Arm China Zhouyi Compass NN Compiler

Version: 1.0

User Guide

Confidential

arm CHINA

Arm China Zhouyi Compass NN Compiler

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0001-00	30 September 2023	Confidential	First release

Confidential Proprietary Notice

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm China and the party that Arm China delivered this document to.

Product Status

The information in this document is Final, that is a for a developed product.

Web Address

https://www.armchina.com

Contents
Arm China Zhouyi Compass NN Compiler User Guide

Preface

This preface introduces the *Arm China Zhouyi Compass NN Compiler User Guide*.

It contains the following sections:

About this book

This book describes how to use the NN compiler for Zhouyi NPUs.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This chapter describes the basic concept of the Zhouyi NN compiler.

Chapter 2 Quantization

This chapter describes the theory of quantization using in the NN compiler.

Chapter 3 Auto-tiling

This chapter describes the tiling mechanism of the NN compiler.

Chapter 4 Using the NN compiler

This chapter describes the basic usage of the NN compiler.

Chapter 5 IR definition

This chapter describes the IR definition and the relationship with the NN compiler.

Chapter 6 Profiling network

This chapter shows how to profile a network with the NN compiler.

Chapter 7 Supporting customized operators

This chapters shows how to support a customized operator.

Appendix A NPU auxiliary tools

This appendix describes the NPU auxiliary tools.

Glossary

The Arm® Glossary is a list of terms used in Arm China documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm China meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm China publications

The following confidential documents are only available to licensees:

- *Arm China Zhouyi Compass Assembly Programming Guide*.
- *Arm China Zhouyi Compass C Programming Guide*.
- *Arm China Zhouyi Compass Debugger User Guide*.
- *Arm China Zhouyi Compass Getting Started Guide*.

- *Arm China Zhouyi CompassStudio User Guide*.
- *Arm China Zhouyi Compass Software Technical Overview*.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content, send an e-mail to *errata@armchina.com*. Give:

- The title *Arm China Zhouyi Compass NN Compiler User Guide*.
- The number 61010024_0001_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm China also welcomes general suggestions for additions and improvements.

_____ Note_____

Arm China tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter describes the basic concept of the Zhouyi NN compiler.

It contains the following sections:

1.1 About the Zhouyi NN compiler

The NPU NN compiler converts a neural network model into graph that can be run on the NPU.

The following figure shows the workflow of the NPU NN compiler.

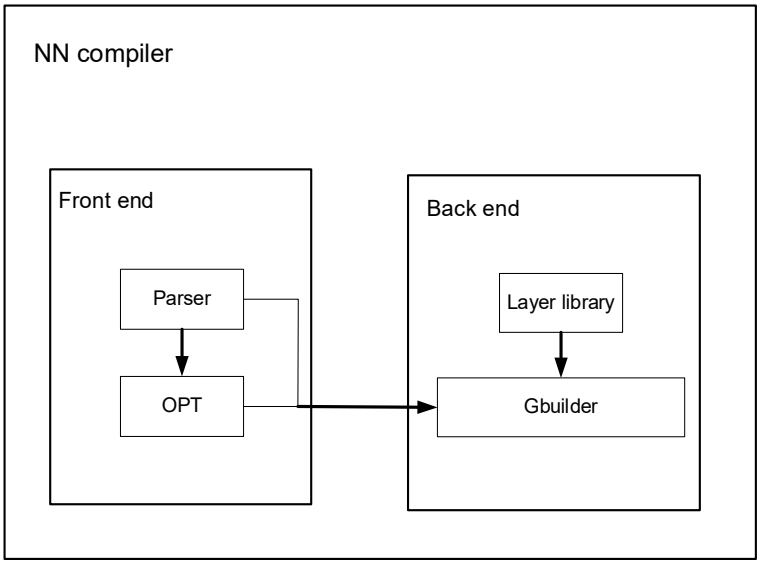

Figure 4-1 Workflow of the NN compiler

1.2 Workflows

The NN compiler is a serial tool for compiling a neural network model to an NPU executable file.

The running of the NN compiler mainly includes three steps:

1. Convert a pretrained model to an *Intermediate Representation* (IR).
   The IR can be a float IR or quantized IR (if the input model is a *Quantization Aware Training* (QAT) model. For more information, see the *Arm China Zhouyi Compass IR Definition Application Note*.
2. Convert a float IR to a quantized IR if needed.
3. Generate an executable file for the NPU with the IR.

There are the following corresponding three modules for the three steps:

- Model parse module—The model parse module (also called the Parser) will parse a third-party model and convert to an NPU internal IR.
- Quantization module—The quantization module (also called the OPT, short for Optimizer) will quantize the model into the quantized model if needed, reorganize the quantized data, and then transfer the quantized model to the generation module.
- Generation module—The generation module (also called the GBuilder) will optimize the execute path and lower the model to fit NPU instructions, and then build the model to an executable file.

In addition to these modules, there is an IR-to-IR optimization module, called GSim. The GSim module works after the parse module and quantization module to optimize the float IR and quantized IR.

1.2.1 Model parse module

The model parse module is designed for converting pretrained models to the IR. The following model formats are supported in this version:

- TensorFlow frozen model (.pb, .h5 and SaveModel type) for TensorFlow version 1.0–2.6.
- TensorFlow Lite model (.tflite) for TensorFlow version 1.0–2.6.
- Caffe model (.caffemodel (required), .prototxt (optional)) for Caffe version 1.
- ONNX model (.onnx) up to opsets 18.

For the TensorFlow model, LSTM and GRU operators only support TensorFlow version 1.0-1.15.

For the TensorFlow Lite model, LSTM operators only support TensorFlow version 1.0-1.15.

For the Caffe model, some non-official layers such as shuffle and detection output are included.

In the model parse module, a pretrained model should be present, as well as some key information about the model:

- Input node or tensor name of the model, generally this input node is a placeholder in TensorFlow.
- Input tensor shape(s).
- Output node name(s).

To parse a model, the module will do the following:

1. Reading the model file (for example, the frozen pb file) and build a raw graph.
2. Converting the raw graph nodes to unified nodes.
3. Optimization in graph level, for example, fusing Convolution+BatchNorm.
4. Inference of shape and adding some addition nodes such as post-process nodes if necessary.

1.2.2 Quantization module

For lower memory storage and lower computation cost, the quantization module can convert the float models to quantized models.

For the operations that need quantization, Arm China provides the corresponding quantization method.

The quantization module does the following:

1. Receive the float *Internal Representation* (IR) and build the float graph.
2. Calibrate each operation to get the output range and the constant data range (like weight and bias) of each operation.
3. Quantize each operation.
   - Get the output scale and output datatype according to the output range and the operator definition. The quantization precision is determined by 'activation_bits' which can be set in the OPT configuration file.
   - For operations with weight and bias, quantize weights to 'weight_bits' (default value=8) and biases to 'bias_bits' (default_value=32). The 'weight_bits' and 'bias_bits' can be set in the OPT configuration file.
   - For operations with complicated computation, like the exponent or logarithm operation, provide a quantized *Look Up Table* (LUT). The LUT length will affect the degree of approximation, and is determined by 'lut_items_in_bits' (LUT length = 2^lut_items_in_bits. The default value is 8, that is, the LUT length is 256. It is recommended to set it to 10–12 when quantizing a model to int16) which can be set in the OPT configuration file.
4. Serialize the quantized IR and perform the optional metrics based on the corresponding assigned dataset.

The OPT automatically uses CUDA to accelerate calculation when it is available. To disable this feature, set the environment variable CUDA_VISIBLE_DEVICES to '-1' or ''.

For the QAT models, you do not need to perform the quantization, because the model data is already quantized. However, it still requires some processes on the quantization data, so that the data can be recognized and used by the NPU operators. These processes are performed by a sub-module in the Optimizer called the *Quantization Tool Lib* (QTLib). See *4.3.4 QAT model support* for more information.

1.2.3 Generation module

The generation module is the backend of the NN compiler. The generation module is the last step for building a neural network model.

In the generation module, the process is similar to any other compiler:

1. Parsing the IR to a *Directed Acyclic Graph* (DAG).
   In this stage, the IR will be parsed, and you will construct a DAG object to represent the IR.
2. Legalizing/Lowering the DAG to meet the hardware specifications.
   In this stage, you will manipulate the DAG to fit the HW requests, such as splitting a node into two nodes, and transforming a subgraph to another subgraph.
3. Generating the binary.
   In this stage, you will generate all parameters for all nodes and construct a binary.

Note

The generation module will select AIFF operators with the priority for the operator emitting stage.

1.2.4 IR optimization module

The IR optimization module (GSim) is an IR-to-IR optimization module that uses common compiler optimization techniques and deep learning optimization techniques. It supports both float IR and quantized IR.

In the GSim module, many passes are performed to optimize the IR:

1. Parsing the IR to DAG.
2. Performing passes such as common sub-expression elimination, operation canalization, transpose elimination, and constant folding.
3. Generating the optimized IR.

Chapter 2
Quantization

This chapter describes the theory of quantization using in the NN compiler.

It contains the following sections:

2.1 Quantization algorithm

According to the quantization scheme, the quantized model can be obtained from a pre-trained model in float, which means there is forward pass in the integer while back-propagation is still in the float pipeline, so that there is no network re-training required.

The following figure shows the 8-bit quantization general pipeline of an operation.

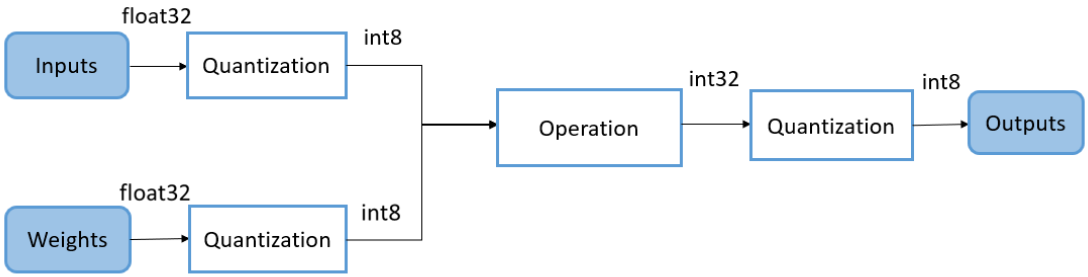

Figure 4-2 Quantization pipeline of a general operation

As shown in the figure, all weights can be quantized offline first, and then during the inference, the input in float will be quantized to an 8-bit value before each operation like convolution, elementwise add and so on. After that, a re-quantization follows to ensure that the result of the operation is back to 8-bit again.

The quantization of a single value is equivalent to mapping the value in the range of float32 to the range of int8.

$$x_{int8} = round\left(x_{float32} \cdot \Delta_x\right) - z$$

$$\Delta_x = \frac{2^b - 1}{x_{max} - x_{min}}$$

Where Δ_x is the quantization scale, z is a constant value that makes float value 0 which is exactly represented by a quantized value (z == 0 when using symmetric quantization schema), b is the bit width (for example, b=8 here), x_{max} and x_{min} are the calibrated maximum and minimum values of x in the range of float.

Take convolution as an example (z is omitted for simplicity), convolution is one of the most frequently used operations in the deep learning model, so the quantization for it is crucial.

The following figure shows the pipeline.

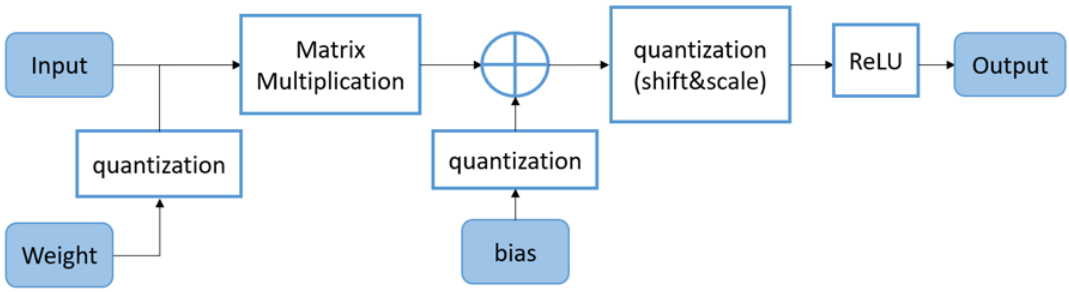

Figure 4-3 Quantization pipeline of convolution

The quantization of weight is the same as the fundamental value quantization:

$$\omega_q = round(\omega \cdot \Delta_\omega)$$

$$\Delta_{\omega}=\frac{2^{b-1}-1}{max(abs(\omega))}$$

The quantization of bias is a little different because there is an addition operation followed:

$$b_q = round(b \cdot \Delta_b)$$

$$\Delta_b=\Delta_{input}\Delta_{\omega}$$

Where Δ_{input} and Δ_{ω} are the quantization scales of input and weight.

The quantization after matrix multiplication and addition is a little bit complicated, which is called 'Feature map quantization'.

The maximum and minimum value of the feature map should be collected in the float pipeline in advance to get the quantization scale of the feature map.

$$\Delta_{map}=\frac{2^{\mathrm{b}}-1}{x_{max}-x_{min}}$$

The following shows the calculation of feature map quantization:

$$x_{out} = round(x_{in} \cdot \Delta_x)$$

$$\Delta_x=\frac{\Delta_{input}\cdot\Delta_{weight}}{\Delta_{map}}$$

Where Δ_{input} and Δ_{ω} are the quantization scales of input and weight.

Because division takes more time than multiplication, to optimize the process, the quantization step will be converted to a multiplier M. Furthermore, M can be torn down to a scale and a bit shift, as shown in the following equation:

$$M = \frac{\Delta_{map}}{\Delta_{input}\cdot\Delta_{weight}}$$

$$\mathrm{M} = 2^{-n}M_0$$

Eventually, the quantization is implemented as:

$$x_{out} = round(x_{in} \cdot 2^{-n}M_0)$$

Where n is the shift value and M_0 is the integer scale value.

For more information, see the *Zhouyi NPU Quantization Whitepaper*.

2.2 Quantization procedure

The quantization module in the NN compiler performs the following steps:

1. Receive the float *Internal Representation* (IR) and build the float graph.
2. Calibrate the feature map and the constant data (if any, like weight and bias) of each layer to get the range of the data.
3. Quantize each layer using the corresponding quantization algorithm.
4. Conduct similarity statistic between float and quantized feature map tensors, and dump all tensors (optional).
5. Use the metric plugin to metricize the float and quantized model.

Calibration methods

The calibration in the NN compiler supports two methods to get tensors statistic values. One is to use an existing statistic file, and the other is auto-statistic using calibration data and corresponding calibration parameters. The two methods are in order of priority.

- Use the statistic file which contains all the tensors statistic information.
  - To get this statistic file, you just use the auto-statistic method first. The tool will output this statistic file (a NumPy file named `modelname_statistic_info.npy`) if '`save_statistic_info=True`' is set in OPT.
  - This statistic file is grouped with dicts. The dicts are formatted as {node_name: {tensorname_port: {'extrema_min': min_v, …}}, for example:

    ```
    Statistic_file_dict={'data':{'data': {'extrema_min': -122.67890167236328,
    'extrema_max': 150.9929962158203, 'running_min': -122.67890167236328,
    'running_max': 150.9929962158203, 'running_mean': 4.95888049978091,
    'running_std': 69.66834421921259, 'running_histc': [8996.705, 641.7012,
    574.97363, ...,  278.732, 292.6069, 8994.181]}}}
    ```

  - If you want to customize the tensor range in a statistic file, you can run the auto-statistic tool first and use the output statistic file to customize the tensor range.

    As you see in the format, there are several keys. The following table shows a map between some calibration strategies and the keys. When you set one of the calibration strategies in the .cfg file, the corresponding statistic values of the tensor should be written in the statistic file.

    Table 4-1 Map between some calibration strategies and the keys

Calibration strategy	Statistic values of tensor
'extrema'	'extrema_min', 'extrema_max'
'mean'	'running_min', 'running_max'
'Nstd'	'running_mean', 'running_std'
'KLD'	'running_histc', 'running_min', 'running_max',

  - If you want to speed up the tool running time, you can use an existing statistic file for calibration, which saves the calibration time.
- Auto-statistic: Use the calibration dataset to get values of all the needed tensors in the NN compiler.
  - Select a subset of the training dataset as the calibration dataset, which tries to contain all the categories.

— Infer the model on the calibration dataset and statistic values of the output tensors. You can configure the calibration_data=, dataset= and calibration_batch_size= fields in the [Optimizer] part of the .cfg file to enable auto-statistic.

— The auto-statistic process can be determined using different calibration strategies on activation and weights. Usually the default strategy is general enough.

Quantization methods

The quantization in the NN compiler supports two methods to get quantization parameters for quantizing each operation. The quantization parameters (listed in the following table) include quantization strategy, and bits for weight/activation/bias.

Table 4-2 Quantization parameters and config fields

No.	Quantization parameters	Config field in the config file
1	bits for tensors	weight_bits bias_bits activation_bits lut_items_in_bits
2	strategy	per_tensor_symmetric_restricted_range per_tensor_symmetric_full_range per_tensor_asymmetric per_channel_symmetric_restricted_range per_channel_symmetric_full_range

After you run the module quantization, an `opt_template.json` file will be outputted. This file includes a detailed description of the quantization parameters of each layer. The following shows an example of a layer.

To use this JSON file, you can customize the quantization parameters of every node. For example, if you want weight_bits=4 bits of the 3rd node:

1. Find the 3rd node and modify `q_bits_weight` (4 in this JSON file).
2. Set the JSON file path to `opt_config` in the config file.
3. Run the tool again.

```
  "resnet_v1_50/predictions/Reshape": {
          "histc_bins": 2048,
          "lut_items_in_bits": 8,
          "q_bits_activation": 8,
          "q_bits_bias": 32,
          "q_bits_weight": 8,
          "q_mode_activation": "per_tensor_symmetric_full_range",
          "q_mode_weight": "per_tensor_symmetric_restricted_range",
          "q_strategy_activation": "mean",
          "q_strategy_weight": "extrema",
          "running_statistic_momentum": 0.9,
          "just_for_display": {
              "quantization_info": "{'resnet_v1_50/predictions/Reshape_0':
{'scale': '3.576941967010498', 'zerop': '0.0', 'qbits': '8', 'dtype': 'int8',
```

```
'qmin': '-128', 'qmax': '127', 'qinvariant': 'False', 'similarity':
0.9937090277671814}}",
            "optimization_info": "{}",
            "brief_info": "layer_id = 76, layer_type = OpType.Reshape,
similarity=0.9937090277671814, "
        }
    }
```

For details of all the settings, see *Using the NN compiler*.

2.3 Mixed precision quantization

To reduce the accuracy drop of quantization, you can configure the model to be quantized using mixed bits precision. For example, you can quantize some layers with 16-bit precision and the remaining layers with 8-bit precision to get a balance between accuracy and speed. The OPT offers two ways to configure mixed precision quantization:

- Automatically search. After the maximum allowed accuracy loss threshold is set, the OPT will automatically recommend which layers should be quantized to 8-bit, which layers should be quantized to 16-bit, and which layers should not be quantized when you trigger the 'mixed_precision_auto_search' option. For more information, see *Using the NN compiler*.

  For example, for tflite_mobilenet_v3 (tested on 5000 ImageNet pictures), when quantized to 8-bit, the top1 accuracy drop is 0.0184. If you want to narrow it down to 0.01, you can simply set 'mixed_precision_auto_search = 1,0.01,L' (use 1 batch data to quantize, and score(baseline model) - score(target model) <= 0.01) in the configuration file of OPT. After that, you get a mixed precision quantized model (quantized to 8-bit: layer 0 to 72, quantized to 16-bit: layer 73 to 146).
- Manually set. By configuring the JSON file that the OPT generates, you can freely set each layer's bits of weights, bias, and activation.

You can manually change the quantization parameters of some layers, which is based on an automatically searched result (because the built-in auto-search algorithm may not always find the best configuration due to time limits), or based on your own analysis (for example, by using the 'fake_quant_scopes_for_debug' and 'fake_quant_operators_for_debug' options, you can easily know which layers are more sensitive to quantization).

2.4 QAT model support

The NN compiler can automatically recognize a QAT model such as a quantized TFLite model or quantized ONNX model and then compile it to a Compass IR. When the input model is a QAT model, the compass IR generated by the Parser will have 'compat_quantized_model=true' in the IR header.

In the quantization process, the Optimizer module will call the QTLib submodule to implement quantization transformation when 'compat_quantized_model=true' is in the IR. The QTLib submodule is written by C++, and can be called by the Android or NN complier workflow. The quantization transformation has mainly two processes:

- Dequantize process:

  This process mainly handles the constants data. If the constants data needs to be dequantized to float data, the QTLib uses the following formula:

  $$\text{f\_data} = (\text{q\_data} - \text{qat\_zp}) * \text{qat\_scale}$$

  Where, `qat_scale` and `qat_zp` are the quantization parameters from the QAT model.

- Quantize process:

  The QTLib quantizes the float data to Compass quantized data using the following formula:

  $$\text{compass\_q\_data} = \text{f\_data} * \text{compass\_scale} - \text{copass\_zp}$$

  Where, `compass_scale` and `compass_zp` are the quantization parameters using the Compass quantization method.

  1. Calculate `compass_scale` and `compass_zp` of each tensor according to the Compass quantization formula:

     $$compass\_scale = \frac{1.0}{qat\_scale}$$
     $$compass\_zp = 0 - qat\_zp$$

  2. Quantize the constants data.
     - `weights`/`biases`: If `weights` and `biases` are float data, quantize these data to quantized data. When the input tensor is asymmetric quantization (input_zp≠0), the biases will absorb `input_zp` to `biases` data:

       $$biases = biases + \sum(weights * input\_zp)$$

     - Other constants data: These data will quantize using the Compass quantization method which can be found in the Compass Quantization Whitepaper.
  3. Generate the quantization parameters needed for the Compass IR (also can be found in the Compass Quantization Whitepaper). For example, the Tanh node will generate the LUT, and the convolution node will generate the scale and shift values.

The QTLib will perform the following Dequantize and Quantize processes.

- For weights and biases constant data:

  If the weights data is with per-tensor or per-channel symmetric quantization, the QTLib will keep these data unchanged, and use it directly in inference. If the weights data is with asymmetric quantization, the QTLib will dequantize these data to float data, and then quantize these float data

using the symmetric quantization method in the Quantize process. However, this dequantize-quantize procedure may cause an accuracy drop. If the accuracy drop is unacceptable, it is strongly recommended configuring ‘force_float_ir=true’ in the Parser section of the Compass cfg file to convert the QAT model to Compass float IR (‘compat_quantized_model=false’ in the IR header). The Optimizer uses the calibration data to quantize Compass float IR to Compass quantized IR. Normally the two quantization modes can satisfy the accuracy requirement.

- For other constants data like negative slope data:

  These constants data will be dequantized to float data using `qat_scale` and `qat_zp` from QAT quantization parameters.

Chapter 3
Auto-tiling

This chapter describes the tiling mechanism of the NN compiler.

It contains the following section:

3.1 Tiling procedure

Tiling can slice some layers in the model into multiple parallel sub-layers, which can improve the performance of the following two scenarios.

- Multi-core systems (for example, X2_1204MP3). The sliced parallel sub-layers can better support multi-core parallel computing.
- Systems with high-efficient memory (for example, DTCM on X1_1204 or GM on X2_1204). After slicing the tensor into smaller copies, the footprint can be significantly reduced by putting some inter-layer tensors completely into on-chip memory.

However, there are a large number of convolution-like layers in neural networks, which will generate an overlap after h/w directional slicing. If multiple convolution layers are cascaded, there will be a spread of overlap, leading to a significant increase in computation, which may even offset the performance gain from multiple cores. Therefore, it becomes a difficult problem to balance the overlap according to the size of on-chip memory, and the characteristics of the model.

The Zhouyi Compass build tool provides an auto-tiling scheme that can adaptively complete tiling based on the model structure, and the hardware information about the target (including on-chip memory size, number of hardware cores, and other information) to improve the system performance. To enable the auto-tiling feature, you need to set '`tiling = fps`' in the `aipubuild` GBuilder section, or '`--tiling fps`' in `aipugb`/`aipurun`. For more information about the configuration fields, see *4.5 Using the NN compiler*.

The following figure shows an example of auto-tiling.

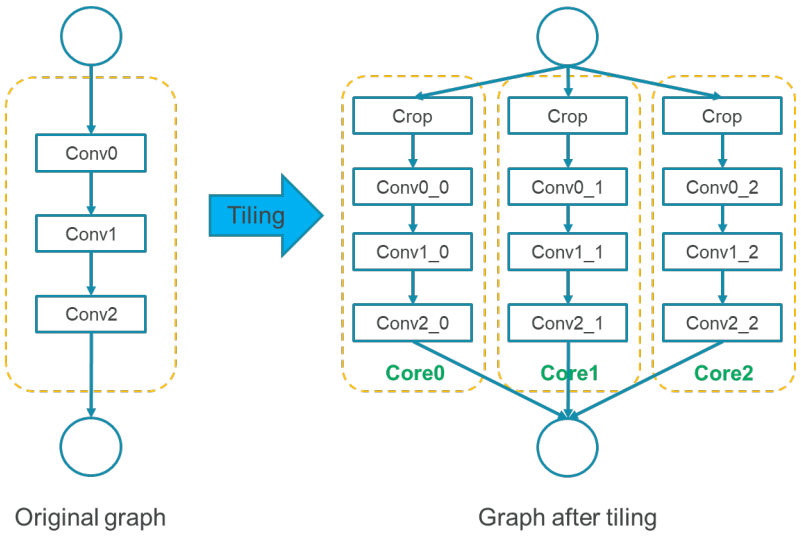

Figure 4-4 An example of auto-tiling

In addition to the auto-tiling method, Arm China also provides a manual tiling method which can support customized tiling parameters for each layer of a model. To enable the manual tiling, you can set the following four parameters in `aipubuild`/`aipuopt`:

- featuremap_tiling_param
- featuremap_splits_item_x
- featuremap_splits_item_y
- featuremap_splits_overlap_rate

For more information about the configuration fields, see *4.5 Using the NN compiler*.

Note that the manual tiling setting of a model requires large amount of analysis of each layer of a model, otherwise, the tiling may lead to a negative optimization.

Chapter 4
Using the NN compiler

This chapter describes the basic usage of the NN compiler.

It contains the following sections:

4.1 Installation

The NN compiler is packed as a Python wheel package, that is, WHL file. Therefore, you can just use pip to install it.

```
$pip install AIPUBuilder-xxx-xxx.whl
```

Note that:

- You need to use a compatible Python version to install WHL.
- You can use the `sudo` or add `--user/--target <dir>/--prefix <dir>` option when you do not have the root permission. Note that when you using the --target or --prefix option, PYTHONPATH needs to be set to install directory.
- The package will depend on TensorFlow, NumPy, NetworkX and other third-party packages. Ensure that the dependencies in `Out-Of-Box/out-of-box-nn-compiler/lib_dependency.txt` are met.

4.2 Usage

Before running the NN compiler tool, you should understand some environment variables for global options setting:

- AIPULIB_PATH: A list of paths separated by colons (:) for the NPU C library searching.
- AIPUPLUGIN_PATH: A list of paths separated by colons (:) for the NPU plugins searching.
- AIPUBUILDER_LOG: A positive number for setting the log level. By default, it is 2.
  - 1: Show all logs.
  - 2: Show WARN, INFO and ERROR logs.
  - 3: Show INFO ERROR logs.
  - 6: Show only ERROR logs.
  - 10: No logs.
- ENABLE_TCM: Enables/Disables *Data Tightly Coupled Memory* (DTCM) for Zhouyi X1 series when set to 1/0.

  The Zhouyi X1 series supports DTCM. Therefore, for Zhouyi X1 series, you can use DTCM options to configure the DTCM. The difference between the DTCM and external SRAM is that the DTCM has a fixed address for the NPU, while the external SRAM has no fixed address for the NPU. You should ensure that the DTCM is fully occupied by the NPU and other devices will not change the data during NPU running. For more information about the DTCM, see *A3.1 Memory space* of the *Arm China Zhouyi NPU X1 Technical Reference Manual*.

Use the following commands to set the environment variables:

- For `bash`: `$export ENV_VAL=…`
- For `csh`: `$setenv ENV_VAL …`

After installation, the NN compiler provides several command line tools—`aipubuild`, `aipuparse`, `aipuopt`, `aipurun`, and `aipugb`.

aipubuild

The `aipubuild` is the entry point of the package. You can use `-h` or `--help` to get the help information. The tool uses a configuration file for its parameters, in the config file. You need to specify all the related parameters to run it.

The following is a simple example:

```
$aipubuild test.cfg
```

The configuration file is a standard `ini` configuration file. It consists of four sections:

- Common
- Parser
- Optimizer
- GBuilder

Common section

The common section just contains one key mode, which will tell `aipubuild` to run in which mode. This key is on run mode by default, and there are just two modes available:

- `run`: Run the third-party model on the NPU simulator.
- `build`: Build the third-party model to the NPU executable file.

Parser section

The section is for the model parse module:

- model_type [**optional**]

  The model format of the input model. By default, it is TensorFlow. The supported types are:

  — TensorFlow
  — TFLite
  — ONNX
  — Caffe

- model_name [**required**]

  The name for the input model, must be a string. It is used to identify the model.

- model_domain [**required**]

  The domain of the model, there are only 5 available options:

  — image_classification
  — object_detection
  — keyword_spotting
  — speech_recognition
  — image_segmentation

  detection_postprocess is required when model_domain is object_detection.

- detection_postprocess [**optional**]

  If your model_domain is object_detection, and if you are using the official detection model, you need to specify your detection post process. Currently, only the following types of post process are supported:

  — SSD
  — SSD_RESNET
  — YOLO2
  — YOLO3_TINY
  — YOLO3_FULL
  — CAFFE_FASTERRCNN

- input_model [**required**]

  File path of the input third-party model. Currently, it supports TensorFlow frozen pb/h5/SaveModel (the input will be a folder)/tflite/onnx/caffemodel.

- caffe_prototxt **[optional]**

  Prototxt file path of the Caffe model.

- input [**optional**]

  The input node name of the model. If you have several inputs, use commas to separate them.

  The input tensor name is experimentally supported for this field. If you do not know the node name or the node name is empty, you can try to use the tensor name here. It should be ‘required’ if the input node information is not included in the model.

- output [**optional**]

  The output node name of the model. If you have several outputs, use commas to separate them.

  The output tensor name is supported for this field. If you do not know the node name or the node name is empty, you can try to use the tensor name here. It should be ‘required’ if the output node information is not included in the model.

- input_shape [**optional**]

The input shape of the model. Usually it is a single tensor shape, for example, input_shape =[1,224,224,3].

If you have several inputs, use commas to separate them, for example, input_shape=[1,224,224,3],[1,112,112,3].

If this field is present, the 'input' field must also be present, and the number of input shapes must be equal to the number of inputs.

- force_float_ir[optional]

  If this field is set to `true` in configuration, the Parser will convert the quantized op in the QAT model into float op and the output IR will be a float IR.

  For example, `force_float_ir=true`.

  If this field is set to `false` in configuration or is not present, the Parser will keep the quantized op in the model if it exists and the output IR will be a quantized IR or a float IR according to different models. For example, `force_float_ir=false`.

- iou_threshold **[optional]**

  Overlap threshold value between two boxes.

- obj_threshold **[optional]**

  Confidence threshold value of the current box.

- max_box_num **[optional]**

  Maximum box number of some detection layers (such as Region/DecodeBox).

- image_width **[optional]**

  Input image width.

- image_height **[optional]**

  Input image height.

- preprocess_type [**optional**]

  The pre-processing (running on the NPU) to be applied on the model inputs. Currently it supports RESIZE, RGB2BGR, BGR2RGB, RGB2GRAY, BGR2GRAY, and STANDARDIZATION. When several pre-processing types are set, they will run on the NPU sequentially.

- preprocess_params [**optional**]

  The parameters needed in the pre-processing. The parameters of each pre-processing should be set with a brace. When the pre-processing does not need parameters, an empty brace {} should be used.

  For RGB2BGR, BGR2RGB, RGB2GRAY, and BGR2GRAY, there is no need of parameters.

  For RESIZE, it needs two parameters. One is the resize method, which can be 'bilinear' or 'nearest'. The other is the input image shape. The destination shape is the original model shape that you do not need to set. For example, the setting of a bilinear RESIZE, with an input image shape [1,448,448,3], should be {bilinear,[1,448,448,3]}.

  For STANDARDIZATION, it needs one parameter—axis. For the axis setting, it can be any combination of different dimensions. For example, for a 3-dimensional input, {[0]} means that only the dimension-0 axis is processed. {[1,2]} means that both the dimension-1 axis and dimension-2 axis are processed. {[0,1,2]} means that all the three dimension axis are processed. When the setting is empty, such as {} or {[]}, it also means that all the axis are processed.

  Therefore, for an input that applies the RESIZE, then RGB2BGR, and then STANDARDIZATION pre-processing, the preprocess_type should be 'RESIZE, RGB2BGR, STANDARDIZATION. The preprocess_params should be something like '{bilinear,[1,448,448,3]}, {}, {[1,2]}'.

Optimizer section

The section is for the quantization module:

- dataset [**optional**]

  A dataset plugin name used to create a dataset to handle the data and label. It is recommended that you create a customized dataset plugin for your own particular model.

- calibration_data [**optional**]

  A dataset path for calibrating a model.

- calibration_batch_size [**optional**]

  The batch_size of inferencing the graph in quantization calibration. It may affect the quantization accuracy if the selected calibration strategy uses statistic values that depend on the batch_size.

- calibration_strategy_for_activation [**optional**]

  The activation calibration strategy. Currently, it supports ‘mean’, ‘extrema’, ‘Nstd’, ‘KLD’, and other strategies. It defaults to ‘mean’.

- calibration_strategy_for_weigth [**optional**]

  The weight calibration strategy. Currently, it supports ‘extrema’, ‘Nstd’, ‘KLD’, and other strategies. It defaults to ‘extrema’.

- weight_bits [**optional**]

  Weight bits for quantizing weight data. It supports 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and16. It defaults to 8. Currently only 8 and 16 are supported in Zhouyi Compass Operators. See the *Zhouyi Compass Operators Specification Application Note* for more information.

- bias_bits [**optional**]

  Bias bits for quantizing bias data. It supports 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47 and 48. It defaults to 32.

- activation_bits [**optional**]

  Activation bits for quantizing activation data. It supports 8, 9, 10, 11, 12, 13, 14, 15 and 16. It defaults to 8. Currently only 8 and 16 are supported in Zhouyi Compass Operators. See the *Zhouyi Compass Operators Specification Application Note* for more information.

- running_statistic_momentum [**optional**]

  Momentum (range[0.0, 1.0]) used for calculating weighted average of some statistics when the calibration dataset has multiple batches, like min_value = momentum * pre_min_value + (1-momentum) * cur_min_value. It defaults to 0.9.

  When tuning a model, you can change it by using the calibration strategies such as ‘mean’.

- histc_bins [**optional**]

  Bins for statistic histograms of each tensor.

  When tuning a model, you can change it by using the calibration strategies such as ‘KLD’.

- trim_infinity_before_statistic [**optional**]

  Exclude the infinite or equivalent very large/small values from statistic if they affect quantization.

  For uniform configuration, it must be ‘(min_value, max_value):method’.

  For specific configuration, it must be ‘[operator_type:(min_value, max_value):method, ...]’ or ‘[(i, j):(min_value, max_value):method, ...]’.

  Where values <= min_value or >= max_value will be treated as infinite values (should be min_value <= 0 <= max_value). The ‘method’ decides how to deal with infinite values and currently it supports ‘clip’ (infinite values will be clamped into [min_value, max_value]) and ‘second’ (infinite values will

be replaced by min/max values after excluding infinite values). The ‘operator_type’ is the valid operator type name. The ‘I’ and ‘j’ stand for ‘layer_id’ in the input IR.

Valid examples: '(-inf, inf):second', '(-65536, 65535):clip', '[Abs:(0, inf):second, BatchNorm:(-32767, 65535):clip]', '[(1,10):(-inf,inf):second, (20,30):(-32767,32767):clip]'.

This field is disabled by default.

- save_statistic_info [**optional**]

  Indicates whether to save and dump the statistic information file. If set to False, it will further adjust statistic information which is necessary for the corresponding calibration strategy for time saving. The default value is False.

- quantize_method_for_weight [**optional**]

  Quantization method for weight and bias. It can be set to ‘per_tensor_symmetric_restricted_range’, ‘per_tensor_symmetric_full_range’, ‘per_channel_symmetric_restricted_range’ and ‘per_channle_symmetric_full_range’. It defaults to ‘per_tensor_symmetric_ restricted_range’.

- quantize_method_for_activation [**optional**]

  Quantization method for weight and bias. It can be set to ‘per_tensor_symmetric_restricted_range’, ‘per_tensor_symmetric_full_range’ and ‘per_tensor_asymmetric’. It defaults to ‘per_tensor_symmetric_full_range’. Currently only some of the Zhouyi Compass Operators support asymmetric quantization. See the *Zhouyi Compass Operators Specification Application Note* for more information.

- statistic_file [**optional**]

  A statistic file path. The statistic file has the statistic feature map and weight range. If using auto-statistic, this field will not be used.

- quant_ir_name [**optional**]

  The quantized IR file name. If setting ‘resnet_50_quant’, tools will output the quantized IR: `resnet_50_quant.txt` and `resnet_50_quant.bin`.

- output_dir [**optional**]

  The output directory for quantized IR, calibration statistic file and quantization configuration JSON file. If it is not set, the quantized IR will be outputted in ‘./’.

- metric [**optional**]

  The metric plugin name for metricizing a model. Currently, the optimizer supports 'cosdistancemetric', 'topkmetric', 'wermetric', and other plugins. It is recommended that you create a customized metric plugin for your own particular model.

- data [**optional**]

  A dataset path for metricizing a model.

- label [**optional**]

  A label path for metricizing a model.

- metric_batch_size [**optional**]

  Batch size when metricizing a model. It is irrelevant to the quantization accuracy.

- unify_scales_for_concat [**optional**]

  Indicates whether to unify scales of each concat's branch when possible.

- unify_scales_for_concat_max_depth [**optional**]

  When enabling unify_scales_for_concat(=True), the max search depth can be settled for speeding up. The default value is 20.

- with_winograd [**optional**]

Indicates whether to enable Winograd algorithm when possible.

- opt_config [**optional**]

  A file path. This JSON file stores quantization configurations of each node.

- dump [**optional**]

  Indicates whether to enable the feature of dumping all tensors and other data.

- dump_dir [**optional**]

  A path. All dump data files will save to this path when 'dump=true’.

- dataloader_workers [**optional**]

  Number of workers for DataLoader. You should set it to 0 when debugging to exclude multi-thread influence.

- resize_degrade_to_nearest [**optional**]

  Indicates whether to degrade the resize method to the nearest neighbor to speed up resizing.

- bias_effective_bits [**optional**]

  The effective high bits for bias data which really takes part in computation (lower bits will be set to 0), due to hardware restriction (for a 32-bit platform, it is less than 32. It is usually 16 in the Zhouyi NPU).

- fake_quant_scopes_for_debug [**optional**, and only for debug usage]

  It will run float forward in related layers in assigned scopes when calling quantization forward for debug usage. It must be a list of unsigned integer (corresponding to layer_id in a float IR) tuples like '[(i,j)]' or '[(i,j),(n,m),...,(k,l)].

- fake_quant_operators_for_debug [**optional**, and only for debug usage]

  It will run float forward in related layers in assigned operators when calling quantization forward for debug usage. It must be a list of valid operator type names (case insensitive, corresponding to layer_type in a float IR), for example, ‘Abs,reshape,tile’.

- mixed_precision_auto_search [**optional**]

  If the maximum allowed accuracy loss threshold is set, it automatically recommends which layers should be quantized to 8-bit, which layers should be quantized to 16-bit, and which layers should not be quantized. It must be set as 'batches,threshold,greater_equal_or_less_equal_than_baseline' (batches < 1 means disabled), for example, '1,0.02,L' means using one batch of data to quantize, with score(baseline model) - score(target model) <= 0.02) or '1,-0.02,G' means using one batch of data to quantize, with score(baseline model) - score(target model) >= -0.02). The default value is '0,0.,L'.

- cast_dtypes_for_lib [**optional**]

  Indicates whether to cast dtypes of operators to adapt to the specification of the operator lib's dtypes (may cause model accuracy loss due to the corresponding specification restriction). 'False' means no. 'True' means yes for all operators. It includes a list of valid operator type names (case insensitive, corresponding to layer_type in float IR), for example, ‘Abs,reshape,tile’ means yes for all the specified operators.

- without_batch_dim [**optional**]

  Indicates whether the model has a batch dim. Most models do have a batch dim. However, if this field is set to True (means that the model does not have a valid batch dim), you should implement a customized dataset plugin to transform the original data layout (determined by the input IR) to the format required by the Torch Dataset (Torch always assumes that the input data has a batch dim). The default value is False.

- featuremap_tiling_param [**optional**]

  The featuremap_tiling_param is used to assign tiling configurations for specific layers (corresponding to layer_id in float IR): '[(i,j,h,w)]' or '[(i,j,h,w),...,(k,l,x,y)]', where 'i,j' and 'k,l' mean that the

corresponding layers in the range will be tiled. 'h,w' and 'x,y' mean the number of parts that the feature map will be split into in the corresponding dimension.

- featuremap_splits_item_x [**optional**]

  Item number for feature map data parallel in the x dimension (that is the w dimension).

- featuremap_splits_item_y [**optional**]

  Item number for feature map data parallel in the y dimension (that is the h dimension).

- featuremap_splits_overlap_rate [**optional**]

  Maximum allowed overlap rate for feature map data parallel. The value should be in the range [0, 100). The default value is 50.

GBuilder section

This section describes some options for controlling the GBuilder module behavior.

- simulator [**optional**]

  The path of the simulator when the mode is `run`. If you leave it empty, it will use the last run's simulator path.

- inputs [**required** in run mode]

  The input file. It is only used when the mode is `run`. The input file is for running on the simulator. Input files are usually some binary files, for example, a raw image, or a raw tensor.

  If you have several inputs, use commas to separate them.

- local_lib [**optional**]

  The path of the local lib if you want to use a customized lib. Usually, you can leave it blank.

- target [**optional**]

  The target hardware that you want to build/run. By default, it is X2_1204. It supports all the Zhouyi series platforms. Currently it supports Z2_0901, Z2_1002, Z2_1004, Z3_0901, Z3_1104, X1_1204, X2_1204, and X2_1204MP3.

- profile [**optional**]

  Define whether to enable the profiler. For the usage of the profiler, see *Chapter 6 Profiling network*. When set to True, some profiling data will be created in the running path if the related simulator supports the profiling feature. With these data, you can create a simulator profiler report with the aipu_simulator_profiler. For more information about the profiling feature of the simulator, see *6.1 About the NPU simulator* in the *Arm China Zhouyi Compass Software Technical Overview*.

- prof_unit [**optional**]

  Define which hardware unit the profiler will profile. Currently, it supports TPC, DMA and AIFF. For the detailed setting rules, see *6.1 Overview*.

- fast_perf [**optional**]

  Disable the DMA/AIFF profile, which may cost much extra time/cycle. Only for Zhouyi X2.

- max_sample_point_num [**optional**]:

  Maximum sample point number of the profiler, only for Zhouyi X2.

- max_aiff_sample_point_num [**optional**]

  Maximum AIFF sample point number of the profiler, only for Zhouyi X2.

- max_dma_sample_point_num [**optional**]

  Maximum DMA sample point number of the profiler, only for Zhouyi X2.

- max_custom_sample_point_num [**optional**]

Maximum custom sample point number of the profiler, only for Zhouyi X2.

- stack_size [**optional**]

  The stack_size (KB) for the NPU. The default value is 257 (KB).

- png [**optional**]

  A file name with suffix .png for visualization of the graph in PNG format. To generate the PNG image, you must install the graphviz package. It is recommended to use SVG which is faster and smaller.

- svg [**optional**]

  A file name with suffix .svg for visualization of the graph in SVG format. To generate the SVG image, you must install the graphviz package.

- tcm_size [**optional**]

  The DTCM size of the X1_1204 SoC platform. By default, it is 4MB. It also can be a value in kilobytes.

- gm_size [**optional**]

  The Global Memory size of Zhouyi X2 or later SoC platforms. It accepts a number in Kbytes. The default value will be the same as the hardware configuration, that is, 4MB for X2_1204MP3 and 1MB for X2_1204.

- tiling [**optional**]

  Select which mode of auto-tiling to apply. Currently, only `fps` mode is supported, and the supported platforms are Zhouyi X1 and Zhouyi X2. In this mode, GBuilder will apply the auto-tiling module to tile the entire graph and try to get better FPS.

- ocm_size [**optional**]

  Set the SoC SRAM size or *On-Chip Memory* (OCM) size of the Zhouyi X2 SoC platform. The allocator will try to use these memory. It may cause slightly higher memory consumption, and there is no guarantee that the OCM or SoC SRAM will be used at runtime. The default value is 0. The value can be a number in KBytes.

- ocm_addr [**optional**]

  Set the OCM address. Only for `run` mode.

Example configuration file

The following is an example file.

```
[Common]
mode=run

[Parser]
model_name=resnet_50
detection_postprocess=
model_domain=image_classification
output=resnet_v1_50/predictions/Reshape
input_model=./resnet_50/frozen.pb
input=Placeholder
input_shape=[1,224,224,3]
preprocess_type=RESIZE, RGB2BGR
preprocess_params={bilinear,[1,448,448,3]},{}
```

```
[Optimizer]
model_name=resnet_50
calibration_data=./Dataset/ImageNet/preprocess/resnet/calibration_100.npy
data=./Dataset/ImageNet/preprocess/resnet/dataset_1000.npy
label=./Dataset/ImageNet/preprocess/resnet/label_1000.npy
calibration_batch_size=20
metric_batch_size=50
dataset=NumpyDataset
metric=TopKMetric

[GBuilder]
simulator=./aipu_simulator
target=Z2_1104
inputs=./resnet_50/input.bin
outputs=./resnet_50/output_resnet_50.bin
profile=True
prof_unit=DMA
tiling=fps
```

aipuparse

The `aipuparse` is the parser module of `aipubuild`. It converts a model from deep learning framework into IR.

Usage:

```
$aipuparse -c net.cfg
```

The configuration file is a standard `ini` configuration file. It only consists of the Common section.

Common section

All keys in the `aipubuild` Parser section are keys of the `aipuparse` Common section. In addition, there is one more key:

- `output_dir`: The output directory for storing the output IR. The output IR has two files <model_name>.txt and <model_name>.bin for graph IR and weights data.

The following is an example configuration file for `aipuparse`:

```
[Common]
model_type=tensorflow
model_name=alexnet
model_domain=image_classification
input_model=alexnet.pb
input=Placeholder
input_shape=[1, 28, 28, 1]
```

```
output=alexnet_v2/fc8/squeezed
output_dir=./IR_float
```

aipuopt

The `aipuopt` is the quantization module of `aipubuild`. It converts a float IR to a quantized IR.

Usage:

```
$ aipuopt -c net.cfg
```

The configuration file is a standard `ini` configuration file. It only consists of the Common section.

Common section

All keys in the `aipubuild` Optimizer section are keys of the `aipuopt` Common section. In addition, there are five more keys:

- `output_dir`: The output directory for storing the output IR. The output IR has two files <*model_name*>.txt and <*model_name*>.bin for graph IR and weights data.
- `graph`: The graph IR file path, for example, the txt file of the output of `aipuparse`.
- `bin`: The IR weights file path, for example, the bin file of the output of `aipuparse`.
- `model_name`: The name of the model.
- `quant_ir_name`[**optional**]: The quant IR name. The output file is named with this value.

The following is an example configuration file for `aipuparse`:

```
[Common]
graph=alexnet_float32.txt
bin=alexnet_float32.bin
model_name=alexnet
dataset=NumpyDataset
calibration_data=./caffe_alexnet/dataset_200.npy
calibration_batch_size=20
metric_batch_size=100
data=./caffe_alexnet/dataset_5000.npy
label=./caffe_alexnet/label_5000.npy
metric=TopKMetric
output_dir=./IR_int
dataloader_workers=4
```

aipurun

The `aipurun` is a tool that help you run a quantized IR in simulator. You can use `aipurun` without a real device to test the model. You can use `-h` or `--help` to get the help information.

Usage:

```
$aipurun ir_path [-w weight_path]  [-i inputs] [other options]
```

The `aipurun` accepts the following options:

- `-v, --version` [**optional**]: Print the tool version and plugin versions and exit.
- `-w, --weight` [**optional**]: The weight file path for the quantized IR.

- `-i, --input <InputFiles>` [**required**]: The input files for running the model. If you have several inputs, separate the file names with commas. The order of the inputs file must be the same as the IR described in global sections with key 'input_tensors'.
- `-o, --output <OutputFileName>` [**optional**]: By default, it is `output.bin`. If you have several outputs, the file will add a suffix 1, 2, … for the second, third, … outputs.
- `-h, --help` [**optional**]: Print the help message and exit.
- `--dump_all_tensors` [**optional**]: Dump all tensors (including output tensors of graph) except inputs.
- `--dump_tensors <layer_ids>` [**optional**]: Dump output tensors of specified layers (layer IDs). By default, it is empty. If you have several layers, separate the layer IDs with commas. Internal tensors of subgraph and input/output tensors of graph will be skipped.
- `--target <TargetName>` [**optional**]: The target to be built. By default, it is X2_1204. The supported targets include Z2_0901, Z2_1002, Z2_1004, Z2_1104, Z3_0901, Z3_1104, X1_1204, X2_1204, and X2_1204MP3.
- `--png <ImageName>` [**optional**]: Output a PNG format for the graph structure. The feature depends on graphviz. Ensure that the dot command of graphviz is available.
- `--svg <ImageName>` [**optional**]: Output an SVG format for the graph structure. The feature depends on graphviz. Ensure that the dot command of graphviz is available.
- `-L, --lib <LibPath>` [**optional**]: The lib search path for searching for the using operator libs.
- `-p, --profile` [**optional**]: Enable the profile counter.
- `--prof_unit <ProfUnit>` [**optional**]: Select the hardware unit to profile. Currently, it supports TPC, DMA and AIFF. For the detailed setting rules, see *6.1 Overview*.
- `--fast_perf` [**optional**]: Disable the DMA/AIFF profile, which may cost much extra time. Only for Zhouyi X2.
- `--max_sample_point_num <MAX_SAMPLE_POINT_NUM>` [**optional**]: Maximum sample point number of the profiler, only for Zhouyi X2.
- `--max_aiff_sample_point_num <MAX_AIFF_SAMPLE_POINT_NUM>` [**optional**]: Maximum AIFF sample point number of the profiler, only for Zhouyi X2.
- `--max_dma_sample_point_num <MAX_DMA_SAMPLE_POINT_NUM>` [**optional**]: Maximum DMA sample point number of the profiler, only for Zhouyi X2.
- `--max_custom_sample_point_num <MAX_CUSTOM_SAMPLE_POINT_NUM>`[**optional**]: Maximum custom sample point number of the profiler, only for Zhouyi X2.
- `--stack_size <StackSize>` [**optional**]: The stack_size (KB) for the NPU. By default, it is 257 (KB).
- `--simulator <SimualtorPath>` [**optional**]: The simulator executable path.
- `--tcm_size <TCM_SIZE>` [**optional**]: The DTCM size of the X1_1204 SoC platform. By default, it is 4MB. It also can be a value in kilobytes.
- `--gm_size <GM_SIZE>` [**optional**]: The Global Memory size of Zhouyi X2 or later SoC platforms. It accepts a number in Kbytes. The default value will be the same as the hardware configuration, that is, 4MB for X2_1204MP3 and 1MB for X2_1204.
- `--tiling <auto-tiling mode>` [**optional**]: Select which mode of auto-tiling to apply. Currently, only '`fps`' mode is supported, and the supported platforms are Zhouyi X1 and Zhouyi X2. In this mode, GBuilder will apply the auto-tiling module to tile the entire graph and try to get better FPS.
- `--ocm_size <OCM_SIZE>` [**optional**]: Set the SoC SRAM size or *On-Chip Memory* (OCM) size. The allocator will try to use these memory. It may cause slightly higher memory consumption, and there is no guarantee that the OCM or SoC SRAM will be used at runtime. The default value is 0. The value can be a number in KBytes.
- `--ocm_addr <OCM_ADDR>` [**optional**]: Set the OCM address.

aipugb

The `aipugb` is a tool which help you build the IR as an `aipu.bin` for executing on a real device with runtime. You can use `-h` or `--help` to get the help information. The tool shares many options with `aipurun`.

Usage:

```
$aipugb ir_path [-w weight_path]  [other options]
```

The `aipugb` accepts below options:

- `-v, --version` [**optional**]: Print the tool version and plugin versions and exit.
- `-w, --weight` [**optional**]: The weight file path for the quantized IR.
- `-o, --output <AipuBinFileName>` [**optional**]: By default, it is `aipu.bin`.
- `-h, --help` [**optional**]: Print the help message and exit.
- `--dump_all_tensors` [**optional**]: Dump all tensors (including output tensors of graph) except inputs.
- `--dump_tensors <layer_ids>` [**optional**]: Dump output tensors of specified layers (layer IDs). By default, it is empty. If you have several layers, separate the layer IDs with commas. Internal tensors of subgraph and input/output tensors of graph will be skipped.
- `--target <TargetName>` [**optional**]: The target to be built. By default, it is X2_1204. The supported targets include Z2_0901, Z2_1002, Z2_1004, Z2_1104, Z3_0901, Z3_1104, X1_1204, X2_1204, and X2_1204MP3.
- `--png <ImageName>` [**optional**]: Output a PNG format for the graph structure. The feature depends on graphviz. Ensure that the dot command of graphviz is available.
- `--svg <ImageName>` [**optional**]: Output an SVG format for the graph structure. The feature depends on graphviz. Ensure that the dot command of graphviz is available.
- `-L, --lib <LibPath>` [**optional**]: The lib search path for searching for the using operator libs.
- `-p, --profile` [**optional**]: Enable the profile counter.
- `--prof_unit <ProfUnit>` [**optional**]: Select the hardware unit to profile. Currently, it supports TPC, DMA and AIFF. For the detailed setting rules, see *6.1 Overview*.
- --fast_perf [**optional**]: Disable the DMA/AIFF profile, which may cost much extra time. Only for Zhouyi X2.
- `--max_sample_point_num <MAX_SAMPLE_POINT_NUM>` [**optional**]: Maximum sample point number of the profiler, only for Zhouyi X2.
- `--max_aiff_sample_point_num <MAX_AIFF_SAMPLE_POINT_NUM>` [**optional**]: Maximum AIFF sample point number of the profiler, only for Zhouyi X2.
- `--max_dma_sample_point_num <MAX_DMA_SAMPLE_POINT_NUM>` [**optional**]: Maximum DMA sample point number of the profiler, only for Zhouyi X2.
- `--max_custom_sample_point_num <MAX_CUSTOM_SAMPLE_POINT_NUM>`[**optional**]: Maximum custom sample point number of the profiler, only for Zhouyi X2.
- `--stack_size <StackSize>` [**optional**]: The stack_size (KB) for the NPU. By default, it is 257 (KB).
- `--tcm_size <TCM_SIZE>` [**optional**]: The DTCM size of the X1_1204 SoC platform. By default, it is 4MB. It also can be a value in kilobytes.
- `--gm_size <GM_SIZE>` [**optional**]: The Global Memory size of Zhouyi X2 or later SoC platforms. It accepts a number in Kbytes. The default value will be the same as the hardware configuration, that is 4MB for X2_1204MP3 and 1MB for X2_1204.
- `--tiling <auto-tiling mode>` [**optional**]: Select which mode of auto-tiling to apply. Currently, only '`fps`' mode is supported, and the supported platforms are Zhouyi X1 and Zhouyi X2. In this mode, GBuilder will apply the auto-tiling module to tile the entire graph and try to get better FPS.

- `--ocm_size <OCM_SIZE>` [**optional**]: Set the SoC SRAM size or *On-Chip Memory* (OCM) size. The allocator will try to use these memory. It may cause slightly higher memory consumption, and there is no guarantee that the OCM or SoC SRAM will used at runtime. The default value is 0. The value can be a number in Kbytes.

Chapter 5
IR definition

This chapter describes the IR definition and the relationship with the NN compiler .

It contains the following sections:

5.1 Introduction

An *Intermediate Representation* (IR) is the representation of network in various deep learning frameworks, which describes the flow of data from the network input data to inference results.

An IR is an important middleware which runs through the whole NN compiler workflow:

1. The parser tool in the NN compiler can parse a third-party model and convert it to a standard float (data type) IR.
2. The quantization and optimization tool in the NN compiler will try to quantize the float IR into 8-bit/16-bit IR (currently only int8/uint8/int16/uint16 operators are supported), and then transfer the 8-bit/16-bit IR to the GBuilder.
3. The GBuilder will optimize the execution sequence, lower the 8-bit/16-bit model to fit NPU instructions or acceleration operation, call the build-in operator library, and build the model to an executable file for the NPU or simulator in the end.

5.2 IR format

The NPU IR consists of two files normally:

- A .txt (text) file, called the IR definition, which represents the model architecture.
- A .bin (binary) file, which stores the model raw parameter data (with float data or int8/uint8/int16/uint16 data), such as the weights data, and the bias data. All the different data are stacked one by one (without specific order) and can be located by the *_offset (which indicates the beginning position) and the *_size (which indicates the data size).

Generally, the IR definition means the .txt file. The IR definition is divided two parts. One is the common information and the other specifies all the necessary layer parameters.

- Common parameters—Define some basic parameters to describe the NN model, which include the model name, layer numbers, precision, ….
- Layer parameters—Give a simple introduction to the operator or neural network (fusion) layer and define the input tensor (or 'layer_bottom') and output tensor (or 'layer_top'). Then, the detailed attributes will be specified, which are used for operator library computing.

For, the detailed IR definition and build-in example, see the *Arm China Zhouyi Compass IR Definition Application Note*. The following is an example of int8 IR definition:

```
model_name=argminmax
layer_number=2
input_tensors=[featuremap_in]
output_tensors=[argminmax_0]
precision=int8

layer_id=0
layer_name=featuremap_in
layer_type=Input
layer_bottom=
layer_bottom_shape=
layer_bottom_type=
layer_top=[feature_in]
layer_top_shape=[[1,25,16,64]]
layer_top_type=[int8]

layer_id=1
layer_name=argminmax_0
layer_type=ArgMinMax
layer_bottom=[feature_in]
layer_bottom_shape=[[1,25,16,64]]
layer_bottom_type=[int8]
layer_top=[argminmax_0]
```

```
layer_top_shape=[[1,25,16,1]]
layer_top_type=[uint16]
method=MAX
axis=3
select_last_index=True
```

In this IR, `layer_id=0` means the input tensor. `layer_id=1` means the ArgMaxMin operator which can be identified by the `layer_type` and `method` attribution. Where `method=MAX` means returning the index of the maximum element (Argmax), and when it is modified to `MIN`, the operation is transferred to Argmin.

When running the NN compiler with this int8 IR, it performs an ArgMaxMin operation on the input activation and returns the corresponding index as the result of the output tensor.

These parameters or attribution can be simply set to different values to evaluate performance of the model according to the *Arm China Zhouyi Compass IR Definition Application Note.*

A module is provided to check IR format according to the IR definition. This module will be called as a function by GBuilder by default.

The IR checker includes two parts:

- Checker in graph level—is pre-implemented and not customizable.
- Checker in operator level—supports user-defined implementation by the checker plugin.

The checker plugin is optional, and the checker in operator level will be skipped if there is no plugin for this operator. For more information about the checker plugin, see *4.7.5 Checker plugin*.

Chapter 6
Profiling network

This chapter shows how to profile a network with the NN compiler .

It contains the following sections:

6.1 Overview

When running on the NPU, a profiler tool is provided to get the network performance information.

To enable the profiling function in the `aipu.bin` (the executable bin file of the NPU), you only need to enable the profile in the GBuilder section of the configuration file as described in *4.5.2 Usage*.

After the profile is enabled in the Generation module, the profiler instructions are automatically inserted in the `aipu.bin`. Also, a file named `graph.json` is generated additionally, which records some node information for the use of the generated final profiler report.

Using the `aipu.bin` generated by the NN compiler to enable the profile, the Linux runtime will generate an additional output buffer, which is the profiler result recorded when running the network. You may write this buffer into a file and output it for further profiling jobs.

The profiler is at layer granularity, which means that the profiler will measure the performance of a network layer by layer.

Be cautious that the profiler depends on hardware counters, so you may get all zero profiling result when you enable the profiler but run it on the simulator.

To generate a human-readable report, a tool named `aipu_profiler` is provided. This tool consumes the `graph.json` file and the additionally generated binary buffer file, and then produces a profiler report which is an HTML file. You can open and read it with a browser.

Because different Zhouyi products have different number of performance counters (for example, the Zhouyi Z2/Z3 series has four performance counters, while the Zhouyi X1/X2 series has eight), the setting options will have different functions on different products. The following table shows the details.

Table 4-3 Details of setting options

Setting option	Product series	Profile information
Default	Zhouyi Z2/Z3	AIPU total cycle Cost time TPC busy cycle and utilization ratio MTP0 busy cycle and utilization ratio MTP1 busy cycle and utilization ratio ITP busy cycle and utilization ratio PTP busy cycle and utilization ratio AIPU read bytes AIPU write bytes
	Zhouyi X1	AIPU total cycle Cost time TPC busy cycle and utilization ratio AIFF busy cycle and utilization ratio DMA busy cycle and utilization ratio MTP0 busy cycle and utilization ratio MTP1 busy cycle and utilization ratio ITP busy cycle and utilization ratio PTP busy cycle and utilization ratio DMA read bytes DMA write bytes AIPU read bytes AIPU write bytes AIFF weight read wait cycle

Setting option	Product series	Profile information
		AIFF input read wait cycle AIFF output write wait cycle
	Zhouyi X2	Core ID: Indicates which physical core the subgraph is running on. Level: Subgraphs which have the same level can be run in parallel topologically. Cycles: Cycles that count this layer cost. AIFF busy cycle and utilization ratio AIPU read bytes AIPU write bytes MTP0 busy cycle and utilization ratio MTP1 busy cycle and utilization ratio ITP busy cycle and utilization ratio PTP busy cycle and utilization ratio Input Shapes
AIFF	Zhouyi Z2/Z3	AIPU total cycle Cost time AIFF busy cycle and utilization ratio MTP0 busy cycle and utilization ratio MTP1 busy cycle and utilization ratio ITP busy cycle and utilization ratio PTP busy cycle and utilization ratio AIFF weights read bytes
	Zhouyi X1/X2	Same as Default
DMA	Zhouyi Z2/Z3	AIPU total cycle Cost time DMA busy cycle and utilization ratio MTP0 busy cycle and utilization ratio MTP1 busy cycle and utilization ratio ITP busy cycle and utilization ratio PTP busy cycle and utilization ratio DMA read bytes DMA write bytes
	Zhouyi X1/X2	Same as Default

In addition to the values in the table, a profiling item, Algorithm Ops, is also displayed in different NPU targets. This is a theoretical value that is calculated by the equation of each operator. Currently only some of the AIFF operators are in the count.

The sum statistics of the entire network total count, total operations, total read bytes, total write bytes, and average utilization for the hardware unit are also shown.

For a Zhouyi X2 device, if `fast_perf` in the GBuilder setting section is True, all information about DMA and AIFF will not be displayed.

_____ Note_____

Because Zhouyi X1 and Zhouyi X2 NPUs have two physical ITP units, but only one ITP cycle counter is available (for consistency of the hardware architecture), this leads to some cases of ITP utilization ratio greater than 100%. When these cases occur, it just indicates that the two ITP units are fully used.

6.2 Using the aipu_profiler

The aipu_profiler consumes two inputs and generates one output.

- graph.json [**required**]

  This file is generated by the NN compiler.
- profiler_binary [**required**]

  The scalar processing unit records the hardware counter on the profile buffer when running the network. After finishing running the network, the Linux runtime will dump this buffer. You may write it to a file and make this file as the second parameter of `aipu_profiler`.
- output [**optional**]

  The output HTML report name. If it is not set, the HTML report will be named after the model name.
- method [**optional**]

  The output file format. Currently it supports `html` and `xlsx`. The default value is `html`.
- frequency [**optional**]

  To calculate cost time by number of cycle, frequency is provided. If not provided, use the frequency filled in graph.json.

The following is an example of generating the HTML report.

```
$./aipu_profiler graph.json profile_data.bin -o resnet50.html --method html
```

After running this command, the `resnet50.html` is generated. You can find the following example of the profiler HTML report (the performance data is for example only).

If an XLSX file is needed, use `--method xlsx -o resnet50.xlsx` instead.

_____ Note_____

Do not use the `profile_data.bin` file generated by the simulator, because the simulator has no simulation of the performance counter of the NPU, which is a fake buffer with all the data in it being 0. Zhouyi Compass provides an independent simulator profiler. See *7.3 Using the aipu_simulator_profiler* for more information.

_____

Profiler Report

DETAILED DATA | SUMMARY

Detailed Data

Layer Name	Core ID	Level	Cycles	AIFF Busy Cycles	DMA Busy Cycles	AIFF Utilization	AIPU Read Bytes	AIPU Write Bytes	DMA Read Bytes	DMA Write Bytes	MTP0 Busy Cycles	MTP0 Utilization	MTP1 Busy Cycles	MTP1 Utilization	ITP Busy Cycles	ITP Utilization	PTP Busy Cycles	PTP Utilization	Input Shapes
subgraph_tfl.quantize			2	0	0	0.00%	0	0	0	0	0	0.00%	0	0.00%	0	0.00%	0	0.00%	[1,224,224,3]
subgraph_MobilenetV2/Conv/Relu6;MobilenetV2/Conv/BatchNorm/FusedBatchNormV3;MobilenetV2/expanded_conv_5/project/Conv2D;MobilenetV2/Conv/Conv2D			2	0	0	0.00%	0	0	0	0	0	0.00%	0	0.00%	0	0.00%	0	0.00%	[1,224,224,3]
subgraph_MobilenetV2/expanded_conv/depthwise/Relu6;MobilenetV2/expanded_conv/depthwise/BatchNorm/FusedBatchNormV3;MobilenetV2/expanded_conv/depthwise/depthwise;MobilenetV2/expanded_conv_5/project/Conv2D			2	0	0	0.00%	0	0	0	0	0	0.00%	0	0.00%	0	0.00%	0	0.00%	[1,1,112,112,32]
subgraph_MobilenetV2/expanded_conv/project/BatchNorm/FusedBatchNormV3;MobilenetV2/expanded_conv/project/Conv2D1			2	0	0	0.00%	0	0	0	0	0	0.00%	0	0.00%	0	0.00%	0	0.00%	[1,1,112,112,32]
subgraph_MobilenetV2/expanded_conv_1/expand/Relu6;MobilenetV2/expanded_conv_1/expand/BatchNorm/FusedBatchNormV3;MobilenetV2/expanded_conv_1/depthwise/BatchNorm/FusedBatchNormV3;MobilenetV2/expanded_conv_1/depthwise/depthwise;MobilenetV2/expanded_conv_12/project/Conv2D;MobilenetV2/expanded_conv_1/expand/Conv2D			2	0	0	0.00%	0	0	0	0	0	0.00%	0	0.00%	0	0.00%	0	0.00%	[1,112,112,16]

Figure 4-5 Example profiler report

In this report, there are two parts of data:

- Detailed Data: Shows the per-tensor performance of the model. On this page, you can sort the tensors by different performance data in ascending or descending order. For example, when you click **Count cycle** in this report, all the tensors will be sorted by the count cycle in ascending order. If you click it again, the tensors will be sorted in descending order.
- Summary: Shows the model performance summary.

6.3 Using the profiler API to get profile data

Instead of generating an HTML file, the profiler also provides a function to help you get the profile result.

The function is as follows:

```
def get_profiler_data(json_path, bin_path, **kwargs):
    """Get final profiler result
    Parameters
    ----------
    json_path : str
        The graph.json path, graph.json is generated by gbuiler when enable
perf.

    bin_path : str
        The binary path, which is profiler buffer data generated when run.
        Dumped using UMD or auto dumped by emulator

    Returns
    -------
    result : dict
        The dictionary of the profile result.
    """
```

The following is an example of getting the output Python dictionary.

```
from AIPUBuilder.Profiler import get_profiler_data
result = get_profiler_data("graph.json",
"rtl_profile_odata_0x20e86000_0.bin")
```

The result is a Python dict, which contains the performance information. This dict has the following four keys:

- “graph_name”

  Result [“graph_name”] is a string which shows the model name.
- “frequency”

  Result [“frequency”] is an int value. The profiler gets cycle count from the hardware counter, but usually time (ms) is preferred instead of cycle count. This value helps you convert between time and cycle count. You can set it to pass frequency parameters in the get_profiler_data call, as shown in the preceding example, or just use the default value (for Zhouyi X1, the default value is 1000, which means that the frequency is 1000M).
- “graph”

  The graph contains all graph information which contains the nodes and tensors. You can get node input/output tensor name, node op_type and tensor shape information from it.
- “profile_data”

  Result [“profile_data”] is a list. The element in list order is the runtime order of each nodes. The element in the list is a dict, which contains two keys, “name” and “runtime_information”.

  — “name” is the name of the node which you can find in the graph.

— “runtime_information” contains the profiler data including cycles, AIFF busy cycle, and DMA busy cycles which correspond to the data in HTML files.

Chapter 7
Supporting customized operators

This chapters describes how to support a customized operator.

It contains the following sections:

7.1 Overview

As described in *Chapter 3 NPU toolchain*, after a customized operator is compiled to a .out by the NPU toolchain, it should be integrated into the NN compiler, so that it can finally be used in the NPU executable binary.

The NPU NN compiler uses the plugin mechanism to support customized operators. To support a customized operator, three plugins need to be implemented:

- Parser plugin
- Optimizer plugin
- GBuilder plugin

One plugin is optional:

- Checker plugin

Assume that before implementation of the plugin, you have created your own customized operators for the NPU by using assembly language or C language.

All kinds of plugins can be implemented in Python language. The Python plugin can be a single file module or a package, and the file name or package name must have prefix `aipubt_`. Arm China recommends that you use the following prefixes:

- `aipubt_parser_` for the parser plugin.
- `aipubt_opt_op_` for the optimizer operator plugin.
- `aipubt_gb_` for the GBuilder plugin.

In addition to the operator plugin, the dataset or metric plugin also can be customized in the optimizer. The dataset plugin is mainly used to get item data for calibration and quantization. The metric plugin is used to metricize a model. Arm China recommends that you use the following prefixes:

- `aipubt_opt_dataset_` for the dataset plugin of the optimizer.
- `aipubt_opt_metric_` for the metric plugin of the optimizer

The NN compiler will search for plugins in the following locations in order:

- Environment variable: AIPUPLUGIN_PATH. This variable has the same syntax as LD_LIBRARY_PATH, which is a list of paths separated by colons (:). For example, you can use the following command to add `/home/user/aipubuilder_plugins/` to the environment variable:

  ```
  $export AIPUPLUGIN_PATH=/home/user/aipubuilder_plugins/:$AIPUPLUGIN_PATH
  ```

- The second location is a folder in the current working directory (CWD), which is `./plugin/`.

Arm China recommends that you use the AIPUPLUGIN_PATH environment variable to set the plugin path.

7.2 Parser plugin

The parser plugin is used for parsing the operator types that are not 'builtin' supported in the parser. The plugin needs to update the operator parameters from the framework model and the parser will deploy these parameters into IR, so that the following quantization plugin can get them from IR and quantize the operation.

The parser plugin needs to implement a Python class. You need to derive the class from the provided base class and override some functions.

```
class ParserOp(object):

    op_type=None

    alias_op_type=None

    '''

    the named subgraph is defined by start_nodes and end_nodes,

    the defined subgraph is combined by all nodes which in any of path from
    any of start_nodes to any of end_nodes.

    '''

    start_nodes=None

    end_nodes=None

    pattern_nodes=None

    pattern_edges=None


    def __init__(self,framework,op_info):

        # for hyper

        self.params=OrderedDict()

        # for weights: str,numpy.array

        self.constants=OrderedDict()
```

The base plugin class can support both the single operator plugin and subgraph operator plugin.

- The single operator plugin is a plugin which maps an operator from the original framework to the Zhouyi framework.
- The subgraph plugin can map a subgraph of other frameworks to a single customized operator.

Single operator plugin

To implement a parser plugin, you need to import the base classes from `AIPUBuilder.Parser.plugin_op` and import registers from `AIPUBuilder.Parser.plugin_loader`.

The following is a simple example of a TopK operator:

```
from AIPUBuilder.Parser.plugin_loader import register_plugin, PluginType

from AIPUBuilder.Parser.plugin_op import ParserOp

import tensorflow as tf


@register_plugin(PluginType.Parser, "0.1")
```

```
class TopKOp(ParserOp):
    op_type = "TopKV2"
    alias_op_type = “TopK”

    def __init__(self, framework ,params):
        super(TopKOp, self).__init__(framework, params)
        self.framework = framework.upper()
        self.params.update({'topk_sorted': params.get('sorted', True)})

def infer_shape(self, input_tensors, *args):
        tf.enable_eager_execution()
        if len(input_tensors) >= 2:
            topk_value = input_tensors[1]
        else:
            topk_value = 1
        self.params.update({'topk_value': topk_value})
        value = self.params["topk_value"]
        sorted = self.params["topk_sorted"]
        values, indices = tf.math.top_k(input_tensors[0], k=value,
sorted=sorted)
        return [values, indices]
```

This plugin parses the TopK operation, and deploys its parameters `topk_sorted` and `topk_value` into IR. For more information about this operation and its parameters, visit **https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/math/top_k**.

For each plugin, there are three classes of attributes:

- `op_type`: A string type, which is used to identify which operator your plugin is for. This string must be the same as the operation name in the framework models.
- `alias_op_type`: A string. You can set it if you want to write a simple operator type in IR. For example, for the MeanVarianceNormalization operator, you must set op_type = ‘MeanVarianceNormalization’, but you can set alias_op_type = ‘MVN’, and the op_type will be ‘MVN’ in IR.

The plugin also has three necessary member functions:

- `__init__(self, framework, params)`: `framework` is a string type for telling the plugin which framework’s model you are parsing. The available values of the framework are:
  - TensorFlow
  - TFLite
  - ONNX
  - Caffe

  Usually, you should check whether the plugin supports the framework.

  `params` is a dict type, which indicates the parameters to initialize a plugin instance. The parameters are all key hyper-parameters saved in the framework model.

- `infer_shape(self, input_tensors, *args)`: Sometimes, the operator parameters are not saved in model files but act as a `const` operator. In this case, the `infer_shape` function can get these parameters in `input_tensors` and update in `self.params`. `input_tensors` is a list of NumPy arrays. This function also returns a list of NumPy arrays which are the operator results. `*args` is reserved for future use.
- `remove_inputs_at(self, tensor_index, node_index=0)`: Remove input tensors at the specific tensor index and node index (for subgraph). The specific input tensor is removed after infer_shape is called. Call this function with different arguments if there are multiple input tensors to be removed. Generally, it is used to remove some constant inputs which are already converted parameters in IR.

The following are two important attributes:

- `params`: An ordered dictionary for storing necessary parameters for the quantization plugin.
- `constants`: An ordered dictionary for storing constant tensors, such as weight and biases. The value of this dictionary is a NumPy array.

After you write a plugin, you need to register it by using the decorator `register_plugin`, which has two arguments. The first is the plugin type, and the second is a string type for versioning.

After that, you can put your plugin in the plugin search path. The NN compiler will call your plugin to handle your operator.

In this example, the parser will deploy operator TopKV2 in IR. The parameters with field names `topk_sorted` and `topk_value`, and related values are also shown in IR for the quantization plugin to use.

If the `op_type` name conflicts with some building operator names, the plugin will overwrite these building operators.

Subgraph plugin

The subgraph plugin is a plugin that can merge several operators into a single operator. This plugin can make the graph clearer and make the supporting customized model easier.

The subgraph plugin has two modes:

- Pattern-based
- Name-based

The pattern-based mode can be used to map a pattern subgraph to a single operator. This mode can be used if the pattern appears many times in your graph, so you can just add a single plugin to support them.

The name-based subgraph is just to merge the subgraph by the operator names. It is easier to use but it is usually a mode specific plugin, and it cannot be used for another model.

With the subgraph plugin, you can support all kinds of post/pre-process by customizing the plugin.

You need to specify the following class members for enabling subgraph mode:

- `start_nodes` (name-based)
- `end_nodes` (name-based)
- `pattern_nodes` (pattern-based)
- `pattern_edges` (pattern-based)

All the members are None by default. If some of them are set to a non-None value, the NN compiler will check whether the value is consistent.

The `start_nodes` and `end_nodes` are for the name-based subgraph plugin. The `pattern_nodes` and `pattern_edges` are for the pattern-based subgraph plugin.

If you want to use the name-based subgraph plugin, the `start_nodes` and `end_nodes` must be set correctly, and the `pattern_ndoes` and `pattern_edges` must be None, and vice versa.

Name-based subgraph plugin

For the name-based subgraph plugin, to define a subgraph, you can use the name of the operator. Therefore, two class members are provided—`start_nodes` and `end_nodes`.

- `start_nodes`: A list of strings to define the start nodes of the subgraph.
- `end_nodes`: A list of strings to define the end nodes of the subgraph.

For example, the following figure shows a graph that merges the green operators into one operator.

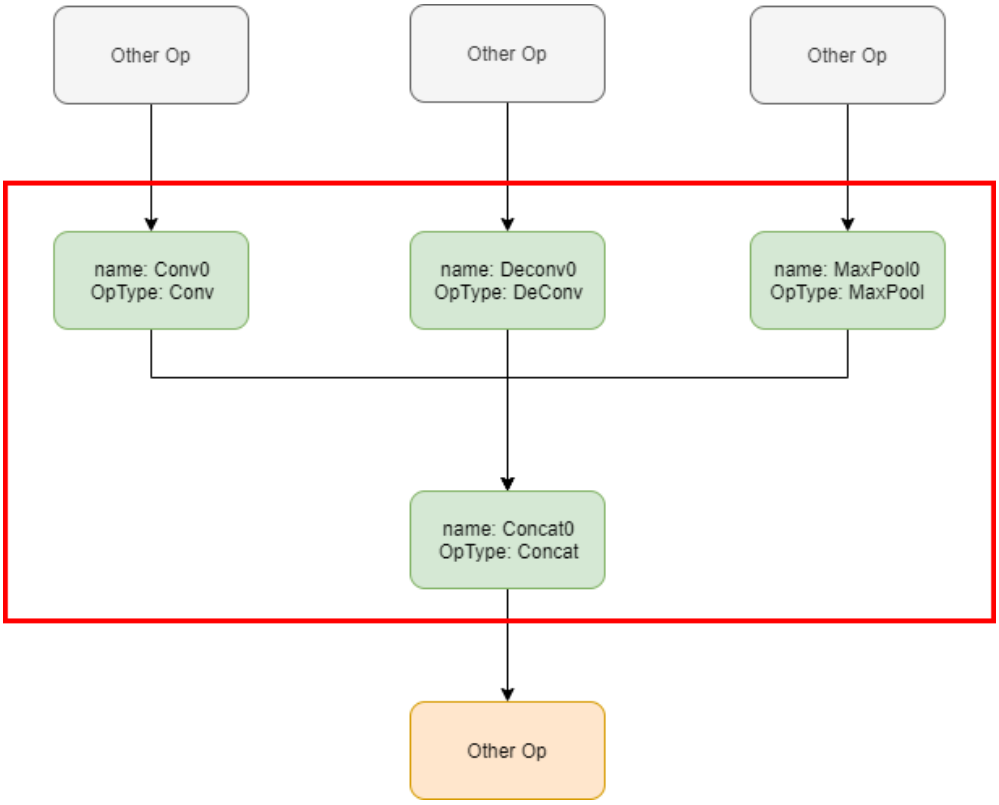

Figure 4-6 Merging the green operators into one operator

In this case, the start nodes include all nodes that their parents are not in the subgraph—`Conv0`, `Deconv0` and `MaxPool0`. The end node is just `Concat0`.

Therefore, you can write a subgraph plugin as follows:

```
class MySubgraphPlugin(ParserOp):
    op_type=MySubgraphPlugin
    start_nodes=["Conv0","Deconv0","MaxPool0"]
    end_nodes=["Concat0"]

    def __init__(self,framework,op_infos):
        super().__init__(framework,op_infos)
        # the op infos is a dict
        conv0_infos=op_infos["Conv0"]
        deconv0_infos=op_infos["Deconv0"]
        maxpool0_infos=op_infos["MaxPool0"]
        concat0_infos=op_infos["Concat0"]
```

```
        # other code to deal with the ops info


    def infer_shape(self,input_tensors,*args):
        input0=input_tensors[0]
        input1=input_tensors[1]
        input2=input_tensors[2]
        # check the shape~
        shape=input0.shape
        # assume that the concat axis =3, which should be read in op_info
        shape+=input1.shape[3]+input2.shape[3]
        return [np.random.randn(shape,dtype=XXX)]
```

After applying the plugin, the graph will change as shown in the following figure.

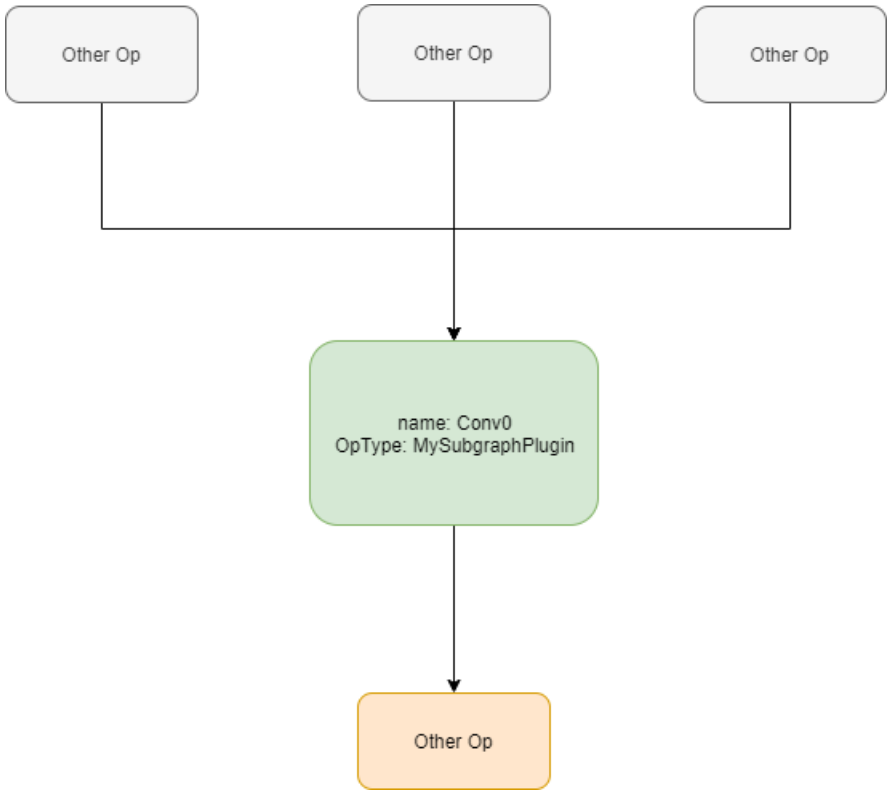

Figure 4-7 The graph after applying the plugin

Pattern-based subgraph plugin

For the pattern-based subgraph, you need to define the patterns—nodes and edges.

- `pattern_nodes`: A list of tuples of string (nick_name, optype). It defines all nodes that appear in the subgraph. The nick name is for identifying the nodes, which is not related to the name of the operator in the graph. The optype name is the operator type.
- `pattern_edges`: A list of tuples. The basic edge can be described as the nick_name of two operators. In addition, you can add some attributes to determinate the input/output tensors of the edges. The attribute is a string dict as the third member of the tuple.

Currently, only two attributes for edges are supported:

- `src_out_port`: An unsigned int to indicate the index of the output for the source node.
  - `0` means the first output of the source node.
  - `2` means the third output.

  By default, the value is `0`.
- `dst_in_port`: An unsigned int to indicate the index of the input for the destination node.
  - `0` means the first input of the destination node.
  - `2` means the third input.

  By default, the value is `0`.

The following is a simple example with the same case as the name-based subgraph plugin:

```
class MySubgraphPlugin(ParserOp):
    op_type=MySubgraphPlugin
    pattern_nodes=[
        ('n0',"Conv"), # the n0 node is a Conv type
        ('n1',"Deconv"), # the n1 node is a Deconv type
        ('n2',"MaxPool"), # the n2 node is a MaxPool type
        ('n3',"Concat"), # the n3 node is a Concat type
]
    pattern_edges=[
        # the edge from n0 to n3, and the tensor is the first input of n3
        ('n0','n3', {'src_out_port': 0, 'dst_in_port': 0}),

        # the edge from n1 to n3, and the tensor is the second input of n3
        ('n1','n3', {'src_out_port': 0, 'dst_in_port': 1}),

        # the edge from n2 to n3, and the tensor is the third input of n3
        ('n2','n3', {'src_out_port': 0, 'dst_in_port': 2}),
    ]
    def __init__(self,framework,op_infos):
        super().__init__(framework,op_infos)
        # the op infos is a dict
        conv0_infos=op_infos["n0"]
        deconv0_infos=op_infos["n1"]
        maxpool0_infos=op_infos["n2"]
        concat0_infos=op_infos["n3"]
        # other code to deal with the ops info

    def infer_shape(self,input_tensors,*args):
```

```
input0=input_tensors[0]
input1=input_tensors[1]
input2=input_tensors[2]
# check the shape~
shape=input0.shape
# assume that the concat axis =3, which should be read in op_info
shape+=input1.shape[3]+input2.shape[3]
return [np.random.randn(shape,dtype=XXX)]
```

7.3 Optimizer plugin

The optimizer plugin includes the operator plugin, model dataset plugin, and model metric plugin.

- The model metric plugin defines the rules for determining whether a model output is correct.
- The model dataset plugin defines a dataset that can be used to run the model. With the model metric plugin, you can get the accuracy of the model on the certain dataset.
- The operator plugin defines the ‘quantize’ and ‘forward’ functions of a customized operator, with which the operator can work correctly in the model.

Model dataset plugin

The dataset plugin of the optimizer inherits from the `torch.utils.data` dataset, so the dataset plugin should be implemented with three methods— `__init__`, `__len__` and `__getitem__`.

To implement an optimizer dataset plugin, you need to import these base classes:

```
from AIPUBuilder.Optimizer.framework import *
from torch.utils.data import Dataset
```

The following is a simple example of a NumPy dataset:

```
from AIPUBuilder.Optimizer.framework import *
from torch.utils.data import Dataset
import numpy as np


@register_plugin(PluginType.Dataset, '0.01')
class NumpyDataset(Dataset):
    ####In very rare situations, you can customize torch's DataLoader
collate_fn
    ####by offering a corresponding static method
    #data_batch_dim = 0
    #label_batch_dim = 0
    #@staticmethod
    #def collate_fn(batch):
    #    return batch_data, batch_label
    #when used as calibration dataset, label_file can be omitted.
    def __init__(self, data_file, label_file=None):
        self.data = None
        self.label = None
        try:
            self.data = np.load(data_file, mmap_mode='c')
        except ValueError:
            self.data = np.load(data_file, allow_pickle=True)
        if label_file is not None:
            try:
```

```
                    self.label = np.load(label_file, mmap_mode='c')
                except ValueError:
                    self.label = np.load(label_file, allow_pickle=True)


    def __len__(self):
        return len(self.data)


    def __getitem__(self, idx):
        #Assume that all preprocesses have been done before save to npy file.
        #If the graph has single input tensor,
        #the data part sample[0] will be passed to the input node as is,
        #if the graph has multiple input tensors,
        #the data part sample[0][i] should be consistent with
input_tensors[i] in IR.
        #If the graph has multiple output tensors,
        #the label part sample[1][i] should be consistent with
output_tensors[i] in IR.
        sample = [[self.data[idx]], float("-inf")]
        if self.label is not None:
            sample[1] = self.label[idx]
        return sample
```

The plugin has three necessary member functions:

- `__init__(self, data_file, label_file=None)`: The data_file and label_file of the dataset plugin initialize a plugin instance.
- `__len__(self)`: Returns the length of dataset.
- `__getitem__(self, idx)`: If the label_file is available, this method will return a list of data[idx] and label[idx]. The data[idx] can also be a list of data for multiple inputs, and label[idx] can be a list or dict of data for multiple outputs.

The format of the data_file and label_file also can be customized (but a NumPy file is recommended). If the model has multiple inputs, the order of inputs in data[idx] should be aligned with the order of `input_tensors` in the header of float32 IR.

The data and the label generated in the dataset plugin are unnecessary to be quantized when used for running a quantized inference forward because the OPT will quantize it automatically.

After you write a plugin, you need to register it by using the decorator `register_plugin`, which has two arguments. The first is the plugin type, and the second is a string type for versioning.

Model metric plugin

The metric plugin of the optimizer inherits from the OptBaseMetric class, and the metric plugin should be implemented with five methods—`__init__`, `__call__`, `reset`, `compute` and `report`.

The base OptBaseMetric class is:

```
class OptMetric(object):
```

```
    def __init__(self, *args):
        pass
    def __call__(self, pred,target,*args):
        raise NotImplementedError()
    def compute(self):
        raise NotImplementedError()
    def reset(self):
        raise NotImplementedError()
    def report(self):
        raise NotImplementedError()
```

The following is a simple example of a TopK metric:

```
from AIPUBuilder.Optimizer.framework import *
from AIPUBuilder.Optimizer.logger import *
import torch

@register_plugin(PluginType.Metric, '0.01')
class TopKMetric(OptBaseMetric):
    def __init__(self, K='1'):
        self.correct = 0
        self.total = 0
        self.K = int(K)
    def __call__(self, pred, target):
        _, pt = torch.topk(pred[0], self.K, dim=-1)
        for i in range(target.numel()):
            if target[i] in pt[i]:
                self.correct += 1
        self.total += target.numel()
    def reset(self):
        self.correct = 0
        self.total = 0
    def compute(self):
        try:
            acc = float(self.correct) / float(self.total)
            return acc
        except ZeroDivisionError:
            return float("-inf")
    def report(self):
```

```
        return "top-%d accuracy is %f" % (self.K, self.compute())
```

The plugin has five necessary member functions:

- `__init__(self, *args)`: Initializes a plugin instance and sets the corresponding attributes
- `__call__(self, pred, target)`: Needs two arguments—`pred` (the outputs of the graph when inferencing) and `target` (the ground truth). If the model has multiple outputs, the order of `pred` outputs is aligned with the order of 'output_tensors' in the header of IR.
- `reset(self)`: Resets the attributes.
- `compute(self)`: Calculates the metric value.
- `report(self)`: Gives a metric value message.

The 'pred' and 'target' passed by OPT are automatically dequantized in advance when running a quantized inference forward.

After you write a plugin, you need to register it by using the decorator register_plugin, which has two arguments. The first is the plugin type, and the second is a string type for versioning.

Operator plugin

The optimizer operator plugin needs to implement two methods—`forward` and `quantize`.

- The `forward` function uses op_register(OpType, version) to register.
- The `quantize` function uses quant_register(OpType, version) to register.

The `forward` function mainly implements the inference and uses the node.quantized parameter to distinguish the float32 inference and quantized inference.

The `quantize` function implements the quantization of all tensors in this operator. The quantization information about a tensor includes `scale`, `zerop`, `qbits`, `qmin`, `qmax`, `dtype`, and `qinvariant`.

When implementing the `forward` and `quantize` methods, the following package must be imported:

```
    from AIPUBuilder.Optimizer.framework import *
```

This package includes some basic core data structures:

Parameter	Main attributes

Dtype	Dtype = {
	.FP16,
	.FP32,
	.FP64,
	.INT8,
	.UINT8,
	.INT16,
	.UINT16,
	.INT32,
	.UINT32,
	.INT64,
	.UINT64,
	.ALIGNED_INT4,
	.ALIGNED_INT12,
	}

PyNode	PyNode = { #Node attributes .name: string,

	.type: OpType, # op type, you can use 'register_optype(string) to register your own op type'
	.inputs: list(PyTensor),
	.outputs: list(PyTensor),
	# params: an ordered-dictionary for storing the necessary parameters
	.params: dict(),
	# constants: an ordered-dictionary for storing constant tensors, such as weights and biases. The value of this dict is also a dict with a format {key: PyTensor}.
	.constants: dict(string: PyTensor),
	# placeholders: a list for storing intermediate tensors.
	.placeholders: list(PyTensor),
	# attrs: an ordered-dictionary for storing the intermediate parameters, which is not writing to IR.
	.attrs: dict(string: value),
	.current_batch_size,
	#Node methods
	#node.get_param(key): get the value from the node.params using the key
	.get_param(string),
	#node.get_ constant (key): get the value from the node.constants using the key
	.get_constant(string)
	#node.get_attrs(key): get the value from the node.attrs using the key
	.get_attrs(string)
	}

PyTensor	PyTensor = {
	.name: string,
	.ir_shape: list,
	.betensor: torch.Tensor, # save the data
	#scale/zerop/qmin/qmax/dtype/qbits/qinvariant are tensor quantization information, and these #information need update in `quantize` method for all tensors.
	.scale: float,
	.zerop: float,
	.qmin: int,
	.qmax: int,
	.qbits: int, #quantization bits, default value: weight_bits=8, activation_bits=8, bias_bits=32
	.dtype: Dtype,
	#if the tensor is quantization invariant (like index values), set it to True.
	.qinvariant: bool,
	.min: float,
	.max: float,
	}

The following table enables you to better understand the relative IR parameters and core attributes.

Table 4-4 IR parameters and core attributes

IR parameters	Attributes
layer_id=2	node.attrs['layer_id']: string
layer_name=norm1	node.name: string
layer_type=LRN	node.type: OpType
layer_bottom=[conv1]	node.inputs

layer_bottom_shape=[[10,55,55,96]]	node.inputs[i].ir_shape: list
layer_bottom_type=[float32]	node.inputs[i].dtype: Dtype
layer_top=[norm1]	node.outputs
layer_top_shape=[[10,55,55,96]]	node.outputs[i].ir_shape: list
layer_top_type=[float32]	node.outputs[i].dtype: Dtype
method=ACROSS_CHANNELS	node.params['method']
depth_radius=2	node.params['detph_radius']
bias=1.000000	node.params['bias']
alpha=0.00020	node.params['alpha']
beta=0.75	node.params['beta']
scale_sum_type=uint8	node.params['scale_sum_type']: Dtype
scale_sum_value=204	node.params['scale_sum_value]
scale_type=uint8	node.params['scale_type']: Dtype
scale_value=138	node.params['scale_value']
lut_type=uint16	node.constants['lut'].dtype
lut_shape=[256]	node.constants['lut'].ir_shape
lut_offset=35232	---
lut_size=512	---

The following uses the LRN operator plugin as an example:

```
from AIPUBuilder.Optimizer.framework import *
from AIPUBuilder.Optimizer.utils import *
from AIPUBuilder.Optimizer.logger import *
import torch

@op_register(OpType.LRN)
def LRN(self, *args):

    _SUPPORT_METHOD = ['ACROSS_CHANNELS', 'WITHIN_CHANNEL']
    inpt = self.inputs[0].betensor
    method = self.get_param('method').upper()
    if method not in _SUPPORT_METHOD:
        (OPT_ERROR("id=%s,op_type=%s DoesNot support '%s', only support %s"
       % (self.attrs['layer_id'], str(self.type), method,
str(_SUPPORT_METHOD))))

    size = self.get_param('size')
    bias = self.get_param('bias')
```

```
    alpha = self.get_param('alpha')
    beta = self.get_param('beta')
    input = nhwc2nchw(inpt)
    if not self.quantized:
        inputs_square = torch.square(input)
        if method == 'ACROSS_CHANNELS':
            inputs_square = inputs_square.unsqueeze(1)
            padding = (0, 0, 0, 0, int(size / 2), int((size - 1) / 2))
            inp_square_with_pad = torch.nn.functional.pad(inputs_square,
padding, mode='constant', value=0)
            square_sum = torch.nn.functional.avg_pool3d(inp_square_with_pad,
            kernel_size=(size, 1, 1),
            stride=1,
            padding=0,
            divisor_override=1,
            count_include_pad=False)
            square_sum = square_sum.squeeze(1) / size
            denominator = torch.pow((bias + alpha * square_sum), beta)
            outt = input / denominator
            if len(self.placeholders) < 1:
                ph0 = Tensor(self.name+"/square_sum",
square_sum.cpu().numpy().astype(dtype2nptype(Dtype.FP32)))
                self.placeholders.append(ph0)
                self.placeholders[0].betensor = square_sum
        else:  # method == 'WITHIN_CHANNEL':
            square_sum = torch.nn.functional.avg_pool2d(inputs_square,
            kernel_size=size,
            stride=1,
            padding=int((size-1.0)/2),
            # divisor_override=1,
            count_include_pad=False)
            denominator = torch.pow((bias + alpha * square_sum), beta)
            outt = input / denominator
    else:
        input = input + self.inputs[0].zerop
        do_scale, do_scale_n = self.get_param('scale_value'),
self.get_param('scale_sum_value')
```

```
        do_shift, do_shift_n = self.get_param('shift_value'),
self.get_param('shift_sum_value')
        lut = self.constants['lut']
        lut_in_bits = 16
        lut_in_qmin, lut_in_qmax = bits2range(lut_in_bits, False)
        lut_out_bits = 16


        inputs_square = torch.square(input)
        if method == 'ACROSS_CHANNELS':
            inputs_square = inputs_square.unsqueeze(1)
            padding = (0, 0, 0, 0, int(size / 2), int((size - 1) / 2))
            inp_square_with_pad = torch.nn.functional.pad(inputs_square,
padding, mode='constant', value=0)
            square_sum = torch.nn.functional.avg_pool3d(inp_square_with_pad,
            kernel_size=(size, 1, 1),
            stride=1,
            padding=0,
            divisor_override=1,
            count_include_pad=False)
        else: #method == 'WITHIN_CHANNEL'
            square_sum = torch.nn.functional.avg_pool2d(inputs_square,
            kernel_size=size,
            stride=1,
            padding=int((size-1.0)/2),
            divisor_override=1,
            count_include_pad=False)


        avg_pool = linear_requantize(square_sum, do_scale_n, do_shift_n, 0,
lut_in_qmin, lut_in_qmax)
        avg_pool = torch.reshape(avg_pool.squeeze(1), (-1,))
        in_is_signed = False # because the square_sum is unsigned.
        lut_v = lookup_lut_powerof2(avg_pool, lut.betensor, lut_in_bits,
in_is_signed, lut_out_bits, is_signed(lut.dtype))
        lut_v = torch.reshape(lut_v, input.shape)
        outt = input * lut_v
        outt = linear_requantize(outt, do_scale, do_shift,
self.outputs[0].zerop, self.outputs[0].qmin, self.outputs[0].qmax)
```

```
    outt = nchw2nhwc(outt)
    self.outputs[0].betensor = outt
    return outt


@quant_register(OpType.LRN)
def LRN_quantize(self, *args):
    inp = self.inputs[0]
    out = self.outputs[0]
    size = self.get_param('size')
    bias = self.get_param('bias')
    alpha = self.get_param('alpha')
    beta = self.get_param('beta')


    q_bits_activation = self.attrs['q_bits_activation']
    q_mode_activation = self.attrs['q_mode_activation']
    if QuantMode.is_per_channel(q_mode_activation) == True:
        OPT_ERROR("Currently not support per-channel quantization of
activations, optimizer will use per-layer quantization for LRN OP")
    out_signed = is_signed(inp.dtype)
    out.qbits = q_bits_activation
    out.scale, out.zerop, out.qmin, out.qmax, out.dtype =
get_linear_quant_params_from_tensor(out, q_mode_activation, out.qbits,
is_signed=out_signed)
    out.qinvariant = False


    placeholder_scale = 1.
    if len(self.placeholders) > 0:
        placeholders = self.placeholders[0]
        placeholders_qinfo =
get_linear_quant_params_from_tensor(placeholders,
       QuantMode.to_symmetric(q_mode_activation), 16, is_signed=False)
        placeholder_scale = placeholders_qinfo[0]


    lut_in_bits = 16
    lut_qmin, lut_qmax = bits2range(lut_in_bits, False)
    lsteps = 2 ** min(inp.qbits, int(self.get_attrs('lut_items_in_bits')))
    lut = 1. / torch.pow(bias + alpha * (torch.linspace(lut_qmin, lut_qmax,
steps=lsteps) / placeholder_scale), beta)
```

```
    lut = Tensor(self.name+"/tmp", lut.cpu().numpy())

    lut.min = lut.betensor.min()

    lut.max = lut.betensor.max()

    # lut output bits fixed to 16bits

    lut_out_bits = 16 #max(q_bits_activation, 16)

    lut.qbits = lut_out_bits

    lut.scale, lut.zerop, lut.qmin, lut.qmax, lut.dtype =
get_linear_quant_params_from_tensor(lut, q_mode_activation, lut.qbits,
is_signed=False)

    qlut = linear_quantize_clip(lut.betensor, lut.scale, lut.zerop, lut.qmin,
lut.qmax)

    self.constants['lut'] = Tensor(self.name+"/lrn_lut",
qlut.cpu().numpy().astype(dtype2nptype(lut.dtype)))


    # quantize the 1./ size

    total_rescale = placeholder_scale / (inp.scale * inp.scale * size)

    scl_n, scl_ntype, sft_n, sft_ntype =
get_scale_approximation_params(total_rescale, mult_bits=q_bits_activation,

 force_shift_positive=self.force_shift_positive)

    total_scale = out.scale / inp.scale / lut.scale

    dscale, dscale_t, dshift, dshift_t =
get_scale_approximation_params(total_scale, mult_bits=q_bits_activation, forc
e_shift_positive=self.force_shift_positive)

    self.params['scale_sum_value'] = int(scl_n)

    self.params['shift_sum_value'] = int(sft_n)

    self.params['scale_sum_type'] = scl_ntype

    self.params['shift_sum_type'] = sft_ntype

    self.params['scale_value'] = int(dscale)

    self.params['shift_value'] = int(dshift)

    self.params['scale_type'] = dscale_t

    self.params['shift_type'] = dshift_t
```

You can put your plugin in the plugin search path. The NN compiler will call your plugin to handle your operators.

Input operator

The input operator is a special operator type which does not contain any calculations and is just a placeholder to represent the starting points of a model. The scale (and other quantization related parameters) of an input tensor is calculated through automatic statistic on the calibration dataset by default. In some cases, if the scale (and other quantization related parameters) is known already, you can directly write a higher version plugin of the input operator. Take a typical image classification model as an example, if the input image of the original float model is normalized to [-1, 1], and the input image (generated by other sensors) on a target platform of the quantized model is in the range [0, 255], then the scale and zero point of this input can be 127.5 and -128. Therefore, the plugin might be as follows:

```
from AIPUBuilder.Optimizer.framework import *
from AIPUBuilder.Optimizer.utils import *

@quant_register(OpType.Input, '1024')
def my_inp_quantize(self, *args):
    out = self.outputs[0]
    out.qbits = 8
    out.dtype = Dtype.UINT8
    out.qmin = 0
    out.qmax = 255
    out.scale = 127.5
    out.zerop = -128
out.qinvariant = False
```

7.4 GBuilder plugin

The GBuilder plugin can be implemented in Python. You need to implement a Python base class as follows:

```
class BuilderOpPlugin(BuilderOpPluginBase):
    def get_score(self):
        return 1
    def setup(self, sgnode, nodes):
        return True
    def generate_code_name(self, sgnode, nodes):
        return ""
    def generate_params(self, sgnode, nodes):
        ro = BuilderParams()
        return ro
    def generate_descriptor(self, sgnode, nodes):
        return BuilderParams()
```

To add a plugin, perform the following steps:

1. Register your operator type, write a pattern for telling GBuilder which layer you want to support, and set up the score.
2. Define the layer parameters.
3. Ensure that the parameter is normal and set up.
4. Generate the `ro` and descriptor. The `ro` should match the `param` of operator implementation.
5. Generate the code name and register this layer.

Pattern and score

The pattern is combined by two lists, a `node_represent` list and an `edge` list:

- A `node_represent` is defined by a list of tuples (name, optype). The name is a string type. The optype is an OpType object defined by the NN compiler. You can also define an optype by calling the NN compiler API. An example of a `node_represent` list is `[{"n1",OpType.Activation}]`.
- An `edge` is defined by a list of names (node1's name, node2's name). The node name must be defined in an existing `node_represent` in advance. You can leave it as an empty list if there is no edge. For example, `[{"n1","n2"}]` means there is one edge from `node_represent` n1 to `node_represent` n2.

Generally, a customized operator is a single node case. For example, if you need to support an operator type named `MyOwnOpType`, you need to register this operator type through the NN compiler API `AIPUBuilder.core.register_optype`.

You can register a customized operator type by using the following code:

```
from AIPUBuilder.core import *
NewOp = register_optype("NewOp")
…
def get_graph_pattern(self):
    return ([("0", NewOp)], [])
```

```
# or you can
def get_graph_pattern(self):
    return ([("0", OpType.NewOp)], [])
```

After registering the customized operator type, you can register a new operator named `NewOp`, which should be the same as 'layer_type=NewOp' in the IR. You can use it in the pattern to define it as `NewOp` or `OpType.NewOp`. Note that the operator type name is case sensitive.

The score is an unsigned int number. If several plugins have the same pattern, the builder will choose the highest-score plugin.

```
def get_score(self):
    return 100
```

Node list creation

A Node list is a list of Node objects that are in the same order of `node_represent` as you defined in the *Pattern and score* stage. You can access the Node parameters through the Node object.

The Node class is defined in `AIPUBuilder.core`, which is defined to represent an operator in the IR. The following is the interface of the Node class:

```
class TensorList(list):
    ...
class TensorMap(dict):
    ...
class NodeParamDict(dict):
    ...
class NodeAttrDict(dict):
    ...
class Node:
    type: OpType
    name: str
    inputs: TensorList
    outputs: TensorList
    constants:TensorMap
    params:NodeParamDict
    attrs:NodeAttrDict
```

The TensorList is a list type that only accepts tensors as the elements.

The Tensor class is defined in `AIPUBuilder.core`, which is defined to represent tensors. The following is the interface of the Tensor class:

```
class Tensor:
    dtype: Dtype
    name: str
    def mem_size(self) -> int: ...
    def offset(self) -> int: ...
```

```
    def size(self) -> int: ...
    def memory_offset(self) -> MemoryObject: ...
    def data(self) -> np.ndarray: ...
    def numpy(self) -> np.ndarray: ...
    def set_numpy(self,d:np.ndarray) -> None: ...
```

The TensorMap is a dict type that only accepts `str` as the key and tensors as values.

The NodeParamDict is a dict type that only accepts `str` as the key and NodeParamValue as a dict value.

The NodeParamValue class is defined in `AIPUBuilder.core`, which is defined to represent a parameter value in the IR. The NodeParamValue is a union type that can accept bool, int, float, TensorShape, and Dtype. It also can be a list of bool, int, float, Dtype, and TensorShape.

The following is the interface of the NodeParamValue class:

```
class NodeParamValue:
    def __init__(self, p:[str,int,bool,float,Dtype,TensorShape]) -> None:...
    def push_back(self, i:[Dtype, bool, int,float]) -> None: ...
    def __getitem__(self, i:[int]) -> object: ...
```

For information on how to get the NodeParamValue information, see the *IR and NodeParamValue* section.

The NodeAttrDict is a dict type that only accepts `str` as the key. The value can be an object type.

After every Node object is created, the nodes will be a list of all the Node objects.

IR and NodeParamValue

The NN compiler will parse the IR, and all attributes in the IR will be represented by a class named `NodeParamValue`. This section describes the `NodeParamValue` class in the relative class.

The following table shows how to get the IR value through the Node object.

In summary:

- A layer in IR is a Node object.
- A layer top/bottom in IR is a Tensor object, which has shape and type attributes.
- A weights/biases/LUT/scale (per channel)/shift (per channel) is a Tensor object in `node.constants`. You can access with key `weights/biases/lut/scale/shift`.
- Other parameters are in `node.params`, and accessible by their keys in the IR.

Table 4-5 IR and Node

IR	Node object
layer_id=3	Ignored
layer_name=featuresPre/Tanh	node.name
layer_type=Activation	node.type
layer_bottom=[featuresPre/conv2d/BiasAdd_0]	node.inputs
layer_bottom_shape=[[1,1,390,128]]	node.inputs[i].shape
layer_bottom_type=[int8]	node.inputs[i].type
layer_top=[featuresPre/Tanh_0]	node->outputs
layer_top_shape=[[1,1,390,128]]	node.outputs[i].shape

layer_top_type=[int8]	node.outputs[i].type
method=TANH	node.params["method"]
scale_type=int8	node.params["scale_type"]
scale_value=1	node.params["scale_value"]
shift_type=int8	node.params["shift_type"]
shift_value=7	node.params["shift_value"]
lut_type=int8	node.constants["lut"]->type
lut_offset=3456	node.constants["lut"].memory_object.offset
lut_size=256	node.constants["lut"].mem_size
lut_shape=256,1	node.constants["lut"].shape

Parameter check and setup

In this section, you need to check whether the parameters are supported by your `asm/c` code for the NPU, such as the height/width of the feature. In the example, it is just to return true, but no checking is conducted. The NN compiler will pass a Node list (`nodes` in the following example) as an argument to the `check_params` function.

```
def check_params(self, nodes):

      print("checking and using params for NewOP")

      return True
```

The `set_target` function is for checking whether your plugin supports this target. The target is the target code such as `Z2_1104`, which is a string type in Python. The following example code shows that only target Z2_1104 is supported.

```
  def set_target(self, target: str):

      return target == "Z2_1104"
```

The `setup` function is mainly used to define the input/output layouts of your operator. It is also for other attributes, such as `workspace_size` and `suppport_mmu`. The layouts information is mandatory.

Required attributes:

- `valid_input_layouts`: For telling GBuilder what kinds of input layouts are supported by the node. It is a list of DataLayout type, where a DataLayout type is a list type as well, for example, [[DataLayout.NCHW], [DataLayout.NHWC]]. This example means that the operator only has one input and the operator supports two combinations of input layouts—the first input layout is NCHW and the second is NHWC.
- `valid_output_layouts`: For telling GBuilder what kinds of output layouts are supported by the node. It is a list of DataLayout type, which is the same as valid_input_layouts, for example, [[DataLayout. NCHW, DataLayout. NHWC], [DataLayout.NCHWC16, DataLayout.NCHWC32]]. This example means that the operator has two outputs, and it supports two kinds of layouts—the first case is that the first output is NCHW format, and the second output is NHWC format. The second case is that the first output is NCHWC16 format, and the second output is NCHWC32 format.
- `keeping_layout`: If your node needs to keep the same layout for input/output tensors, set this attribute to True. This attribute is a bool type.

Optional attributes:

- `workspace_size`: Int type. It indicates how much temporary buffer is needed in this node.
- `support_mmu`: Bool type. If the node supports the segmented MMU, and it is True, GBuilder will allocate a SRAM or MMU buffer for it.

- Others: You can set any other attributes for later use in the RO/Code generation stage.

The NN compiler will pass two arguments to the `setup` function:

- The first is a Node object (`sgnode` in the following example) to represent your operator.
- The second is a Node list (`nodes` in the following example) for you to access the original parameters.

You can only set the attributes to `sgnode`, while `nodes` is read-only (created before the *Node list creation* phase). Any modification of `nodes` is unexpected. The following is an example:

```
def setup(self, sgnode, nodes):
    print("Setting up")
    sgnode.attrs["keeping_layout"] = True
    valid_in = [[DataLayout.NCHWC32],
                [DataLayout.NCHWC16],
                [DataLayout.NCHW],
                [DataLayout.NHWC], ]
    sgnode.attrs["valid_input_layouts"] = valid_in
    sgnode.attrs["valid_output_layouts"] = valid_in
    return True
```

RO/Descriptor generation

In this phase, you will return a RO list. This is the core of the plugin, to generate parameters of `ASM/C` code according to the node information.

Descriptor is the same as `ro`, just named as descriptor. The descriptor is optional. For a customized operator, you can leave it blank.

The `generate_params` and `generate_descriptor` are similar. They both have two arguments, which are the same as the `setup` function. The origin nodes have all the information to generate your `ro`. Note that the Node list is read-only, and you should not modify it.

The `ReadOnlyPram` (`ro` in the following example) is a class defined by the NN compiler, which is for the arguments to pass to `ASM/C` lib. It is a union type of uint32 and Tensor, so you can only use a uint32_tnumber or a Tensor object (feature map) as your `ASM/C` lib parameters.

The following is an example of generating a `ro`:

```
def generate_params(self, sgnode, nodes):
    print("generating ro")
    inp = sgnode.inputs[0]
    out = sgnode.outputs[0]
    ro = BuilderParams()
    ro.append(nodes[0].params["add_n"])
    ro.append(inp.shape.size())
    ro.append(inp)
    ro.append(out)
    return ro
```

Important Notice

Differences between `sgnode` and `nodes`:

- The `sgnode` is a subgraph node which contains all nodes, but it acts as one node. In addition, the NN compiler will treat it as a node.
- All `nodes` in the subgraph are invisible in the NN compiler. All Tensor inputs/outputs and constants should use the `sgnode`.

In the preceding case, the `nodes` argument is a one-element list, because the pattern is a single node pattern. The `sgnode` also represents only one physical Node.

The key of constants (weights/biases/scale/shift/lut) will change because there may be many nodes which have weights. The key will be added by a scope of Node name, for example:

```
nodes[0]'s weights would have a key: nodes[0].name+"/weights",
```

Code generation and register

The section just returns your code name. Ensure that your code name is the same as your function name and object file name if you are using the C compiler. If you are using the assembler, the code name must be the same as your binary file name.

```
def generate_code_name(self, sgnode, nodes):
    return '''addn_Z2_1104.bin'''
```

The NN compiler provides a decorator for registering your code. You can just add the decorator before your class. The decorator is defined in `AIPUBuilder.plugin_loader`, which accepts only two arguments:

- The plugin type for Builder, which must be PluginType.Builder.
- The version of your plugin, which must be a number.

For example:

```
from AIPUBuilder.plugin_loader import *
@register_plugin(PluginType.Builder, 0)
class AddonePlugin(BuilderOpPlugin):
    def get_graph_pattern(self):
    …
    Other code
```

The following is a simple example with all codes:

```
from AIPUBuilder.core import *
from AIPUBuilder.plugin_loader import *

NewOp = register_optype("NewOp")
@register_plugin(PluginType.Builder, 0)
class AddonePlugin(BuilderOpPlugin):
    def get_graph_pattern(self):
        return ([("0", NewOp)], [])

    def set_target(self, target: str):
        return target == "Z2_1104"
```

```
def check_params(self, nodes):
    print("checking and using params for NewOP")
    return True

def setup(self, sgnode, nodes):
    print("Setting up")
    sgnode.attrs["keeping_layout"] = True
    valid_in = [[DataLayout.NCHWC32],
                [DataLayout.NCHWC16],
                [DataLayout.NCHW],
                [DataLayout.NHWC], ]
    sgnode.attrs["valid_input_layouts"] = valid_in
    sgnode.attrs["valid_output_layouts"] = valid_in
    return True
def generate_params(self, sgnode, nodes):
    print("generating ro")
    ro = BuilderParams()
    ro.append(out)
    return ro
def generate_code_name(self, sgnode, nodes):
return '''addn_Z2_1104.bin'''
```

7.5 Checker plugin

The checker plugin can be implemented in Python, which is similar to the GBuilder plugin.

You need to implement a Python-based class as follows:

```
class CheckerOpPlugin:
    def get_op_type(self):
        return None
    def check(self, node):
        errors = []
        return errors
    def infer_shape(self, node):
        return []
```

The base class is defined in `AIPUBuilder.core._checker`.

Perform the following three steps to add a checker plugin:

1. Register your operator type (if it does not exist) and tell the checker which operator you want to support.
2. Check whether the format of this operator is correct.
3. Infer the shapes of output tensors according to the inputs and other parameters.

_____ Note_____

Both the GBuilder plugin and checker plugin need to check parameters. The key difference is whether it is related to the specification and backend implementation. The part that is related to the specification and backend implementation should be checked in the GBuilder plugin. Other parts should be checked in the checker plugin.

Register and get_op_type

Register your operator type as a GBuilder plugin if the operator type does not exist, and tell the checker which operator you want to support.

For example, to support an operator type named `MyOwnOpType`, assume that you have registered it before, ~~you can implement get_op_~~type as follows:

```
def get_op_type(self):
    return OpType.MyOwnOpType
```

Check_IR

In this function, you need to check whether the format of parameters is supported. The return value of this function is a list, and the element of the list is a class which is named IRError. If one of the parameters is not matched, you need to construct an IRError and push the IRError into the list.

Type, key, and message are required to construct an IRError which is defined in `AIPUBuilder.core._checker`.

For example, if you just check the number of input tensors, you can implement check_IR as follows:

```
def check(self, node):
    errors = []
    if not hasattr(node, "inputs"):
```

```
            errors.append(IRError(IRErrorType.MissingParam,
                                  "layer_bottom",
                                  "Inputs required"))
        elif len(node.inputs) != 1:
            errors.append(IRError(IRErrorType.InvalidParam,
                                  "layer_bottom",
                                  "Invalid inputs num " +
                                   str(len(node.inputs)) +
                                   ", expected 1"))
                                     return errors
```

Infer_shape

In this function, you need to infer the shapes of output tensors according to the inputs and other parameters. The checker will compare the returned output shapes with the original output shapes automatically, and then report an error if the results mismatch.

For most operators, output shapes can be inferred by inputs and other parameters.

For some special operators of which the output shapes are not inferred, just the original output shapes are returned.

The following is an example of infer_shape of an element-wise operator:

```
    def infer_shape(self, node):
      inp = node.inputs[0]
      # the element wise output shape is same as input shape
      shape = TensorShape(inp.shape)
      shape.layout = DataLayout.NHWC
      return [shape]
```

The following is a simple example with all codes for the AddOne operator:

```
from AIPUBuilder.plugin_loader import *
from AIPUBuilder.core import *
from AIPUBuilder.core._checker import *
NewOp = register_optype("AddOne")


@register_plugin(PluginType.Checker, 0)
class AddoneCheckerPlugin(CheckerOpPlugin):
    def get_op_type(self):
        return OpType.AddOne


    # return error list
    def check(self, node):
        errors = []
```

```
            errors.append(IRError(IRErrorType.InvalidParam,"stride_x",
"invalid params"))
            return errors


        # return shape list
        def infer_shape(self, node):
            inp = node.inputs[0]
            shape = TensorShape(inp.shape)
            shape.layout = DataLayout.NHWC
            return [shape]
```

7.6 Tutorial

An abs operator is used as an example in the out-of-the-box example to show you how to develop a customized operator on the NPU. You can find the example in the `AI610-SDK-1003/customized-op-example/` directory.

Appendix A
NPU auxiliary tools

This appendix describes the NPU auxiliary tools.

It contains the following section:

A.1 AIPU dumper

The AIPU dumper (`aipudumper`) is a tool that enables you to modify the data location in DDR manually.

- For Zhouyi Z2/Z3/X1, it reads the AIPU bin file.
- For Zhouyi X2, it reads the AIPU elf bin file.

It also reads the input bin file, updates the data location by the address configuration file, and then runs in the simulator or emulator.

The following figure shows the input and output files for `aipudumper`.

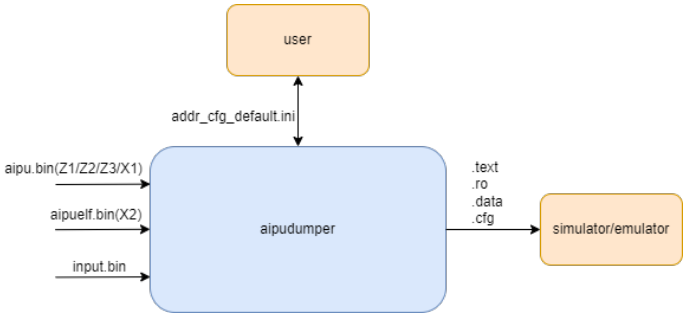

Figure A-1 Input and output files for aipudumper

Usage:

```
$aipudumper aipu_bin_path [-i inputs] [other options]
```

The following shows the simple usage of aipudumper:

- To generate the default address configuration file:

  ```
  $aipudumper aipu.bin -i input.bin --dump_addr_cfg
  ```

- To apply the updated address configuration file and run in the simulator:

  ```
  $aipudumper aipu.bin -i input.bin --load_addr_cfg addr_cfg_default.ini --
  simulator path/to/simulator
  ```

The following figure shows the format of the `addr_cfg_default.ini` file for Zhouyi Z2/Z3/X1.

Figure A-2 Format of the default address configuration file for Zhouyi Z2/Z3/X1

The following figure shows the format of the `addr_cfg_default.ini` file for Zhouyi X2.

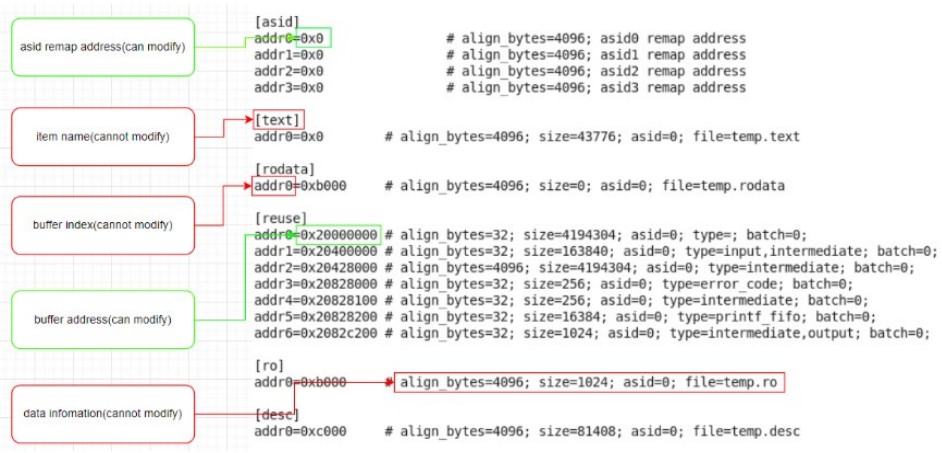

Figure A-3 Format of the default address configuration file for Zhouyi X2

For the `[asid]` section, you can set the ASID remap address in the corresponding `addr` item. For other sections, the `addr` item is the relative address. Therefore, the absolute address is `addr` + `asidx` remap address.

Each buffer is located in a fixed ASID region, for example, the buffer `addr3` in `[reuse]` is located in `asid0`. Therefore, the absolute address of this buffer is `addr3` in `[reuse]` + `addr0` in `[asid]`.

The following describes the options for the tool:

- `-v` [optional]: Print the tool version.
- `-h` [optional]: Print the help message.
- `-i` [optional]: The input files for the model.
- `-o` [optional]: Specify the output file name. By default, it is `output.bin`. If there are several outputs, the files will be added a suffix 1, 2, … as the second, third, … output.
- `--dump_addr_cfg` [optional]: Dump the default address configuration file. It is automatically assigned by aipudumper based on the AIPU bin file.
- `--load_addr_cfg` [optional]: Load the address configuration file.
- `--simulator` [optional]: Specify the simulator executable path.
- `--multi_core` [optional]: For Zhouyi X2 only, specify the number of cores. By default, it is single core.
- `--disable_gm` [optional]: For Zhouyi X2 only, disable global memory optimization.
- `--multi_batch` [optional]: For Zhouyi X2 only, specify the number of batches. By default, it is 1 for a single batch.
- `--ocm_addr` [optional]: For Zhouyi X2 only, set the SoC SRAM or *On-Chip Memory* (OCM) address.

Note

The OCM address range should be within the address range of ASID 0.

- `--ocm_size` [optional]: For Zhouyi X2 only, set the SoC SRAM or *On-Chip Memory* (OCM) size. The unit is KBytes. Be default, it is 0.