TE Custom Operator Development Guide (Using CLI)

Issue 01

Date 2020-05-30

HUAWEI TECHNOLOGIES CO., LTD.

Contents

1 Before You Start

1.1 Overview

Ascend 310 provides a deep-learning framework that converts models under open-source frameworks such as Caffe and TensorFlow into models supported by Ascend 310. During model conversion, if operators in the models are not implemented in the built-in operator library, an error is reported during the conversion. In this case, the operators that are not implemented need to be customized. The custom operators can be added to the custom operator library to allow successful model conversion. You can also optimize existing operators in the current model, add the optimized operators to the custom operator library, and load the optimized operators for model conversion.

Ascend 310 provides the Tensor Engine (TE) operator development framework for developing custom operators. TE is a custom operator development framework based on the Tensor Virtual Machine (TVM). It provides the DSL language based on the Python syntax for developing custom operators. Custom TE operators can run on the AI CPU and AI Core.

This document describes the development process of a custom Tensor Engine (TE) operator in command line interface (CLI) mode by using the custom caffe_reduction_layer operator (which is an extended built-in reduction operator in the Caffe network model LeNet-5) as an example. The process includes how to develop a custom TE operator, verify the operator, register the operator, and load the custom operator for model conversion.

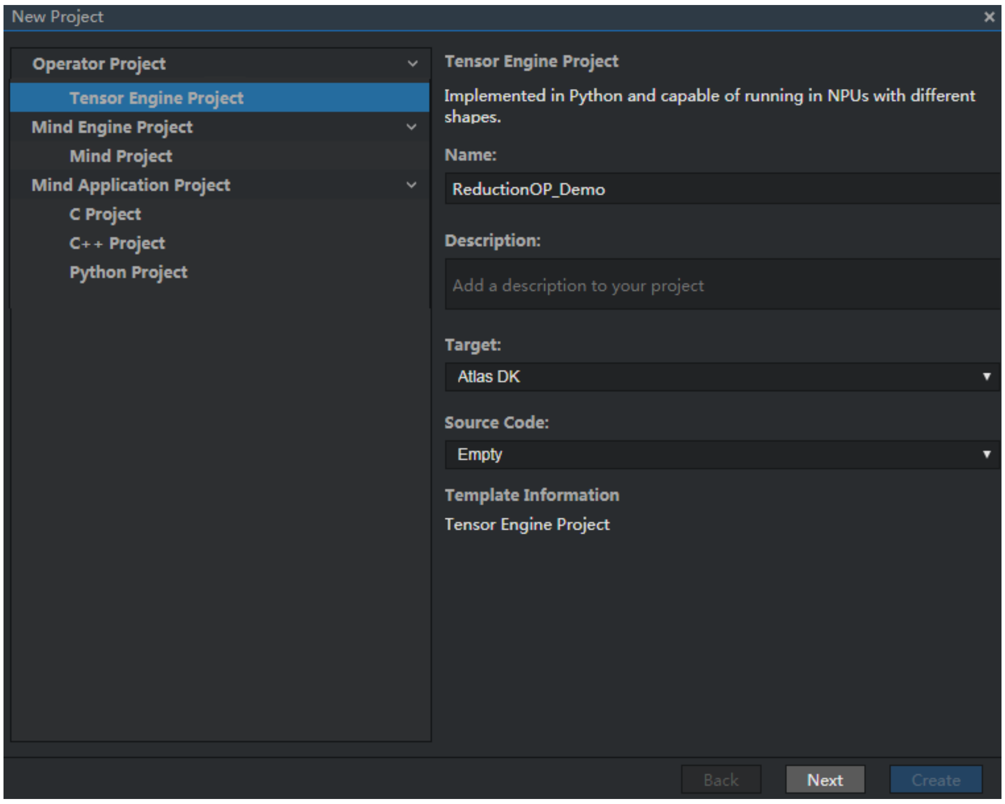
Table 1-1 describes the custom TE operator development modes supported by Ascend 310. This document describes only the development in command line interface (CLI) mode.

Table 1-1 Custom TE operator development modes supported by Ascend 310

Development Mode	Tool Dependency	Description	Compilation Mode	Reference Document
CLI	This mode does not depend on Mind Studio. It only requires the device development kit (DDK) installation. **NOTE** The DDK provides the NPU-based digital development kit that contains project code samples and integrates related dependent libraries and header files. You can compile project files by using **Makefile**.	There is no UI or automatically generated framework code. The implementation code of all operators and operator plug-in code need to be developed by users or modified based on code samples in the DDK.	The **Makefile** file is compiled based on the DDK code sample.	*TE Custom Operator Development Guide*

Development Mode	Tool Dependency	Description	Compilation Mode	Reference Document
Mind Studio	This mode depends on Mind Studio. The following figure shows the operation entry.	The framework code is automatically generated based on the model file imported during project creation and selected operators. You only need to pay attention to the operator implementation code.	The **Makefile** file does not need to be compiled because it is generated based on the configurations on Mind Studio.	*Ascend 310 TE Custom Operator Development Guide (Mind Studio)*

Figure 1-1 Tensor engine project creation



Tensor Engine Project

Implemented in Python and capable of running in NPUs with different shapes.

Name:

ReductionOP_Demo

Description:

Add a description to your project

Target:

Atlas DK

Source Code:

Empty

Template Information

Tensor Engine Project

1.2 Intended Audience

This document is intended for developers who program operators using Ascend 310. It aims to help you:

- Understand the end-to-end (E2E) development process of a custom TE operator, including developing a custom operator and integrating it into the network to run on E2E applications.
- Learn the operator development and operator plug-in development principles by referring to the reduction operator sample in this document, and be able to develop other custom operators.

To better understand this document, you should have:

- Capability of developing Python/C++/C language programs
- Good understanding of mathematical expressions
- Good understanding of machine learning and deep learning
- Good understanding of the Ascend 310 architecture
- Good understanding of TVM and open source framework Caffe

2 Precautions for Operator Development

Precautions for Developing Custom Operators for an SSD Network with the Post-Processing Node

The PriorBox operator and detection output operator of the SSD network are automatically integrated during model conversion. If the implementation of the PriorBox operator of the SSD network has been customized, model conversion will fail due to the integration failure. In this situation, you need to delete the detection output operator from the network model file, and customize a post-processing node in engine orchestration to implement the function of the detection output operator.

Others

- Do not modify the implementation of the built-in operators of the framework (all files are saved in **ddk/include/inc/custom**). Otherwise, exceptions may occur. For example, the system fails to be started or the model cannot be converted.
- When developing an operator plug-in, customers shall be responsible for the source code to avoid backdoor implantation.

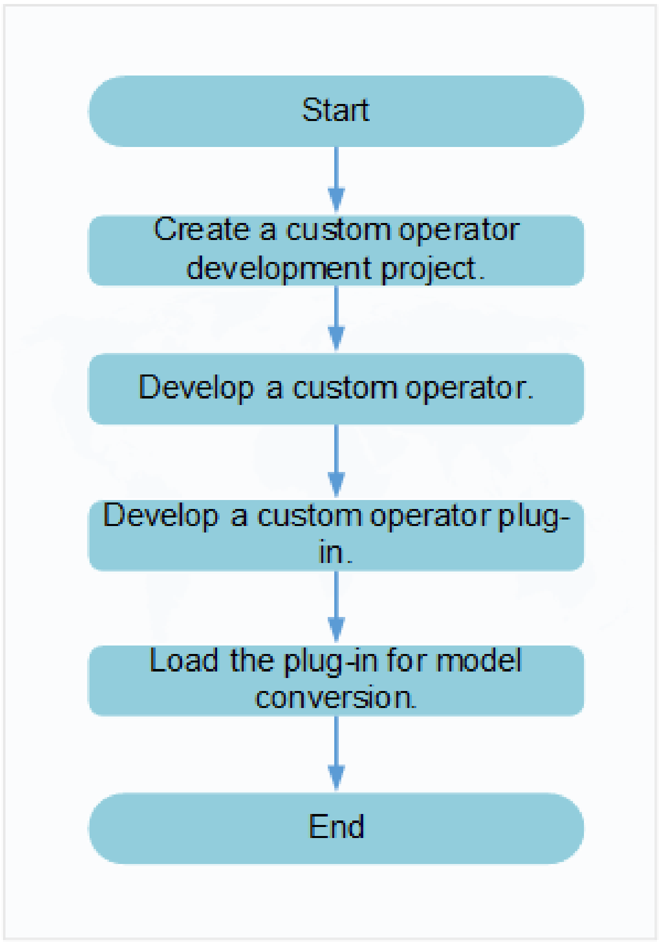
3 Environment Preparation

The DDK has been deployed. It provides developers with an Ascend 310-based digital development kit that contains project code samples and integrates related dependent libraries and header files. You can compile project files by using **Makefile**.

4 Overall Development Process

Figure 4-1 shows the development process of a custom TE operator.

Figure 4-1 Developing a custom TE operator


Start
Create a custom operator development project.
Develop a custom operator.
Develop a custom operator plug-in.
Load the plug-in for model conversion.
End

The operating procedure is as follows:

1. Create a custom operator development project.
2. Customize unimplemented operators, including developing operator code, and compiling, running, and verifying a single operator.

3. Develop the custom operator plug-in and register the custom operator with the Framework. After the operator plug-in is compiled, the *.so plug-in file of the custom operator is generated.
4. Load the *.so plug-in file to identify the custom operator and convert the model.

5 Setting Environment Variables

Step 1 Log in to the DDK server as the DDK installation user *ascend*.

Step 2 Set the environment variables.

- In the Atlas 200 DK environment, run the following commands in the CLI to set environment variables:

```
export DDK_HOME=$HOME/tools/che/ddk/ddk
export LD_LIBRARY_PATH=$DDK_HOME/uihost/lib/
export PYTHONPATH=$DDK_HOME/site-packages/te-0.4.0.egg:$DDK_HOME/site-
packages/topi-0.4.0.egg
export PATH=$PATH:$DDK_HOME/toolchains/ccec-linux/bin
export TVM_AICPU_LIBRARY_PATH=$DDK_HOME/uihost/lib/:$DDK_HOME/uihost/
toolchains/ccec-linux/aicpu_lib
export TVM_AICPU_INCLUDE_PATH=$DDK_HOME/include/inc/tensor_engine
export TVM_AICPU_OS_SYSROOT=/usr/lib/aarch64-linux-gnu
```

- In the ASIC environment, run the following commands in the CLI to set environment variables:

```
export DDK_HOME=$HOME/tools/che/ddk/ddk
export LD_LIBRARY_PATH=$DDK_HOME/uihost/lib/
export PYTHONPATH=$DDK_HOME/site-packages/te-0.4.0.egg:$DDK_HOME/site-
packages/topi-0.4.0.egg
export PATH=$PATH:$DDK_HOME/toolchains/ccec-linux/bin
export TVM_AICPU_LIBRARY_PATH=$DDK_HOME/uihost/lib/:$DDK_HOME/uihost/
toolchains/ccec-linux/aicpu_lib
export TVM_AICPU_INCLUDE_PATH=$DDK_HOME/include/inc/tensor_engine
export TVM_AICPU_OS_SYSROOT=$DDK_HOME/uihost/toolchains/aarch64-linux-
gcc6.3/sysroot
```

NOTE

Change the path of **DDK_HOME** to the actual installation path of the DDK.

You can write the commands for setting environment variables to the operator customization script for future use.

----End

6 Creating a Custom Operator Development Project

Step 1 Log in to the DDK server as the DDK installation user *ascend*.

Step 2 Create a workspace directory.

```
mkdir $HOME/tools/projects
```

Step 3 Create a directory for custom projects.

```
$ mkdir $HOME/tools/projects/customop_te
```

Create an operator code directory.

```
mkdir $HOME/tools/projects/customop_te/operator
```

Create a directory for storing operator plug-in code.

```
mkdir $HOME/tools/projects/customop_te/plugin
```

Step 4 Copy the sample code in the DDK to the directory where the operator development project is stored.

NOTE

- This example describes how to develop a custom operator by using the code of the built-in reduction operator in the DDK and its plug-in. You can create related files for editing.
- In the example, the installation path of the DDK is **$HOME/tools/che**. If the actual installation path is different from the example path, replace the path in the example command with the actual installation path.

- Copy the sample code file **reduction.py** of the reduction operator and the sample data generation script **data_gen.py** used for operator verification in the DDK to the directory where the operator code of the custom project is stored.

  ```
  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/python/reduction.py
  $HOME/tools/projects/customop_te/operator/
  ```

  ```
  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/python/data_gen.py
  $HOME/tools/projects/customop_te/operator/
  ```

- Copy the sample code file of the reduction operator plug-in and the **Makefile** file in the DDK to the directory where the operator plug-in code of the custom project is stored.

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/caffe_reduction_layer.cpp $HOME/tools/projects/customop_te/plugin/

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/Makefile $HOME/tools/projects/customop_te/plugin/

- Copy the Caffe network model file in the DDK used in this example to the custom project directory.

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/model/ $HOME/tools/projects/customop_te/

Table 6-1 shows the directory structure of the custom operator development project.

Table 6-1 Directory structure

Sample Directory	File	Description
customop_te/operator	reduction.py	Sample code of the reduction operator
	data_gen.py	File generated based on reduction operator sample data
customop_te/plugin	caffe_reduction_layer.cpp	Sample code of the reduction operator plug-in (needs to be modified)
	Makefile	Compilation rule file of the reduction operator plug-in (needs to be modified)
customop_te/model	deploy_mylenet-1.prototxt	Original model file of the MyLeNet network
	mylenet-1.caffemodel	Pre-trained model file of the MyLeNet network

----End

7 Developing a Custom Operator

7.1 Operator Basics

A deep learning algorithm consists of multiple compute units, that is, operators (Ops). In Caffe, an operator describes the computation logic of the layer, for example, the convolution that performs convolution and the Fully-Connected (FC) layer that multiplies the input by a weight matrix.

The following introduces some basic terms about operators.

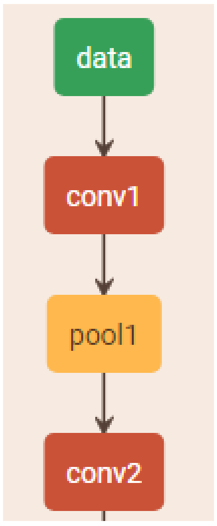
Operator Type

Type of an operator. For example, the type of a convolution operator is convolution. A network can have different operators of the same type.

Operator Name

The name of an operator identifies the operator on a network. An operator name must be unique on a network. As shown in the following figure, conv1, pool1, and conv2 are the names of operators on the network. They are of the same type, convolution. Each indicates a convolution operation.

Figure 7-1 Network topology



Tensor

As data in a TE operator, the tensor includes the input data and output data. **TensorDesc** (the tensor descriptor) describes the input data and output data. **Table 7-1** describes the attributes of the **TensorDesc** struct.

Table 7-1 Description of the TensorDesc attributes

Attribute	Definition
name	Indexes a tensor. The **name** of each tensor must be unique.
shape	Specifies the shape of a tensor, for example, **(10)**, **(1024,1024)**, or **(2,3,4)**. For details, see **Shape**. Default value: N/A Format: **(i1, i2, ... i*n*)**, where, **i1** to **i*n*** are positive integers.
dtype	Specifies the data type of a tensor object. Default value: N/A Value range: **float16**, **int8**, **int16**, **int32**, **uint8**, **uint16**, **bool** **NOTE** • The supported data types vary with the operation. For details, see the *TE API Reference*. • TE APIs support both the float16 and float32 types. However, OMG converts the float32 type to the float16 type during model conversion. Therefore, the current version does not support the float32 type for custom operator development.
format	Specifies the data layout format. For details, see **Format**.

- Shape

  The shape of a tensor is described in the format of **(D0, D1, ..., D*n* – 1)**, where, **D0** to **D*n*** are positive integers.

  For example, the shape (3, 4) indicates a 3 x 4 matrix, where the first dimension has three elements, the second dimension has four elements.

  The number count in the bracket equals to the dimension count of the tensor. The first element of shape depends on the element count in the outer bracket of the tensor, and the second element of shape depends on the element count in the second bracket of the tensor starting from the left, and so on.

Table 7-2 Tensor shape examples

Tensor	Shape
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

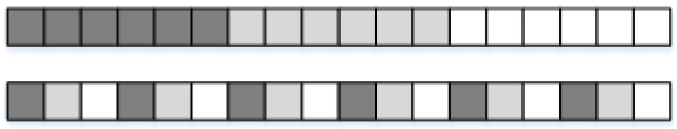
- Format

  In the deep learning framework, n-dimensional data is stored by using an n-dimensional array. For example, a feature graph of a convolutional neural network is stored by using a four-dimensional array. The four dimensions are N, H, W and C, which stand for batch, height, width, and channels, respectively.

  Data can be stored only in linear mode because the dimensions have a fixed order. Different deep learning frameworks store feature graph data in different sequences. For example, data in Caffe is stored in the order of [Batch, Channels, Height, Width], that is, NCHW. Data in TensorFlow is stored in the order of [Batch, Height, Width, Channels], that is, NHWC.

  As shown in **Figure 7-2**, an RGB picture is used as an example. In the NCHW order, the pixel values of each channel are clustered in sequence as RRRGGGBBB. In the NHWC order, the pixel values are interleaved as RGBRGBRGB.

Figure 7-2 NCHW and NHWC



To improve data access efficiency, the tensor data is in the Ascend AI software stack is stored in the 5D format NC1HWC0. C0 is closely related to the micro architecture and is equal to the size of the matrix computing unit in AI Core. For the FP16 type, C0 is **16**; for the INT8 type, C0 is **32**. The C0 data needs to be stored consecutively. That is, C1=(C+C0-1)/C0. If the division result is not an integer, the last data record is padded with zeros to be aligned with C0.

The NHWC-to-NC1HWC0 conversion process is as follows:

a. Split the NHWC data along dimension C into C1 pieces of NHWC0.

b. Arrange the C1 pieces of NHWC0 in the memory consecutively into NC1HWC0.

Operator Attributes

Different operators have different attribute values. The following describes some common operator attributes.

- Axis

  The axis represents the subscript of a dimension of a tensor. For a two-dimensional tensor with five rows and six columns, that is, with shape (5, 6), axis 0 represents the first dimension in the tensor, that is, the row; axis 1 represents the second dimension of tensor, that is, the column.

  For example, for tensor [[[1,2],[3,4]], [[5,6],[7,8]]] with shape (2, 2, 2), axis 0 represents data in the first dimension, that is, matrices [[1,2],[3,4]] and [[5,6],[7,8]], axis 1 represents data in the second dimension, that is, arrays [1,2], [3,4], [5,6], and [7,8], and axis 2 indicates the data in the third dimension, that is, numbers 1, 2, 3, 4, 5, 6, 7, and 8.

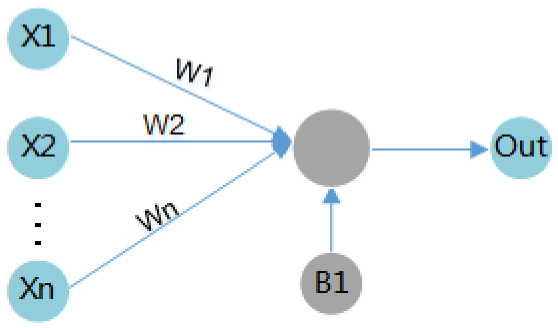
  Negative **axis** is interpreted as a dimension counted from the end.

- Bias

  The bias, along with the weight, is a linear component to be applied to the input data. It is applied to the result of multiplying the weight by the input data.

  As shown in **Figure 7-3**, assume that the input data is **X1**, the associated weight is **W1**, and the bias is **B1**. After the data passes the compute unit, the data changes to **(X1 * W1 + B1)**.

  Figure 7-3 Bias computation example

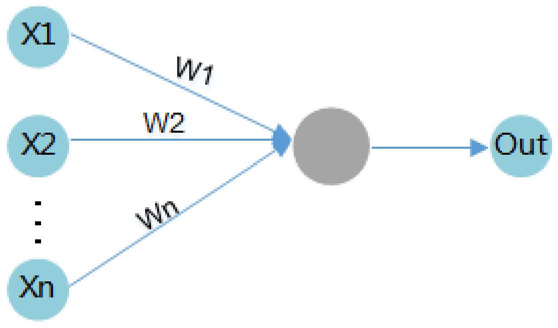
  

- Weight

  The input data is multiplied by a weight value in the compute unit. For example, if an operator has two inputs, an associated weight value is allocated to each input. Generally, it is considered that relatively important data is assigned a relatively greater weight value, and a weight value of zero indicates a specific feature can be ignored.

  As shown in **Figure 7-4**, assume that the input data is **X1** and the associated weight is **W1**. After the data passes the compute unit, the data changes to **(X1 * W1)**.

Figure 7-4 Weight computation example



Sample Operator: Reduction

Reduction is a Caffe operator that reduces the specified axis and its subsequent axes of a multi-dimensional array.

- Attributes of the reduction operator
  - **ReductionOp**: operation type. Four operation types are supported.

Table 7-3 Operation types supported by the reduction operator

Operator Type	Description
SUM	Sums the values of all reduced axes.
ASUM	Sums the absolute values of all reduced axes.
SUMSQ	Squares the values of all reduced axes and then sums them.
MEAN	Averages the values of all reduced axes.

  - **axis**: first axis to reduce. The value range is [–N, N – 1].

    For example, for an input tensor with shape (5, 6, 7, 8):

    - If axis = 3, the shape of the output tensor is (5, 6, 7).
    - If axis = 2, the shape of the output tensor is (5, 6).
    - If axis = 1, the shape of the output tensor is **(5)**.
    - If axis = 0, the shape of the output tensor is **(1)**.
  - **coeff**: scalar, scaling coefficient for output. The value **1** indicates that the output is not scaled.
- Data of the reduction operator
  - Input data

    The input contains the tensor data and tensor description.

    Input data: tensor **x**

    The description of tensor **x** contains the following attributes:

Table 7-4 Input of the reduction operator

Input Parameter	Description
x	Name of the input tensor, whose shape is determined by the **shape** parameter
shape	Shape of the input data, N-dimensional
dtype	Type of the input data, either Indicates the data type. The value is float16 or float32.

– Output data

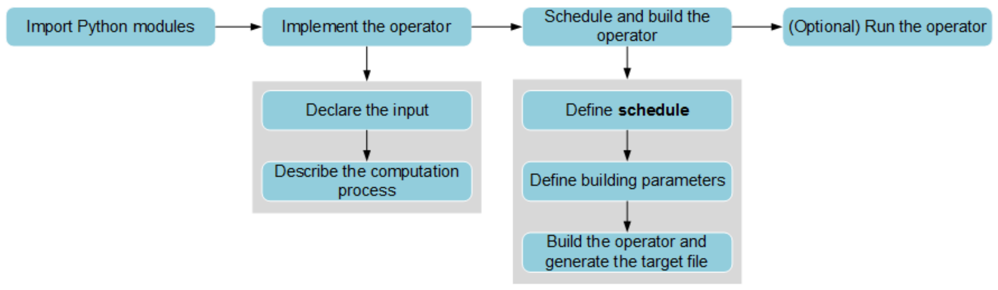
y: tensor of the identical data type as input **x**, whose shape is determined by the input tensor shape and the specified **axis**

7.2 Implementing an Operator

7.2.1 Procedure

The code of a TE operator is developed in Python. **Figure 7-5** shows the implementation procedure.

Figure 7-5 Implementation process of a TE custom operator


Import Python modules
Implement the operator
Schedule and build the operator
(Optional) Run the operator
Declare the input
Describe the computation process
Define **schedule**
Define building parameters
Build the operator and generate the target file

NOTE

- The supported input data types for custom operators are as follows: float16, int8, int16, int32, uint8, uint16, and bool.
  - The supported data types vary with the operation. For details, see the *TE API Reference*.
  - TE APIs support both the float16 and float32 types. However, OMG converts the float32 type to the float16 type during model conversion. Therefore, the current version does not support the float32 type for custom operator development.
- TE provides sample code of some custom operators for user reference or direct use in **ddk/site-packages/topi-0.4.0.egg/topi/cce** in the DDK installation directory.

7.2.2 Importing Python Modules

Import the Python modules provided by the Ascend AI software stack. The sample code is provided as follows.

```
import te.lang.cce
from te import tvm
from topi import generic
```

In the preceding information:

- **te.lang.cce**: introduces the SDL APIs supported by TE, including common ones such as **vmuls**, **vadds**, and **matmul**.

  For details about the interface definition, see the Python functions in the **/site-packages/te-0.4.0.egg/te/lang/cce/** directory in the DDK installation path. For details about how to use the Python functions, see the *TE API Reference*.
- **te.tvm**: introduces the code generation mechanism of the TVM.

  For details about the interface definition, see the Python functions in the **/site-packages/te-0.4.0.egg/te/tvm** directory in the DDK installation path. For details about how to use the Python functions, visit **https://docs.tvm.ai/**.
- **topi.generic**: provides the API for automatic operator scheduling.

  For details about the interface definition, see the Python functions in the **/site-packages/topi-0.4.0.egg/topi/generic** directory in the DDK installation path. For details about how to use the Python functions, see the *TE API Reference*.

7.2.3 Implementing an Operator

Function Definition for Operator Implementation

As described below, the implementation function of an operator contains the input tensor shape, data type, operator attributes, kernel name, and build and print configurations. This function is called by plug-in code and is executed when OMG converts the model.

```
def operationname(shape, dtype, attribute1, attribute2, ... , kernel_name="KernelName", need_build=True,
need_print=False)
```

In the preceding information:

- **shape**: input tensor shape. If an operator has multiple input tensors and each tensor has a unique shape, multiple shapes need to be defined as placeholders for the tensors. If multiple input tensors have a same shape, define one shape.
- **dtype**: data type of the input tensor.
- **attribute1**, **attribute2**, ...: operator attributes. Edit the code based on the operator definition.
- **kernel_name**: name of the operator in the kernel, that is, the name of the generated binary file. The value is user-defined and unique. The value can contain only uppercase letters, lowercase letters, digits, and underscores (_). Enter a maximum of 200 characters starting with a letter or underscore (_).
- **need_build**: build enable, either **True** or **False**

- **need_print**: intermediate representation (IR) print enable, either **True** or **False**

Examples:

Reduction operator:

```
def reduction(shape, dtype, axis, operation, coeff, kernel_name="Reduction", need_build=True,
need_print=False)
```

Matmul operator:

```
def matmul(shape_a, shape_b, dtype, kernel_name="matmul", trans_a=False,
trans_b=False,need_build=False, need_print=False):
```

Operator Implementation Logic

The TE operator implementation logic is summarized as follows:

Define placeholders for input tensors, and then call the SDL interfaces in **te.lang.cce** to describe the computation process. The following is a code example:

```
data = tvm.placeholder(shape, name="data_input", dtype=inp_dtype)
with tvm.target.cce():
    cof = coeff
   data_tmp_input = te.lang.cce.vmuls(data, cof)    // Process the scaling parameter and multiply the input
tensor by a scalar.
    tmp = data_tmp_input
    res_tmp = te.lang.cce.sum(tmp, axis=axis)    // Perform the summation operation on the axis.
    res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag=True)   // Convert the data type.
```

In the preceding information:

- **data** indicates the input tensor, which is defined by using the **placeholder** interface of the TVM. A Tensor object is returned, indicating a group of input data.

  If the operator has multiple input tensors, multiple tensor objects need to be defined. For example:

  ```
  tensor_a = tvm.placeholder(shape_a, name='tensor_a', dtype=dtype)
  tensor_b = tvm.placeholder(shape_b, name='tensor_b', dtype=dtype)
  ```

- **vmuls** (vector multiplication) and **sum** (summation) constitute the intermediate computation logic.
- **cast_to** is used to convert the data type. The output tensor must be of the identical data type as the input tensor. If the data type is changed during computation, you need to use the **cast_to** interface to convert the data type of the output tensor to that of the input tensor.

  For example: If the data type of the input tensor is int8, it is converted to float16 for the vmuls operation. In this case, the **cast_to** interface must be called to convert the data type of the output tensor from float16 to int8 after the computation logic is complete. **vmuls** converts int8 values to float16, padding the decimal part with zeros. Therefore, **f1628IntegerFlag** is set to **True**. The sample code is as follows:

  ```
  res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag=True)
  ```

  For details about how to use the **te.lang.cce.cast_to** API, see **Compute APIs** in *TE API Reference*.

- **res** indicates the output tensor, of the identical data type as the input tensor.

Before implementing the operator logic, you can customize the code for pre-processing the input data. The sample code is as follows:

```
# basic check
check_list = ["float16", "float32"]
if not (dtype.lower() in check_list):
    raise RuntimeError("Reduction only support %s while dtype is %s" % (
        ",".join(check_list), dtype))
reduction_op = ("SUM", "ASUM", "SUMSQ", "MEAN")

# axis parameter check
if type(axis) != int:
    raise RuntimeError("type of axis value should be int")
if axis >= len(shape) or axis < -len(shape):
    raise RuntimeError(
        "input axis is out of range, axis value can be from %d to %d" % (
            -len(shape), len(shape) - 1))
# operation parameter check
if operation not in reduction_op:
    raise RuntimeError("operation can only be one of SUM, ASUM, SUMSQ , MEAN")
# coeff parameter check
if type(coeff) != int and type(coeff) != float:
    raise RuntimeError("coeff must be a value")
# Preprocess
if axis < 0:
    axis = len(shape) + axis
shape = list(shape)
shape1 = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])]
inp_dtype = dtype.lower()
```

7.2.4 Scheduling and Building an Operator

As shown in the following code, after the computation logic is defined, the Auto schedule mechanism performs auto scheduling. You can check the computation IR through the print mechanism provided by the TVM. The configuration information includes the print switch status, build switch status, operator name in the kernel, and input and output tensors.

```
sch = generic.auto_schedule(res)
config = {
    "print_ir": need_print,
    "need_build": need_build,
    "name": kernel_name,
    "tensor_list": [data, res]
}
te.lang.cce.cce_build_code(sch, config)
```

- Use the **auto_schedule** interface of **generic** to perform auto scheduling (defining **schedule**). The argument of the **auto_schedule** interface is the output tensor of the operator.

  The schedule object defines how to efficiently execute the described computation process on hardware. That is, the related computations are mapped to the corresponding instructions on a hardware device. A schedule object contains an IR, which uses code similar to pseudo code to describe a computation process. You can print the object by using the parameter **need_print**.
- The input and output tensors are stored in **tensor_list**. The input and output tensors must be arranged in the input and output sequences of the operator.

  For example: **"tensor_list": [tensor_a, tensor_b, res]**, where, **tensor_a** and **tensor_b** are the input tensors, and **res** is the output tensor.
- The **cce_build_code** interface provided by **te.lang.cce** is used to build the operator based on scheduling and configuration. During operator building

when OMG converts the model, a dedicated kernel is built based on the input data shape, type, and operator parameters.

– **sch**: schedule object to be calculated by the generated operator.

– **config**: map configured by compilation parameters.

After building, an operator target file *.o (the running target of the operator is AI Core) or *.so (the running target of the operator is AI CPU) and an operator description file *.json are generated.

7.2.5 Running an Operator

After the custom operator code is written, you can append the operator calling statement to the *.py code of the operator as follows. Construct the input data by referring to **7.4 Operator Running Verification** and use it to check the operator execution result.

For example:

```
if __name__ == "__main__":
    reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2,kernel_name = "Reduction")
```

7.3 Code Examples

Open the copied operator sample file **reduction.py**.

For details about the code description, see the following comments:

```
#coding=utf-8

import te.lang.cce
from te import tvm
from topi import generic
from topi.cce import util
def reduction(shape, dtype, axis, operation, coeff, kernel_name="Reduction",
                          need_build=True, need_print=False):
    """
    Reduce a tensor on a certain axis, and scale output with coeff
    Parameters
    ----------
    shape : shape of data
    dtype : source data type, only support float16, float32
    axis : the first axis to reduce, may be negative to index from the end (e.g., -1 for the last axis).
          If axis == 0, the output Blob always has the empty shape (count 1), performing reduction across the
entire input.
    op : can only be one of "SUM, ASUM (sum of abs), SUMSQ (sum of sqr), MEAN"
    coeff : scale for output
    kernel_name : cce kernel name, default value is "cce_reductionLayer"
    need_buid : if need to build CCEC kernel, default value is False
    need_print : if need to print the ir, default value is False
    Returns
    -------
    None
    """
    # Basic parameter verification
    #shape parameter verification. The check_shape_rule() function definition is stored in ddk/ddk/site-
packages/topi-0.4.0.egg/topi/cce/util.py.
    util.check_shape_rule(shape)
    check_list = ["float16", "float32"]
    if not (dtype.lower() in check_list):
        raise RuntimeError("Reduction only support %s while dtype is %s" % (",".join(check_list), dtype))

    reduction_op = ("SUM", "ASUM", "SUMSQ", "MEAN")
    # Axis parameter verification
    if type(axis) != int:
        raise RuntimeError("type of axis value should be int")
    if axis >= len(shape) or axis < -len(shape):
        raise RuntimeError(
            "input axis is out of range, axis value can be from %d to %d" % (-len(shape), len(shape) - 1))
   # op parameter verification
    if operation not in reduction_op:
        raise RuntimeError("op can only be one of SUM, ASUM, SUMSQ , MEAN")
   # coeff parameter verification
    if type(coeff) != int and type(coeff) != float:
        raise RuntimeError("coeff must be a value")
   # Parameter pre-processing
    if axis < 0:
        axis = len(shape) + axis
    shape = list(shape)
    shape1 = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])]
    inp_dtype = dtype.lower()
   # Define the tensor object of the input data. The data is only used as a placeholder and no actual
memory is allocated.
    data = tvm.placeholder(shape1, name="data_input", dtype=inp_dtype)
   # Define the operator calculation process.
    with tvm.target.cce():
        if operation == "ASUM":
            data_tmp_input = te.lang.cce.vabs(data)
            cof = coeff
            tmp = te.lang.cce.vmuls(data_tmp_input, cof)
        elif operation == "SUMSQ":
            data_tmp_input = te.lang.cce.vmul(data, data)
            cof = coeff
            tmp = te.lang.cce.vmuls(data_tmp_input, cof)
        elif operation == "MEAN":
            size = shape1[-1]
            cof = float(coeff) * (size ** (-0.5))
            tmp = te.lang.cce.vmuls(data, cof)
        elif operation == "SUM":
            cof = coeff
            data_tmp_input = te.lang.cce.vmuls(data, cof)
            tmp = data_tmp_input

        #Sum up data by axis to reduce dimensions.
        res_tmp = te.lang.cce.sum(tmp, axis=axis)
        #Convert the data type.
        res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag = True)
        if operation == "MEAN":
            size = shape1[-1]
            sqrt_size = size ** (-0.5)
            res = te.lang.cce.vmuls(res_tmp, sqrt_size)
        #Generate the schedule object to be calculated by the operator.
        sch = generic.auto_schedule(res)
    #Define compilation parameters.
    config = {"print_ir": need_print,
              "need_build": need_build,
              "name": kernel_name,
              "tensor_list": [data, res]}
    #Compile the operator and generate the target file.
    te.lang.cce.cce_build_code(sch, config)

#Call the reduction operator by using the parameters: shape (2, 3, 4), datatype (float16), axis (1), op
(SUM), coeff (2), and operator name (Reduction).
if __name__ == "__main__":
    reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2,kernel_name = "Reduction")
```

7.4 Operator Running Verification

This section describes how to independently run a custom operator to verify its correctness.

7.4.1 Compiling an Operator

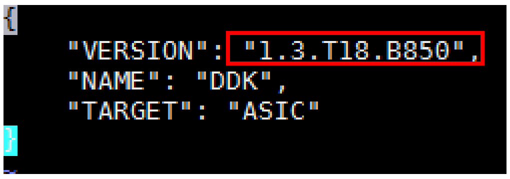
Compile the operator code to generate the operator binary file and operator description file as follows:

Step 1 Obtain the DDK version number, which is required for compiling the operator.

Check the DDK version number in **$HOME/tools/che/ddk/ddk/ddk_info**.

As shown in **Figure 7-6**, the **VERSION** field indicates the current DDK version.

Figure 7-6 Checking the DDK version number



```
{
    "VERSION": "1.3.T18.B850",
    "NAME": "DDK",
    "TARGET": "ASIC"
}
```

Step 2 Set the version number and compile the operator.

1. Run the following command in the **customop_te/operator** directory to enter the python interaction mode:

   python

2. In python interaction mode, run the following commands in sequence to set the DDK version number:

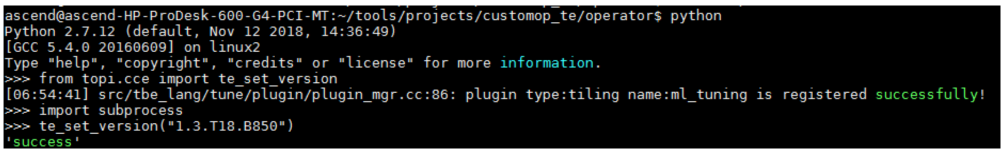
   from topi.cce import te_set_version

   import subprocess

   te_set_version("1.3.T18.B850")

   Figure 7-7 shows an example.

   Figure 7-7 Example of setting the DDK version number

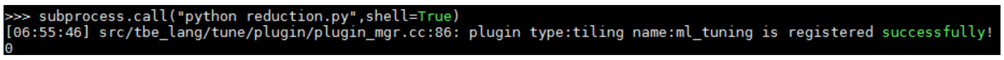
   

   ```
   ascend@ascend-HP-ProDesk-600-G4-PCI-MT:~/tools/projects/customop_te/operator$ python
   Python 2.7.12 (default, Nov 12 2018, 14:36:49)
   [GCC 5.4.0 20160609] on linux2
   Type "help", "copyright", "credits" or "license" for more information.
   >>> from topi.cce import te_set_version
   [06:54:41] src/tbe_lang/tune/plugin/plugin_mgr.cc:86: plugin type:tiling name:ml_tuning is registered successfully!
   >>> import subprocess
   >>> te_set_version("1.3.T18.B850")
   'success'
   ```

3. Run the following commands to compile the **reduction.py** file to generate the binary file and description file of the operator:

   subprocess.call("python reduction.py",shell=True)

   Figure 7-8 Compiling operator files

   ```
   >>> subprocess.call("python reduction.py",shell=True)
   [06:55:46] src/tbe_lang/tune/plugin/plugin_mgr.cc:86: plugin type:tiling name:ml_tuning is registered successfully!
   0
   ```

4. Exit the python interaction mode.

   quit(0)

Step 3 After the operator compilation is completed, the **kernel_meta** folder is generated in the current operator directory. This folder contains the operator binary file *.o (with operators to run on AI Core) or *.so (with operators to run on AI CPU) and operator description file *.json.

- The *.o or *.so file is the binary file of the operator.
- The .json file is the operator description file, which is used to define operator properties and resources required for running.

  The .json file is parsed as follows:

```
{
"binFileName":"Reduction",            // Binary file name of the generated operator
"binFileSuffix":".o",             // Extension of the generated operator binary file. For the operators to be running on AI CPU, the generated binary file name is suffixed by .so.
"blockDim":1,                         // Number of AI Cores used for calculation
"kernelName":"Reduction__kernel0",    // Name of the kernel function of the operator
"magic":"RT_DEV_BINARY_MAGIC_ELF",    // The operation target is AI Core. If the value of magic is RT_DEV_BINARY_MAGIC_ELF_AICPU, the operation target is AI CPU.
"sha256":"d5158df2ff2e64743eec7f527ddd9078b99c1d670f5adbd8b789224657ab0f91"  // Value obtained after the .o file is encrypted
}
```

----End

7.4.2 Building an Input Data File

To run a single operator, you need to build input data and save it in binary format to the operator project.

Step 1 Go to the operator code directory of the custom operator project as the DDK installation user.

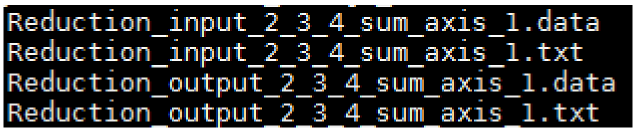
cd $HOME/tools/projects/customop_te/operator/

Step 2 Run the script compiled based on sample data to generate a sample data file for the reduction operator.

python data_gen.py reduction

The data files shown in **Figure 7-9** are generated in the operator project.

Figure 7-9 Generated sample data



```
Reduction_input_2_3_4_sum_axis_1.data
Reduction_input_2_3_4_sum_axis_1.txt
Reduction_output_2_3_4_sum_axis_1.data
Reduction_output_2_3_4_sum_axis_1.txt
```

Table 7-5 describes the data files.

Table 7-5 Description of data files

Data File	Description
Reduction_input_2_3_4_sum_axis_1.data	Indicates the input data file in binary format of the reduction operator.
Reduction_input_2_3_4_sum_axis_1.txt	Displays the binary file of the reduction operator in .txt format to facilitate result viewing.

Data File	Description
Reduction_output_2_3_4_sum_axis_1.data	Indicates the output data verification file in binary format of the reduction operator, which is used to verify whether the output of the operator is correct after the operator runs.
Reduction_output_2_3_4_sum_axis_1.txt	Displays the binary file of the reduction operator in .txt format to facilitate result viewing.

----End

7.4.3 Running a Single Operator

Step 1 Generate a single-operator compilation binary file in the DDK sample project **customop_runner**.

1. Assign the write permission to the **customop_runner** sample project.

   chmod -R +w $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/

2. Go to the **build** directory of the DDK sample project **customop_runner**.

   cd $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/build

3. Set the environment variable of **DDK_PATH**.

   export DDK_PATH=$HOME/tools/che/ddk/ddk

4. Generate the **Makefile** file.
   - For the Atlas 200 DK developer board, run the following command:

     cmake -Dtarget=OI .
   - In the ASIC environment, run the following command:

     cmake .

5. Run the **make** command to generate a binary file.

   make

   The executable file **main** and dynamic library **libcustom_engine.so** are generated in **$HOME/tools/che/ddk/ddk/sample/customop/customop_runner/out/**.

Step 2 Copy the **out** directory to the directory of the custom TE operator project.

cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/out $HOME/tools/projects/customop_te/

Step 3 Configure the input data, output data, and verification description file.

Go to the **out** folder in the directory of the custom TE operator project.

cd $HOME/tools/projects/customop_te/out

- In the **input.txt** file, configure the path and name of the input data file.

  For example:

  ```
  dataPath=../operator/Reduction_input_2_3_4_sum_axis_1.data
  ```

  If the custom operator has multiple inputs, you need to define multiple .txt files according to **input.txt**, for example, **input1.txt**, **input2.txt**, and more.

- Configure the output data in the **output.txt** file.

  For example:

  ```
  size=4
  dataPath=./output/out0.data
  dtype=1
  ```

  – **size**: expected size of the output data file, in bytes
  – **dataPath**: path and name of the output data file
  – **dtype**: output data type. The value **1** indicates **float16**, and the value **2** indicates **float32**.

- In the **expect.txt** file, configure the path and file name of the data to be generated.

  For example:

  ```
  dataPath=../operator/Reduction_output_2_3_4_sum_axis_1.data
  ```

Step 4 Copy the **operator** and **out** folders to the host of the developer board or ASIC device as the **HwHiAiUser** user. The **operator** and **out** folders must be placed in the same directory.

For example, copy **operator** and **out** folders to the **/home/HwHiAiUser/projects** directory.

Step 5 Run the operator.

Log in to the host of the developer board or ASIC device as the **HwHiAiUser** user and go to the **out** directory, for example, **/home/HwHiAiUser/projects/out**.

1. Create the **output** folder in the current directory to store the generated data file. The file path and name are defined in the **output.txt** file.

   mkdir output

2. Run the following command to assign the execution permission to the **main** file:

   chmod +x main

3. Run the following command to run a single operator and compare the operators:

   ./main -i input.txt -o output.txt -e expect.txt -b ../operator/kernel_meta/Reduction.o -p *0.2* -d *0.2* -k *Reduction__kernel0* -t 0

   In the preceding command:

   – **-i**: input data configuration file, which specifies the path and file name of the input data

     If the custom operator has multiple inputs, you need to define multiple .txt files according to **input.txt**, for example, **input1.txt**, **input2.txt**, and more.

   – **-o**: output data configuration file, which specifies the size, path, file name, and type of the output data
   – **-e**: expected data configuration file, which specifies the path and file name of the expected output data for result comparison
   – **-b**: operator binary file *.o (operator to be running on AI Core) or *.so (operator to be running on AI CPU)
   – **-p**: allowed precision deviation. The value range is [0, 1). A smaller value indicates higher precision.

– **-d**: statistical discrepancy, that is, the percentage of the data whose precision deviation is above the threshold. The value range is (0, 1). A smaller value indicates higher precision.
– **-k**: kernel name. The first letter must be in uppercase. The kernel name of the TE operator must be the same as **kernelName** in the.json file. The kernel name of the C++ operator must be the same as the object name **opetype** registered in Framework for the operator.
– **-t**: The value can be **0** (TE_AI Core operator), **1** (TE_AI CPU operator), or **2** (custom C++ operator or AI CPU operator).

  You can check whether the TE operator is running on AI Core or AI CPU based on the value of **magic** in the ****.json** file generated after the operator is compiled.

After the command is executed successfully, the **out0.data** result file and the **vertifyResult.txt** verification data comparison result file are generated in the **output** folder of the current directory.

The following is an example of the **vertifyResult.txt** file, indicating that the actual output data is consistent with the expected output data.

```
Output file ./output/out0.data compare result true
```

----End

8 caffe.proto File Operator Definition (Optional)

If a model under the Caffe open source framework is used, define the **caffe.proto** file by referring to the description in this section. If a model under the TensorFlow open-source framework is used, skip this section.

After the operator is developed, add the definition of the custom operator to the built-in **caffe.proto** file of the DDK. If the definition of this operator already exists in the built-in **caffe.proto** file, skip this section.

If there are multiple unsupported custom operators in the same model, add related operator definitions at a time by referring to this section. During the implementation of operator plug-ins, related operator parameters are read from the **caffe.proto** file based on the operator name, and then the data structures of operators are converted into those supported by the offline model supported by the Ascend AI processor.

The built-in **caffe.proto** file of the DDK is stored in **$HOME/tools/che/ddk/ddk/include/inc/custom/proto/caffe/caffe.proto**. You can modify the file and add the definition of the custom operator.

The following describes how to add the definition of the reduction operator (the definition of the reduction operator has been added to the built-in **caffe.proto** file of the DDK):

Step 1 Add the definition of the reduction operator to **LayerParameter**.

Add the definition of the reduction layer to **LayerParameter** and set its ID.

```
message LayerParameter {
...
optional ReductionParameter reduction_param = 136;   // The ID must be unique.
...
}
```

Step 2 Add the parameter definition of the reduction layer to the **caffe.proto** file.

```
message ReductionParameter {
  enum ReductionOp {          // Operation types supported by the operator
    SUM = 1;                          // Sums the values of all the axes on which the reduce operation is
performed.
    ASUM = 2;                      // Sum the calculated absolute values of all axes on which the reduce
operation is performed.
    SUMSQ = 3;                    // Sum the squared values of all axes on which the reduce operation is
performed.
    MEAN = 4;                        // Average the values of all axes on which the reduce operation is performed.
  }
  optional ReductionOp operation = 1 [default = SUM];   // Defines the operation of the operator.
  optional int32 axis = 2 [default = 0];                              // Defines the axis for which the reduce
operation is to be performed.
  optional float coeff = 3 [default = 1.0]; // coefficient for output  // Scalar, indicating the scaling multiple of
the result
}
```

----End

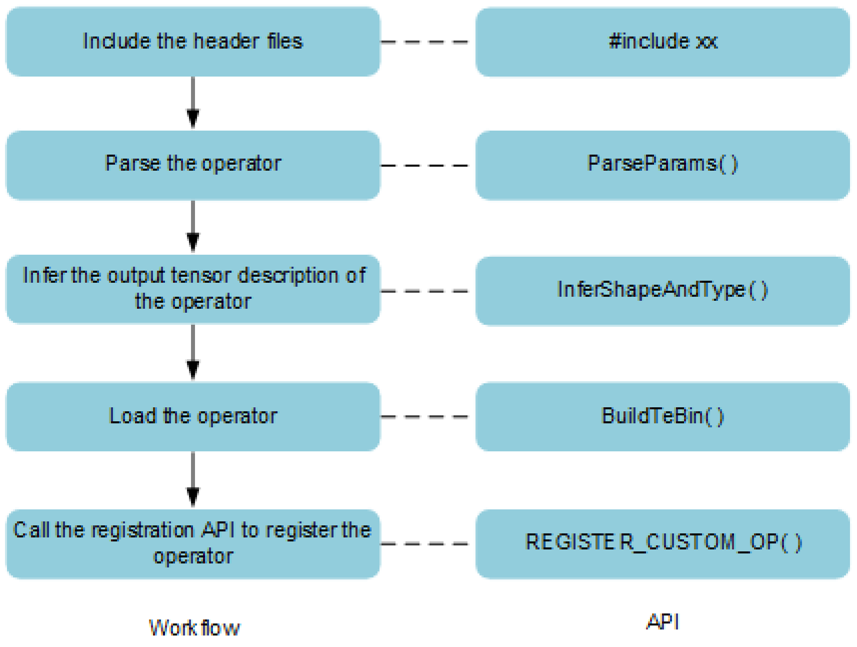
9 Developing the Plug-In of a Custom Operator

9.1 Implementing a Plug-In

9.1.1 Implementing the Plug-In

After the custom operator is developed, OMG should be able to adapt the attribute values of the custom operator to the offline model so that the custom operator can run on the offline model. Therefore, you need to develop the custom operator plug-in for parsing operator attributes, inferring the shapes and types, and registering the custom operator through the registration mechanism provided by OMG.

Figure 9-1 Implementing a custom operator plug-in



9.1.2 Including Header Files

Use the **#include** command in the header of the plug-in implementation file to include the header files related to the plug-in implementation functions in the plug-in implementation source file.

```
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
```

Table 9-1 Description of header files

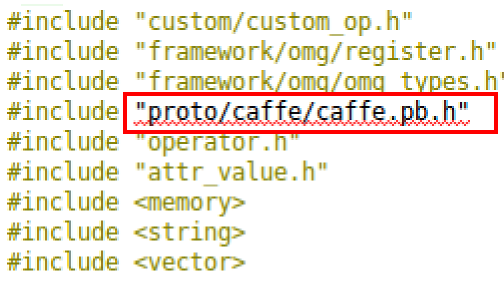
Header File	Category	Function
custom/custom_op.h	**/include/inc/custom/custom_op.h** in the DDK installation directory	User-defined functions for building, operator debugging, and operator verification can be called once this header file is included.
framework/omg/register.h	**/include/inc/framework/omg/register.h** in the DDK installation directory	Class operator registration can be used and APIs of class operator registration can be called once this hearer file is included.

Header File	Category	Function
framework/omg/ omg_types.h	**/include/inc/ framework/omg/ omg_types.h** in the DDK installation directory	TBE custom operator information structure **TEBinInfo** can be used once this header file is included.
proto/caffe/caffe.pb.h	**proto/caffe/caffe.pb.h** generated in the directory of the operator project during the building of an operator plug-in	When the operator plug-in is built, the **/ include/inc/custom/ proto/caffe/caffe.proto** file in the DDK installation directory is automatically built, and the **proto/caffe/ caffe.pb.h** file is generated in the directory of the operator project for the plug-in code to call to parse operator parameters.
operator.h	**include/inc/graph/ operator.h** in the DDK installation directory.	APIs for setting/ obtaining operator attributes and setting input/output can be called once this header file is included.
attr_value.h	**/include/inc/graph/ attr_value.h** in the DDK installation directory	Data types of class **AttrValue** can be used once this header file is included.
memory	C++ standard library	Smart pointers, memory allocators, temporary functions for allocating and releasing dynamic memory, and functions for constructing memory objects in the C++ standard library can be called once this header file is included.
string	C++ standard library	Class **string** can be used to construct objects APIs of class **string** can be called once this header file is included.

Header File	Category	Function
vector	C++ standard library	Vector templates can be used and APIs of class **vector** can be called once this header file is included.

Before the operator plug-in is built, ignore the following message (the project does not have the **proto/caffe/caffe.pb.h** file).

Figure 9-2 Message indicating a caffe.pb.h parsing failure



```
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
```

9.1.3 Parsing an Operator

For a newly developed custom operator, you need to customize a function for parsing operator attributes and converting the operator attribute definitions in the source model to the operator attribute definitions in the offline model supported by the Ascend AI processor. If you are rewriting a built-in operator of the Ascend AI processor, skip this step. A built-in operator is automatically parsed by its plug-in.

Function Declaration

The operator parsing function is declared as follows:

```
Status ParseParamsxx(const Message* op_origin, ge::Operator& op_dest)
```

- **ParseParamsxx**: function name, which is user-defined and must be unique
- **op_origin**: source operator model. It is a data struct in the protobuf format. It is derived from the .proto file of the Caffe model in the **/include/inc/custom/proto/caffe/caffe.proto** directory under the DDK installation directory. If the custom operator is not defined in the **caffe.proto** file, add the operator definition by referring to **8 caffe.proto File Operator Definition (Optional)**.

  The operator plug-in reads operator attributes based on the operator name from the **proto/caffe/caffe.pb.h** file and **caffe.pb.cc** file generated after **caffe.proto** building and parses the operator attributes to convert the operator data structure to a data structure supported by the offline model.

  Find the preset **caffe.proto** file in **/include/inc/custom/proto/caffe/caffe.proto** in the DDK installation path. You can modify the file and add the definition of the custom operator.

- **op_dest**: target operator model. As the operator data struct of the offline model supported by the Ascend AI processor, it stores operator information. For details about class **Operator**, see **Class Operator** in *GE API Reference*.

Procedure

Implement the **ParseParamsxx** function as follows:

Step 1 Define the object that points to **LayerParameter** and obtain the handle to the current operator layer.

```
const caffe::LayerParameter* layer =dynamic_cast<const caffe::LayerParameter*>(op_origin);
const caffe::xxxParameter& param = layer->reduction_param();
```

In the preceding information:

- **xxxParameter** in **caffe::xxxParameter** of the **param** object must be the same as the type declared in the **LayerParameter** object.
- The name of the member function **xxx_param()** of the **layer** object must be the same as the object name declared in the **LayerParameter** object.

The following uses the Reduction and convolution operators in **caffe.proto** as an example:

```
message LayerParameter {
optional ReductionParameter reduction_param = 136;
optional ConvolutionParameter convolution_param = 106;
...
}
```

The code for obtaining the handles to the **Reduction** and **Convolution** operator layers are as follows:

const caffe::**ReductionParameter**& param = layer->**reduction_param**()

const caffe::**ConvolutionParameter**& param = layer->**convolution_param()**

Step 2 Parse the operator attributes and assign the attributes to the **op_dest** object of the **Operator** type.

You can call the **CreateFrom<AttrValue::T>(DT&& val)** API to convert **DT** parameters to **T** parameters of class **AttrValue** and call the **SetAttr(const string& name, const AttrValue& value)** API to assign the converted values of the **AttrValue** object to the corresponding attributes of the **op_dest** object.

Type **T** is introduced to the Ascend AI software stack to simplify the type definition. The supported data types are renamed. For the mapping between type **T** and the source data type, see Table 9-2. For the prototype definitions, see the **include/inc/graph/attr_value.h** file in the DDK installation directory.

Table 9-2 Mapping between type **T** and source data types

T Type	Source Data Type
INT	int64_t
FLOAT	float
STR	std::string

T Type	Source Data Type
TENSOR	TensorPtr
TENSOR_DESC	TensorDesc
GRAPH	ComputeGraphPtr
BYTES	Buffer
NAMED_ATTRS	NamedAttrs
BOOL	bool
LIST_INT	vector<INT>
LIST_FLOAT	vector<FLOAT>
LIST_BOOL	vector<BOOL>
LIST_STR	vector<STR>
LIST_TENSOR	vector<TENSOR>
LIST_TENSOR_DESC	vector<TENSOR_DESC>
LIST_GRAPH	vector<GRAPH>
LIST_BYTES	vector<BYTES>
LIST_NAMED_ATTRS	vector<NAMED_ATTRS>

For details about the **SetAttr** API, see the *GE API Reference*.

The following are examples of parsing common parameters:

- Parameters of the int or float type

  For example, the operator parameters in the **caffe.proto** file are defined as follows:

  ```
  message ReductionParameter {
    ......
    optional int32 axis = 2 [default = 0];
    optional float coeff = 3 [default = 1.0];
  }
  ```

  Call the **SetAttr** API to assign the value of **param.axis()** to the **axis** attribute of the **op_dest** object and convert the type to INT. Assign the value of **param.coeff()** to the **coeff** attribute of the **op_dest** object and convert the type to FLOAT, as follows:

  ```
  op_dest.SetAttr("axis", AttrValue::CreateFrom<AttrValue::INT>(param.axis()));
  op_dest.SetAttr("coeff", AttrValue::CreateFrom<AttrValue::FLOAT>(param.coeff()));
  ```

  In the preceding information, **CreateFrom<AttrValue::T>(DT&& val)** is used to convert **DT** parameters to **T** parameters of the **AttrValue** class.

- Parameters of the enum type

  For example, the operator parameters in the **caffe.proto** file are defined as follows:

  ```
  message ReductionParameter {
    enum ReductionOp {
  ```

```
    SUM = 1;
    ASUM = 2;
    SUMSQ = 3;
    MEAN = 4;
  }
...}
```

a. Convert parameters of the enum type to parameters of the **map** type.

```
std::map<caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
        { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
    { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
    { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
    { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
    };
```

b. Call the **SetAttr** API to assign the *operation_map* parameter of the map type to the **operation** attribute of the **op_dest** object.

```
op_dest.SetAttr("operation",
AttrValue::CreateFrom<AttrValue::STR>(operation_map[param.operation()]));
```

For details about the **SetAttr** API, see the *GE API Reference*.

- Parameters of the repeated type

  For example, the operator parameters in the **caffe.proto** file are defined as follows:

```
message xxxParameter {
...
  repeated float min_size = 1;
  repeated uint32 offset = 2;
....
}
```

a. Convert parameters of the repeated float type to parameters of the vector<float> type, convert parameters of the repeated uint32 type to parameters of the vector<uint32> type, and assign values to the parameters of the vector type.

```
vector<float> min_size;
vector<uint32> offset;
for(int i = 0; i < param.min_size_size(); ++i)
{
   min_size.push_back(param.min_size(i));  // Call the push_back function of the
vector type to assign a value to the min_size parameter of the repeated object.
}
for(int i = 0; i < param.offset_size(); ++i)
{
   offset.push_back(param.offset(i)); // Call the push_back function in the vector
object to assign a value to the offset parameter of the repeated object.
}
```

b. Call the **SetAttr** API to assign the value of the **min_size** parameter of the vector<float> type to the **min_size** attribute of the **op_dest** object. Assign the value of the **offset** parameter of the vector<uint32> type to the **offset** attribute of the LIST_INT type in the **op_dest** object.

```
op_dest.SetAttr("min_size",
ge::AttrValue::CreateFrom<ge::AttrValue::LIST_FLOAT>(min_size));
op_dest.SetAttr("offset", ge::AttrValue::CreateFrom<ge::AttrValue::LIST_INT>(offset));
```

For details about the **SetAttr** API, see the *GE API Reference*.

----End

9.1.4 Inferring the Output Tensor Description of an Operator

Infer the output tensor description of the operator based on the input tensor description, operator logic, and operator attributes. The output tensor description includes the tensor shape, data type, and data layout format. In this way, all tensors can be statically allocated with memory during offline model conversion, thereby avoiding overhead caused by dynamic memory allocation.

Function Declaration

The function is declared as follows:

```
Status InferShapeAndTypexx(const ge::Operator& op, vector<ge::TensorDesc>& v_output_desc)
```

- **InferShapeAndTypexx**: function name, which is user-defined and must be unique
- **op**: compute node definition, which stores the input tensor description and operator attributes. For details about the ge::Operator type, see **Class Operator** in *GE API Reference*
- **v_output_desc**: output tensor description of the compute node, including the shape, data layout format, and data type. For details about the **TensorDesc** type, see **Class TensorDesc** in *GE API Reference*

The following describes the implementation of the **InferShapeAndTypexx** function in different scenarios.

Operator with Same-Shape Output and Input Tensors

For an operator whose output and input tensors have the identical shape, you can directly insert the description of the input tensor into the vector space in which the output tensor description is located.

A code sample is provided as follows:

```
v_output_desc.push_back(op.GetInputDesc(0));
```

The **GetInputDesc** API is used to obtain the input tensor description based on the operator input name or input index in class Operator. For details about the API, see **Class Operator** in *GE API Reference*.

Operator with Reduced Dimensions

For common dimension reduction operations such as Reduction and Reduce, compute the shape of the output tensor (including the output tensor dimensions and element count of each dimension) according to information such as the operator input attribute **axis**, and then insert the shape of the output tensor into the **v_output_desc** vector.

A code sample is provided as follows:

Step 1 Obtain the input tensor description and shape of the input tensor of the operator.

```
auto tensorDesc = op.GetInputDesc(0); // Obtain the input tensor description, including the shape, data layout format, and data type.
auto shape = tensorDesc.GetShape(); // Obtain the shape of the input tensor.
```

For details about the **GetShape** API, see **Class TensorDesc** in *GE API Reference*.

Step 2 Obtain the attribute values of the operator according to the computation logic, and compute the shape of the output tensor of the operator.

For example, for the reduction operator in the MyLeNet network, because the upper layer of Reduction is Softmax and the output from Softmax is padded to 4-dimensional from 2-dimensional in the offline model, you need to adjust **axis** so that it points to a 2-dimensional position. After the reduce operation, and assign the adjusted **Shape** to the output tensor description.

- Obtain the key-value pair of the **axis** attribute from the operator object, obtain the **axis** attribute value from the key-value pair, convert the attribute from the INT type to the int64_t type, assign the attribute value to the variable **axis**, verify and adjust the value of **axis**, and point the value to axis 1, as follows:
```
int64_t axis = -1;
ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS != 
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("Get axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to 
point to the original 2d.
    if (axis < 0) axis -= 2;

    if (axis < 0) axis += shape.GetDimNum();

    if (axis < 0 || axis >= shape.GetDimNum())
    {
        printf("invalid axis:%d, dim_size:%d\n", (int32_t)axis, (int32_t)shape.GetDimNum());
        return PARAM_INVALID;
    }
```
- Adjust **Shape** and set the dimensions from axis 1 to **1**. For example, if the input tensor is with shape (2, 3, 4, 5), adjust the shape to (2, 1, 1, 1).
```
int32_t dimsize = (int32_t)shape.GetDimNum();
int32_t idx = 0;
for(idx=axis; idx<dimsize; idx++)
{
    shape.SetDim(idx, 1);
}
```
- Set the adjusted shape to the **tensorDesc** object.
```
tensorDesc.SetShape(shape);
```

For details about APIs **GetDimNum** and **SetDim**, see **Class Shape** in *GE API Reference*.

Step 3 Set the output tensor description of the operator.
```
v_output_desc.push_back(tensorDesc)
```
Assign **tensorDesc** to the description object **v_output_desc** of the output tensor.

----End

Network with Multiple Operators of the Identical Type

A network can have multiple operator layers of the same type, such as the convolution operator. Sometimes, you need to customize the shape of an operator layer (redefine an existing operator). In this case, the output tensor description inference needs to be performed according to different situations of the network, determine the operator layers to be customized based on the attributes such as

num_output, **kernel**, **stride**, and **pad**, and obtain the tensor information of the operators.

A code sample is provided as follows:

Step 1 Assign the input tensor description to the output tensor description, and obtain the input tensor description and the shape of the input tensor of the operator.

```
v_output_desc.push_back(op.GetInputDesc(0)); // Assign the input tensor description to the output tensor description object. Alternatively, assign the shape obtained after inference to the tensorDesc object, and then assign the tensorDesc object to the output tensor description object.
auto tensorDesc = op.GetInputDesc(0); // Obtain the input tensor description, including the shape, data layout format, and data type.
auto shape = tensorDesc.GetShape(); // Obtain the shape of the input tensor.
```

For details about the **GetShape** API, see **Class TensorDesc** in *GE API Reference*.

Step 2 Obtain the attribute values of the operator according to the computation logic, match the operator according to the attribute values of the operator and shape, and compute the shape of the output tensor of the operator.

For example, for operators of type **convolution** in a network, match the convolution operator whose **num_outputs** is **128**, **shape.GetDim(0)** is **1**, **shape.GetDim(1)** is **128**, **shape.GetDim(2)** is **28**, and **shape.GetDim(3)** is **28**, and reassign the shape in the output tensor description of the operator.

- Obtains the value of **num_outputs**.

```
ge::AttrValue num_outputsAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("num_output", num_outputsAttrValue)) ||
        (ge::GRAPH_SUCCESS != num_outputsAttrValue.GetValue<AttrValue::INT>(num_outputs)))
    {
        printf("GetOpAttr num_outputs failed!\n");
    }
```

- Match the convolution operator whose **num_outputs** is **128**, **shape.GetDim(0)** is **1**, **shape.GetDim(1)** is **128**, **shape.GetDim(2)** is **28**, and **shape.GetDim(3)** is **28** , and reassign the shape in the output tensor description of the operator.

```
 if(shape.GetDim(0) == 1 && shape.GetDim(1) == 128 &&
        shape.GetDim(2) == 28 && shape.GetDim(3) == 28 && num_outputs == 128)
    {
        shape.SetDim(0, 1);
        shape.SetDim(1, 128);
        shape.SetDim(2, 28);
        shape.SetDim(3, 28);
        v_output_desc[0].SetShape(shape);
        return SUCCESS;
        return FAILED;
    }
```

For details about APIs **GetDimNum** and **SetDim**, see **Class Shape** in *GE API Reference*.

NOTE

To use **op_name** for operator matching, obtain **op_name** as follows:

```
auto op_name = op.GetName();
```

The method for obtaining other attributes **kernel_w**, **kernel_h**, **stride_w**, **stride_h**, **pad_w**, and **pad_h** are similar. You only need to change the value of **key** in **op.GetAttr**.

----End

9.1.5 Building an Operator

Function Declaration

The operator building function is declared as follows:

```
Status BuildTeBinxx(const ge::Operator& op, TEBinInfo& te_bin_info)
```

In the preceding information:

- **BuildTeBinxx**: function name, which is user-defined and must be unique
- **op**: target operator model. As the operator data struct of the offline model supported by the Ascend AI processor, it stores operator information. For details about class **Operator**, see **Class Operator** in *GE API Reference*.
- **te_bin_info**: path of the operator binary file, operator description file path, and DDK version information For details about the **TEBinInfo** struct, see **TEBinBuildFn** in *Framework API Reference*.

Implementation Procedure

The operator building function is called during model conversion by OMG as follows:

- Obtain the operator tensor description and operator attributes. During model conversion, the information must be fixed values for operator matching.

  For example, in model conversion, match the reduction operator whose **axis** is **1** and **Dim** of the input tensor **Shape** is **4**.

```
// Parse the operator attribute operation.
    ge::AttrValue operationAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("operation", operationAttrValue)) || (ge::GRAPH_SUCCESS != 
operationAttrValue.GetValue<AttrValue::STR>(operation)))
    {
           printf("GetOpAttr operation failed!\n");
    }

// Parse the operator attribute axis, and adjust axis to point to the actual output dimension of the 
Softmax operator at the upper layer of the reduction operator in the MyLeNet network, that is, axis 1.
    ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS != 
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("GetOpAttr axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to 
point to the original 2d.
    if(axis < 0)
        axis -= 2;

// Parse the operator attribute coeff.
    ge::AttrValue coeffAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("coeff", coeffAttrValue)) || (ge::GRAPH_SUCCESS != 
coeffAttrValue.GetValue<AttrValue::FLOAT>(coeff)))
    {
        printf("GetOpAttr coeff failed!\n");
    }
// Obtain the input tensor description of the operator.
    TensorDesc input_desc      = op.GetInputDesc(0);

    // Parse the input shape and check whether Dim of the operator is 4.
    if(input_desc.GetShape().GetDimNum() != 4)
    {
        printf("The shape size is %d, which is not 4!", (int32_t)input_desc.GetShape().GetDimNum());
        return FAILED;
    }
```

- Specify the operator implementation file, operator implementation function, and operator name in the kernel.

```
    FilePath = "project_path/operator/reduction"; // Absolute path of the operator implementation file
+ name of the .py operator file
    FuncName = "Reduction"; // Name of the operator implementation function in the operator
implementation file
    KernelName = "Reduction"; // kernel_name defined in the operator implementation function of
the operator implementation file, that is, the name of the generated binary file
```

- Specify the path of the operator description file (*.json) generated during operator compilation. Use the following fixed configuration.

```
  te_bin_info.json_file_path = "./kernel_meta/" + KernelName + ".json";
```

  During model conversion, operator information will be obtained from the operator description file in this path.

  When the **omg** model conversion command is executed, the **kernel_meta** folder generated after operator building is copied to the path where the **omg** command is executed based on the operator implementation path configured in the **FilePath** file. Therefore, the path of the *.json file relative to the path where the **omg** command is executed is fixed to **./kernel_meta**.

- Call the **te::BuildCustomop** function to call the Python function in the operator implementation file to build the operator.

  Call the **BuildCustomop** function as follows:

```
te::BuildTeCustomOp(te_bin_info.ddk_version, op.GetName(), FilePath, FuncName,"(i,i,i,i), s, i, s, f, s",
input_desc.GetShape().GetDim(0),input_desc.GetShape().GetDim(1),input_desc.GetShape().GetDim(2),
input_desc.GetShape().GetDim(3), "float16", axis, operation.c_str(), coeff,KernelName.c_str());
```

  In the preceding information:

  – **te_bin_info.ddk_version**: DDK version information (unconfigurable), which will be automatically filled during model conversion
  – **op.GetName()**: obtaining operator name (unconfigurable),
  – **FilePath**: relative path of the operator file
  – **FuncName**: name of the operator implementation function in the operator implementation file
  – **(i, i, i, i), s, i, s, f, s**: parameter placeholders of the implementation functions in the operator implementation file, where, **i** indicates the integer type, **s** indicates the string type, **f** indicates the single-precision floating point number type, and **o** indicates the **PyObject*** type. The placeholders must be consistent with the sequence and types of the succeeding parameters, and must be consistent with the definition of the operator implementation function in the operator implementation file. **BuildCustomop** calls the operator implementation function based on these parameters and generates the kernel using the TVM mechanism.

9.1.6 Registering an Operator

As the framework manager, Framework provides the **REGISTER_CUSTOM_OP** macro to register an operator based on the specified operator name.

The code of custom operator registration is as follows:

```
REGISTER_CUSTOM_OP("test_layer")
    .FrameworkType(CAFFE)
    .OriginOpType("Test")
```

```
.ParseParamsFn(ParseParamsxx)
.InferShapeAndTypeFn(InferShapeAndTypexx)
.TEBinBuildFn(BuildTeBinxx)
.ImplyType(ImplyType::TVM)
.Formats({DOMI_TENSOR_NC1HWC0}, {DOMI_TENSOR_NC1HWC0})
.WeightFormats({DOMI_TENSOR_FRACTAL_Z, DOMI_TENSOR_NC1HWC0});
```

In the preceding information:

- **REGISTER_CUSTOM_OP**: Registers a custom operator. Replace **test_layer** with the operator name in the offline model file. The operator name can be random but must not conflict with existing operator names.
- **FrameworkType**: The operator parameter parsing logic varies depending on the framework. Therefore, models under different frameworks require different plug-ins. The plug-in registration code must specify the model framework. Set this parameter to **CAFFE**.
- **OriginOpType**: operator type, which must be the same as the operator type defined in Caffe Prototxt. Otherwise, parsing fails. Find the preset **caffe.proto** file in **/include/inc/custom/proto/caffe/caffe.proto** in the DDK installation path.
- **ParseParamsFn**: Registers the function for model parsing. **ParseParamsxx** has been implemented in **9.1.3 Parsing an Operator**. This step is required for a plug-in developed for the Caffe framework. If you are rewriting a built-in operator of the Ascend AI processor, skip this step. If the custom operator is not supported by the Ascend AI processor, this step is mandatory.
- **InferShapeAndTypeFn**: Registers the function for shape and class inference. **InferShapeAndTypexx** has been implemented in **9.1.4 Inferring the Output Tensor Description of an Operator**.
- **TEBinBuildFn**: Registers the TBE operator building function. **BuildTeBinxx** has been implemented in **9.1.5 Building an Operator**.
- **ImplyType**: Specifies the operator implementation. **ImplyType::TVM** indicates that the operator is a TE operator.
- **Formats**: Specifies the layout formats of the input data and output data of the operator. The first list is the input data format list, and the second list is the output data format list. If there are multiple inputs, list the layout format of each input data in the first list. For example, if there are two pieces of input data in the NC1HWC0 format, call the **Formats** function as follows:

  ```
  .Formats({DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0},
  {DOMI_TENSOR_NC1HWC0})
  ```

  For details, see **Formats** in *Framework API Reference*.
- **WeightFormats**: Sets the layout format of operator weight data. For details about the supported data formats, see **WeightFormats** in *Framework API Reference*. For example, the data layout format of the filter of convolution is **fractal_Z**, and the data layout format of the filter of bias is **NC1HWC0**.

  If quantization during model conversion is enabled, the constant formats for Framework processing need to be added to this API. Currently, Framework supports the following quantization operators: Conv, FC, and Depthwise Conv. If quantization is enabled for these operators during model conversion, you need to add six **DOMI_TENSOR_NC1HWC0** data formats to the end of the parameter list of the **WeightFormats** API. (During quantization, Framework adds six constants whose data layout format is NC1HWC0. The following is a code sample of the **WeightFormats** API for the convolution operator with quantization enabled:

```
.WeightFormats({DOMI_TENSOR_FRACTAL_Z, DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0,
DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0,DOMI_TENSOR_NC1HWC0,
DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0})
```

9.2 Plug-In Code Example

Step 1 Modify the code file in **customop_te/plugin/caffe_reduction_layer.cpp**.

- Change **FilePath** to the relative path of the current operator file+name of the operator py file.

  FilePath = "../operator/reduction";

- Change **FuncName** to the name of the operator implementation function in the **reduction.py** file.

  FuncName ="reduction"

- Change the value of **KernelName** to the **KernelName** of the operator configured in the **reduction.py** file.

  KernelName = "Reduction";

- Change the path of the binary file *.o and description file *.json of the operator.

  te_bin_info.bin_file_path = "./operator/kernel_meta/" + KernelName + ".o";

  te_bin_info.json_file_path = "./operator/kernel_meta/" + KernelName + ".json";

  The path is relative to the current project directory. For example, the **Reduction.o** file is in the **operator/kernel_meta** directory under the current project directory.

Step 2 Check the modified code file. The following is a code example of the modified reduction operator plug-in:

```
#include <Python.h>
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
using namespace ge;

namespace domi
{
// Rewrite the ParseParams function to parse operator parameters.
Status CaffeReductionParseParams(const Message* op_origin, ge::Operator& op_dest)
{
    // Convert op_origin into the layer object.
    const caffe::LayerParameter* layer =
        dynamic_cast<const caffe::LayerParameter*>(op_origin);

    // Verify the validity of the input parameters.
    if (nullptr == layer)
    {
        printf("Dynamic cast op_src to LayerParameter failed\n");
        return FAILED;
    }
    // Convert the enumeration type of the operator parameters to the map type.
    std::map<caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
        { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
    { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
    { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
    { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
    };
    // Obtain the handle to the current operator layer.
    const caffe::ReductionParameter& param = layer->reduction_param();
    // Assign all parameter values of the operator to the object of the operator data structure op_dest of the
Da Vinci model.
    if(param.has_axis())
    {
        op_dest.SetAttr("axis", AttrValue::CreateFrom<AttrValue::INT>(param.axis()));
    }
    if(param.has_coeff())
    {
        op_dest.SetAttr("coeff", AttrValue::CreateFrom<AttrValue::FLOAT>(param.coeff()));
    }
    if(param.has_operation())
    {
        op_dest.SetAttr("operation",
AttrValue::CreateFrom<AttrValue::STR>(operation_map[param.operation()]));
    }
    return SUCCESS;
}

// Rewrite the InferShapeAndType function to obtain the output description of the operator.
Status CaffeReductionInferShapeAndType(const ge::Operator& op, vector<ge::TensorDesc>& v_output_desc)
{
    // Obtain the TensorDesc object input by the operator.
    auto tensorDesc      = op.GetInputDesc(0);
    // Obtain the input shape.
    auto shape = tensorDesc.GetShape();
    int64_t axis = -1;

    ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("Get axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to point to
the original 2d.
    if (axis < 0) axis -= 2;

    if (axis < 0) axis += shape.GetDimNum();

    if (axis < 0 || axis >= shape.GetDimNum())
    {
        printf("invalid axis:%d, dim_size:%d\n", (int32_t)axis, (int32_t)shape.GetDimNum());
        return PARAM_INVALID;
    }
    int32_t dimsize = (int32_t)shape.GetDimNum();
    int32_t idx = 0;
    for(idx=axis; idx<dimsize; idx++)
    {
        shape.SetDim(idx, 1);
    }
    // Set the adjusted shape information to tensorDesc.
    tensorDesc.SetShape(shape);
    v_output_desc.push_back(tensorDesc);

    return SUCCESS;

}


// Compile the operator and obtain the binary file and description file of the operator.
Status CaffeReductionBuildTeBin(const ge::Operator& op, TEBinInfo& te_bin_info)
{
    // Declare operator parameters.
    std::string FilePath   = "";
    std::string FuncName   = "";
    std::string KernelName = "";

    std::string operation  = "";
    int64_t     axis       = -1;
    float       coeff      = 1;
    // Obtain operator parameter values.
    ge::AttrValue operationAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("operation", operationAttrValue)) || (ge::GRAPH_SUCCESS !=
operationAttrValue.GetValue<AttrValue::STR>(operation)))
    {
        // Add exception handling and maintenance information.
        printf("GetOpAttr operation failed!\n");
    }

    // Parse the axis parameter.
    ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("GetOpAttr axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to point to
the original 2d.
    if(axis < 0)
        axis -= 2;

    // Parse the coeff parameter.
    ge::AttrValue coeffAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("coeff", coeffAttrValue)) || (ge::GRAPH_SUCCESS !=
coeffAttrValue.GetValue<AttrValue::FLOAT>(coeff)))
    {
        printf("GetOpAttr coeff failed!\n");
    }
    // Obtain the operator input description.
    TensorDesc input_desc      = op.GetInputDesc(0);

    // Parse the input shape and check whether Dim of the operator is 4.
    if(input_desc.GetShape().GetDimNum() != 4)
    {
        printf("The shape size is %d, which is not 4!", (int32_t)input_desc.GetShape().GetDimNum());
        return FAILED;
    }
    // Set the operator file path.
    FilePath   = "../operator/reduction";
    // Set the name of the operator implementation function.
    FuncName   = "reduction";
    // Set the operator name defined in the operator implementation file.
    KernelName = "Reduction";

    // i => int; s => string; f => dobule; O => bool, and bool value is Py_True or Py_False. Call the compilation
interface of Tensor Engine to compile the operator.
    te::BuildTeCustomOp(te_bin_info.ddk_version, op.GetName(), FilePath, FuncName,
                  "(i,i,i,i), s, i, s, f, s", input_desc.GetShape().GetDim(0), input_desc.GetShape().GetDim(1),
                  input_desc.GetShape().GetDim(2), input_desc.GetShape().GetDim(3), "float16", axis,
operation.c_str(), coeff,
                  KernelName.c_str());    // The parameter sequence must be the same as that in the operator
implementation function.

    // set te op json to te_bin_info
    te_bin_info.bin_file_path  = "./kernel_meta/" + KernelName + ".o";
    te_bin_info.json_file_path = "./kernel_meta/" + KernelName + ".json";

    return SUCCESS;
}

REGISTER_CUSTOM_OP("custom_reduction") //####test_reduction is the type name of the operator in the
OM model. It can be specified randomly and cannot be the same as an existing type name. It is case
sensitive.
```

```
    .FrameworkType(CAFFE)  // Enumerated type. The options are as follows: CAFFE, TENSORFLOW
    .OriginOpType("Reduction")  // Reduction indicates the type name of the operator in the caffe
framework.
    .ParseParamsFn(CaffeReductionParseParams)  // Op parameters parse function
    .InferShapeAndTypeFn(CaffeReductionInferShapeAndType)       // Set output description and datatype
function
    .TEBinBuildFn(CaffeReductionBuildTeBin)           // Build Te op binary function
    .ImplyType(ImplyType::TVM);        // Implementation type. Enumerated type, The options are as follows:
TVM, AI_CPU.

}  // namespace domi
```

----End

9.3 Compiling the Operator Plug-In

Step 1 Log in to the DDK server as the DDK installation user.

Step 2 Modify the **Makefile** file in **projects/customop_te/plugin**.

cd $HOME/tools/projects/customop_te/plugin/

vi Makefile

- Change the name of the generated operator plug-in.

```
ll : libcaffe_reduction_layer.so lib_caffe_parser.so
......bian
libcaffe_reduction_layer.so: $(OBJS_customop)
        $(CC) -c $(CC_FLAGS) -o proto/caffe/caffe.pb.o proto/caffe/caffe.pb.cc
        $(CC) $^ $(LNK_FLAGS) -o $@

lib_caffe_parser.so: $(OBJS_no_customop)
        $(CC) -c $(CC_FLAGS) -o proto/caffe/caffe.pb.o proto/caffe/caffe.pb.cc
        @if [ -f $(LOCAL_DIR)/proto/caffe/caffe.proto ]; then $(CC) $^ proto/caffe/caffe.pb.o $
(LNK_FLAGS) -o $@; fi;
```

  libcaffe_reduction_layer.so is the name of the generated operator plug-in. You can change the name as required.

  lib_caffe_parser.so is the library file generated during the parsing of the **caffe.proto** file, and its name cannot be changed. For Caffe operators, ensure that all unsupported custom ones in the same model have been defined in **8 caffe.proto File Operator Definition (Optional)**.

- Set **TOPDIR** to the installation directory of the DDK.

```
ifeq ($(DDK_PATH),)
TOPDIR      := $(HOME)/tools/che/ddk/ddk
else
TOPDIR := $(DDK_PATH)
endif
```

- Use the default values for other parameters in **Makefile**, which serve as the common template for the operator plug-in compilation.

Step 3 Compile the operator plug-in.

Run the following command in the plug-in directory to compile the operator plug-in:

make

After the compilation is complete, the operator plug-in file **libcaffe_reduction_layer.so** is generated in the current directory.

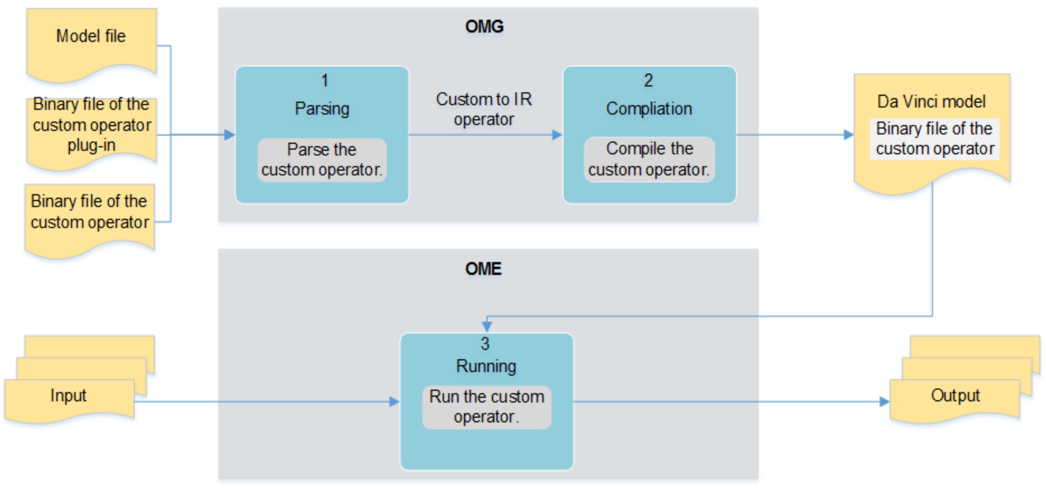
----End

10 Loading the Plug-In for Model Conversion

Principle Description

Figure 10-1 shows the model conversion process when the custom operator plug-in is loaded.

Figure 10-1 Loading a plug-in for model conversion


Model file
Binary file of the custom operator plug-in
Binary file of the custom operator
OMG
1
Parsing
Parse the custom operator.
Custom to IR operator
2
Compliation
Compile the custom operator.
Da Vinci model
Binary file of the custom operator
OME
3
Running
Run the custom operator.
Input
Output

1. The offline model generator (OMG) loads the model file and custom operator plug-in, parses the operator in the model file, and converts the custom operator into the intermediate representation (IR) operator.
2. The OMG converts the data of the custom operator based on the running environment, calculates the running memory, compiles and generates the binary file (*.o) of the custom operator, and generates the Da Vinci offline model file (*.om).
3. During application running, the offline model executor (OME) obtains the input data, loads the offline model file, calls the operator cyclically, and outputs the result data.

Operating Procedure

Step 1 Go to the root directory of the custom operator development project as the DDK installation user.

cd $HOME/tools/projects/customop_te/

Step 2 Run the following command to convert the model:

omg --model=*model/deploy_mylenet-1.prototxt* --weight=*model/mylenet-1.caffemodel* --framework=0 --plugin_path=*plugin* --output=*mylenet* --ddk_version=*1.3.T18.B850*

- **--model**: relative path of the original model file of the MyLeNet network
- **--weight**: relative path of the pre-trained model file of the MyLeNet network
- **--framework**: original framework type
  - 0: Caffe
  - 3: TensorFlow
- **--plugin**: directory where the custom operator plug-in is located
- **--output**: name of the output model file, which can be customized
- **--ddk_version**: version number of the matched DDK for running the custom operator. You can view the version number of the DDK in the **$HOME/tools/che/ddk/ddk/ddk_info** file.

NOTE

For details about other parameters, see *Model Conversion Guide*.

----End

11 Function Reference for Operator Plug-In Development

11.1 ParseParamsFn

Function

Parses parameters.

Syntax

OpRegistrationData& ParseParamsFn(ParseParamFunc parseParamFn);

Parameter Description

Parameter	Input/ Output	Description
parseParamFn	Input	Callback function ParseParamFunc. For details, see **Callback Function ParseParamFunc**.

Callback Function ParseParamFunc

You can customize and implement the ParseParamFunc class functions to convert the parameters and weights of the Caffe model and fill the results in the Operator class.

Syntax

Status ParseParamFunc(const Message* op_origin, ge::Operator& op_dest);

Parameter Description

Parameter	Input/ Output	Description
op_origin	Input	Data structure in protobuf format (from the prototxt file of the Caffe model), including operator parameter information
op_dest	Output	Operator data structure of offline model supported by the Ascend AI processor, which stores operator information For details about the operator class, see **Operator Class APIs** in *GE API Reference*.

11.2 InferShapeAndTypeFn

Function

Shape inference function

Syntax

OpRegistrationData& InferShapeAndTypeFn(InferShapeFunc inferShapeFn);

Parameter Description

Parameter	Input/ Output	Description
inferShapeFn	Input	Callback function InferShapeFunc. For details, see **Callback Function InferShapeFunc**.

Callback Function InferShapeFunc

You can customize and implement the InferShapeFunc class function to obtain the output description of an operator, including the tensor description such as the output shape information and data type.

Syntax

Status InferShapeFunc(const ge::Operator& op, vector<ge::TensorDesc>& v_output_desc);

Parameter Description

Parameter	Input/Output	Description
op	Input	Operator data structure of offline model supported by the Ascend AI processor For details about the operator class, see **Operator Class APIs** in *GE API Reference*.
v_output_desc	Output	Operator output description For details about the TensorDesc class, see **TensorDesc Class APIs** in *GE API Reference*.

11.3 TEBinBuildFn

Function

Build callback function

Syntax

OpRegistrationData& TEBinBuildFn(BuildTeBinFunc buildTeBinFn);

Parameter Description

Parameter	Input/ Output	Description
buildTeBinFn	Input	Callback function BuildTeBinFunc. For details, see **Callback Function BuildTeBinFunc**.

Callback Function BuildTeBinFunc

You can customize and implement the BuildTeBinFunc class function to construct the operator binary file.

Syntax

virtual Status BuildTeBinFunc(const ge::Operator& op, TEBinInfo& teBinInfo);

Parameter Description

Parameter	Input/ Output	Description
op	Input	Operator data structure of offline model supported by the Ascend AI processor, which stores operator information For details about the operator class, see **Operator Class APIs** in *GE API Reference*.

teBinInfo	Output	Path of the binary file of a custom operator and DDK description
		struct TEBinInfo
		{
		std::string bin_file_path; // Automatically obtained from the **binFileName** field in the JSON file. To ensure compatibility with cases written by users, the field is not deleted.
		std::string json_file_path;
		std::string ddk_version;
		};

11.4 WeightFormats

Function

Sets the data formats supported by the operator weight.

Syntax

OpRegistrationData& WeightFormats(

const std::initializer_list<domi::tagDomiTensorFormat>& weight_formats);

Parameter Description

Parameter	Input/ Output	Description
weight_formats	Input	Data formats supported by the weight: typedef enum tagDomiTensorFormat { DOMI_TENSOR_NCHW = 0, /**< NCHW */ DOMI_TENSOR_NHWC, /**< NHWC */ DOMI_TENSOR_ND, /**< Nd Tensor */ DOMI_TENSOR_NC1HWC0, /**< NC1HWC0 */ DOMI_TENSOR_FRACTAL_Z, /**< FRACTAL_Z */ DOMI_TENSOR_NC1C0HWPAD, DOMI_TENSOR_NHWC1C0, DOMI_TENSOR_FSR_NCHW, DOMI_TENSOR_FRACTAL_DECONV, DOMI_TENSOR_BN_WEIGHT, DOMI_TENSOR_CHWN, /*Android NN Depth CONV*/ DOMI_TENSOR_FILTER_HWCK, /* filter input tensor format */ DOMI_TENSOR_RESERVED } domiTensorFormat_t;

11.5 Registration Macro of the Operator Building Function

Function

Registers the operator by calling DOMI_REGISTER_OP.

Prototype

REGISTER_CUSTOM_OP(name)

Parameter Description

Parameter	Input/ Output	Description
name	Input	Type of an operator in the Da Vinci model, which can be specified randomly and must be unique. The value is case sensitive.

12 Appendix

12.1 Change History

Release Date	Description
2020-05-30	This issue is the first official release.