
    threading.Thread(target=perf_count, args=(pack_runner, args.iterations))
    for _ in range(clients)
]
for t in thp:
    t.start()
for t in thp:
    t.join()
```

(continues on next page)

(continued from previous page)

```
    durations_from_th = []
    while not q.empty():
        durations_from_th += q.get()
    max_timestamp = max(y for _, y in durations_from_th)
    min_timestamp = min(x for x, _ in durations_from_th)
    clients * (sample_num * count_percent) / (max_timestamp - min_timestamp)
    times_range = [y - x for x, y in durations_from_th]

    times_range.sort()
    tail_latency = round(times_range[int(len(times_range) * 0.99)] * 1000, 2)
    avg_latency = round(sum(times_range) / len(times_range) * 1000, 2)

    print(f"Average Latency: {avg_latency}ms, P99 latency: {tail_latency}ms.")
    return tail_latency, avg_latency


# no pack, padding each line with 0 if input length is not long enough.
# samples num equals to batch at every running turn
def run_original_model_with_model_runner(args, datasets):
    run_queue = queue.Queue()
    start_time = time.time()
    for i in range(0, args.dataset_size, args.batch_size):
        feed_dicts = padding_data(datasets, i, args)
        run_queue.put((args.batch_size, feed_dicts))
    run_queue.put((0, None))
    duration_of_padding_s = time.time() - start_time

    mean_latency_of_padding_us = duration_of_padding_s * 1e6 / args.dataset_size
    print(f"Mean latency of padding data: {mean_latency_of_padding_us} us/sam")
    print(f"Total latency of padding data: {duration_of_padding_s} s")

    sess = get_session(args.model_without_packing, 1, "poprt").load()

    asy_results = []

    def execute(feed_dicts, valid_num):
        outputs = sess.run([OUTPUT1, OUTPUT2], feed_dicts)
        res = []
        for i in range(valid_num):
            res1 = outputs[0][i].copy().tolist()
            res2 = outputs[1][i].copy().tolist()
            res.append({OUTPUT1: res1, OUTPUT2: res2})
        return res

    # execute
    pool = ThreadPool(processes=1)
    total_start_time = time.time()
    while True:
        valid_num, feed_dicts = run_queue.get()
        if feed_dicts is None:
            break
        async_result = pool.apply_async(execute, (feed_dicts, valid_num))
        asy_results.append(async_result)
```

(continues on next page)

(continued from previous page)

```
    results = []
    for asy in asy_results:
        for res in asy.get():
            results.append(res)
    total_end_time = time.time()

    tput = len(results) / (total_end_time - total_start_time)
    latency = (total_end_time - total_start_time) / len(results)

    if args.dump_results:
        dump_results("original_" + args.model_without_packing, results)

    print(f"[Original] Throughput: {tput} samples/s, Latency: {latency *1000} ms")

    return results


def calculate_mae(expected_results, output_results, datasets, enable_debug):
    assert len(datasets) == len(expected_results)
    assert len(datasets) == len(output_results)
    maes = []
    zipped_data = zip(datasets, expected_results, output_results)
    for i, (data, expected, output) in enumerate(zipped_data):
        np.testing.assert_equal(len(expected), len(output))
        input_len = data.input_len
        output_1_mae = mean_absolute_error(
            expected[OUTPUT1][:input_len], output[OUTPUT1][:input_len]
        )
        output_2_mae = mean_absolute_error(
            expected[OUTPUT2][:input_len], output[OUTPUT2][:input_len]
        )
        maes.append([i, output_1_mae, output_2_mae])

    k = 10 if len(datasets) > 10 else len(datasets)

    def print_topk(k, out_name, out_index):
        for i in range(1, k + 1):
            print(f"Sample: {maes[-i][0]}, {out_name} mae : {maes[-i][out_index]}")

    if enable_debug:
        maes.sort(key=lambda e: e[1])
        print(f"\n***** Top {k} mae of output: {OUTPUT1} *****")
        print_topk(k, OUTPUT1, 1)

        maes.sort(key=lambda e: e[2])
        print(f"\n***** Top {k} mae of output: {OUTPUT2} *****")
        print_topk(k, OUTPUT2, 2)

    print(f"{OUTPUT1} average mae: {np.mean(maes,axis=0)[1]}")
    print(f"{OUTPUT2} average mae: {np.mean(maes,axis=0)[2]}")


def main():
    parser = argparse.ArgumentParser(description='packed bert-base-squad')
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument(
        '--avg_seq_len', type=int, default=128, help='average sequence length of input'
    )
    parser.add_argument(
        '--batch_size', type=int, default=16, help='batch size of model'
    )
    parser.add_argument('--dump_results', action='store_true', help='dump results')
    parser.add_argument(
        '--dynamic_input_name', type=str, default=INPUT_IDS, help='dynamic input name'
    )
    parser.add_argument(
        '--emb_size', type=int, default=30522, help='word embedding table size'
    )
    parser.add_argument(
        '--enable_debug', action='store_true', help='enable output debug info'
    )
    parser.add_argument(
        '--iterations', type=int, default=100, help='number of batches to run'
    )
    parser.add_argument(
        '--max_seq_len', type=int, default=256, help='max sequence length of input'
    )
    parser.add_argument(
        '--max_valid_num', type=int, default=40, help='max valid num for pack'
    )
    parser.add_argument(
        '--model_without_packing', help='model without pack, unpack, repack op'
    )
    parser.add_argument(
        '--model_with_packing_unpack_repack',
        help='model with pack, unpack, repack op converted by PopRT',
    )
    parser.add_argument(
        '--model_with_packing_attention_mask',
        help='model with AttentionMask op converted by PopRT',
    )
    parser.add_argument(
        '--timeout_microseconds',
        type=int,
        default=15000,
        help='timeout in microseconds',
    )

    args = parser.parse_args()
    args.dataset_size = args.iterations * args.batch_size

    # generate synthetic dataset
    datasets = get_synthetic_data(args)
    original_result = run_original_model_with_model_runner(args, datasets)

    offline_pack_result_unpack_repack = run_packing_model_with_model_runner(
        args, datasets, args.model_with_packing_unpack_repack, True
    )
    online_pack_result_unpack_repack = run_packing_model_with_pack_runner_unpack_repack(
```

(continues on next page)

(continued from previous page)

```
        args, datasets
    )

    offline_pack_result_attention_mask = run_packing_model_with_model_runner(
        args, datasets, args.model_with_packing_attention_mask, False
    )
    online_pack_result_attention_mask_first_fit = (
        run_packing_model_with_pack_runner_attention_mask(args, datasets, "first_fit")
    )
    online_pack_result_attention_mask_next_fit = (
        run_packing_model_with_pack_runner_attention_mask(args, datasets, "next_fit")
    )
    latency_distribuion_with_pack_runner_attention_mask(args, datasets, "first_fit")

    # compare the results
    print("\nCompare results between original and online pack(with unpack repack)")
    calculate_mae(
        original_result, online_pack_result_unpack_repack, datasets, args.enable_debug
    )
    print("\nCompare results between offline and online pack with unpack repack op")
    calculate_mae(
        offline_pack_result_unpack_repack,
        online_pack_result_unpack_repack,
        datasets,
        args.enable_debug,
    )

    print(
        "\nCompare results between original and online_first_fit pack with attention_mask op"
    )
    calculate_mae(
        original_result,
        online_pack_result_attention_mask_first_fit,
        datasets,
        args.enable_debug,
    )
    print(
        "\nCompare results between original and online_next_fit pack with attention_mask op"
    )
    calculate_mae(
        original_result,
        online_pack_result_attention_mask_next_fit,
        datasets,
        args.enable_debug,
    )

    print(
        "\nCompare results between offline and online_next_fit pack with attenttion_mask op"
    )
    calculate_mae(
        offline_pack_result_attention_mask,
        online_pack_result_attention_mask_next_fit,
        datasets,
        args.enable_debug,
```

(continues on next page)

(continued from previous page)

```
    )


if __name__ == "__main__":
    sys.exit(main())
```

Downloading the model

Before downloading the model, you need to install the dependencies:

```
pip install torch==1.10.0
pip install transformers[onnx]==4.25.1
```

Download the model:

```
python -m transformers.onnx --model=csarron/bert-base-uncased-squad-v1 . --feature question-answering
```

Converting the model

The downloaded model does not contain position_ids in the input. To use packing on the IPU, you need to pack the input on the host first. Therefore, you need to add position_ids to the input of the model. The code is as follows:

Listing 5.4: add_position_ids.py

```
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import argparse
import copy
import os

import onnx

# Download model from huggingface
# - python -m transformers.onnx --model=csarron/bert-base-uncased-squad-v1 . --feature question-answering
# reference: https://huggingface.co/csarron/bert-base-uncased-squad-v1


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Preprocess Bert-Squad Model')
    parser.add_argument(
        '--input_model', type=str, default='', help='path of input model'
    )
    args = parser.parse_args()

    if not os.path.exists(args.input_model):
        parser.print_usage()
        raise FileNotFoundError(f'Unable to find model: {args.input_model}')

    model = onnx.load(args.input_model)

```

(continues on next page)

(continued from previous page)

```
    # for packed bert, we need to export position_ids to model's input
    # step 1: remove unneed node
    rm_node_names = [
        'Shape_7',
        'Gather_9',
        'Add_11',
        'Unsqueeze_12',
        'Slice_14',
        'Constant_8',
        'Constant_10',
        'Constant_13',
    ]
    rm_nodes = []
    for node in model.graph.node:
        if node.name in rm_node_names:
            rm_nodes.append(node)

    assert len(rm_node_names) == len(rm_nodes)

    for node in rm_nodes:
        model.graph.node.remove(node)

    # step 2: add position_ids to model's input
    position_ids = copy.deepcopy(model.graph.input[0])
    position_ids.name = 'position_ids'
    model.graph.input.append(position_ids)

    for node in model.graph.node:
        if node.op_type == 'Gather' and node.name == 'Gather_18':
            node.input[1] = position_ids.name

    print(f'Save preprocessed model to bert_base_squad_pos.onnx')
    onnx.save(model, 'bert_base_squad_pos.onnx')
```

To generate the model without packing, run:

```
poprt \
    --input_model squad_bert_base_pos.onnx \
    --output_model squad_bert_base_bs16_sl256.onnx \
    --precision fp16 \
    --input_shape input_ids=16,256 attention_mask=16,256 token_type_ids=16,256 position_ids=16,256
```

To generate model with packing, run:

```
poprt \
    --input_model squad_bert_base_pos.onnx \
    --output_model squad_bert_base_bs16_sl256_pack.onnx \
    --precision fp16 \
    --input_shape input_ids=16,256 attention_mask=16,256 token_type_ids=16,256 position_ids=16,256 \
    --pack_args max_valid_num=40 segment_max_size=256
```

max_valid_num is used to specify the maximum batch size after unpacking, and segment_max_size indicates the maximum length.

Running the model

Run the model:

```
python packed_bert_example.py \
    --model_with_packing squad_bert_base_bs16_sl256_pack.onnx \
    --model_without_packing squad_bert_base_bs16_sl256.onnx
```

When the example has run, you should see the an output similar to the following:

```
[Original] Throughput: 1860.9792005501781 samples/s, Latency: 0.5373515188694 ms
....
[Pack Offline] Throughput: 2830.8140869025283 samples/s, Latency: 0.3532552719116211 ms
....
[Pack Online] Throughput: 2782.587696947809 samples/s, Latency : 0.3593777120113373 ms
....
```

5.6 CPU packing

5.6.1 Background

The IPU runs static graphs and processes input data of a fixed size each time. If the input length is not fixed, additional zero padding operations are required; otherwise, the data processing power of the IPU cannot be effectively utilised.

The CPU packing functionality is implemented through the $\mathrm{popart.PackRunner}$ class and is specifically designed for a dynamic sequence length. After the user request is packed, the number of zero paddings is greatly reduced. When packing is combined with model modifications to achieve the equivalent transformations, the data processing efficiency of the IPU can be effectively utilised.

5.6.2 Functional modules

In essence, packing accumulates user requests of unfixed lengths within a fixed period of time and then packs the data together and sends it to the IPU in one batch.

Timeout processing

In order to increase the amount of valid data sent to the IPU, it is necessary to accumulate batches over a certain period of time. This is achieved by setting a timeout.

User data preprocessing

If the data has already been filled with zeros, this zero-filled data will be removed during preprocessing based on mask data provided in the request.

Data accumulation

Pack maintains an internal queue, where user requests will be queued. When the timeout module issues a request or the data being packed reaches the maximum value (reaches the maximum number of entries or about to exceed the model input size), PackRunner will send the accumulated requests to the IPU (out of queue).

Post-packing processing

After the IPU operation is completed, the packing program will be notified to retrieve the data and copy the output back to the user address. The user program will obtain the result corresponding to the input. No special processing is required.

There are two situations when packing copies data:

1. If the unpack operator is inserted in the network, then the output length is fixed. Simply copy it back according to the fixed length.
2. If there is no unpack operator in the network, then the default output length copied back is equal to the effective length of the input data (the input excludes the length of the user pad).

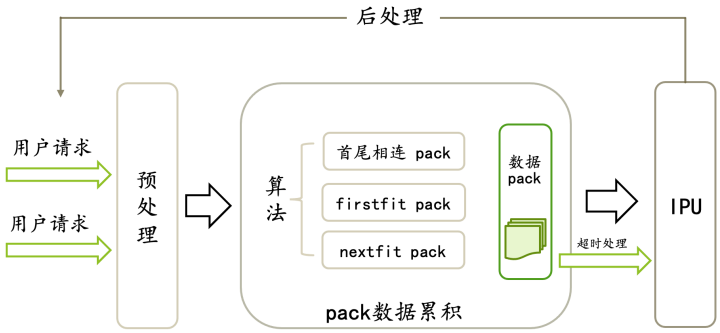


Fig. 5.8: Schematic diagram of packing functional modules

5.6.3 Packing algorithms

CPU packing currently supports the following methods:

- end-to-end
- FirstFit
- NextFit

Packing has the following three restrictions built in:

- The total number of data entries in the packing cannot exceed the value set for $\mathrm{maxValidNum}$.
- The total data length of the packing cannot exceed the maximum length of the input when compiling the model.
- The time interval between the first and last data of the packing cannot exceed the maximum value set by the user.

End-to-end method

The end-to-end packing method (Fig. 5.9) is used to concatenate the data end-to-end until the set maximum number of entries is reached or the maximum input value (input size when compiling the model) is about to be exceeded. With the end-to-end packing method, data can cross lines, and the unpack and repack operators need to be inserted into the network.

Fig. 5.9: End-to-end packing

FirstFit method

The FirstFit packing method (Fig. 5.10) is an "intraline" method, which means that data cannot cross lines. When a user request arrives, the scanning will start from the first line, until a line is found to put this data in. If a line is not found, then it will start a new line. According to network requirements, it may be necessary to insert several zeros as separators between two entries of data. A separator is not required before the first entry of data in the line. Although FirstFit changes the relative order of user requests within the packing, the result order will be consistent with the input order for the user because the built-in program will adjust the result order. Compared to NextFit, FirstFit improves space utilisation.

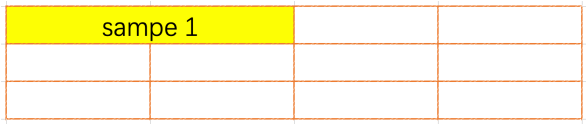


Fig. 5.10: FirstFit packing

NextFit method

NextFit packing (Fig. 5.11) is also an intraline packing method, and data cannot cross lines. When a user request arrives, the scanning will start from the end of the previous entry of data until a line is found to put this data in. If a line is not found, then it will start a new line.

Similarly, the total data length of the packing cannot exceed the maximum length of the input at the compiling time. Like FirstFit, it may be necessary to insert separators between two entries of data.

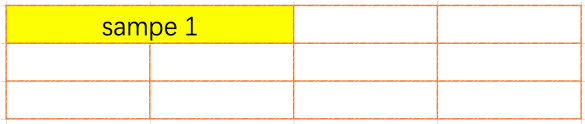


Fig. 5.11: NextFit packing

5.6.4 Examples

PopRT contains two end-to-end examples to illustrate how to use packing on the IPU. Each example includes the *three packing algorithms* and the corresponding performance data. PopRT examples are available on the examples code page.

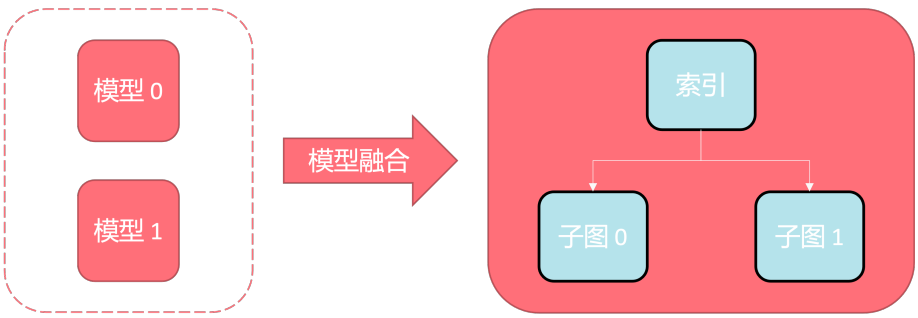
1. BERT packing example: packed_bert_example
2. DeBERTa packing example: packed_deberta_example

For detailed instructions, see README.md in each directory.

5.7 Model fusion

For Poplar, the model is first compiled into a binary executable that can be executed on the IPU, and then loaded into the IPU for execution. Since Poplar currently only supports the loading of one executable at the same time, executing multiple models on one IPU can cause repeated loading of different executables, leading to a sharp decrease in performance.

The function of model fusion is to fuse multiple user models into one model in the compilation stage, with each model serving as a subgraph of the fused model. The subgraphs are isolated from each other by branch operators. At runtime, controlling which subgraph to run by inputting the model index can greatly reduce the latency of model switching.



Before reading this chapter, it is necessary to be familiar with the following:

- PopRT Runtime
- PopEF

5.7.1 Implementing PopRT model fusion

At present, this feature can only be enabled through the YAML configuration file. You need to define the ONNX model to be fused and other related parameters in the YAML configuration file.

An example of the YAML file is as follows:

Listing 5.5: config.yaml

```yaml
output_dir: './'
output_model: 'model_fusion.onnx'
export_popef: True
max_tensor_size: -1
ipu_version: 'ipu21'
model_fusion:
  - input_model: 'model0.onnx'
    input_shape: ['X=1,2', 'Y=1,2']
    precision: 'fp32'

  - input_model: 'model1.onnx'
    input_shape: ['X=1,1']
    precision: 'fp16'
```

You can configure independent parameters for each model in model_fusion. Parameters other than model_fusion are global parameters, which apply to each model to be fused.

For instance, max_tensor_size = -1 in the example, since it is defined in global parameters, it will apply to all models.

The fused model needs to be run through the PopRT Runtime; therefore, export_popef must be set to True to generate the PopEF model.

Furthermore, in order to ensure the running of the subsequent inference code, ipu_version needs to be set correctly based on the running device; for example, ipu21 needs to be set for the C600 platform..

An example of model fusion is:

Listing 5.6: compile.py

```python
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import os

import numpy as np
import onnx

from onnx import TensorProto, checker, helper, mapping


```

(continues on next page)

(continued from previous page)

```
def create_model0(opset_version=11):
    g0_dtype = TensorProto.FLOAT
    g0_add = helper.make_node("Add", ["X", "Y"], ["Z"])
    g0_reshape = helper.make_node("Reshape", ["Z", "C"], ["O"])
    g0 = helper.make_graph(
        [g0_add, g0_reshape],
        'graph',
        [
            helper.make_tensor_value_info("X", g0_dtype, (1, 2)),
            helper.make_tensor_value_info("Y", g0_dtype, (1, 2)),
        ],
        [
            helper.make_tensor_value_info("O", g0_dtype, (2,)),
        ],
    )

    g0_const_type = TensorProto.INT64
    g0_const = helper.make_tensor(
        "C",
        g0_const_type,
        (1,),
        vals=np.array([2], dtype=mapping.TENSOR_TYPE_TO_NP_TYPE[g0_const_type])
        .flatten()
        .tobytes(),
        raw=True,
    )
    g0.initializer.append(g0_const)
    m0 = helper.make_model(g0, opset_imports=[helper.make_opsetid("", opset_version)])
    checker.check_model(m0)
    return m0


def create_model1(opset_version=11):
    g1_dtype = TensorProto.FLOAT16
    g1_concat = helper.make_node("Concat", ["X", "C"], ["O"], axis=1)
    g1 = helper.make_graph(
        [g1_concat],
        'graph',
        [
            helper.make_tensor_value_info("X", g1_dtype, (1, 1)),
        ],
        [
            helper.make_tensor_value_info("O", g1_dtype, (1, 3)),
        ],
    )

    g1_const = helper.make_tensor(
        "C",
        g1_dtype,
        (1, 2),
        vals=np.array([[1.5, 2.0]], dtype=mapping.TENSOR_TYPE_TO_NP_TYPE[g1_dtype])
        .flatten()
        .tobytes(),
        raw=True,
```

(continues on next page)

(continued from previous page)

```
    )
    g1.initializer.append(g1_const)
    m1 = helper.make_model(g1, opset_imports=[helper.make_opsetid("", opset_version)])
    checker.check_model(m1)
    return m1


def create_onnx(opset):
    model0 = create_model0(opset)
    model1 = create_model1(opset)

    onnx.save(model0, "model0.onnx")
    onnx.save(model1, "model1.onnx")


if __name__ == '__main__':
    abs_path = os.path.abspath(os.path.dirname(__file__))
    if os.getcwd() != abs_path:
        raise RuntimeError(f"Please run program in {abs_path}")

    create_onnx(opset=11)

    cmd = "poprt \
          --config_yaml config.yaml "

    os.system(cmd)
```

This example will create the ONNX model required for the YAML file shown above, call PopRT to read the YAML file for model fusion and generate the executable.popef set in the YAML. After the compilation is completed, you can view the metadata of the fused model through popef_dump, which is the necessary information for the subsequent runtime.

An example of the command is:

```
popef_dump executable.popef
```

After the successful running of the above command, the terminal will display the relevant information of this PopEF file. Under the Anchors keyword, you can see the input/output name, type, shape and other information of the compiled static graph, of which, there will be an input named index which is used to control the selection of subgraphs.

5.7.2 Implementing PopRT Runtime fusion model inference

To run the PopEF file compiled by the PopRT model fusion, you need to write a program for inference through the PopRT Runtime.

An example of the program is:

Listing 5.7: run.py

```python
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import os

import numpy as np
import numpy.testing as npt

from poprt.runtime import Runner, RuntimeConfig

if __name__ == '__main__':
    abs_path = os.path.abspath(os.path.dirname(__file__))
    if os.getcwd() != abs_path:
        raise RuntimeError(f"Please run program in {abs_path}")

    popef_path = f"{abs_path}/executable.popef"

    config = RuntimeConfig()
    config.timeout_ns = 0
    config.validate_io_params = False
    runner = Runner(popef_path, config)

    g0_X = np.ones([1, 2], dtype=np.float32)
    g0_Y = np.ones([1, 2], dtype=np.float32) * 2
    g0_O = np.zeros([2], dtype=np.float32)

    runner.execute(
        {
            "graph0/X": g0_X,
            "graph0/Y": g0_Y,
        },
        {
            "graph0/O": g0_O,
        },
    )
    npt.assert_array_equal(g0_O, np.ones([2], dtype=np.float32) * 3)

    g1_X = np.zeros([1, 1], dtype=np.float16)
    g1_O = np.zeros([1, 3], dtype=np.float16)

    runner.execute(
        {
            "graph1/X": g1_X,
        },
        {
            "graph1/O": g1_O,
        },
    )
    npt.assert_array_equal(g1_O, np.array([[0.0, 1.5, 2.0]], dtype=np.float16))
```

This example shows how to run the PopEF file generated above.

The first step is to create the RuntimeConfig instance and configure the running parameters of ModelRunner. Note: be sure to set validate_io_params to False, since the fusion graph does not need to process data for each

input/output. Setting this parameter to True will cause errors.

The second step is to create the ModelRunner instance and load the PopEF file, so as to load the model into the IPU.

The third step is to set the input/output of the subgraph and specify the subgraph through index. If set to 0, the subgraph 0 can be specified. If index is not specified, PopRT Runtime will fill it automatically, as shown in the example.

5.8 Custom operations

PopRT supports creation of custom operators. You will need to create a custom operation when your model contains an operator that is not supported in PopRT. In this case, you can write a custom operator and compile it into a dynamic link library. PopRT supports dynamic linking of this custom operator into PopRT with the command line.

Since PopRT uses PopART as the backend, the process of developing custom operators for PopRT is the same as that for PopART. Please refer to Creating Custom Op in PopART .

This section describes the process of developing custom operators for PopRT with an example.

5.8.1 Writing custom operators

Taking the custom operator named LeakyRelu as an example, you first need to write the C++ code for it:

Listing 5.8: leaky_relu_custom_op.cpp

```
1  // Copyright (c) 2020 Graphcore Ltd. All rights reserved.
2
3  // This example demonstrates how to create a custom operator for PopART, in this
4  // case a Leaky ReLU op that returns `x` for any element `x >= 0` and `x *
5  // alpha` for any element `x < 0`, where `alpha` is provided as a scalar
6  // attribute to the operator.
7  #include <popart/operatoridentifier.hpp>
8  #include <popart/opmanager.hpp>
9  #include <popart/opserialiser.hpp>
10 #include <popart/popx/opxmanager.hpp>
11
12 #include <popops/ElementWise.hpp>
13 #include <popart/popx/opx.hpp>
14
15 namespace CustomOperators {
16 const popart::OperatorIdentifier LeakyReluId = {popart::Domain::ai_graphcore,
17                                                 "LeakyRelu",
18                                                 1};
19 } // namespace CustomOperators
20
21 class LeakyReluOp;
22 class LeakyReluOpx;
23
24 class LeakyReluOp : public popart::Op {
```

(continues on next page)

(continued from previous page)

```
public:
  LeakyReluOp(const popart::OperatorIdentifier &_opid,
              float _alpha,
              const popart::Op::Settings &settings_)
      : popart::Op(_opid, settings_), alpha(_alpha) {}

  std::unique_ptr<Op> clone() const final {
    return std::make_unique<LeakyReluOp>(*this);
  }

  void setup() final { outInfo(0) = inInfo(0); }

  void appendAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }

  void appendOutlineAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendOutlineAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }

  float getSubgraphValue() const final { return getHighSubgraphValue(); }

  bool requiresRandomSeed() const override { return false; }

  // Attributes
  float getAlpha() const { return alpha; }

private:
  float alpha;
};

namespace {
using popart::DataType;
using popart::OpDefinition;

static OpDefinition::DataTypes T = {DataType::FLOAT16, DataType::FLOAT};

static OpDefinition
    leakyReluOpDef({OpDefinition::Inputs({{"input", T}}),
                    OpDefinition::Outputs({{"output", T}}),
                    OpDefinition::Attributes({{"alpha", {"*"}}})});

static popart::OpCreator<LeakyReluOp> leakyReluOpCreator(
    popart::OpDefinitions({{CustomOperators::LeakyReluId, leakyReluOpDef}}),
    [](const popart::OpCreatorInfo &info) {
      // default alpha is 10**(-2)
      float alpha = info.attributes.getAttribute<popart::Attributes::Float>(
          "alpha", 1e-2f);
      return std::make_unique<LeakyReluOp>(info.opid, alpha, info.settings);
    },
    true);
} // namespace
```

(continues on next page)

(continued from previous page)

```

namespace pe = popops::expr;

class LeakyReluOpx : public popart::popx::Opx {
public:
  LeakyReluOpx(popart::Op *op, popart::popx::Devicex *devicex)
      : popart::popx::Opx(op, devicex) {
    verifyOp<LeakyReluOp>(op, {CustomOperators::LeakyReluId});
  }

  void grow(poplar::program::Sequence &prog) const final {

    auto op = getOp<LeakyReluOp>();

    poplar::Tensor input = getInTensor(0);

    float alpha = op.getAlpha();

    // x < 0.0f ? alpha * x : x
    auto expression = pe::Select(pe::Mul(pe::Const(alpha), pe::_1),
                                 pe::_1,
                                 pe::Lt(pe::_1, pe::Const(0.0f)));

    popops::mapInPlace(graph(),
                       expression,
                       {input},
                       prog,
                       debugContext("LeakyRelu"),
                       poplar::OptionFlags());

    setOutTensor(0, input);
  }
};

static popart::popx::OpxCreator<LeakyReluOpx>
    LeakyReluOpxCreator({CustomOperators::LeakyReluId});
```

Create a Makefile and generate custom_ops.so using the make command:

Listing 5.9: Makefile

```
CXX ?= g++
CXXFLAGS = -std=c++14 -fPIC -g
LDLIBS = -shared -lpopart
ONNX_NAMESPACE = -DONNX_NAMESPACE=onnx

BUILD_DIR = build
SOURCES = leaky_relu_custom_op.cpp
TARGET = $(BUILD_DIR)/custom_ops.so

all: create_build_dir leaky_relu_custom_op

.PHONY: create_build_dir
create_build_dir:
```

(continues on next page)

(continued from previous page)

```
	mkdir -p $(BUILD_DIR)

leaky_relu_custom_op: leaky_relu_custom_op.cpp
	$(CXX) $(SOURCES)  $(LDLIBS) $(CXXFLAGS) $(ONNX_NAMESPACE) -o $(TARGET)

.PHONY: clean
clean:
	rm -rf  $(BUILD_DIR)
```

Create a shape-inference file for the custom operator:

Listing 5.10: custom_shape_inference.py

```
# Copyright (c) 2022 Graphcore Ltd. All rights reserved.
from typing import Tuple

import onnx
import onnx.helper
import onnx.shape_inference

from poprt.passes import ShapeFunc, get_dtype, get_shape, register_shape_func


@register_shape_func(['LeakyRelu'])
class LeakyRelu(ShapeFunc):
    """Function based on ONNX to infer the shape and dtype of custom op."""

    def __init__(self) -> None:
        super().__init__()

    def __call__(
        self,
        model: onnx.ModelProto,
        node: onnx.NodeProto,
    ) -> Tuple[onnx.ModelProto, bool]:
        graph = model.graph
        input_name = node.input[0]
        output_name = node.output[0]
        # If the Op already has known shape and dtype of output, return True
        if get_shape(model.graph, output_name) and get_dtype(model.graph, output_name):
            return model, True

        input_dtype = get_dtype(graph, input_name)
        input_shape = get_shape(graph, input_name)
        # If the Op is able to be inferred shape and dtype, return True
        if input_dtype and input_shape and 0 not in input_shape:
            # ![Shape-Inference Function begin]

            # Step.1: Write the method following ONNX-Protobuf standard,
            #         to calc shape and dtype of output in terms of shape and dtype of input
            # The LeakyRelu Op has same shape and dtype with input and output

            # Step.2: Create new TensorProto with inferred shape and dtype of output
            output_tensor = onnx.helper.make_tensor_value_info(
```

(continues on next page)

(continued from previous page)

```
            output_name, input_dtype, input_shape
        )
        # Step.3: Call update_value_info to update
        model = self.update_value_info(model, output_tensor)
        # Step.4: Call infer_shapes function
        model = onnx.shape_inference.infer_shapes(model)
        # ![Shape-Inference Function end]
        return model, True
    # If the Op is not able to be inferred, return False
    else:
        return model, False
```

Create the ONNX model file with the LeakyRelu op

Run the following test code with Python3 to generate the ONNX model file custom_op_test.onnx for testing:

Listing 5.11: create_onnx_with_custom_op.py

```
# Copyright (c) 2022 Graphcore Ltd. All rights reserved.
import argparse
import os

import onnx

from onnx import helper


def create_onnx_model_with_custom_op():
    TensorProto = onnx.TensorProto

    attributes = {"alpha": 0.01}
    leaky_relu = helper.make_node(
        "LeakyRelu", ["X"], ["Y"], domain="ai.graphcore", **attributes
    )
    relu = helper.make_node("Relu", ["Y"], ["Z"])

    graph = helper.make_graph(
        [leaky_relu, relu],
        "custom_op_test",
        [
            helper.make_tensor_value_info("X", TensorProto.FLOAT, (8, 8)),
        ],
        [
            helper.make_tensor_value_info("Z", TensorProto.FLOAT, (8, 8)),
        ],
    )
    opset_imports = [helper.make_opsetid("", 11)]
    model = helper.make_model(graph, opset_imports=opset_imports)
    model.opset_import.append(onnx.helper.make_opsetid("ai.graphcore", 1))
    return model


```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Convert ONNX model and run it on the IPU.'
    )
    parser.add_argument(
        '--output_dir',
        type=str,
        default='./',
        help="Full path of the directory the ONNX model will be saved to.",
    )
    args = parser.parse_args()

    if not os.path.isdir(args.output_dir):
        raise ValueError("--output_dir should be an existing folder")

    model_path = os.path.join(args.output_dir, 'custom_op_test.onnx')

    model = create_onnx_model_with_custom_op()
    onnx.save(model, model_path)

    # Convert and Run
    compile_cmd = "bash build.sh"
    os.system(compile_cmd)
    abs_path = os.path.abspath(os.path.dirname(__file__))
    run_cmd = rf"""poprt \
--input_model {model_path} \
--custom_shape_inference {abs_path}/custom_shape_inference.py \
--custom_library_so_paths {abs_path}/custom_ops.so \
--run"""
    os.system(run_cmd)
    # 2022-12-30 07:01:54,408 INFO cli.py:446] Bs: 8
    # 2022-12-30 07:01:54,408 INFO cli.py:449] Latency: 0.23ms
    # 2022-12-30 07:01:54,408 INFO cli.py:450] Tput: 35469
```

5.8.2 Using custom operators in PopRT

The library files of custom operators can be dynamically linked with the PopRT command line option --custom_library_so_paths, and the shape-inference for the custom operator can be registered with --custom_shape_inference.

The ONNX model file generated above can be executed with the following command:

```
poprt \
    --input_model custom_op_test.onnx \
    --custom_library_so_paths custom_ops.so \
    --custom_shape_inference custom_shape_inference.py \
    --run
```

5.9 Custom passes

In PopRT, a pass is used for various processes such as model conversion and optimisation. The functionality of PopRT can be expanded by adding custom passes.

The implementation/use of custom passes is subject to the general specification for PopRT pass, as detailed in the *Passes* chapter.

5.9.1 Implementing custom passes

To add custom passes in PopRT, at least one Python file needs to be written. The example code is as follows:

Listing 5.12: replace_add_with_sub.py

```python
# Copyright (c) 2022 Graphcore Ltd. All rights reserved.
import onnx

from poprt.passes import Pass, register


@register('replace_add_with_sub')
class ReplaceAddwithSub(Pass):
    """Replace Add with Sub."""

    def __init__(self):
        super().__init__()

    # define the transform
    def run_transform(
        self,
        graph: onnx.GraphProto,
        is_main_graph: bool,
    ) -> onnx.GraphProto:
        for node in graph.node:
            if node.op_type == 'Add':
                node.op_type = 'Sub'
        return graph

    # define the run method
    def run(self, onnx_model: onnx.ModelProto) -> onnx.ModelProto:
        onnx_model.graph.CopyFrom(
            self.traverse_graph(onnx_model.graph, self.run_transform)
        )
        return onnx_model
```

As shown in the example code, the ReplaceAddwithSub pass is implemented. The function of this pass is to replace all Add nodes in the ONNX model with Sub nodes. The pass is registered as replace_add_with_sub.

Note: A Python file can contain multiple custom passes, but the registered name of each custom pass must be unique.

5.9.2 Using custom passes

Custom passes can be used in the PopRT CLI or the Python API.

Using custom passes in the PopRT CLI

The Python file (multiple files shall be separated by ,) containing custom passes can be specified by using the parameter --custom_pass_config. The pass to be applied can be specified with --passes.

Listing the registered custom passes:

Listing 5.13: load_custom_passes_in_cli.sh

```
poprt \
    --list_all_passes \
    --custom_pass_config replace_add_with_sub.py |
    grep replace_add_with_

# replace_add_with_sub
```

Applying custom passes:

```
poprt \
    --input_model model.onnx \
    --passes replace_add_with_sub \
    --custom_pass_config replace_add_with_sub.py
```

Note:

- Passing of parameters to custom passes is not supported when using custom passes via CLI.
- There may be dependencies among multiple custom passes, and PopRT will execute them in the pass order specified by --passes.
- The passes specified with --passes will be executed after some of the PopRT default conversion processes are completed. This means that the input ONNX model is not the original model of the input.

Using custom passes in the Python API

Custom passes can be registered by importing the custom pass modules into the application code.

Listing 5.14: load_custom_passes.py

```
import replace_add_with_sub  # noqa

import poprt

assert 'replace_add_with_sub' in poprt.get_registered_passes()
```

A custom pass is used in the same way as a built-in pass. Please refer to the *Passes* chapter for more details.

5.10 Custom patterns

The concept of a "pattern" comes from the PopART framework. The role of a pattern is to match ops in the PopART IR and modify or replace the matched ops, as a means of graph optimisation.

A typical application scenario for the custom PopART pattern is when a user implements a high-performance op and replaces some performance bottleneck ops with this high-performance op through the custom PopART pattern, thereby improving the performance of the entire model.

This chapter helps users understand how to implement custom PopART patterns and use them in PopRT.

Before reading this chapter, it is necessary to be familiar with the PopART framework.

5.10.1 Implementing custom PopART patterns

To implement a custom PopART pattern in PopRT, at least one C++ file needs to be written.

This example shows how the Relu Op in the diagram is replaced with the Negative Op, which can be compiled into a standalone dynamic library and linked when using PopRT:

Listing 5.15: replace_relu_with_neg_pattern.cpp

```
// Copyright (c) 2022 Graphcore Ltd. All rights reserved.
#include <string>

#include <popart/graph.hpp>
#include <popart/op/negate.hpp>
#include <popart/op/relu.hpp>
#include <popart/op/sqrt.hpp>
#include <popart/operators.hpp>
#include <popart/patterns/pattern.hpp>
#include <popart/patterns/patterns.hpp>

namespace popart {
class IArray;
class Tensor;
} // namespace popart

using namespace popart;

class ReplaceReluWithNeg : public PreAliasPattern {
public:
  bool matches(Op *op) const override { return op->isConvertibleTo<ReluOp>(); }

  std::vector<const Tensor *> touches(Op *) const override { return {}; }

  bool apply(Op *op) const override {
    std::cout << "Custom pattern ReplaceReluWithNeg applied in "
              << op->debugName() << std::endl;

    auto negOp = makeReplacementOpInIr(Onnx::Operators::Neg_6, op);

    auto inputId  = op->inId(ReluOp::getInIndex());
```

(continues on next page)

(continued from previous page)

```
    auto outputId = op->outId(ReluOp::getOutIndex());
    op->disconnectAllInputs();
    op->disconnectAllOutputs();
    op->getGraph().eraseOp(op->id);

    negOp->connectInTensor(NegateOp::getInIndex(), inputId);
    negOp->connectOutTensor(NegateOp::getOutIndex(), outputId);
    negOp->setup();

    return true;
  }
};

namespace {
static PatternCreator<ReplaceReluWithNeg>
    ReplaceReluWithNegPatternCreator("ReplaceReluWithNeg",
                                     /* default enabled = */ false,
                                     /* default mandatory = */ false);
} // namespace
```

To implement custom PopART patterns, two parts are typically required as follows:

- Inherit `PreAliasPattern` and implement its `apply` method.
- Register the pattern with `PatternCreator`.

5.10.2 Using custom PopART patterns in PopRT

First, the source code of the pattern needs to be compiled into a dynamic link library. The following is the command for compiling the example:

```
g++ \
    -std=c++14 \
    -fPIC \
    -O3 \
    -DONNX_NAMESPACE=onnx \
    replace_relu_with_neg_pattern.cpp \
    -shared \
    -lpopart \
    -o custom_pattern.so
```

Then, the dynamic library containing custom patterns can be linked with the `--custom_library_so_paths` parameter of the PopRT CLI:

```
poprt  --custom_library_so_paths path/to/shared/library
```

There are several ways to use custom PopART patterns in PopRT. When setting a pattern, you can use `:value` after the pattern name to specify whether to disable or enable the pattern.

- If `:value` is not used after the pattern name, the pattern is enabled.
- If `:value` is specified after the pattern name, and `value` is one of `True`, `true` or `1`, the pattern will be enabled.

- Otherwise the pattern is disabled.

Method 1: Use PatternCreator to enable the pattern by default

```
static PatternCreator<ReplaceReluWithNeg>
    ReplaceReluWithNegPatternCreator("ReplaceReluWithNeg",
                                     /* default enabled = */ true);
```

With this method, the use of pattern is determined during the compilation period. After linking with the dynamic library of this pattern, it will be executed by default and cannot be dynamically controlled and during run time.

Method 2: Configure a pattern using the Python API

You can use the CompilerOptions().custom_patterns API to configure patterns in Python. In the following case, the ReplaceReluWithNeg pattern is enabled, and the InPlace pattern is disabled.

```
from poprt.compiler import Compiler, CompilerOptions

opts = CompilerOptions()
opts.custom_patterns = ["ReplaceReluWithNeg", "InPlace:0"]
Compiler.compile_and_export(model, outputs, popef_path, opts)
```

Method 3: Config the specified pattern using CLI

You can also enable custom patterns using the PopRT CLI. Multiple patterns can be separated by commas. In the following case, the ReplaceReluWithNeg pattern is enabled and the InPlace pattern is disabled.

```
poprt \
    --input_model model.onnx \
    --output_model model_export.onnx \
    --export_popef \
    --output_dir model \
    --custom_library_so_paths build/custom_patterns.so \
    --compiler_options custom_patterns=ReplaceReluWithNeg,InPlace:0
```

5.11 Custom transforms

The concept of a transform comes from the PopART framework and is a means of graph optimisation. Unlike patterns which only match ops in the PopART IR, transforms carry out optimisations of the entire graph in the PopART IR, and therefore change the PopART IR in a more complex way.

For example, the role of the SubgraphOutline transform is to extract duplicate Op structures into new graphs and use CallOps calls to save memory.

This chapter helps users understand how to implement custom PopART transforms and use them in PopRT.

Before reading this chapter, it is necessary to be familiar with the following:

- PopART framework.
- Infrastructure of the PopART IR

5.11.1 Implementing custom PopART transforms

To create a custom transform in PopART, at least one C++ file needs to be written.

This example is used to print the current serialised IR, which can be compiled into a standalone dynamic library and linked when using PopRT:

Listing 5.16: ir_serialise_transform.cpp

```
// Copyright (c) 2022 Graphcore Ltd. All rights reserved.

#include <iostream>
#include <string>
#include <popart/graph.hpp>
#include <popart/ir.hpp>
#include <popart/op.hpp>
#include <popart/transforms/transform.hpp>

namespace popart {
class Graph;

class IrSerialise : public Transform {
public:
  static std::size_t id();

  IrSerialise() : Transform() {}
  virtual ~IrSerialise() override {}

  virtual bool apply(Graph &graph) const final;

  virtual std::size_t getId() const final { return id(); }

  virtual std::string getName() const final { return "IrSerialise"; }
};

std::size_t IrSerialise::id() { return typeid(IrSerialise).hash_code(); }

bool IrSerialise::apply(Graph &graph) const {
  const auto &ir = graph.getIr();
  std::stringstream ss;
  ir.serialise(Ir::SerialiseFormat::JSON, ss);
  const auto modelStr = ss.str();
  std::cout << "SerializedIr : " << std::endl;
  std::cout << modelStr << std::endl;
  return true;
}

namespace {
bool init = Transform::registerTransform(new IrSerialise);
}
```

(continues on next page)

(continued from previous page)

```
42
43 } // namespace popart
```

In order to implement a custom PopART transform, it is necessary to inherit and override popart::Transform or implement the main methods:

- apply() implements IR conversion and other functions
- getId() is the unique transform ID
- getName() defines the name of the custom transform. It is necessary to avoid conflicts with existing transform names in PopART.
- registerTransform() registers custom transform with PopART

Please refer to the default transform in PopART contained in the public PopART repo on GitHub.

5.11.2 Using custom transforms in PopRT

After the custom transform has been written, the next thing to do is to use it in PopRT.

The source code of the custom transform needs to be compiled into a standalone dynamic link library and linked when using PopRT. The following is an example command for compilation:

```
g++ \
  -std=c++14 \
  -fPIC \
  -O3 \
  -DONNX_NAMESPACE=onnx \
  ir_serialise_transform.cpp \
  -shared \
  -lpopart \
  -o custom_transform.so
```

Then, the dynamic library containing the custom transform can be linked with the --custom_library_so_paths PopRT CLI parameter:

```
poprt  --custom_library_so_paths path/to/shared/library
```

Since transforms change PopART IR in a more complex way, they need to be executed in a predefined order. Usually, before writing the transform, it is necessary to consider which execution position to put it in.

The execution order of the transform is divided into several stages, and PopART allows a user-defined custom transform to be called at the checkpoint after each stage is completed.

Predefined checkpoints are as follows:

- Fwd0: Initial IR after ONNX Lowering to PopART IR
- Fwd1: After pre-alias patterns are applied to FWD0
- Bwd0: After the backward pass
- Prealias: After pre-alias patterns are applied to BWD0

- MainLoops: After applying the MainLoops transform
- Final: After the final IR of all transforms is applied

See the C++ code in the PopART public GitHub repo for more details

Configure the custom transform with the --compiler_options parameter of the PopRT CLI, as:

```
poprt \
    --input_model model.onnx \
    --output_model model_export.onnx \
    --export_popef \
    --output_dir model \
    --custom_library_so_paths build/custom_transforms.so \
    --compiler_options custom_transform_applier_settings="{'Fwd0': ['IrSerialise']}"
```

5.12 Manual sharding

PopRT manual sharding supports dividing the model into different subgraphs through sharding points provided by users to achieve model parallelism and pipeline parallelism.

5.12.1 Sharding and model parallelism

PopRT supports sharding the ONNX graph across different devices based on the sharding points provided by users to achieve model parallelism. Sharding is suitable for large models that exceed the memory limit of a single device and require multiple devices.

For more information, refer to the section on sharding in the technical note.

Note: To use model parallelism, the PopRT backend options need to be set as follows:

- options.virtual_graph_mode = "manual"
- options.num_ipus = number of devices

5.12.2 Pipelining and pipeline parallelism

PopRT supports sharding the ONNX graph across different pipeline stages based on the sharding points provided by users to achieve pipeline parallelism and improve throughput.

For more information, refer to the sections on pipelining in the technical note and in the IPU Programmer's Guide.

Note: To use pipeline parallelism, it is necessary to enable model parallelism and set the PopRT backend options as follows:

- options.enable_pipelining = True

- options.batches_per_step = integer multiple of the number of pipeline stages

5.12.3 Manual sharding process

PopRT manual sharding shards the ONNX graph based on the ONNX node, and the sharding point can be any ONNX node.

1. The nodes in the ONNX graph are arranged in topological sorting order. PopRT manual sharding first performs topological sorting of the sharding points set by the user.
2. Traverse the sharding point. Take the sharding point as the starting point to traverse the ONNX graph in the direction of input, and put all the traversed ONNX nodes into a subgraph. If there is no input node or the node has already set sharding information, then stop the traversal of such branch.
3. After the traversal is completed, you will get the subgraph. Set the sharding information of the subgraph using ONNX attribute:
   - ___ipu_number specifies the device serial number corresponding to each subgraph in model parallelism
   - ___pipeline_stage specifies the pipeline stage corresponding to each subgraph in pipeline parallelism.

Note:

- Different sharding points can have the same device serial number and pipeline stage. For example, if there are two parallel branches started from different sharding points, and we want to put them onto a single device, then these two sharding points will have same device serial number.
- After the sharding information is set based on the sharding point, the remaining nodes without sharding information are automatically set:
  - ___ipu_number will be set to the currently set maximum device serial number +1.
  - ___pipeline_stage will be set to the currently set maximum pipeline stage +1.

5.12.4 Configuring manual sharding

There are two methods for configuring manual sharding:

- with the PopRT CLI
- with the poprt.converter.Sharder class.

Configuring manual sharding with the PopRT CLI

- Specify the sharding point name, device serial number and pipeline stage with the yaml file:

Listing 5.17: shard.yaml

```
-
  node: resnetv17_stage1___plus0
  device: 0
  stage: 0
-
  node: resnetv17_stage4_batchnorm2_fwd
  device: 1
  stage: 1
-
  node: resnetv17_stage4___plus0
  device: 2
  stage: 2
```

- Configuring sharding information with --manual_sharding_config in the PopRT CLI:

```
poprt \
    --input_model model.onnx \
    --manual_sharding_config shard.yaml
```

- Determine whether to perform manual sharding only on input_model with --only_manual_sharding in the PopRT CLI, which is not set by default.

  Not setting --only_manual_sharding means that manual sharding is performed after the Convert phase optimisation on input_model.

  Setting --only_manual_sharding means that only manual sharding is performed on input_model. Only --input_model, --output_model, --output_dir and --manual_sharding_config are supported; other parameters are invalid.

```
poprt \
    --input_model model.onnx \
    --manual_sharding_config shard.yaml \
    --only_manual_sharding
```

Configuring manual sharding with the Python API

You can use poprt.converter.Sharder to configure manual sharding.

```
sharding_info = {
    "resnetv17_stage1___plus0": 0,
    "resnetv17_stage4_batchnorm2_fwd": 1,
    "resnetv17_stage4___plus0: 2,
}
pipelining_info = {
    "resnetv17_stage1___plus0": 0,
    "resnetv17_stage4_batchnorm2_fwd": 1,
```

(continues on next page)

(continued from previous page)

```
    "resnetv17_stage4___plus0: 2,
}

sharded_model = poprt.converter.Sharder(
                        sharding_info=sharding_info,
                        pipelining_info=pipelining_info
                    ).run(converted_model)
```

Note:

- The PopRT CLI with --only_manual_sharding set or the use of poprt.converter.Sharder API needs to guarantee that every node in the ONNX graph has unique name.
- The PopRT CLI without --only_manual_sharding set does not need to guarantee that every node in the ONNX graph has unique name. The Convert optimisation process will guarantee that every node has unique name.

5.12.5 Example

The following is a simple example of manual sharding:

Take ResNet50 as an example.

Listing 5.18: shard.py

```
# Copyright (c) 2023 Graphcore Ltd. All rights reserved.
import numpy as np
import onnx
import requests

from poprt import runtime
from poprt.compiler import Compiler, CompilerOptions
from poprt.converter import Sharder


def load_model():
    # Download model
    url = 'https://github.com/onnx/models/raw/main/vision/classification/resnet/model/resnet50-v1-7.onnx'
    response = requests.get(url)
    if response.status_code == 200:
        model = onnx.load_model_from_string(response.content)
    else:
        raise Exception(
            f"Failed to download model with status_code {response.status_code}"
        )
    return model


def manual_sharding(model):
    # Fix the batch size to 1
```

(continues on next page)

(continued from previous page)

```
    model.graph.input[0].type.tensor_type.shape.dim[0].dim_value = 1

    # Sharding and pipelining info
    sharding_info = {
        "resnetv17_stage1___plus0": 0,
        "resnetv17_stage4_batchnorm2_fwd": 1,
        "resnetv17_stage4___plus0": 2,
    }
    pipelining_info = {
        "resnetv17_stage1___plus0": 0,
        "resnetv17_stage4_batchnorm2_fwd": 1,
        "resnetv17_stage4___plus0": 2,
    }
    model = Sharder(sharding_info=sharding_info, pipelining_info=pipelining_info).run(
        model
    )

    return model


def compile(model):
    # Compile the model with backend options
    model_bytes = model.SerializeToString()
    outputs = [o.name for o in model.graph.output]

    options = CompilerOptions()
    options.ipu_version = runtime.DeviceManager().ipu_hardware_version()
    # Sharding into 4 IPUs
    options.num_ipus = 4
    # Enable Sharding and Pipelining
    options.enable_pipelining = True
    options.virtual_graph_mode = "manual"
    options.batches_per_step = 16

    executable = Compiler.compile(model_bytes, outputs, options)
    runner_config = runtime.RuntimeConfig()
    runner_config.timeout_ns = 0
    runner = runtime.Runner(executable, runner_config)
    return runner


def run(runner):
    inputs_info = runner.get_execute_inputs()
    outputs_info = runner.get_execute_outputs()

    inputs = {}
    for i in inputs_info:
        inputs[i.name] = np.ones(i.shape, dtype=i.numpy_data_type())

    outputs = {}
    for o in outputs_info:
        outputs[o.name] = np.zeros(o.shape, dtype=o.numpy_data_type())

    runner.execute(inputs, outputs)
```

(continues on next page)

(continued from previous page)

```
80
81
82 if __name__ == '__main__':
83     model = load_model()
84     model = manual_sharding(model)
85     runner = compile(model)
86     run(runner)
```

5.13 Error handling

This section describes how PopRT handles errors that are raised by Poplar while running an inference application. These Poplar exceptions can be raised by either the application or by the IPU hardware. A full description of all Poplar errors can be found in the Exceptions section of the Poplar and PopLibs API Reference.

Note: To ensure the stability and reliability of your system, it is recommended that you configure PopRT runtime according to your requirements and error handling strategies.

PopRT handles Poplar errors as follows:

- application_runtime_error
  - If config.auto_reset is true, then the IPU is automatically reset before the next inference.
    - An IPU reset will be performed before the next execution. This reset doesn't affect other running IPUs.
    - Any new requests will be processed after the IPU reset is complete.
  - If config.auto_reset is false, then an error is raised.
    - The error message contains the reason why the error occurred.
    - All requests which have already been enqueued before the exception occurred will return the error.
- recoverable_runtime_error
  - If poplar::RecoveryAction is IPU_RESET and if config.auto_reset is true, then the IPU is automatically reset before the next inference.
    - An IPU reset will be performed before the next execution. This reset doesn't affect other running IPUs.
    - Any new requests will be processed after the IPU reset is complete.
  - If poplar::RecoveryAction is not IPU_RESET or if config.auto_reset is false, then an error is raised.
    - The error message contains the reason why the error occurred.
    - All requests which have already been enqueued before the exception occurred will return the error.
- Unknown runtime errors
  - An error is raised.

  - The error message might contain the reason why the error occurred.
  - When these errors occur manual intervention is required before the system is operational again.
  - The IPU will not be reset and all requests will return the error.
- All other runtime errors
  - An error is raised.
  - The error message might contain the reason why the error occurred.
  - When these errors occur manual intervention might be required before the system is operational again.
  - The error message might contain a required recovery action.
  - The IPU will not be reset and all requests will return the error.

5.14 PopRT frontend

The PopRT frontend is used to support the loading of model files generated by different frameworks. You can currently load only ONNX and TensorFlow models. You can specify the frontend for loading models with the PopRT CLI or the Python API (poprt.frontend).

5.14.1 ONNX frontend

The PopRT ONNX frontend is the default frontend in PopRT, and you do not need to specify additional parameters when using this frontend.

For more information on the ONNX frontend, refer to the poprt.frontend.OnnxFrontend class API description.

5.14.2 TensorFlow frontend

The PopRT TensorFlow frontend uses the API provided by tf2onnx to extend PopRT's support for TensorFlow models.

For more information on the Tensorflow frontend, refer to the poprt.frontend.TensorflowFrontend class API description.

Currently, TensorFLow SavedModel and the following CLI parameters are supported. For detailed information on these parameters, refer to tf2onnx subcommand.

Note: Except for the saved_model parameter, all other parameters have the same meaning as the parameters provided by tf2onnx. saved_model is used to indicate whether the loaded model is a TensorFlow SavedModel. The path to the saved model is specified with input_model.

- saved_model
- signature_def

- tag
- outputs
- opset
- inputs_as_nchw
- outputs_as_nchw

Loading a TensorFlow model with the PopRT CLI

The PopRT CLI uses the tf2onnx subcommand to process the parameters related to the conversion of a TensorFlow model to ONNX. You need to input these command line parameters immediately after the tf2onnx subcommand, for example:

```
poprt \
    --input_model resnet_v2_50 \
    --framework tensorflow \
    --input_shape inputs=1,224,224,3 \
    tf2onnx \
    --saved_model \
    --outputs Identity:0 \
    --tag serve \
    --signature_def serving_default \
    --inputs_as_nchw inputs:0 \
    --outputs_as_nchw Identity:0
```

Loading a TensorFlow model with Python API

You can also load a TensorFlow model with the poprt.frontend.TensorflowFrontend module, for example:

```
frontend = poprt.frontend.get_frontend(
    saved_model_path,
    framework='tensorflow',
    saved_model=True,
    signature_def=signature,
    tag=tag,
    opset=opset,
    inputs_as_nchw = inputs_as_nchw,
    outputs_as_nchw = outputs_as_nchw,
    input_shape=input_shape,
)
model = frontend.load_model()
```

5.15 Auto-sharding

PopRT auto-sharding supports the automatic selection of model sharding nodes to achieve model parallelism.

5.15.1 Model parallelism

PopRT supports sharding the ONNX graph across different devices based on provided sharding nodes to achieve model parallelism. It is suitable for large models that exceed the memory limits of a single device and require multiple devices to run.

Refer to the sharding section in the technical note Model Parallelism on the IPU with TensorFlow: Sharding and Pipelining for more information about sharding.

Note: To use model parallelism, the following PopRT backend options need to be configured as follows:

- options.virtual_graph_mode = "manual"
- options.num_ipus = number of devices

5.15.2 Principle of auto-sharding

Auto-sharding is based on a *manual sharding* increasing sharding scheme traversal strategy.

Alternative nodes strategy

Auto-sharding selects the alternative sharding nodes from the ONNX graph. The selection strategy selects a node in the ONNX graph that has multiple intermediate inputs, which does not include model inputs and constant inputs.

The list of alternative nodes satisfies topological sorting, and the subgraph corresponding to the alternative node is found by traversing the path from each alternative node to its input direction.

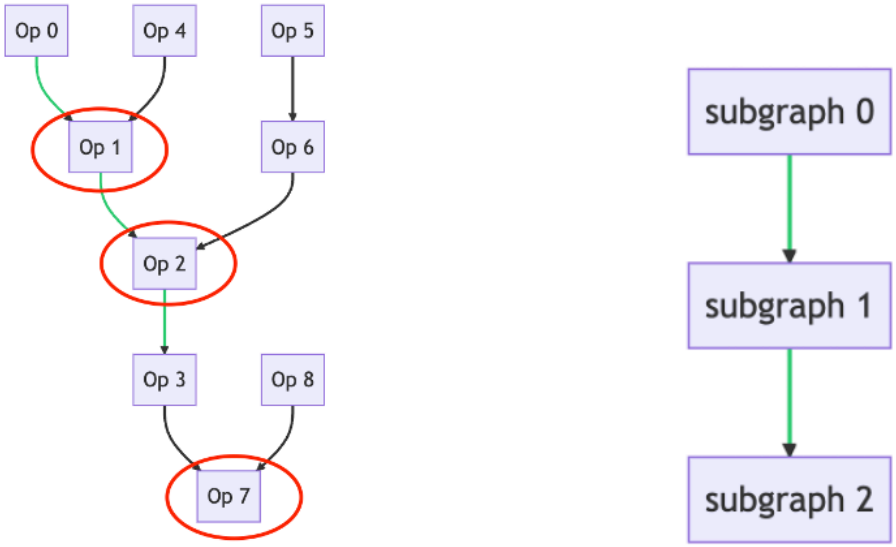
Each alternative node corresponds to a subgraph, and each subgraph records the corresponding **memory (bytes_cost)** and **computation (FLOPs_cost)**:

- **memory (bytes_cost)**: sum of initialiser input sizes for all nodes in the subgraph.
- **computation (FLOPs_cost)**: sum of FLOPs of all nodes in the subgraph. The corresponding FLOPs of each node is obtained through poprt.profile.Profiler.get_profiler().

Merge the alternative nodes that have less memory or computation, so as to reduce the number of alternative nodes and improve the traversal efficiency of the sharding scheme.

Note: The traversal strategy of the sharding scheme is to traverse the **subgraph** corresponding to the alternative node as the minimum unit. The list of subgraphs still satisfies the topological sorting.

As shown in the following figure, the alternative nodes are Op1, Op2, Op7. The subgraph corresponding to Op1 is [Op1, Op0, Op4], the subgraph corresponding to Op2 is [Op2, Op5, Op6], and the subgraph corresponding to Op7 is [Op7, Op3, Op8]. Traverse from the left image to the right image.



Traversal strategy of sharding scheme

Auto-sharding will select the sharding scheme from the alternative nodes:

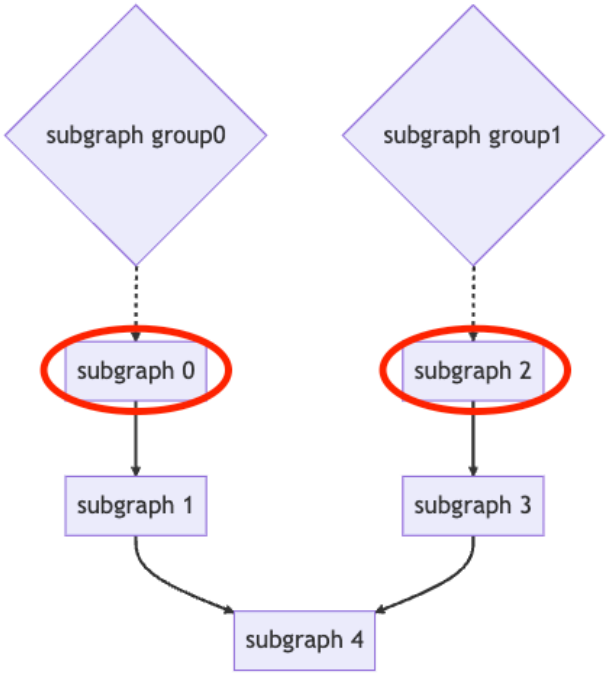
1. Initial sharding: The list of subgraphs satisfies the topological sorting. The list of subgraphs is directly divided by subgraph **memory (bytes_cost)** to ensure memory balance. As shown below, divide the list of subgraphs into four **subgraph groups**, with each **subgraph group** corresponding to one IPU.

| subgraph, subgraph, ··· | subgraph, subgraph, subgraph ··· | subgraph ··· | subgraph, subgraph··· |

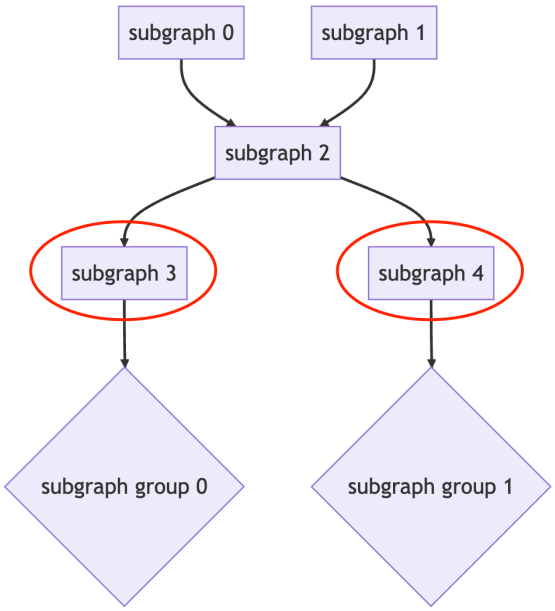
2. Traversal strategy:
    - Start traversal from the initial scheme.
    - If Out Of Memory (OOM) occurs, adjust the **subgraph group** corresponding to the IPU that went OOM according to the memory, and try to put the removable subgraphs in this **subgraph group** into the adjacent or parallel **subgraph group** respectively for compilation.
    - If the sharding scheme with better performance is selected, balance the **subgraph group** according to the computation, try to put the removable subgraphs in the **subgraph group** with the largest computation into the adjacent or parallel **subgraph group** respectively, and search for a more computationally balanced sharding scheme for compilation.

The update of the **subgraph group** has the following situations:

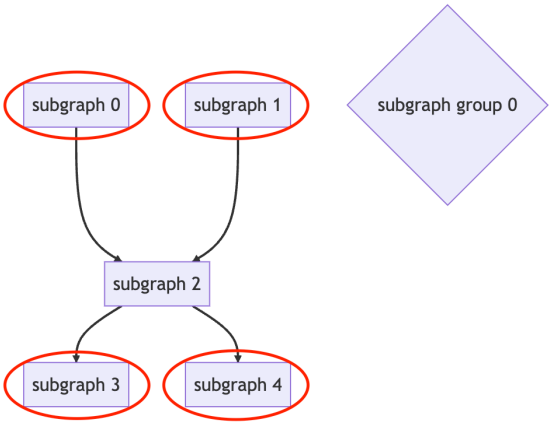
- The starting subgraph of each **subgraph group** can be moved into the corresponding **parent subgraph group**. As shown below, subgraph 0 can be moved into the parent subgraph group, subgraph group 0, and subgraph 2 can be moved into parent subgraph group, subgraph group 1.



- The ending subgraph of each **subgraph group** can be moved into the corresponding **child subgraph group** . As shown below, subgraph 3 can be moved into child subgraph group, subgraph group 0, and subgraph 4 can be moved into child subgraph group, subgraph group 1.



- The starting subgraph or ending subgraph of each **subgraph group** can be moved into the corresponding **parallel subgraph group**. As shown below, subgraphs 0, 1, 3 and 4 can be moved into parallel subgraph group subgraph group 0.



3. Adjustment of available memory promotion: If an OOM occurs in each sharding scheme in the traversal strategy, select the sharding scheme with the smallest OOM size, and try to reduce the available memory proportion to 0.3 and 0.1 for compilation.

5.15.3 Using auto-sharding

Note:

- Auto-sharding requires that the input ONNX model be converted in PopRT (*Quick start*).
- Regarding the compilation options, if the input ONNX model is FP16, partials_type defaults to half. available_memory_proportion performs the sharding strategy traversal based on the default values. If all the sharding strategies in the traversal are OOM, adjust the value for available_memory_proportion according to the aforementioned method and re-try the compilation. Apart from that, do not consider other compilation options. You can manually adjust compilation options based on the auto-sharding sharding scheme.
- Since the actual performance of the sharding scheme Profiling is not considered for adjustment, the sharding scheme found by auto-sharding attempting to traverse the compilation strategy may be a locally effective solution, but not necessarily a globally optimal solution. You can use *Manual Sharding* to manually adjust the sharding nodes based on the sharding scheme selected by auto-sharding.
- The time taken for auto-sharding is proportional to the graph compilation time. This time can be very long.

Auto-sharding tool

You can download the auto-sharding tool from GitHub: auto_sharding.py .

The parameters are:

- --input_model ${INPUT_MODEL}: The input ONNX model.
- --num_ipus ${NUM_IPUS}: Specify the number of IPUs
- --output_model ${OUTPUT_MODEL}: The output ONNX model. If not set, this will default to ${input_model}.auto_sharded.onnx.
- --optimal_perf: Indicates whether to enable performance optimisation. If not enabled, the traversal will stop after finding the first successfully compiled sharding scheme. If enabled, the best performing sharding scheme will be selected after traversing all sharding schemes.
- --num_processes ${NUM_PROCESSES}: The number of processes for parallel compilation. Default: 2.

Examples

1. Performance optimisation is not enabled. Shard the model across two IPUs. All the traversal sharding schemes are OOM. Select the sharding scheme with the smallest OOM size, try available_memory_proportion=0.3, and the compilation is successful.

```
python auto_sharding.py --input_model ../debug/deberta.onnx.optimized.onnx --num_ipus 2
...
[Success] Compile successfully available_memory_proportion 0.3 and solution:
Sharding nodes: Device 0 - ['Add_938']
[Success] The latency 336.2899446487427 ms, tput 2.9736244449548064
[Success] Save the sharded model to ../debug/deberta.onnx.optimized.onnx.auto_sharded.onnx
```

2. Performance optimisation is enabled. Shard the model across two IPUs, and the optimal solution is returned after traversal.

```
python auto_sharding.py --input_model=vit_l_16_16.onnx.optimized.onnx --num_ipus=2 --optimal_perf
...
[Success] The optimal solution:
Sharding nodes: Device 0 - ['/encoder/layers/encoder_layer_11/Add_1']
[Success] The optimal latency 2.712721824645996 ms, tput 368.6334481164495
[Success] Save the sharded model to vit_l_16_16.onnx.optimized.onnx.auto_sharded.onnx
```

5.16 Model debugger

PopRT provides an accuracy tracking tool (Model Debugger) to help debug accuracy issues. Model Debugger can be used to compare the accuracy of ONNX models or TensorFlow pb models, as well as to print and modify the specified ONNX models.

5.16.1 Model Debugger tool

You can download the Model Debugger tool from GitHub: model_debugger.py .

Model Debugger provides three subcommands (compare, modify and print), each with their own parameters:

- The compare subcommand compares the accuracy between the source model and the target mode and has the following parameters:
  - --src_model: specifies the path to the source model. Use --dst_model to specify the second model for accuracy comparison.
  - --src_backend: specifies the backend used to run the source model. Options are tensorflow (for TensorFlow), onnxruntime (for ONNX) and poprt (for PopRT). Default: onnxruntime.
  - --dst_model: specifies the target model path.
  - --dst_backend: specifies the backend used to run the target model. Options are tensorflow (for TensorFlow), onnxruntime (for ONNX) and poprt (for PopRT). Default: poprt.
  - --compare_method: specifies the comparison method. Separate multiple methods with a comma (",") . Options are: mse (mean square error), rmse (root mean squared error), mae (mean absolute error), mape (mean absolute percentage error) and r2 (R^2 coefficient of determination between two sets of results).
  - --input_data: specifies input data for each input tensor. It can be a path to the NumPy file or a character string from zeros, ones, or rand.
  - --mark_outputs: marks the name of the tensor to be compared. Separate multiple names with a comma (",") . If "all" is specified, then all intermediate tensors will be compared.
  - --mark_io_mapping: specifies input and output mappings to compare results of the same tensor with different names in the source model and target model.

  --dump_data: dumps the tensor data to be compared to a local file.

  --dump_dir: the directory to save the dumped data to.
- The modify subcommand modifies the input model using the specified operation and has the following parameters:
  - --src_model or --model: specifies the path to the source model.
  - --update: uses key=value to map and update the specified weight, where key is the weight name and value can be a path to a NumPy file or a character string from zeros, ones, or rand.
  - --extract: extracts subgraphs into the model. Use --extract input_names=in_name output_names=out_name to extract the subgraphs between in_name and out_name.
  - --sort: performs topological sorting on models.
  - --output_model: the path to save the modified model to.

The print subcommand prints the input model and has the following parameters:

- --src_model or --model: specifies the path to the source model.

- --level: sets the printing level. Options are:
  - model to print the model.
  - graph to print the graph.
  - node to print the nodes.
  - tensor to print the tensors.
- --name: specifies the name of the tensor or node to be printed.

5.16.2 Examples

Here are some examples

Listing 5.19: Comparing the accuracy of two ONNX models

```
python model_debugger.py compare --src_model model1.onnx --dst_model model2.onnx --input_data input=rand --mark_
↪outputs all
```

Listing 5.20: Comparing the accuracy of an ONNX model and a TensorFlow pb model

```
python model_debugger.py compare --src_model model.onnx --src_backend onnxruntime --dst_model model.pb --dst_backend␣
↪tensorflow --input_data x:0=rand --compare_method mse,rmse,mae --dump_data --dump_dir temp_output
```

Listing 5.21: Extracting the subgraphs between the specified input (Conv_output_0) and output (Mul_output_0)

```
python model_debugger.py modify --src_model model.onnx --extract input_names=Conv_output_0 output_names=Mul_
↪output_0 --output_model extracted.onnx
```

Listing 5.22: Updating the value of the specified weight

```
python model_debugger.py modify --src_model model.onnx --update onnx::Conv_770=zeros
```

Listing 5.23: Printing the specified weight

```
python model_debugger.py print --src_model model.onnx --name onnx::Conv_770
```

Listing 5.24: Printing the graph of the model

```
python model_debugger.py print --src_model ./debug_tools_dev/extracted.onnx --level graph
```

CHAPTER

SIX

PYTHON API

6.1 poprt module

class poprt.Converter(*, *input_shape=None*, *convert_version=11*, *precision='fp32'*, *checkpoints=None*, *eightbitsio=False*, *fp16_skip_op_types=None*, *skip_passes=None*, *used_passes=[]*, *check=False*, *disable_fast_norm=False*, *pack_args=None*, *fp8_skip_op_names=None*, *fp8_params='F143,F143,0,0'*, *quantize=False*, *enable_insert_remap=False*, *enable_erf_gelu=False*, *serialize_matmul=None*, *serialize_matmul_add=None*, *merge_matmul=None*, *merge_matmul_add=None*, *merge_moe=None*, *remap_mode=['after_matmul']*, *max_tensor_size=-1*, *infer_shape_ahead=False*, *enable_avoid_overflow_patterns=False*, *disable_progress_bar=False*, *batch_size=None*, *batch_axis=None*, *remove_outputs=[]*, *fold_periodic_initializer=False*, *enable_compress_pattern=False*, *merge_if_with_same_cond=False*, *apply_hcsr=None*, *logger=<Logger poprt (WARNING)>*)

Convert a general ONNX model to an IPU-friendly ONNX model.

Construct a new Converter.

Parameters

- input_shape (Dict[str, List[int]]) –The shape of the inputs.
- convert_version (int) –The version of the opset to convert to.
- precision (str) –The precision to convert the model to. Supported precisions: fp32, fp16, fp8.
- checkpoints (str) –Names to set output tensors to.
- eightbitsio (bool) –If True, enable 8-bit I/O. Default: False.
- fp16_skip_op_types (str) –The list of ops which will remain fp32 in the fp16 precision mode.
- skip_passes (str) –The list of passes which will be skipped.
- used_passes (List[str]) –The list of user-specified passes.
- disable_fast_norm (bool) –If True, disables transfer of the LayerNorm op to the FastNorm op.

- pack_args (Dict) –Enable the packed transformer.
- fp8_skip_op_names (str) –The names of ops which will remain at fp32 and fp16 in fp8 mode, such as ‘Conv_1, Conv_2’.
- fp8_params (str) –Set parameters for the fp8 model. The format is ‘input_format, weight_format ,‘input_scale‘ ,‘weight_scale‘’.
- quantize (bool) –If True, use the quantization method. Default: False.
- enable_insert_remap (bool) –If True, enables insertion of remap automatically to improve tensor layout.
- enable_erf_gelu (bool) –If True, enables replacement of Erf Gelu patterns with the Gelu op.
- serialize_matmul (Dict[str, str]) –Enable to serialize MatMul op to save memory on chip.
- serialize_matmul_add (Dict[str, str]) –Enable to serialize MatMul weights and Add bias with weights on last dim to save memory on chip.
- merge_matmul (str) –Enable to merge MatMul ops with last dim to save cycles.
- merge_matmul_add (str) –Enable to merge MatMul/Add ops with last dim to save cycles.
- merge_moe (str) –Enable to merge Mixture-of-Experts structure to save cycles.
- remap_mode (List[str]) –The insert position of the remap. Supported options: after_matmul, before_add and after_slice.
- max_tensor_size (int) –Max tensor size (in bytes) generated by constant_folding. -1 means do not set the max tensor size by default.
- infer_shape_ahead (bool) –Fix input shape and infer shapes at the beginning.
- batch_size (int) –Value to set for the batch size for all inputs. Works with the batch_axis parameter.
- batch_axis (int) –Value to set for the batch axis for all inputs. Works with the batch_size parameter.
- remove_outputs (List[str]) –List of outputs to remove from the graph.
- fold_periodic_initializer (bool) –Fold period initializer to save Always-Live memory.
- enable_compress_pattern (bool) –Enable replace Compress Ops with MaskCompress Op.
- check (bool) –
- enable_avoid_overflow_patterns (bool) –
- disable_progress_bar (bool) –
- merge_if_with_same_cond (bool) –
- apply_hcsr (Dict) –
- logger (Logger) –

convert(*model*)

Convert a general ONNX model to an IPU-friendly ONNX model.

Parameters model (ModelProto) –An ONNX ModelProto class object to be converted.

Returns An ONNX ModelProto class object representing the ONNX model.

Return type *ModelProto*

6.2 poprt.compiler module

class poprt.compiler.Compiler(*self: poprt._compiler.Compiler*) → None

Compile ONNX model to PopEF.

Return type None

static compile(*model: str, outputs: List[str], options: poprt._compiler.CompilerOptions = <poprt._compiler.CompilerOptions object at 0x7f27006cd5b0>*) → poprt::Executable

Parameters

- model (Union[AnyStr, onnx.ModelProto]) –
- outputs (List[str] | None) –
- options (CompilerOptions) –

Return type Executable

static compile_and_export(*model: str, outputs: List[str], filename: str, options: poprt._compiler.CompilerOptions = <poprt._compiler.CompilerOptions object at 0x7f27006c88b0>*) → None

Parameters

- model (Union[AnyStr, onnx.ModelProto]) –
- outputs (List[str] | None) –
- filename (str | None) –
- options (CompilerOptions) –

Return type None

static compile_and_get_summary_report(*model: str, outputs: List[str], options: poprt._compiler.CompilerOptions = <poprt._compiler.CompilerOptions object at 0x7f27006cd530>, reset_profile: bool = True*) → str

Parameters

- model (Union[AnyStr, ModelProto]) –
- outputs (List[str]) –

- options (CompilerOptions) –
- reset_profile (bool) –

Return type str

class poprt.compiler.CompilerOptions(*self: poprt._compiler.CompilerOptions*) → None

Return type None

6.3 poprt.runtime module

class poprt.runtime.Runner(*popef*, *config=None*)

Load PopEF model, and execute.

Parameters

- popef (Union[str, Executable]) –The input PopEF model.
- config (Union[RuntimeConfig, PackRunnerConfig]) –The runtime config.

Return type None

execute(*input*, *output*)

Execute the runner.

Parameters

- input (Dict[str, ndarray]) –
- output (Dict[str, ndarray]) –

Return type None

class poprt.runtime.DeviceManager(*self: poprt._runtime.DeviceManager*) → None

Device manager.

Return type None

get_device(*num_ipus*)

Get devices containing the required number of IPUs.

Parameters num_ipus (int) –The number of IPUs.

Returns The device.

Return type *Device*

get_num_devices()

Get the number of devices.

Return type int

get_specific_device(*device_id*)

Get specific devices.

Parameters device_id (int) –The ID of the target device.

Returns The device.

Return type *Device*

ipu_hardware_version()

Get IPU version.

The possible outputs are:

- ipu21: C600 cards
- ipu2: M2000 or Bow IPUs

Return type str

6.4 poprt.frontend module

class poprt.frontend.OnnxFrontend(*path*, ***kwargs*)

The ONNX frontend.

Parameters path (str) –The path to the input ONNX model.

Return type None

get_onnx_name(*dir_or_name*)

Filter out non-ONNX file.

Parameters

- files –A list of filenames.
- dir_or_name (str) –

Returns The ONNX model if there are only one ONNX model, otherwise throw an error.

Return type *Optional*[str]

load_model()

Load ONNX Model.

Parameters dir_or_name –The directory containing the ONNX model or the name of the model. If a directory is specified, then there should only be one ONNX model in the directory.

Returns The ONNX model.

Return type *ModelProto*

class poprt.frontend.TensorflowFrontend(*path*, ***, *saved_model=True*, *signature_def=''*, *tag=''*, *opset=11*, *inputs_as_nchw=None*, *outputs_as_nchw=None*, *inputs=None*, *input_shape={}*, *outputs=None*, ***kwargs*)

The TensorFlow frontend.

Parameters

- path (str) –The path to the input model.
- saved_model (bool) –If True, then this is a TensorFlow SavedModel, otherwise not.
- signature_def (str) –The SignatureDef from the SavedModel to use.
- tag (str) –The tag to use for the SavedModel.
- opset (int) –The opset version to use for the ai.onnx domain in the TensorFlow frontend.
- inputs_as_nchw (str) –Transpose inputs from nhwc to nchw.
- outputs_as_nchw (str) –Transpose outputs from nhwc to nchw
- output_names –The names of the output models (This parameter is optional if the model is a SavedModel.)
- inputs (str) –
- input_shape (dict) –
- outputs (str) –

Return type None

get_inputs_meta(*graph*)

Get the metadata of the input tensors of the TensorFlow graph.

Returns None

Return type None

get_outputs_meta(*graph*)

Get the metadata of the output tensors of the TensorFlow graph.

Returns None

Return type None

load_model()

Load TensorFlow model and convert to an ONNX ModelProto.

Return type *ModelProto*

split_nodename_and_shape(*name*)

input name with shape into name and shape.

6.5 poprt.backends module

class poprt.backends.Backend(*path_or_bytes*, ***, *export_popef=None*, *compiler_options=<poprt.compiler.CompilerOptions object>*, *runtime_options={"autoReset":false, "batchSizeTimeoutNS":9223372036854775807, "batchingDim":4294967295, "checkPackageHash":true, "dataParallelTimeoutNS":9223372036854775807, "deviceWaitConfig.sleepTimeSec":6, "deviceWaitConfig.timeoutSec":1, "flushOnWaitingOutputs":false, "isBatchSizeTimeoutEnabled":false, "requestTracepointsBufferSize":1000, "ringBufferSizeMultiplier":2, "runnerType":0, "threadSafe":true, "timeoutNS":10000000, "validateIOParams":true}*, *align_output_dtype=False*, *logger=None*)

The PopRT backend.

Parameters

- path_or_bytes (Union[AnyStr, IO[bytes], onnx.ModelProto]) –The input ONNX model.
- export_popef (str) –The path for the exported PopEF model.
- compiler_options (compiler.CompilerOptions) –Compiler options. See poprt.compiler.CompilerOptions.
- runtime_options (runtime.AnyConfig) –Runtime options. See poprt.runtime.RuntimeConfig
- align_output_dtype (bool) –If True, align output data type based on the ONNX model. Backend.run also has the parameter align_output_dtype. The output data type will be aligned if either of these two parameters is set to True.
- logger (logging.Logger) –A custom logger.

Return type None

get_io_info()

Get meta info of input and outputs, including data type, name and shape.

Return type tuple[Dict[str, Any], Dict[str, Any]]

run(*output_names*, *inputs*, *align_output_dtype=False*)

Run the model.

Parameters

- output_names (List[str]) –The names of the output tensors.
- inputs (Dict[str, ndarray]) –The input tensor data.
- align_output_dtype (bool) –If True, align output data type based on the ONNX model.

Return type *List*[*ndarray*]

class poprt.backends.ORTBackend(*path_or_bytes*, *sess_options=None*, *providers=None*, *provider_options=None*, *lazy_load=False*, ***kwargs*)

Bases: Backend

Backend compatible with onnxruntime.InferenceSession.

Parameters

- path_or_bytes –The input ONNX model.
- sess_options –onnxruntime.InferenceSession compatible API, not used
- providers –onnxruntime.InferenceSession compatible API. Not used.
- provider_options –onnxruntime.InferenceSession compatible API. Not used.
- lazy_load –If False, ORTBackend will load the ONNX model by default. Set to True to prevent this behaviour.
- **kwargs –See poprt.Backend for more args.

Return type None

run(*output_names*, *input_feed*, *run_options=None*)

Run the model.

Parameters

- output_names –The names of the output tensors.
- inputs –The input tensor data.
- align_output_dtype –If True, align output data type based on the ONNX model.

Return type *List*[*ndarray*]

6.6 poprt.quantizer module

poprt.quantizer.quantize(*onnx_model*, *input_model*, *output_dir*, *data_preprocess=None*, *precision='fp8'*, *quantize_loss_type='kld'*, *num_of_layers_keep_fp16=0*, *options=None*)

Quantize the model according to the specified strategy.

Only SimpleQuantizer is supported.

Parameters

- onnx_model (ModelProto) –The ONNX ModelProto.
- input_model (str) –The original model.
- output_dir (str) –the output dir
- data_preprocess (Optional[str]) –The path to the pickle format file for data preprocessing. The storage format is {input_name_1: ndarray_1, input_name_2: ndarray_2, ···}.
- precision (typing_extensions.Literal[fp8, fp8_weight]) –The precision to be used in the conversion.

- quantize_loss_type (str) –choose the calibration method, default is kld.
- num_of_layers_keep_fp16 (int) –set the layer whose loss is topk to fp16 in fp8 quantization.
- options (Optional[Dict[str, Any]]) –Options for the conversion.

Returns A quantized ONNX ModelProto.

Return type *ModelProto*

class poprt.quantizer.FP8Quantizer(*output_dir*, *loss_type*, *data_preprocess=None*, *precision='fp8'*, *num_of_layers_keep_fp16=0*, *options=None*)

Return the input model.

Parameters

- output_dir (str) –
- loss_type (str) –
- data_preprocess (str) –
- precision (typing_extensions.Literal[fp8, fp8_weight]) –
- num_of_layers_keep_fp16 (int) –
- options (Dict[str, Any]) –

6.7 poprt.passes module

class poprt.Pass(**args*, ***kwargs*)

Abstract base class for passes.

A new pass could be as follows:

```
import onnx
from poprt.passes import register, Pass


@register('dummy_pass')
class Dummy(Pass):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

    def run(self, onnx_model: onnx.ModelProto) -> onnx.ModelProto:
        print(f"producer_name: {onnx_model.producer_name}")
        return onnx_model
```

Return type None

static against_passes(*pass_names*)

Register against property for a pass.

Passes can't work with against passes.

Parameters pass_names (List[str]) –

Return type *Callable*[[*Any*], *PassReg*]

static constraint_passes(*constraint_name*, *pass_names*)

Register constraints for a pass.

Valid constraints are against, depend and before.

Parameters

- constraint_name (str) –
- pass_names (List[str]) –

Return type *Callable*[[*Any*], *PassReg*]

static get_pass(*name*, **args*, ***kwargs*)

Get a pass by its registered name.

Parameters name (str) –The registered name of the pass.

Returns An instance of Pass.

Return type Pass

Example:

```
import poprt

# get a Pass with parameters
onnx_model = poprt.get_pass('float_to_half', skip_op_types=['Gelu'])(onnx_model)

poprt.get_pass('model_overview')(onnx_model)
```

static get_registered_passes()

Get all registered passes.

Return type *Dict*[str, *PassReg*]

static get_typed_registered_passes(*pass_type*)

Get typed registered passes.

Parameters pass_type (Any) –

Return type *Dict*[str, Pass]

static override_register_pass(*pass_name*)

Register a pass, override a registered pass if needed.

Parameters pass_name (str) –

Return type *Callable*[[*Any*], *PassReg*]

static property_register(*k*, *v*)

Register a property for a pass.

Parameters

- k (str) –
- v (Any) –

Return type *Callable*[[*Any*], *PassReg*]

static register_pass(*pass_name*)

Register a pass.

Parameters pass_name (str) –

Return type *Callable*[[*Any*], *PassReg*]

run(*onnx_model*)

Run a pass.

Inherited subclasses should override this method.

Parameters onnx_model (ModelProto) –The input ONNX model.

Returns The optimized ONNC model.

Return type *ModelProto*

traverse_graph(*graph*, *transform*, *is_main_graph=True*)

Traverse a and transform a GraphProto.

Parameters

- graph (GraphProto) –The input graph.
- transform (Callable[[GraphProto, bool], GraphProto]) –The transform function.
- is_main_graph (bool) –

Return type *GraphProto*

class poprt.PassManager(*used_passes=[]*, *gather_ir_passes=False*)

Manage passes.

Parameters

- used_passes (List[Union[str, Pass]]) –List of passes that will be used.
- gather_ir_passes (bool) –If True, gather the ONNX IR passes and execute it in one turn.

Return type None

Example:

```
import poprt

pm = poprt.PassManager(
    [
```

(continues on next page)

(continued from previous page)

```
        'model_overview',
        'float_to_half',
        poprt.get_pass('model_overview'),
    ]
)

pm.run(onnx_model)
```

add_passes(*used_passes=[]*, *gather_ir_passes=False*)

Add passes for PassManager.

Parameters

- used_passes (List[Union[str, Pass]]) –The list of passes that will be used.
- gather_ir_passes (bool) –If True, gather the ONNX IR passes and execute it in one turn.

Return type None

get_all_pass_names()

Get names of all passes.

Return type *List*[str]

get_passes()

Get all passes.

Return type *List*[Pass]

run(*onnx_model*)

Apply passes to the ONNX model.

Parameters onnx_model (ModelProto) –The ONNX model that will be optimized.

Returns The optimized ONNX model.

Return type *ModelProto*

sort_passes()

Solve pass dependency.

6.7.1 Built-in passes

Refer to the Section 5.1, Passes section for more details about passes.

class poprt.passes.add_checkpoints.AddCheckpoints(*checkpoints*)

Add an intermediate tensor to the output.

Parameters checkpoints (List[str]) –

Return type None

This pass is registered as add_checkpoints.

class poprt.passes.apply_host_concat_split.ApplyHostConcatSplit(*merged_inputs*, *remap=False*)

Merge model inputs with the same shape.

NOTE: this is an experimental feature. Only tested on merging 2D inputs.

For example, a onnx graph has 3 inputs with same shape [512, 1] and dtype fp16. Before this Pass(only show inputs info):

```
+--+-----+---+
+-512*1*fp16-+

+--+-----+---+
+-512*1*fp16-+

+--+-----+---+
+-512*1*fp16-+
```

After applying this Pass:

```
                +------+  -> +--+-----+---+
+------------+     |      |     +-512*1*fp16-+
|            |     |      |
| 3*512*fp16 | --> | HCSR |  -> +--+-----+---+
|            |     | Node |     +-512*1*fp16-+
+------------+     |      |
                   |      |  -> +--+-----+---+
                   +------+     +-512*1*fp16-+
```

The raw inputs will be replaced with one new input with shape [3, 512] and same dtype. The HCSR Node is a custom operation equal to Split + Reshape. it has 3 new outputs, these outputs have same shape and data from raw model inputs.

Parameters merged_inputs (List[Any]) –

This pass is registered as apply_host_concat_split.

class poprt.passes.apply_ir_pass.ApplyIrPass(*passes=[]*)

Apply passes based on the ONNX IR.

Parameters passes (List[str]) –

Return type None

This pass is registered as apply_ir_pass.

class poprt.passes.attention_padding.AttentionPadding(**args*, ***kwargs*)

Recognise the attention pattern and pad inputs of MatMul Op to speed up.

Return type None

This pass is registered as attention_padding.

class poprt.passes.auto_insert_remap.AutoInsertRemap(*remap_mode=['after_matmul']*)

Insert a Remap op after or before the assigned op.

This is an experimental feature. Pass 'before/after' + '_' + 'op_type' to assign remap_mode. For example, there are two different insert modes:

- **after_matmul: Add the Remap op after the MatMul. This mode is more** general but is more likely to go OOM.
- **before_add. Add the Remap op before the Add op. This mode is targetted at reducing cycles of attention + mask** in transformer-based models.

Parameters remap_mode (List[str]) –

Return type None

This pass is registered as auto_insert_remap.

class poprt.passes.workarounds.BatchNormWorkaround(**args*, ***kwargs*)

Workaround for the BatchNorm op.

Return type None

This pass is registered as batchnorm_workaround.

class poprt.passes.check_unspported_attribute.ReplaceUnsupportedAttr(**args*, ***kwargs*)

Check the attributes that popart does not supported.

Return type None

This pass is registered as check_unsupported_attribute.

class poprt.passes.check_with_fake_data.CheckWithFakeData(*origin_model*)

Check model with fake data using the ONNX runtime.

Parameters origin_model (ModelProto) –

Return type None

This pass is registered as check_with_fake_data.

class poprt.passes.compress_pattern.CompressPattern(**args*, ***kwargs*)

Return type None

This pass is registered as compress_pattern.

class poprt.passes.const_batch_size.ConstBatchSize(*const_batch_size=1*)

Convert an unknown batch size to a const value.

Parameters const_batch_size (int) –

Return type None

This pass is registered as const_batch_size.

class poprt.passes.const_input_shape.ConstInputShape(*const_input_shape={}*, *batch_size=None*, *batch_axis=None*)

Convert the input shape to const values.

Parameters

- const_input_shape (Dict[str, Any]) –
- batch_size (int) –
- batch_axis (int) –

This pass is registered as const_input_shape.

class poprt.passes.constant_folding.ConstantFolding(*max_tensor_size=- 1*)

Support constant folding.

Parameters max_tensor_size (int) –

Return type None

This pass is registered as constant_folding.

class poprt.passes.workarounds.CumSumWorkaround(**args*, ***kwargs*)

Workaround for the CumSum op.

Return type None

This pass is registered as cumsum_workaround.

class poprt.passes.double_to_float.DoubleToFloat(**args*, ***kwargs*)

Convert double to float (only for initializer).

Return type None

This pass is registered as double_to_float.

class poprt.passes.eight_bits_io.EightBitsIO

Insert norm operator after input image.

This pass is registered as eight_bits_io.

class poprt.passes.apply_ir_pass.eliminate_deadend(**args*, ***kwargs*)

Return type None

This pass is registered as eliminate_deadend.

class poprt.passes.apply_ir_pass.eliminate_duplicate_initializer(**args*, ***kwargs*)

Return type None

This pass is registered as eliminate_duplicate_initializer.

class poprt.passes.apply_ir_pass.eliminate_duplicate_nodes(**args*, ***kwargs*)

Return type None

This pass is registered as eliminate_duplicate_nodes.

class poprt.passes.apply_ir_pass.eliminate_identity(**args*, ***kwargs*)

Return type None

This pass is registered as eliminate_identity.

class poprt.passes.apply_ir_pass.eliminate_nop_arithmetic(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_arithmetic.

class poprt.passes.apply_ir_pass.eliminate_nop_cast(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_cast.

class poprt.passes.apply_ir_pass.eliminate_nop_expand(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_expand.

class poprt.passes.apply_ir_pass.eliminate_nop_flatten(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_flatten.

class poprt.passes.apply_ir_pass.eliminate_nop_if(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_if.

class poprt.passes.apply_ir_pass.eliminate_nop_pad(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_pad.

class poprt.passes.apply_ir_pass.eliminate_nop_reshape(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_reshape.

class poprt.passes.apply_ir_pass.eliminate_nop_transpose(*args, **kwargs)

Return type None

This pass is registered as eliminate_nop_transpose.

class poprt.passes.apply_ir_pass.eliminate_unused_initializer(*args, **kwargs)

Return type None

This pass is registered as eliminate_unused_initializer.

class poprt.passes.erf_gelu_pattern.ErfGeluPattern(*args, **kwargs)

Recognise the pattern of the Erf Gelu op and replace the pattern with Erf Gelu.

Return type None

This pass is registered as erf_gelu_pattern.

class poprt.passes.expand_if.ExpandIf(**args*, ***kwargs*)

Merge if nodes that use same condition input.

Return type None

This pass is registered as expand_if.

class poprt.passes.apply_ir_pass.extract_constant_to_initializer(**args*, ***kwargs*)

Return type None

This pass is registered as extract_constant_to_initializer.

class poprt.passes.fill_squeeze_axes.FillSqueezeAxes(**args*, ***kwargs*)

Fill the empty axes of the Squeeze op to ensure that shape-inference works.

Return type None

This pass is registered as fill_squeeze_axes.

class poprt.passes.final_check.FinalCheck(**args*, ***kwargs*)

Final check for data type and shape of the converted model.

Return type None

This pass is registered as final_check.

class poprt.passes.workarounds.FloatOpsWorkaround(**args*, ***kwargs*)

Workaround for ops which are required with float32 or float16 inputs.

Return type None

This pass is registered as float_ops_workaround.

class poprt.passes.float_to_fp8.Float2FP8(*fp8_params=['F143', 'F143', 0, 0]*, *skip_op_names=[]*, *convert_model='fp8'*, *fp8_input_dict=None*, *fp8_weight_dict=None*, *skip_op=True*)

Convert a model from fp32 or fp16 to fp8.

Parameters

- fp8_params (List[Union[typing_extensions.Literal[F143, F152], str]]) –Set of parameters for the fp8 model. The format is [input_format, weight_format, input_scale, weight_scale].
- skip_op_names (List[str]) –The names of ops which will not be converted in fp8 mode. They will remain as fp16 or fp32. Example names [‘Conv_1’, ‘Conv_2’].
- convert_model (typing_extensions.Literal[fp8, fp8_weight]) –Specifies which fp8 type the model is converted to, can be set to ‘fp8’ or ‘fp8_weight’
- fp8_input_dict (Dict[str, int]) –Set parameters for each fp8 input node of the fp8 model. If it’s not None, fp8_params will be discarded.

- fp8_weight_dict (Dict[str, int]) –Set parameters for each fp8 weight node of the fp8 model. If it's not None, fp8_params will be discarded.
- skip_op (bool) –

Return type None

This pass is registered as float_to_fp8.

class poprt.passes.float_to_half.Float2Half(*skip_op_types=[]*, *enable_avoid_overflow_patterns=False*)

Convert a model from fp32 to fp16.

Create a Float2Half instance.

Parameters

- skip_op_types (List[str]) –The op types which will remain as fp32 in the fp16 mode.
- enable_avoid_overflow_patterns (bool) –Enable to keep fp32 for several specific patterns in the fp16 model.

Returns A Float2Half instance

Return type None

This pass is registered as float_to_half.

class poprt.passes.float_to_int8.Float2Int8(*int8_offset_scale=None*)

Convert a model from fp32 or fp16 to int8.

Parameters int8_offset_scale (Dict) –a dict to store int8 value, offset and scale. For example:

Return type None

```
{
   "weight_name": {
      "value": np.int8
      "offset": np.float32,
      "scale": np.float32,
      }
}
```

This pass is registered as float_to_int8.

class poprt.passes.float_to_mixed.Float2Mixed

Convert a model from fp32 to mixed precision.

Return type None

This pass is registered as float_to_mixed.

class poprt.passes.fold_periodic_initializer.FoldPeriodicInitializer

Fold periodic initializer to save Always-Live memory.

Return type None

This pass is registered as fold_periodic_initializer.

class poprt.passes.apply_ir_pass.fuse_bn_into_conv(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_bn_into_conv.

class poprt.passes.fuse_bn_into_gemm.FuseBnIntoGemm

Fuse BatchNormalization to MatMul/Gemm.

Conditions: Condition 1: MatMul/Gemm uses an initializer. Condition 2: No multi outputs in Gemm/MatMul. Condition 3: Initializers used across operators is not supported.

Return type None

This pass is registered as fuse_bn_into_gemm.

class poprt.passes.fuse_cast_into_onehot.FuseCastIntoOnehot

Fuse Cast into OneHot.

Return type None

This pass is registered as fuse_cast_into_onehot.

class poprt.passes.apply_ir_pass.fuse_consecutive_arithmetic(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_arithmetic.

class poprt.passes.apply_ir_pass.fuse_consecutive_cast(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_cast.

class poprt.passes.apply_ir_pass.fuse_consecutive_reshape(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_reshape.

class poprt.passes.apply_ir_pass.fuse_consecutive_squeeze(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_squeeze.

class poprt.passes.apply_ir_pass.fuse_consecutive_transpose(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_transpose.

class poprt.passes.apply_ir_pass.fuse_consecutive_unsqueeze(**args*, ***kwargs*)

Return type None

This pass is registered as fuse_consecutive_unsqueeze.

class poprt.passes.fuse_mul_into_matmul.FuseMulIntoMatmul

Fuse Mul into MatMul.

Return type None

This pass is registered as fuse_mul_into_matmul.

class poprt.passes.fused_attention.FusedAttention(**args*, ***kwargs*)

Recognise the pattern of MultiHeadAttention and replace it with Fused MultiHeadAttention. A Attention pattern is as follows:

```
Add
|
Reshape  --    --    --
|          \          \
MatMul      MatMul     MatMul
|          |          |
Reshape     Reshape    Reshape
|          |          |
Add         Add        Add
|          |          |
Reshape     Reshape    Reshape
```

A Fused Attention pattern is as follows:

```
Add
|
Concat
|
MatMul
|
Add
|
Reshape
|
Transpose
|
Split
```

Return type None

This pass is registered as fused_attention.

class poprt.passes.gelu_pattern.GeluPattern(**args*, ***kwargs*)

Recognise the pattern of the Gelu op and replace the pattern with Gelu.

Return type None

This pass is registered as gelu_pattern.

class poprt.passes.workarounds.IndicesWorkaround(**args*, ***kwargs*)

Workaround for the Gather and GatherElements ops.

Return type None

This pass is registered as indices_workaround.

class poprt.passes.insert_attention_mask.InsertAttentionMask(**args*, ***kwargs*)

Replace Reshape-Cast-Sub-Mul with Cast-AttentionMask.

Return type None

This pass is registered as insert_attention_mask.

class poprt.passes.int64_to_int32.Int64ToInt32(**args*, ***kwargs*)

Convert int64 to int32.

Return type None

This pass is registered as int64_to_int32.

class poprt.passes.layer_norm_pattern.LayerNormPattern(**args*, ***kwargs*)

Recognise the pattern of the LayerNorm op and replace the pattern with GroupNorm.

Return type None

This pass is registered as layer_norm_pattern.

class poprt.passes.layer_precision_compare.LayerPrecisionCompare(*origin_model*, *data_preprocess=None*, *options=None*, *output_dir='./'*)

Compare the output of the Conv/MatMul/Gemm op of the original model and the fp8 model.

It will randomly take a batch of data from the calibration for inference, and then records the output of the original model and the converted model. We use cosine distance to evaluate the error because it is a normalized number that measures the angle between vectors. The closer the value is to 0, the smaller the error. The log is written to a log file.

Create LayerPrecisionCompare instance.

Parameters

- data_preprocess (str) –The pPath of the pickle format file for data preprocessing.
- options (Dict[str, Any]) –The options for the session.
- output_dir (str) –The save path of the log.
- origin_model (ModelProto) –

Returns A LayerPrecisionCompare instance

Return type None

This pass is registered as layer_precision_compare.

class poprt.passes.manual_sharding.ManualSharding(*sharding_info=None*, *pipelining_info=None*)

Shard the graph into several subgraphs manually in terms of specific nodes.

Parameters

- sharding_info (Dict[str, int]) –
- pipelining_info (Dict[str, int]) –

Return type None

This pass is registered as manual_sharding.

class poprt.passes.matmul_rotary_embedding.MatmulRotaryEmbedding

Recognise the pattern of element-wise rotary embedding and replace the pattern with the equivalent matmul.

Return type None

This pass is registered as matmul_rotary_embedding.

class poprt.passes.merge_if_with_same_cond.MergeIfWithSameCond(**args*, ***kwargs*)

Merge if nodes that use same condition input.

Return type None

This pass is registered as merge_if_with_same_cond.

class poprt.passes.merge_matmul.MergeMatmul(*merge_str=None*)

Parameters merge_str (str) –

Return type None

This pass is registered as merge_matmul.

class poprt.passes.merge_matmul_add.MergeMatmulAdd(*merge_str=None*)

Parameters merge_str (str) –

Return type None

This pass is registered as merge_matmul_add.

class poprt.passes.merge_moe.MergeMoE(*merge_str=None*)

Parameters merge_str (str) –

Return type None

This pass is registered as merge_moe.

class poprt.passes.merge_multi_slice.MergeMultiSlice(**args*, ***kwargs*)

Return type None

This pass is registered as merge_multi_slice.

class poprt.passes.merge_variant_if.MergeVariantIf(**args*, ***kwargs*)

Move nodes to subgraph of if operator.

Return type None

This pass is registered as merge_variant_if.

class poprt.passes.model_io_info_overview.modelIOInfoOverview(*use_print=True*, **args*, ***kwargs*)

Show the input/output information statistics of the model.

Return type None

This pass is registered as model_io_info_stat.

class poprt.passes.model_overview.ModelOverview(*use_print=True*, **args*, ***kwargs*)

Print overview information of the model to stdout.

Return type None

This pass is registered as model_overview.

class poprt.passes.move_subgraph_initializer.MoveSubgraphInitializer

Move the subgraph initializers into the main graph.

PopART only searches initializers in the main graph.

Return type None

This pass is registered as move_subgraph_initializer.

class poprt.passes.workarounds.OneHotWorkaround(**args*, ***kwargs*)

Workaround for the OneHot op which needs to have an int32 depth and a positive axis.

Return type None

This pass is registered as onehot_workaround.

class poprt.passes.overlap_io.OverlapIO

Enable overlap IO.

Return type None

This pass is registered as overlap_io.

class poprt.passes.packed_transformer.PackedTransformer(*args*)

Recognise the pattern of SelfAttention and replace it with Packed SelfAttention.

This pass is registered as packed_transformer.

class poprt.passes.post_expand.PostExpand(**args*, ***kwargs*)

Return type None

This pass is registered as post_expand.

class poprt.passes.pre_scale.PreScale(**args*, ***kwargs*)

Pre scale: Attention matrix Q to Q/sqrt(d), and remove 1/sqrt(d) node.

Return type None

This pass is registered as pre_scale.

class poprt.passes.remove_duplicated_initializer.RemoveDuplicatedInitializer

Remove duplicated initializer to save memory.

Return type None

This pass is registered as remove_duplicated_initializer.

class poprt.passes.workarounds.RemoveEmptyConcatInputs(**args*, ***kwargs*)

Workaround for the Concat op which does not support empty inputs in PopART.

Return type None

This pass is registered as remove_empty_concat_inputs.

class poprt.passes.remove_initializer_from_input.RemoveInitializerFromInput(**args*, ***kwargs*)

Remove initializer from model inputs.

Model: https://github.com/onnx/models/blob/main/vision/classification/resnet/model/resnet50-v1-7.onnx

Return type None

This pass is registered as remove_initializer_from_input.

class poprt.passes.remove_input_cast.RemoveInputCast(**args*, ***kwargs*)

Remove input cast: input(fp16)->cast(fp16->int32)->gather to input(int32)->gather.

Return type None

This pass is registered as remove_input_cast.

class poprt.passes.remove_outputs.RemoveOutputs(*outputs=[]*)

Remove specific outputs and useless structures of the graph.

Parameters outputs (List[str]) –

Return type None

This pass is registered as remove_outputs.

class poprt.passes.replace_bn_with_mul_add.ReplaceBNWithMulAdd(**args*, ***kwargs*)

Replace BatchNormalization op with Mul + Add ops.

Return type None

This pass is registered as replace_bn_with_mul_add.

class poprt.passes.replace_castlike.ReplaceCastLike(**args*, ***kwargs*)

Replace ONNX CastLike op with Cast.

Return type None

This pass is registered as replace_castlike.

class poprt.passes.replace_clip_empty_inputs.ReplaceClipInputs(**args*, ***kwargs*)

Replace empty inputs in Clip op.

Return type None

This pass is registered as replace_clip_empty_inputs.

class poprt.passes.replace_consecutive_cast_with_notzero.ReplaceConsecuiveCastWithNotZero(**args*, ***kwargs*)

Recognise the pattern of consecutive Cast ops and replace the pattern with a NotZero op.

Return type None

This pass is registered as replace_consecutive_cast_with_notzero.

class poprt.passes.replace_div_with_mul.ReplaceDivWithMul(**args*, ***kwargs*)

Replace Div with Mul if the divisor is constant.

Model: https://github.com/onnx/models/blob/main/text/machine_comprehension/gpt-2/model/gpt2-10.onnx

Return type None

This pass is registered as replace_div_with_mul.

class poprt.passes.apply_ir_pass.replace_einsum_with_matmul(**args*, ***kwargs*)

Return type None

This pass is registered as replace_einsum_with_matmul.

class poprt.passes.replace_erf_with_erfv2.ReplaceErfWithErfV2(**args*, ***kwargs*)

Replace the Erf op with ErfV2.

ErfV2 is more efficient with bigger errors.

Return type None

This pass is registered as replace_erf_with_erfv2.

class poprt.passes.replace_gemm_with_matmul.ReplaceGemmWithMatMul(**args*, ***kwargs*)

Replace Gemm with MatMul in the ONNX model.

Return type None

This pass is registered as replace_gemm_with_matmul.

class poprt.passes.replace_greater_or_equal.ReplaceGreaterOrEqual(**args*, ***kwargs*)

Replace the GreaterOrEqual op with the Less and Not ops.

Return type None

This pass is registered as replace_greater_or_equal.

class poprt.passes.replace_groupnorm_with_fast_norm.ReplaceGroupNormWithFastNorm(**args*, ***kwargs*)

Replace GroupNormalization with FastNorm if datatype is fp16 and num_groups =1.

Return type None

This pass is registered as replace_groupnorm_with_fast_norm.

class poprt.passes.replace_half_reducemean.ReplaceHalfReduceMean(**args*, ***kwargs*)

Replace the ReduceMean op in fp16 mode with ReduceSum + Mul in case of overflow.

Return type None

This pass is registered as replace_half_reducemean.

class poprt.passes.replace_hardswish.ReplaceHardSwish(**args*, ***kwargs*)

Replace the HardSwish op with the HardSigmoid and Mul ops.

Replacement is required for opsets earlier than 14 since HardSwish is only supported from opset version 14.

Return type None

This pass is registered as replace_hardswish.

class poprt.passes.replace_isinf.ReplaceIsInf(**args*, ***kwargs*)

Replace the IsInf op with the IsInfV2 op (support detect_negative and detect_positive).

Return type None

This pass is registered as replace_isinf.

class poprt.passes.replace_less_or_equal.ReplaceLessOrEqual(**args*, ***kwargs*)

Replace the LessOrEqual op with the Less and Not ops.

Return type None

This pass is registered as replace_less_or_equal.

class poprt.passes.replace_nonzero.ReplaceNonZero(**args*, ***kwargs*)

Replace NonZero with ArgMax when the number of non-zero elements is known.

Only a single element is supported.

Return type None

This pass is registered as replace_nonzero.

class poprt.passes.replace_pow.ReplacePow(**args*, ***kwargs*)

Replace Pow op with the Square and Mul ops.

Return type None

This pass is registered as replace_pow.

class poprt.passes.replace_round.ReplaceRound(**args*, ***kwargs*)

Replace the Round op with the RoundV2 op (half to even mode).

Return type None

This pass is registered as replace_round.

class poprt.passes.replace_softmax.ReplaceSoftmax(**args*, ***kwargs*)

Replace the Softmax op with the SoftmaxV2 op when the axis is the lowest dim and the lowest dim is odd.

Return type None

This pass is registered as replace_softmax.

class poprt.passes.replace_where_mask.ReplaceWhereMask(**args*, ***kwargs*)

Change the attention mask method from "where" to "add".

Return type None

This pass is registered as replace_where_mask.

class poprt.passes.replace_where_with_mul_add.ReplaceWhereWithMulAdd

Replace the Where op with the Add and Mul ops.

Where(condition, X, Y) = Add(Mul(condition, X), Mul(neg_condition, Y)).

This pass is registered as replace_where_with_mul_add.

class poprt.passes.replace_where_with_wherev2.ReplaceWhereWithWhereV2(**args*, ***kwargs*)

Replace the Where op with WhereV2.

Return type None

This pass is registered as replace_where_with_wherev2.

class poprt.passes.serialize_matmul.SerializeMatmul(*serialize_dict=None*)

Enable serializing of the Matmul op to save memory on chip.

Parameters serialize_dict (Dict) –

Return type None

This pass is registered as serialize_matmul.

class poprt.passes.serialize_matmul_add.SerializeMatmulAdd(*serialize_dict=None*)

Parameters serialize_dict (Dict) –

Return type None

This pass is registered as serialize_matmul_add.

class poprt.passes.shape_inference.ReplacePow(**args*, ***kwargs*)

Run shape inference.

Return type None

This pass is registered as shape_inference.

class poprt.passes.sort_graph.SortGraph(**args*, ***kwargs*)

sort subgraph to keep topological order in popart.

Return type None

This pass is registered as sort_graph.

class poprt.passes.workarounds.TopKWorkaround(**args*, ***kwargs*)

Workaround for the TopK op which needs to have a positive axis.

Return type None

This pass is registered as topk_workaround.

class poprt.passes.apply_ir_pass.trace_folding(**args*, ***kwargs*)

Return type None

This pass is registered as trace_folding.

class poprt.passes.apply_ir_pass.unique_name_for_nodes(**args*, ***kwargs*)

Return type None

This pass is registered as unique_name_for_nodes.

CHAPTER

SEVEN

C++ API

7.1 PopRT Compiler

class Compiler

Public Static Functions

static void compileAndExport(const std::string &model, const std::vector<std::string> &outputs, std::ostream &out, const *CompilerOptions* &options = *CompilerOptions*())

Compile model and export PopEF model to stream.

Parameters

- model –**[in]** An ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- outputs –**[in]** The names of output tensors.
- out –**[out]** The stream that the compiled PopEF model will be written to.
- options –**[in]** The user configuration options for the *Compiler* class. Default: CompilerOptions().

static void compileAndExport(const std::string &model, const std::vector<std::string> &outputs, const std::string &fileName, const *CompilerOptions* &options = *CompilerOptions*())

Compile model and Export PopEF to file.

Parameters

- model –**[in]** An ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- outputs –**[in]** The names of the output tensors.
- fileName –**[out]** The filename that the compiled PopEF model will be written to.
- options –**[in]** The user configuration options for the *Compiler* class. Default: CompilerOptions().

static std::shared_ptr<*Executable*> compile(const std::string &model, const std::vector<std::string> &outputs, const *CompilerOptions* &options = *CompilerOptions*())

Compile and return an executable object.

Parameters

- model –**[in]** An ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- outputs –**[in]** The names of the output tensors.
- options –**[in]** The user configuration options for the *Compiler* class. Default: CompilerOptions().

static std::string compileAndGetSummaryReport(const std::string &model, const std::vector<std::string> &outputs, const *CompilerOptions* &options, bool resetProfile = true)

Compile the model and return a summary report.

Parameters

- model –**[in]** An ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- outputs –**[in]** The names of the output tensors.
- options –**[in]** The user configuration options for the *Compiler* class. Default: CompilerOptions().
- resetProfile –**[in]** If True, resets the execution profile. Default = True.

Returns A string containing the report.

struct CompilerOptions

Public Members

int64_t numIpus = 1

The number of IPUs to select.

std::string ipuVersion = ""

IPU version. Auto detect if empty.

int64_t batchesPerStep = 1

The number of batches to run on the chip before returning.

std::string partialsType = "half"

The partials type to set globally for MatMuls and convolutions. Valid values are "float" and "half".

float availableMemoryProportion = 0.6

The value to set globally for the the available memory proportion for matmuls and convolutions. Valid values are between 0 and 1 (inclusive) [=0.6].

int64_t numIOTiles = 0

The number of IPU tiles dedicated to I/O.

bool enableModelFusion = false

Enable model fusion. By default, model fusion is disabled.

bool enablePrefetchDatastreams = true

Enable prefetching for input data streams. Prefetching is enabled by default.

Poplar will speculatively read data for a stream before it is required in order to allow the 'preparation' of the data to occur in parallel with compute. Enabled when True. Default: True.

unsigned streamBufferingDepth = 1

Specify the default buffering depth value used for streams.

bool enablePadConvChannel = false

Custom patterns.

bool serializeIr = false

Enable (set to True) to serialize the IR. Disabled by default.

std::string serializedIrDest = ""

Destination to dump IR serialization stream.

bool enableGatherSimplifier = true

Enable (set to True) to simplify Gather op. Enabled by default.

std::map<std::string, std::string> engineOptions

Poplar engine options.

bool showCompilationProgressBar = true

Show progress bar when compiling.

std::function<void(int, int)> compilationProgressLogger

Callback function used to indicate PopART compilation 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.

Param int The progress value.

Param int The maximum value for the progress.

std::vector<std::string> customPatterns

Specify custom patterns.

std::map<std::string, std::vector<std::string» customTransformApplierSettings

Specify custom transforms.

std::map<std::string, std::string> opaqueBlobs

Specify opaque blob messages.

bool use128BitConvUnitLoad = false

Bit-width of the conv load.

bool enableMultiStageReduce = true

If true, perform the reduction following the convolution in multiple stages if it would significantly reduce code size. This comes at the cost of increasing the number of cycles.

bool enableFastReduce = false

Enable fast reduce.

bool enableOutlining = true

Enable (set to True) outlining. Enabled by default.

bool groupHostSync = false

Specify to group the h2d streams at the beginning of the schedule and the d2h streams at the end of the schedule.

When True, tensors will stay live for longer.

Default: False.

bool rearrangeStreamsOnHost = false

Enable rearrangement of h2d tensors to be done on the host.

Default: False (Rearrangement done on device).

bool rearrangeAnchorsOnHost = true

Enable rearrangement of d2h tensors to be done on the host.

Default: True (Rearrangement done on host to save device memory).

float outlineThreshold = 1.0f

Specify the incremental value that a subgraph requires, relative to its nested subgraphs (if any), to be eligible for outlining.

Default: 1.0f.

bool enableNonStableSoftmax = false

Enable the non-stable softmax Poplar function.

Default: False (not enabled).

bool enablePipelining = false

Enable pipelining of virtual graphs.

Default: False (not enabled).

bool enableEngineCaching = false

Enable Poplar executable caching. The file is saved to the location defined with cachePath.

Default: False (not enabled).

std::string cachePath = ""

Folder to save the Poplar executable to.

Default: "" .

bool constantWeights = true

Specify an optimization for an inference session to have constant weights.

Default: True.

uint64_t subgraphCopyingStrategy = 0

Specify how copies for inputs and outputs for subgraphs are lowered.

Default: popart::OnEnterAndExit.

std::string virtualGraphMode = "off"

Specify how to place ops on virtual graphs to achieve model parallelism, either manually using model annotations, or automatically.

Default: popart::VirtualGraphMode::Off.

std::vector<float> virtualGraphSplitRatios

Specify split ratios when VirtualGraphModel::Auto is enabled.

These values represent split ratios in each device and each of the values is in the range (0, 1). For example, to uniformly split the whole graph on 4 IPUs, the values should be [0.25, 0.25, 0.25, 0.25].

double timeLimitScheduler = 1e9

The maximum allowed time (in seconds) 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.

int transitiveClosureOptimizationThreshold = 100000

Specify the transitive closure optimization threshold.

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 this threshold, the transitive closure optimization pass is not run.

bool createHostTransferableTensorWithOffset = false

Accumulate the created tensors bytes, rotate the start tile of the next tensor to balance the tile mapping. Especially when there are a lot of small input tensors, enable it can avoid mapping on tile0 all the time.

Default=false.

bool enableEfficientOverlapIOTopoCons = false

Enable simplified and equivalent overlapIO constraints.

Suppose we have the N bins in each of three stage(8 for before loop /7 for insdie loop /6 for after loop), and L ops for each bins, vallina implementaiton of overlapio creates topocons of complexity O(N*N*L*L).

To make sure InitOps in each step are scheduled before HostLoadOps, we only need to keep topo constrains in each bin and let the last of op of each bin Bin0 is scheduled before the first op of Bin1 next to Bin0. Then total complexity O(N*N*L*L) is reduced to (N*L).

Default: false (not enabled).

7.2 Executable

class Executable

Executable object of the PopEF model.

Public Functions

Executable() = delete

~Executable() = default

Executable(const *Executable*&) = delete

Executable &operator=(const *Executable* &other) = delete

Executable(*Executable*&&) = default

Executable &operator=(*Executable*&&) = default

explicit Executable(const std::string &popefPath)

Create a new *Executable* object.

Parameters popefPath –**[in]** The PopEF model file.

explicit Executable(const std::vector<std::string> &popefPaths)

Create a new *Executable* object.

Parameters popefPaths –**[in]** The PopEF model files.

explicit Executable(const std::shared_ptr<std::istream> &in, std::streamoff size = 0)

Create a new *Executable* object.

Parameters

- in –**[in]** The stream to parse.
- size –**[in]** The number of bytes to read from the stream. If 0, parse.

Executable(std::shared_ptr<popef::Model> popefModel)

Create a new *Executable* object.

Parameters popefModel –**[in]** The PopEF model object.

void appendOpaqueBlobs(const std::map<std::string, std::string> &opaqueBlobs)

Append opaque blobs.

void appendOpaqueBlobs(const std::pair<std::string, std::string> &opaqueBlob)

Append opaque blobs.

std::shared_ptr<popef::Model> getPopefModel()

Get the PopEF Model.

const std::map<std::string, std::string> &getOpaqueBlobs()

Get the opaque blobs.

7.3 PopRT Runtime

class BaseRunner

Subclassed by poprt::runtime::LightRunner, *poprt::runtime::ModelRunner*, *poprt::runtime::PackRunner*

Public Functions

BaseRunner() = default

virtual ~BaseRunner() = default

BaseRunner &operator=(const *BaseRunner*&) = delete

BaseRunner(*BaseRunner*&&) = default

virtual *BaseRunner* &operator=(*BaseRunner*&&) = default

virtual void execute(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) = 0

Run a model synchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.
- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors

virtual *OutputFutureMemoryView* executeAsync(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) = 0

Run a model asynchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.
- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors.

Returns The future result of an asynchronous call for all executable output tensors.

virtual std::vector<*InputDesc*> getExecuteInputs() const = 0

Get a description of the input data required in the execute class methods.

Returns A vector of *DataDesc* instances.

virtual std::vector<*OutputDesc*> getExecuteOutputs() const = 0

Get a description of the output data required in the execute class methods.

Returns A vector of *DataDesc* instances.

enum class poprt::runtime::RunnerType

Values:

enumerator ModelRunner = 0

enumerator PackRunner

enumerator ModelFusionRunner

class ModelRunner : public poprt::runtime::*BaseRunner*

Subclassed by poprt::runtime::ModelFusionRunner

Public Functions

ModelRunner(const *ModelRunner*&) = delete

ModelRunner &operator=(const *ModelRunner*&) = delete

ModelRunner(*ModelRunner*&&) = default

ModelRunner &operator=(*ModelRunner*&&) = default

~ModelRunner() override

ModelRunner(const std::string &popefPath, const *RuntimeConfig* &config = *RuntimeConfig*())

Create a new *ModelRunner* object.

Parameters

- popefPath –**[in]** The path to PopEF files from which the model will be loaded.
- config –**[in]** The runtime configuration.

ModelRunner(const std::shared_ptr<*Executable*> &executable, const *RuntimeConfig* &config = *RuntimeConfig*())

Create a new *ModelRunner* object.

Parameters

- executable –**[in]** The executable which will load the model.
- config –**[in]** The runtime configuration.

virtual void execute(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) override

Run a model synchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.

- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors

virtual *OutputFutureMemoryView* executeAsync(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) override

Run a model asynchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.
- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors.

Returns The future result of an asynchronous call for all executable output tensors.

virtual std::vector<*InputDesc*> getExecuteInputs() const override

Get a description of the input data required in the execute class methods.

Returns A vector of *DataDesc* instances.

virtual std::vector<*OutputDesc*> getExecuteOutputs() const override

Get a description of the output data required in the execute class methods.

Returns A vector of *DataDesc* instances.

class PackRunner : public poprt::runtime::*BaseRunner*

Public Functions

PackRunner(const *PackRunner*&) = delete

PackRunner &operator=(const *PackRunner*&) = delete

PackRunner(*PackRunner*&&) = default

PackRunner &operator=(*PackRunner*&&) = default

~PackRunner() override

PackRunner(const std::string &popefPath, const *PackRunnerConfig* &config)

Create a new *PackRunner* object.

Parameters

- popefPath –**[in]** The path to PopEF files from which the model will be loaded.
- config –**[in]** The pack runner configuration.

PackRunner(const std::shared_ptr<*Executable*> &executable, const *PackRunnerConfig* &config)

Create a new *PackRunner* object.

Parameters

- executable –**[in]** The *Executable* which will load the model.

- config –**[in]** The pack runner configuration.

virtual void execute(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) override

Run a model synchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.
- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors

virtual *OutputFutureMemoryView* executeAsync(*InputMemoryView* &inputData, *OutputMemoryView* &outputData) override

Run a model asynchronously. The user allocates and passes pointers to output memory. Non-const MemoryView is used here to avoid possible copy by some runners.

Parameters

- inputData –**[in]** The user-allocated tensor buffer for all executable input tensors.
- outputData –**[in]** The user-allocated tensor buffer for all executable output tensors.

Returns The future result of an asynchronous call for all executable output tensors.

virtual std::vector<*InputDesc*> getExecuteInputs() const override

Get a description of the input data required in the execute class methods.

Returns A vector of *DataDesc* instances.

virtual std::vector<*OutputDesc*> getExecuteOutputs() const override

Get a description of the output data required in the execute class methods.

Returns A vector of *DataDesc* instances.

struct RuntimeConfig

Public Members

RunnerType runnerType = *RunnerType*::*ModelRunner*

The type of the target runner.

DeviceWaitConfig deviceWaitConfig

By default, the model runner throws an exception when it is not able to attach to any device required by the given model. This behavior can be changed by setting a custom *DeviceWaitConfig*.

bool threadSafe = true

If true, the mutex will be locked on each execution call. If false, the mutex will not be locked. By default the model runner is not thread-safe and each replica has an independent mutex. Default: true.

std::chrono::nanoseconds $\mathrm{timeoutNS}$ = std::chrono::milliseconds(10)

Duration in nanoseconds to wait before calling timeout callback when the IPU is waiting for input data, which is not available. If 0, never call the timeout, in other words, wait forever for the data.

bool $\mathrm{validateIOParams}$ = true

If true, the I/O parameters will be checked during the execution *ModelRunner* "execute" functions. If false, this check is not done. Default: true.

uint32_t $\mathrm{batchingDim}$ = std::numeric_limits<uint32_t>::max()

The dimension which the input data will extend with the batch size. For example, the PopEF model with shape [4, 4, 3, 3] and batchingDim=0, means the batch size is extended on dimension 0. Then, input data with shapes [?, 4, 3, 3] will be allowed, where ? can be [1, 2, ⋯, N].

The default value is std::numeric_limits<uint32_t>::max(), which means dynamic batch sizing is disabled, and the input data can only be $\mathrm{N} * \mathrm{batch\_size\_of\_popef\_model}$, for example [n * 4, 4, 3, 3] for the above model, where n can be [1, 2, ⋯, N]

bool $\mathrm{checkPackageHash}$ = true

If true, the Poplar hash will be checked before the executable is loaded onto the device. If false, this check is not done. Default: true.

int64_t $\mathrm{ringBufferSizeMultiplier}$ = 2

The multiplier used to determine the size of the ring buffer. The ring buffer size is given by $\mathrm{ringBufferSizeMultiplier} * \mathrm{batch\ size}$.

bool $\mathrm{autoReset}$ = false

If true, the IPU will be reset automatically before next inference when application runtime error or recoverable error of that recovery action is IPU_RESET occur. If false, the error will be raised and no action will be token. Default: false.

bool $\mathrm{flushOnWaitingOutputs}$ = false

If true, only flush the dummy data when there is a user request waiting for outputs from queues. Otherwise, wait forever for user-requested data in the callback. If false, flush the dummy data anyway.

std::chrono::nanoseconds $\mathrm{batchSizeTimeoutNS}$ = std::chrono::nanoseconds::max()

Duration in nanoseconds before the timeout callback is called when the IPU is waiting for input data to compose a batch to load.

If $\mathrm{max}()$, the callback will never be called, in other words, the model will wait forever for the data.

std::chrono::nanoseconds $\mathrm{dataParallelTimeoutNS}$ = std::chrono::nanoseconds::max()

Duration in nanoseconds before the timeout callback is called when the IPU is waiting in parallel for input data.

If $\max()$, the callback will never be called, in other words, the model will wait forever for the data.

bool $\mathrm{isBatchSizeTimeoutEnabled}$ = false

Indicates whether the batch size timeout is enabled. If true, then check the *batchSizeTimeoutNS* and *dataParallelTimeoutNS* in the timeout callback when the IPU is waiting for input data which doesn't arrive.

size_t $\mathrm{requestTracepointsBufferSize}$ = 1000

Size of the request tracepoint buffer.

If set to 0, the request tracepoint buffer size will be infinite. Note: if $\mathrm{flush\_on\_waiting\_outputs}$ is set to false, no request tracepoints will not be recorded.

ReplicaIdToDevice $\mathrm{replicaIdToDevice}$ = {}

Mapping between replica ID and *Device*. This allows you to set a specific device for a given replica. By default, the model runner assigns devices automatically.

std::pair<*InputMemoryView*, *OutputMemoryView*> $\mathrm{flushByName}$

The config option that specifies the data to the callback function for flushing data that is not available.

$\mathrm{flushByName}$ indicates to runtime which data to flush.

If specified, a $\mathrm{flush\_callback}$ function will be called once when the IPU has timed out after waiting for input data, which is not available.

The application which defines the flushByName function must ensure that the inputs and outputs it specifies for runtime to flush are correct. Otherwise the results of this and following inferences will be corrupted.

struct $\mathrm{PackRunnerConfig}$

Public Functions

inline explicit $\mathrm{PackRunnerConfig}$(int timeoutInMicroSeconds = 0, int maxValidNum = 0, std::string dynamicInputName = "", std::string unpackInfoInputName = "")

void $\mathrm{enablePaddingRemovePattern}$(std::string maskName, std::vector<std::string> dynamicGroup)

Used to remove padding from user based on mask.

void $\mathrm{enableSingleRowMode}$(std::string maskName, std::string unpackInfoName = "", int delimiterNum = 0)

Enable pack mode in which data can no across rows.

Public Members

RunnerType runnerType = *RunnerType*::*PackRunner*

The type of the target runner.

DeviceWaitConfig deviceWaitConfig

By default, the model runner throws an exception when it is not able to attach to any device required by the given model. This behavior can be changed by setting a custom *DeviceWaitConfig*.

int timeoutInMicroSeconds = 0

Used to determine when to force to push the user input data into the queue, even if the *PackRunner* can receive more data. The value of timeoutInMicroSeconds should be greater than 0.

int compileBatchSize = -1

Used for notifying the pack runner about the model batch size during compile time, as it may not always be obtainable from the model.

int maxValidNum = 0

maxValidNum is the maximum samples that *PackRunner* can reach. *PackRunner* will stop pack when reached maxValidNum samples or reached the maximum space that user allowed or reached limited time.

std::string dynamicInputName = ""

Dynamic sequence input name.

std::string unpackInfoInputName = ""

Unpack info input name.

std::string maskName = ""

Attention mask name, used to remove padding of user input.

std::vector<std::string> dynamicGroup

Used to specify group of dynamic inputs when remove padding(e.g., {input_ids, mask, token_type, position_ids}). Fixed size input name should not be in dynamicGroup.

PackAlgorithm algo = PackAlgorithm::NextFit

bool disableDataAcrossRows = false

User input cannot across rows in this pack mode.

bool enablePaddingRemove = false

Remove pad from user.

int delimiterNum = 0

Used to insert delimiter before pack.

bool enablePrefetchDatastreams = true

is prefetch datastreams

struct DeviceWaitConfig

Public Functions

DeviceWaitConfig() = default

inline DeviceWaitConfig(int timeoutSec, int sleepTimeSec)

Public Members

std::chrono::seconds timeoutSec = std::chrono::seconds(1)

The time in seconds to wait for a device.

std::chrono::seconds sleepTimeSec = std::chrono::seconds(6)

The time in seconds between attach attempts.

struct DataDesc

The description of data used by *ModelRunner*.

Public Functions

DataDesc(std::string name, int64_t sizeInBytes, std::vector<int64_t> shape, popef::DataType dataType, bool popefContainsTensorData = false)

Create description of input/output data.

Parameters

- name –**[in]** The name of the input/output tensor.
- sizeInBytes –**[in]** The size of the tensor measured in bytes.
- shape –**[in]** A vector defining the shape of the tensor. The size of the vector is equal to the number of tensor dimensions. Each element of the vector indicates the size of a single dimension.

- dataType –**[in]** The data type of a single tensor element.
- popefContainsTensorData –**[in]** If true, the model has a tensor data blob associated with the tensor. If false, the model does not have a tensor data blob associated with the tensor. Default: false.

Public Members

std::string name

The name of the input/output tensor.

int64_t sizeInBytes

The size of the tensor measured in bytes.

std::vector<int64_t> shape

A vector defining the shape of the tensor. The size of the vector is equal to the number of tensor dimensions. Each element of the vector indicates the size of a single dimension.

popef::DataType dataType

The data type of a single tensor element.

bool popefContainsTensorData

If true, the model has a tensor data blob associated with the tensor. If false, the model does not have a tensor data blob associated with the tensor.

using poprt::runtime::InputDesc = *DataDesc*

Description of input data required by *ModelRunner*.

using poprt::runtime::OutputDesc = *DataDesc*

Description of output data required by *ModelRunner*.

using poprt::runtime::InputMemoryView = std::unordered_map<std::string, *ConstTensorMemoryView*>

Mapping between a tensor name and an immutable memory view. Used as input to *ModelRunner::execute*.

using poprt::runtime::OutputMemoryView = std::unordered_map<std::string, *TensorMemoryView*>

Mapping between a tensor name and a memory view. Used as output from *ModelRunner::execute*, when the output memory is allocated and managed by the *ModelRunner* client.

using poprt::runtime::OutputFutureMemoryView = std::unordered_map<std::string, std::shared_future<*TensorMemoryView*»

Mapping between a tensor name and a future memory view. Used as output from *ModelRunner::executeAsync*, when the output memory is allocated and managed by the *ModelRunner* client.

struct ConstTensorMemoryView

Immutable view to already allocated memory.

Public Functions

ConstTensorMemoryView() = default

Default constructor.

ConstTensorMemoryView(const *TensorMemoryView* &other)

Default copy constructor.

ConstTensorMemoryView(const void *data, uint64_t dataSizeBytes)

Immutable view to const memory.

Parameters

- data –**[in]** The pointer to the allocated memory.
- dataSizeBytes –**[in]** The size of the memory block, in bytes.

Public Members

const void *data = nullptr

Pointer to the allocated memory.

uint64_t dataSizeBytes = 0

The size of the memory block, in bytes.

struct TensorMemoryView

Mutable view to already allocated memory.

Public Functions

TensorMemoryView() = default

Default constructor.

TensorMemoryView(void *data, uint64_t dataSizeBytes)

Immutable view to memory.

Parameters

- data –**[in]** The pointer to the allocated memory.
- dataSizeBytes –**[in]** The size of the memory block, in bytes.

Public Members

void *data = nullptr

The pointer to the allocated memory.

uint64_t dataSizeBytes = 0

The size of the memory block, in bytes.

struct TensorMemory

Tensor memory manager responsible for allocating, storing, sharing and releasing tensor memory.

Public Functions

TensorMemory() = default

Default constructor.

TensorMemory(int64_t dataSizeBytes)

Allocate the user-requested memory block and store it in a shared_ptr.

Parameters dataSizeBytes –**[in]** The size of the memory block, in bytes.

TensorMemoryView getView()

Get the mutable memory view.

ConstTensorMemoryView getConstView()

Get the immutable memory view.

ConstTensorMemoryView getView() const

Get the immutable memory view.

Public Members

uint64_t dataSizeBytes = 0

The size of the memory block, in bytes.

std::shared_ptr<void> data = nullptr

Pointer to the allocated memory.

7.3.1 DeviceManager

class Device

Public Functions

Device(const *Device*&) = delete

Device &operator=(const *Device* &other) = delete

Device(*Device*&&) = default

Device &operator=(*Device*&&) = default

Device(std::shared_ptr<model_runtime::Device>)

~Device()

std::string ipuVersion() const

Get the version of the device.

const std::shared_ptr<model_runtime::Device> device() const

Get model_runtime::Device.

class DeviceManager

Public Functions

DeviceManager(const *DeviceManager*&) = delete

DeviceManager &operator=(const *DeviceManager* &other) = delete

DeviceManager(*DeviceManager*&&) = default

DeviceManager &operator=(*DeviceManager*&&) = delete

DeviceManager()

~DeviceManager()

std::string ipuHardwareVersion()

Get the version of the IPU on the physical system.

std::size_t getNumDevices() const

Get the number of devices attached to this host.

std::shared_ptr<*Device*> getDevice(int64_t numIpus)

Get a device matching the requested configuration.

Parameters numIpus –**[in]** The number of IPUs the device must contain.

Returns The device, if attachment is successful, otherwise throw an error.

std::shared_ptr<*Device*> getSpecificDevice(int64_t deviceId)

Get a specific device matching the requested configuration.

Parameters deviceId –**[in]** The ID of the device to acquire.

Returns The device, if attachment is successful, otherwise throw an error.

CHAPTER

EIGHT

REVISION HISTORY

Date	Version	Description
16th of Oct 2023	v1.5.0	Release v1.5.0
16th of Aug 2023	v1.4.0	Release v1.4.0
5th of Jul 2023	v1.3.0	Release v1.3.0
29th of May 2023	v1.2.0	Release v1.2.0
30th of March 2023	v1.1.0	Release v1.1.0
22th of February 2023	v1.0.0	Release v1.0.0
18th of January 2023	v0.9.0	Release v0.9.0
8th of December 2022	v0.8.0	First release

CHAPTER

NINE

TRADEMARKS & COPYRIGHT

INDEX

F

G

I

L

M

O

P

Q

R

S

T

U