TE 自定义算子开发指导 (命令行方式)

文档版本 01
发布日期 2020-05-09

华为技术有限公司

版权所有 © 华为技术有限公司 2020。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

HUAWEI和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 开发前必读 1
1.1 概述 1
1.2 目标读者 3
2 算子开发注意事项 4
3 环境准备 5
4 开发流程总览 6
5 配置环境变量 8
6 创建自定义算子开发工程 9
7 自定义算子开发 11
7.1 算子特性 11
7.2 算子代码实现 15
7.2.1 实现流程 16
7.2.2 导入 Python 模块 16
7.2.3 算子实现 16
7.2.4 算子调度与编译 18
7.2.5 执行算子 19
7.3 代码样例 19
7.4 算子运行验证 20
7.4.1 算子编译 20
7.4.2 构造输入数据文件 22
7.4.3 运行单算子 22
8 caffe.proto 文件算子定义（可选） 25
9 自定义算子插件开发 26
9.1 插件代码实现 26
9.1.1 插件实现流程 26
9.1.2 包含头文件 27
9.1.3 解析算子 28
9.1.4 推理算子输出张量描述 31
9.1.5 编译算子 34
9.1.6 算子注册 35

9.2 插件代码样例.....36
9.3 编译算子插件.....39
10 加载插件转换模型.....41
11 算子插件开发函数参考.....43
11.1 ParseParamsFn 函数.....43
11.2 InferShapeAndTypeFn 函数.....44
11.3 TEBinBuildFn 函数.....45
11.4 WeightFormats 函数.....46
11.5 算子构建函数注册宏.....47
12 附录.....48
12.1 修订记录.....48

1 开发前必读

1.1 概述

1.2 目标读者

1.1 概述

Ascend 310中提供了深度学习框架Framework，可以将Caffe、Tensorflow开源框架模型转换成Ascend 310支持的模型，在进行模型转换时，如果模型中的算子在系统内置的算子库中未实现，则转换过程会报错。这时，未实现的算子就需要用户进行自定义，自定义算子可加入到自定义算子库中，使得模型转换过程可以正常进行。用户也可以优化当前模型中已有算子，并将优化后的算子加入到自定义算子库中，并加载优化后的算子进行模型转换。

Ascend 310提供了TE（Tensor Engine）算子开发框架，可以开发自定义算子。TE是基于TVM（Tensor Virtual Machine）的自定义算子开发框架，提供了基于Python语法的DSL语言供开发者开发编写自定义算子。TE自定义算子可运行于AI CPU和AI Core。

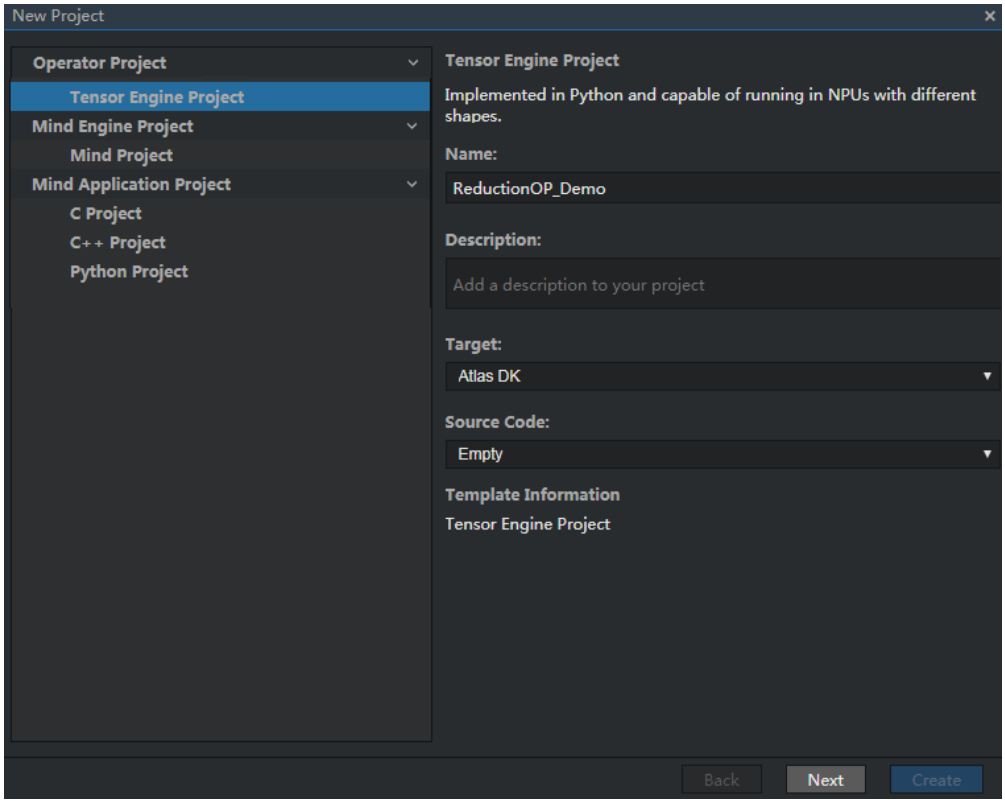
本文以Caffe网络模型Lenet-5内置Reduction算子的扩展自定义caffe_reduction_layer算子为例进行讲解，介绍如何通过后台命令行的方式进行TE自定义算子的开发、算子的验证、算子的注册及如何加载用户自定义的算子进行模型转化。

Ascend 310支持的TE自定义算子开发方式如**表1-1**所示，本文仅介绍“命令行开发方式”。

表 1-1 Ascend 310 支持的 TE 自定义算子开发方式

方式	工具依赖	描述	编译方式	对应文档
命令行开发方式	不依赖Mind Studio工具，仅安装DDK的情况就可进行TE自定义算子开发。 **说明** DDK（Device Development Kit）为用户提供基于NPU的数字开发者套件，内含工程代码样例，并集成了相关依赖库和头文件，用户可以通过makefile脚本编译相应的工程文件。	无UI界面，无自动生成的框架代码，所有算子实现代码及算子插件代码需要用户自行开发，或者基于DDK中代码样例进行编辑修改。	基于DDK代码样例编写Makefile编译脚本。	《TE自定义算子开发指导》
Mind Studio 开发方式	依赖Mind Studio工具，操作入口如下图所示。	框架代码基于工程创建时导入的模型文件及选择的算子自动生成，用户只需要重点关注算子的实现代码。	Makefile编译脚本通过Mind Studio界面配置生成，无需编写。	《Ascend 310 TE自定义算子开发指导（Mind Studio方式）》

图 1-1 Tensor Engine Project 创建



1.2 目标读者

本文档适用于使用Ascend 310进行算子开发的人员，通过本文档您可以达成：

- 通过本文档提供的算子开发样例进行算子开发，并将此算子集成到整网络中进行端到端的应用程序的运行，端到端了解自定义算子的开发流程。
- 参考文档中的Reduction算子样例，学习算子开发及算子插件开发的原理，能够进行其他自定义算子的开发。

掌握以下经验和技能可以更好地理解本文档：

- 具备Python/C++/C语言程序开发能力
- 理解数学表达式
- 对机器学习、深度学习有一定的了解
- 对Ascend 310架构有一定了解
- 对TVM及开源Caffe框架有一定的了解

2 算子开发注意事项

SSD 网络中带后处理节点的自定义算子开发注意事项

SSD网络的PriorBox算子与Detection Ouptut算子模型转换时会自动融合，若用户自定义实现了SSD网络中的PriorBox算子，则无法自动融合导致模型转换失败，此时需要用户删除网络模型文件中的Detection Output层算子，然后在Engine编排流程中自定义后处理节点实现Detection Output算子的功能。

其他

- 请勿修改框架内置算子的实现（路径：ddk/include/inc/custom下的所有文件），否则将可能导致未知风险，例如：系统启动失败、模型无法正常转换等。
- 客户在开发算子插件时需要对自己的源代码负责，避免植入后门。

3 环境准备

已完成DDK（Device Development Kit）的部署，DDK为用户提供基于Ascend 310的开发者套件，内含工程代码样例，并集成了相关依赖库和头文件，用户可以通过makefile编译相应的工程文件。

4 开发流程总览

TE自定义算子开发流程如图4-1所示。

图 4-1 TE 自定义算子开发流程

主要流程如下：

1. 创建自定义算子开发工程。
2. 自定义未实现的算子，包括算子代码开发、单算子的编译、运行、验证。

3. 自定义算子插件开发，将算子注册到Framework中，算子插件编译后，生成自定义算子的*.so插件文件。
4. 再次进行模型转换时，通过加载*.so插件文件，才能识别自定义的算子，进而完成模型转换。

5 配置环境变量

步骤1 以DDK安装用户*ascend*登录DDK所在服务器。

步骤2 设置环境变量。

- 若为Atlas 200 DK开发者板环境，在终端窗口执行如下命令设置环境变量。

```
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
```

- 若为ASIC环境，在终端窗口执行如下命令设置环境变量。

```
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
```

说明

DDK_HOME的路径需要修改为实际的DDK安装路径。

用户可自行将设置环境变量的命令写入自定义脚本，方便后续执行。

----结束

6 创建自定义算子开发工程

步骤1 以DDK安装的普通用户*ascend*登录DDK所在服务器。

步骤2 创建工作空间目录。

mkdir $HOME/tools/projects

步骤3 创建自定义工程目录。

$ **mkdir $HOME/tools/projects/customop_te**

创建算子代码目录：

mkdir $HOME/tools/projects/customop_te/operator

创建算子插件代码目录：

mkdir $HOME/tools/projects/customop_te/plugin

步骤4 拷贝DDK中的样例代码到算子开发工程目录下。

说明

- 本示例使用ddk中内置的reduction算子代码及算子插件代码进行说明如何进行自定义算子的开发，用户可自己新建相关文件进行编辑。
- 本示例中DDK的安装路径为“$HOME/tools/che”，若实际安装路径与示例路径不一致，请自行替换示例命令中路径。

- 将ddk中的reduction算子样例代码reduction.py和算子验证的样例数据生成脚本data_gen.py拷贝到自定义工程的算子代码目录下。

  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/

- 将ddk中的reduction算子插件样例代码文件及makefile文件拷贝到自定义工程的算子插件代码目录下。

  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/

- 将ddk中的本示例使用的caffe网络模型文件拷贝到自定义工程目录下。

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

本示例所示的自定义算子开发工程目录结构如表6-1所示。

表 6-1 目录结构

样例目录	文件	说明
customop_te/operator	reduction.py	reduction算子样例代码
	data_gen.py	reduction算子样例数据生成文件
customop_te/plugin	caffe_reduction_layer.cpp	reduction算子插件样例代码（需要用户修改）
	Makefile	reduction算子插件编译规则文件（需要用户修改）
customop_te/model	deploy_mylenet-1.prototxt	mylenet网络的原始模型文件
	mylenet-1.caffemodel	mylenet网络的预训练模型文件

----结束

7 自定义算子开发

7.1 算子特性

7.2 算子代码实现

7.3 代码样例

7.4 算子运行验证

本节描述如何单独运行用户开发的算子，验证算子实现的正确性。

7.1 算子特性

深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称Op）。在Caffe中，算子对应层中的计算逻辑，例如：卷积层（Convolution Layer）中的卷积算法，是一个算子；全连接层（Fully-connected Layer， FC layer）中的权值求和过程，是一个算子。

下面介绍算子中常用的基本概念。

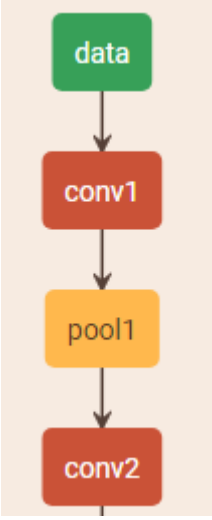
算子类型（Type）

算子的type，代表算子的类型，例如卷积算子的类型为Convolution，在一个网络中同一类型的算子可能存在多个。

算子名称（Name）

算子的name，用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如下图所示conv1，pool1，conv2都是此网络中的算子名称，其中conv1与conv2算子的类型为Convolution，表示分别做一次卷积运算。

图 7-1 网络拓扑示例



张量（Tensor）

Tensor是TE算子中的数据，包括输入数据与输出数据，TensorDesc（Tensor描述符）是对输入数据与输出数据的描述，TensorDesc数据结构包含如下属性如表7-1所示。

表 7-1 TensorDesc 属性解释

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如（10,）或者（1024，1024）或者（2，3，4）等。详细介绍请参见 **形状（shape）**。 默认值：无 形式：(i1, i2,…in)，i1，i2，in为正整数
数据类型（dtype）	功能描述：指定Tensor对象的数据类型。 默认值：无 取值范围：float16, int8, int16, int32, uint8, uint16, bool。 **说明** • 不同计算操作支持的数据类型不同，详细请参见《TE API参考》。 • TE API同时支持float16与float32数据类型，但OMG进行模型转换的时候会将float32数据类型转换成float16，所以当前版本在进行自定义算子开发时，不支持使用float32的数据类型。
数据排布格式（format）	详细请参见 **数据排布格式（format）**。

- 形状（Shape）
  张量的形状，以(D0, D1, … ,Dn-1)的形式表示，D0到Dn是任意的正整数。

如形状(3,4)表示第一维有3个元素，第二维有4个元素，(3,4)表示一个3行4列的矩阵。

在形状的中括号中有多少个数字，就代表这个张量是多少维的张量。形状的第一个元素要看张量最外层的中括号中有几个元素，形状的第二个元素要看张量中从左边开始数第二个中括号中有几个元素，依此类推。例如：

表 7-2 张量的形状举例

张量	形状
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

- 数据排布格式（format）

  在深度学习框架中，多维数据通过多维数组存储，比如卷积神经网络的特征图用四维数组保存，四个维度分别为批量大小（Batch，N）、特征图高度（Height，H）、特征图宽度（Width，W）以及特征图通道（Channels，C）。

  由于数据只能线性存储，因为这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据，比如Caffe，排列顺序为[Batch, Channels, Height, Width]，即NCHW。Tensorflow中，排列顺序为[Batch,Height, Width, Channels]，即NHWC。

  如图7-2所示，以一张格式为RGB的图片为例，NCHW实际存储的是“RRRGGGBBB”，同一通道的所有像素值顺序存储在一起，而NHWC实际存储的则是“RGBRGBRGB”，多个通道的同一位置的像素值顺序存储在一起。

  图 7-2 NCHW 和 NHWC

  昇腾AI软件栈中，为了提高数据访问效率，所有张量数据统一采用NC1HWC0的五维数据格式。其中C0与微架构强相关，等于AI Core中矩阵计算单元的大小，对于FP16类型为16，对于INT8类型则为32，这部分数据需要连续存储；C1=(C+C0-1)/C0。如果结果不整除，最后一份数据需要补零以对齐C0。

  整个NHWC -> NC1HWC0的转换过程为：

  a. 将NHWC数据在C维度进行分割，变成C1份NHWC0。
  b. 将C1份NHWC0在内存中连续排列，由此变成NC1HWC0。

算子属性

不同的算子包含的属性值不同，下面介绍几个常见的算子属性。

- 轴（axis）

axis代表张量中维度的下标，比如张量a是一个5行6列的二维数组，即shape是(5,6)，则axis=0表示是张量中的第一维，即行。axis=1表示是张量中的第二维，即列。

例如张量数据[[[1,2],[3,4]], [[5,6],[7,8]]]，Shape为(2,2,2)，则轴0代表第一个维度的数据即[[1,2],[3,4]]与[[5,6],[7,8]]这两个矩阵，轴1代表第二个维度的数据即[1,2]、[3,4]、[5,6]、[7,8]这四个数组，轴2代表第三个维度的数据即1，2，3，4，5，6，7，8这八个数。

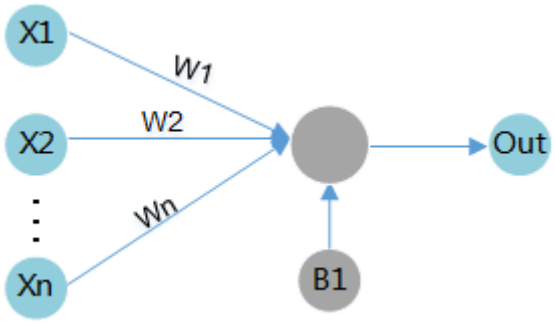
轴axis可以为负数，此时表示是倒数第axis个维度。

- 偏差（bias）

  偏差是除了权重之外，另一个被应用于输入数据的线性分量。它被加到权重与输入数据相乘的结果中，用于改变权重与输入相乘所得结果的范围。

  如图7-3所示，假设输入数据为X1，与其相关联的权重为W1，偏差为B1，那么在通过计算单元后，数据变为了X1*W1+B1。

  图 7-3 偏差计算示例

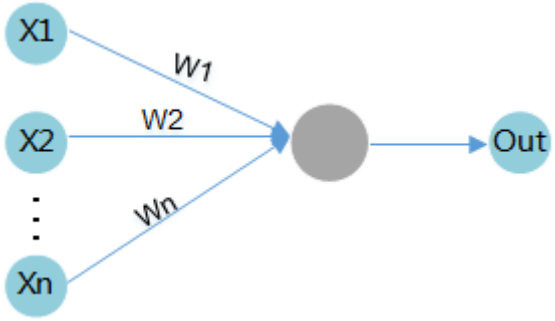


- 权重（weights）

  当输入数据进入计算单元时，会乘以一个权重。例如，如果一个算子有两个输入，则每个输入会分配一个关联权重，一般将认为较重要数据赋予较高的权重，不重要的数据赋予较小的权重，为零的权重则表示特定的特征是无需关注的。

  如图7-4所示，假设输入数据为X1，与其相关联的权重为W1，那么在通过计算单元后，数据变为了X1*W1。

  图 7-4 权重计算示例



算子样例—Reduction

Reduction是caffe中的一个算子，将多维数组的指定轴及之后的数据做reduce操作。

- Reduction的属性
  - ReductionOp：算子支持的操作类型，包含四种类型。

    表 7-3 Reduction 算子操作类型

算子类型	说明
SUM	对被reduce的所有轴求和。
ASUM	对被reduce的所有轴求绝对值后求和。
SUMSQ	对被reduce的所有轴求平方后再求和。
MEAN	对被reduce的所有轴求均值。

  - axis：Reduction需要指定一个轴，会对此轴及其之后的轴进行reduce操作，取值范围为：[-N，N-1]。

    比如，输入的张量的形状为(5,6,7,8)。

    - 如果指定的轴是3，则输出tensor的形状为(5,6,7)。
    - 如果指定的轴是2，则输出tensor的形状为(5,6)。
    - 如果指定的轴是1，则输出tensor的形状为(5,)。
    - 如果指定的轴是0，则输出tensor的形状为(1,)。
  - coeff：标量，对结果缩放，如果值为1，则不进行缩放。
- Reduction算子的数据
  - 输入数据

    输入包含张量数据及针对张量数据描述。

    输入数据：张量x。

    张量x的描述包含如下属性：

    表 7-4 Reduction 算子输入

输入参数	说明
x	张量的name，输入数据，形状由参数shape确定。
shape	输入数据的形状，维度为N。
dtype	输入的数据类型，支持如下类型： float16，float32。

  - 输出数据

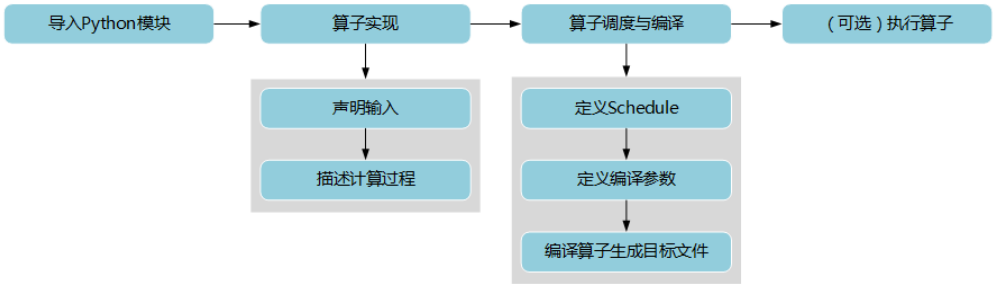
    y：与输入x数据类型相同的张量，其形状由输入张量形状和指定轴决定。

7.2 算子代码实现

7.2.1 实现流程

TE算子代码通过Python语言开发，实现流程如图7-5所示。

图 7-5 TE 自定义算子实现流程图



说明

- 支持的自定义算子的输入数据类型为：float16, int8, int16, int32, uint8, uint16, bool。
  - 不同计算操作支持的数据类型不同，详细请参见《TE API参考》。
  - TE API同时支持float16与float32数据类型，但OMG进行模型转换的时候会将float32数据类型转换成float16，所以当前版本在进行自定义算子开发时，不支持使用float32的数据类型。
- TE提供了一些已经自定义实现的算子样例代码，用户可参考或直接使用，代码存储路径为DDK安装目录下的“ddk/site-packages/topi-0.4.0.egg/topi/cce”。

7.2.2 导入 Python 模块

导入昇腾AI软件栈提供的Python模块，代码示例如下所示。

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

其中：

- “te.lang.cce”：引入TE支持的特定域语言接口，包括常见的运算vmuls、vadds、matmul等。

  具体的接口定义可查看DDK安装路径下“/site-packages/te-0.4.0.egg/te/lang/cce/”目录下的python函数，使用方法请参见《TE API参考》。
- “te.tvm”：引入TVM后端代码生成机制。

  具体的接口定义可查看DDK安装路径下“/site-packages/te-0.4.0.egg/te/tvm”目录下的python函数，使用方法请参见**https://docs.tvm.ai/**。
- “topi.generic”：提供了自动算子调度接口。

  具体的接口定义可查看DDK安装路径下“/site-packages/topi-0.4.0.egg/topi/generic”目录下的所有python函数，使用方法请参见《TE API参考》。

7.2.3 算子实现

算子实现函数定义

如下所示，一个算子的实现函数中包含了输入张量的形状，数据类型，算子属性，内核名称，以及相应的编译、打印等配置。该函数会被插件代码调用，在离线模型生成器进行模型转换时执行。

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

其中：

- shape：输入张量的形状，若算子有多个输入，且每个输入的shape不同，则此处需定义多个shape用于后续对每个输入张量占位，若多个输入张量的shape相同，则可以定义一个shape。
- dtype：输入张量的数据类型。
- *attribute1*，*attribute2*...：算子的属性，根据算子的实际定义进行代码编辑。
- kernel_name：算子在内核中的名称（即生成的二进制文件的名称），用户自定义，保持唯一，只能是大小写字母、数字、“_”的组合，且必须是字母或者“_”开头，长度小于或等于200个字符。
- 编译配置参数“need_build”：取值范围为**True**或者**False**，代表是否需要进行编译。
- 打印配置参数“need_print”：取值范围为**True**或者**False**，代表是否需要打印算子的中间表示（IR：Intermediate Representation）。

例如，

对于Reduction算子，实现函数定义如下：

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

对于Matmul算子，实现函数定义如下：

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

算子实现逻辑

TE的算子实现逻辑总体概括为：

定义好输入数据的张量占位符，然后调用**te.lang.cce**中的各种特定域语言接口进行计算过程的描述，如下代码示例所示：

```
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)        //对缩放参数进行处理，将输入张量乘上一个标量
    tmp = data_tmp_input
    res_tmp = te.lang.cce.sum(tmp, axis=axis)            //在轴axis上做求和操作
    res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag=True)     //进行数据类型的转换
```

其中：

- data为输入张量，使用TVM的**placeholder**接口进行定义，placeholder是一个占位符，返回一个Tensor对象，表示一组输入数据。

  若算子有多个输入张量，此处需要定义多个Tensor对象。例如：

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

- vmuls（向量乘），sum（求和）组成中间的计算逻辑。
- cast_to（转换数据类型）：输出张量需要与输入张量的数据类型保持一致，如果计算过程中对数据类型做了转换，需要使用cast_to接口将输出张量的数据类型转换为输入张量的数据类型。

  例如：如果输入张量的数据类型为int8，则执行vmuls操作时，会将数据类型转换为float16，则需要在计算逻辑结束后调用cast_to接口将输出张量的数据类型由

float16转换为int8，由于使用vmuls将int8转换为float16的数值时，小数部分为0，所以“f1628IntegerFlag”设置为True，代码示例为：

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

“te.lang.cce.cast_to”接口的详细使用方法请参见《TE API参考》中的“Compute接口”。

- res为输出张量，其数据类型与输入张量的数据类型一致。

用户在进行算子逻辑实现前，可以自定义实现代码对输入数据进行预处理，如下代码示例所示：

```
    # 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 算子调度与编译

如下代码所示，当定义完计算逻辑后，使用auto_schedule机制，便可以自动生成相应的调度，此处通过TVM的打印机制可以看到相应计算的中间表示。配置信息包括是否需要打印、编译以及算子内核名以及输入、输出张量。

```
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)
```

- 使用“generic”的“auto_schedule”接口，自动生成相应的调度（schedule），“auto_schedule”接口的参数为算子的输出张量。

  schedule可以理解为：描述的计算过程如何在硬件上高效执行。就是把相关的计算和硬件设备上的相关指令对应起来。schedule对象中包含一个“中间表示”（IR），它用一种类似伪代码来描述计算过程，可以通过“need_print”参数把它打印出来进行查看。

- “tensor_list”（张量列表）中保存输入张量、输出张量，这个顺序需要严格按照算子本身的输入、输出数据顺序排列。

  例如："tensor_list": [tensor_a, tensor_b, res]，tensor_a与tensor_b是输入张量，res为输出张量。

- 根据调度和配置使用“te.lang.cce”提供的“cce_build_code”接口来进行算子编译，算子编译过程会根据输入的数据形状、类别、算子参数等编译出专用内核，这个过程在离线模型生成器转换模型时发生。
  - sch：生成的算子计算schedule对象。
  - config：编译参数配置的map。

  编译完成后，会生成算子目标文件.o文件（运行目标为AI Core的算子）或者.so文件（运行目标为AI CPU的算子）与算子描述文件.json文件。

7.2.5 执行算子

自定义算子代码编写完成后，可以在算子的*.py代码结尾加入调用算子的语句，如下所示，可以参照**7.4 算子运行验证**构造输入数据，验证算子执行结果是否正确。

例如：

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

7.3 代码样例

打开拷贝过来的算子样例文件reduction.py

详细的代码解释请参考如下注释。

```
#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
    """
    #基本参数校验
    #shape参数的校验，check_shape_rule()函数的定义文件为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参数校验
    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参数校验
    if operation not in reduction_op:
        raise RuntimeError("op can only be one of SUM, ASUM, SUMSQ , MEAN")
    # coeff参数校验
    if type(coeff) != int and type(coeff) != float:
        raise RuntimeError("coeff must be a value")
    # 参数预处理
    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()
    # 定义输入数据Tensor，仅占位，未分配实际内存
    data = tvm.placeholder(shape1, name="data_input", dtype=inp_dtype)
    # 定义算子计算过程
    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

        #按axis进行求和，降维
        res_tmp = te.lang.cce.sum(tmp, axis=axis)
        #进行数据类型的转换
        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)
        #生成算子计算的schedule对象
        sch = generic.auto_schedule(res)
    #定义编译参数
    config = {"print_ir": need_print,
              "need_build": need_build,
              "name": kernel_name,
              "tensor_list": [data, res]}
    #编译算子，生成目标文件
    te.lang.cce.cce_build_code(sch, config)

#以shape为(2,3,4)，datatype为float16，axis为1，op为SUM，coeff为2，算子名称为Reduction的参数来调用reduction算子
if __name__ == "__main__":
    reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2,kernel_name = "Reduction")
```

7.4 算子运行验证

本节描述如何单独运行用户开发的算子，验证算子实现的正确性。

7.4.1 算子编译

对算子代码进行编译，生成算子的二进制文件及算子描述文件。

步骤1 算子编译依赖DDK的版本号，所以编译算子前需要获取当前使用的DDK的版本号。

在“$HOME/tools/che/ddk/ddk/ddk_info”文件中查看DDK版本号。

如图7-6所示，VERSION字段即为当前DDK版本。

图 7-6 DDK 版本号查看

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

步骤2 执行如下命令设置版本号信息、编译算子。

1. 在customop_te/operator目录下执行如下命令进入python交互模式。

   python

2. 在python交互模式下依次执行如下命令设置DDK的版本号。

   from topi.cce import te_set_version

   import subprocess

   te_set_version("1.3.T18.B850")

   执行示例如图7-7所示。

   图 7-7 设置 DDK 版本号示例

```
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. 执行如下命令编译reduction.py文件，生成算子二进制文件与描述文件。

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

   图 7-8 编译算子文件

```
>>> 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. 退出python交互模式。

   quit(0)

步骤3 算子编译完成后，会在当前operator目录下生成kernel_meta文件夹，kernel_meta文件夹下为编译生成的算子二进制文件*.o文件（运行目标为AI Core的算子）或者*.so文件（运行目标为AI CPU的算子）及算子描述文件*.json文件。

- *.o或者*.so是算子的二进制文件。
- .json文件是对应的算子描述文件，用于定义算子属性及运行时所需要的资源。

  json文件解析如下所示：

```
{
"binFileName":"Reduction",          //生成的算子的二进制文件名称
"binFileSuffix":".o",           //生成的算子的二进制文件后缀，运行目标为AI CPU时后缀为".so"
"blockDim":1,                   //计算时用到的AI Core的数量
"kernelName":"Reduction__kernel0",   //算子的kernel函数名
"magic":"RT_DEV_BINARY_MAGIC_ELF",   //代表算子运行目标为AI Core，若magic值为
RT_DEV_BINARY_MAGIC_ELF_AICPU，代表运行目标是AI CPU
```

```
"sha256":"d5158df2ff2e64743eec7f527ddd9078b99c1d670f5adbd8b789224657ab0f91"  //对.o文件进行加密得到的值
}
```

----结束

7.4.2 构造输入数据文件

运行单算子需要构造算子输入数据，并以二进制格式保存到算子工程下。

步骤1 以DDK安装用户进入自定义算子工程的算子代码目录下。

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

步骤2 执行样例数据生成脚本，生成reduction算子的样例数据文件。

python data_gen.py reduction

在算子工程中会生成如图7-9所示数据文件。

图 7-9 样例数据生成

```
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
```

数据文件说明如表7-5所示。

表 7-5 数据文件说明

数据文件	说明
Reduction_input_2_3_4_sum_axis_1.data	reduction算子的二进制格式输入数据文件。
Reduction_input_2_3_4_sum_axis_1.txt	将reduction算子的二进制格式文件以txt文件的方式显示出来，方便用户查看结果。
Reduction_output_2_3_4_sum_axis_1.data	reduction算子二进制格式输出数据验证文件，用于验证算子运行后的输出结果是否正确。
Reduction_output_2_3_4_sum_axis_1.txt	将reduction算子的二进制格式输出数据文件以txt文件的方式显示出来，方便用户查看结果。

----结束

7.4.3 运行单算子

步骤1 在DDK 样例**customop_runner**中生成单算子编译二进制文件。

1. 给customop_runner工程样例赋予写权限。

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

2. 进入DDK样例**customop_runner**工程的build目录。

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

3. 设置DDK_PATH的环境变量。

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

4. 执行如下命令生成Makefile文件。
   - 对于Atlas 200 DK开发者板环境，执行如下命令。

     cmake -Dtarget=OI .
   - 对于设备类型为ASIC的环境，执行如下命令。

     cmake .

5. 执行make命令，编译生成二进制文件。

   make

   在“$HOME/tools/che/ddk/ddk/sample/customop/customop_runner/out/”目录下会生成可执行文件**main和动态库libcustom_engine.so**

步骤2 把out目录拷贝到TE自定义算子工程目录。

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

步骤3 配置输入数据、输出数据及验证描述文件。

进入TE自定义算子工程目录中的out文件夹。

cd $HOME/tools/projects/customop_te/out

- 在input.txt文件中配置输入数据文件路径及文件名。

  例如：

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

  如果自定义算子有多个输入数据，需要自定义多个输入数据的txt文件，例如input1.txt、input2.txt等，分别按照input.txt的格式进行定义。
- 在output.txt中配置输出数据的相关信息

  例如：

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

  - size：预期的输出数据文件大小，单位字节。
  - dataPath：输出数据的路径及文件名。
  - dtype：输出数据类型，其中**1**代表**float1**6，**2**代表**float32**。
- 在expect.txt文件中配置期望生成数据的路径及文件名。

  例如：

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

步骤4 将operator文件夹与out文件夹以HwHiAiUser用户拷贝到开发者板或者ASIC设备的Host侧，operator与out文件夹应放置在同一目录下。

例如拷贝到“/home/HwHiAiUser/projects”目录下。

步骤5 运行算子

以HwHiAiUser用户登录开发者板或者ASIC类型设备的Host侧，并进入out目录, 例如“/home/HwHiAiUser/projects/out” 。

1. 在当前目录下创建output文件夹，用于存储生成的数据文件。此文件路径及名称为output.txt文件中定义的。

mkdir output

2. 执行如下命令为main文件添加可执行权限。

   chmod +x main

3. 执行如下命令，运行单算子并进行算子比对。

   ./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

   其中：

   - -i：输入数据配置文件，配置了输入数据的路径及文件名。

     如果自定义算子有多个输入数据，需要自定义多个输入数据的txt文件，例如input1.txt、input2.txt等，分别按照input.txt的格式进行定义。
   - -o：输出数据配置文件，配置了输出数据的大小、路径及文件名，输出数据的类型。
   - -e：期望数据配置文件，配置了期望输出数据的路径及文件名，用于做结果比对。
   - -b：算子实现的二进制文件*.o（运行目标为AI Core的算子）或者*.so（运行目标为AI CPU的算子）。
   - -p：精度偏差，表示数据相对误差的允许范围，取值为[0,1)，精度偏差越小表示精度越高。
   - -d：统计偏差，整个数据集中精度偏差不满足门限的数据量占比，数值范围（0，1），统计偏差越小表示精度越高。
   - -k：核函数名称，TE算子的核函数名称需要与.json文件中的kernelName保持一致；C++算子的内核名称需要与算子在Framework注册的opetype的名称保持一致。
   - -t：有三种取值，0代表TE_AI Core算子，1代表TE_AI CPU算子，2代表C++的自定义算子（AI CPU算子）。

     对于TE算子，是运行在AI Core上还是运行在AI CPU上可以通过算子编译完成后生成的****.json**文件中的“magic”参数值进行查看。

   执行成功后，会在当前目录的output文件夹下会生成名称为out0.data结果文件及名称为vertifyResult.txt的验证数据比对结果文件。

   vertifyResult.txt文件样例如下，代表实际输出数据与期望输出数据内容比对一致。

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

----结束

8 caffe.proto 文件算子定义（可选）

如果使用的是Caffe开源框架的模型，请参考本章节进行caffe.proto文件的定义；如果使用的Tensorflow开源框架的模型，此章节跳过。

算子开发完成后，需要在DDK内置的caffe.proto文件中增加自定义算子的相关定义（如果内置的caffe.proto文件中已存在此算子的定义则此章节跳过）。

若同一模型中存在多个不支持的自定义算子，请参考本章节一次性将相关算子定义进行添加。算子插件实现时会根据算子名称从caffe.proto文件读取相关的算子参数进行解析，进而将算子数据结构转换为适配昇腾AI处理器的离线模型支持的数据结构。

DDK中内置的caffe.proto文件的路径为：“$HOME/tools/che/ddk/ddk/include/inc/custom/proto/caffe/caffe.proto”，用户可基于此文件进行修改，添加自定义算子的定义。

此处以添加Reduction算子的定义为例（DDK中内置的caffe.proto已经添加了Reduction算子的定义，此处以此样例进行说明）。

步骤1 在LayerParameter中添加Reduction算子的定义。

如下所示，在LayerParameter中加入Reduction层的定义，并设置其ID。

```
message LayerParameter {
......
optional ReductionParameter reduction_param = 136;   //此ID需要保持唯一
......
}
```

步骤2 在caffe.proto文件中增加Reduction层的参数定义。

```
message ReductionParameter {
  enum ReductionOp {          //算子支持的操作类型
    SUM = 1;                   //对被reduce的所有轴求和
    ASUM = 2;                  //对被reduce的所有轴求绝对值后求和
    SUMSQ = 3;                 //对被reduce的所有轴求平方后再求和
    MEAN = 4;                  //对被reduce的所有轴求均值
  }
  optional ReductionOp operation = 1 [default = SUM];   // 定义算子的操作
  optional int32 axis = 2 [default = 0];                //定义reduce的轴
  optional float coeff = 3 [default = 1.0]; // coefficient for output  //标量，对结果进行缩放的倍数
}
```

----结束

9 自定义算子插件开发

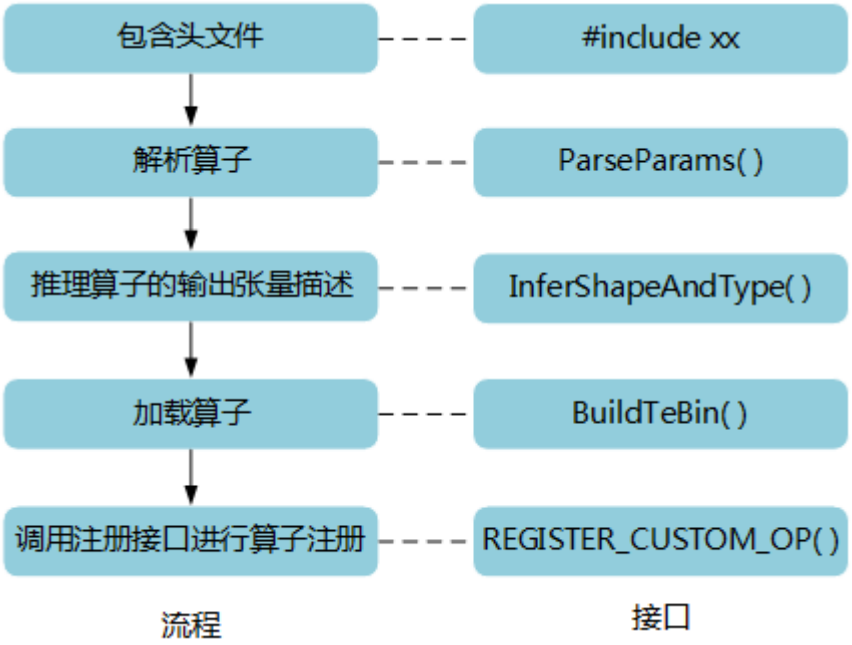
9.1 插件代码实现
9.2 插件代码样例
9.3 编译算子插件

9.1 插件代码实现

9.1.1 插件实现流程

自定义算子开发完成后，离线模型生成器需要支持将自定义算子的属性值转换到离线模型，在离线模型执行阶段能够支持自定义算子的执行，所以用户需要开发自定义算子插件，进行算子属性的解析、形状类别的推断，并通过离线模型生成器提供的注册机制完成自定义算子的注册。

图 9-1 自定义算子插件实现流程图



9.1.2 包含头文件

应在插件实现文件的头部使用预编译命令“#include”将插件实现函数相关的头文件包含到插件实现源文件中。

```
#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>
```

表 9-1 头文件说明

头文件	目录	作用
custom/custom_op.h	DDK安装目录下的/include/inc/custom/custom_op.h。	包含该头文件，可调用自定义的编译函数及算子调测函数、算子校验函数。
framework/omg/register.h	DDK安装目录下的/include/inc/framework/omg/register.h。	包含该头文件，可使用算子注册类相关，调用算子注册相关的接口。
framework/omg/omg_types.h	DDK安装目录下的/include/inc/framework/omg/omg_types.h。	包含该头文件，可使用TE自定义算子信息结构体TEBinInfo。
proto/caffe/caffe.pb.h	算子插件编译时，在算子工程所在目录下生成proto/caffe/caffe.pb.h文件。	算子插件编译时，会自动编译DDK安装目录下的“/include/inc/custom/proto/caffe/caffe.proto”文件，并在算子工程所在目录下生成proto/caffe/caffe.pb.h文件供插件代码调用进行算子参数解析。
operator.h	DDK安装目录下的include/inc/graph/operator.h。	包含该头文件，可调用算子属性存取、输入/输出设置接口。
attr_value.h	DDK安装目录下的/include/inc/graph/attr_value.h。	包含该头文件，可使用AttrValue类下各数据类型。
memory	C++标准库。	包含该头文件，可使用C++标准库中的智能指针、内存分配器、临时性的申请与释放动态内存的函数、在内存上构造对象的函数等。

头文件	目录	作用
string	C++标准库。	包含该头文件，可使用string类构造对象并调用string相关接口。
vector	C++标准库。	包含该头文件，可使用vector类模板并调用vector相关接口。

算子插件编译前，由于工程中不存在proto/caffe/caffe.pb.h文件，所以工程中如果出现如下提示信息，请忽略。

图 9-2 caffe.pb.h 无法解析提示信息

```
#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 解析算子

对于用户新开发的自定义算子，需要自定义实现解析算子中属性的函数，将原始模型中的算子属性定义转换为适配昇腾AI处理器的离线模型中的算子属性定义。如果用户重写昇腾AI处理器内置算子，则此步骤跳过，插件会自动实现对内置算子的解析。

函数声明

算子解析函数的声明如下所示：

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

- ParseParamsxx：函数名称，用户自定义，需要保持唯一。
- op_origin：源算子模型，为protobuf格式的数据结构，来源于Caffe模型的proto文件，存储路径为DDK安装目录下的“/include/inc/custom/proto/caffe/caffe.proto”，若用户自定义的算子在caffe.proto文件中未定义，则需要参考**8 caffe.proto文件算子定义（可选）**增加算子定义。

  算子插件实现时会根据算子名称从caffe.proto编译生成的proto/caffe/caffe.pb.h文件与caffe.pb.cc文件中读取相关的算子属性进行解析，进而将算子数据结构转换为离线模型支持的数据结构。

  DDK中内置的caffe.proto文件的路径为DDK安装目录下的：“/include/inc/custom/proto/caffe/caffe.proto”，用户可基于此文件进行修改，添加自定义算子的定义。
- op_dest：目标算子模型，适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息，Operator类的详细描述请参见《GE API参考》中的“Operator类接口”。

实现流程

ParseParamsxx函数的实现流程如下所示：

步骤1 定义指向LayerParameter的对象，并获取当前算子层的句柄。

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

其中：

- param对象的类型caffe::xxxParameter中的xxxParameter需要与LayerParameter对象中声明的类型保持一致。
- layer对象的成员函数xxx_param()的名字需要与与LayerParameter对象中声明的对象名字保持一致。

例如caffe.proto中Reduction算子及Convolution算子的定义如下：

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

则获取当前Reduction算子层及Convolution算子层的句柄代码定义如下：

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

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

步骤2 解析算子的每一个属性，并将其赋给Operator类型的对象op_dest。

开发者可调用CreateFrom<AttrValue::**T**>(**DT**&& val)接口将**DT**类型的参数转换为AttrValue类的**T**类型的参数，然后调用SetAttr(const string& name, const AttrValue& value)接口将转换后的AttrValue对象的值赋给op_dest对象的相应属性。

T类型是昇腾AI软件栈为了简化类型定义，对当前支持的数据类型做的重命名，T类型与原始数据类型的对应关系如表9-2所示，原型定义可参考DDK安装目录的“include/inc/graph/attr_value.h”文件。

表 9-2 T 类型与原始数据类型对应关系

T类型	原始数据类型
INT	int64_t
FLOAT	float
STR	std::string
TENSOR	TensorPtr
TENSOR_DESC	TensorDesc
GRAPH	ComputeGraphPtr
BYTES	Buffer
NAMED_ATTRS	NamedAttrs
BOOL	bool

T类型	原始数据类型
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>

SetAttr接口的详细描述请参见《GE API参考》。

下面给出几种常见类型的参数解析示例：

- int或者float类型参数

  例如，caffe.proto文件中算子参数定义如下所示：

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

  调用SetAttr接口，将“param.axis()”的值赋给op_dest对象的axis属性，并将类型转换为INT；将“param.coeff()”的值赋给op_dest对象的coeff属性，并将类型转换为FLOAT，如下所示：

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

  其中CreateFrom<AttrValue::**T**>(**DT**&& val)是将**DT**类型的参数转换为AttrValue类的**T**类型的参数。

- enum类型参数

  例如，caffe.proto文件中算子属性定义如下所示：

```
message ReductionParameter {
  enum ReductionOp {
    SUM = 1;
    ASUM = 2;
    SUMSQ = 3;
    MEAN = 4;
  }
...}
```

  a. 首先，需要将enum类型参数转换为map类型参数。

```
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. 调用SetAttr接口，将map类型参数*operation_map*赋值给op_dest对象的operation属性。

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

SetAttr接口的详细描述请参见《GE API参考》。

- repeated类型参数

  例如caffe.proto文件中算子参数定义如下所示：

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

a. 首先，将repeated float类型参数转换为vector<float>类型参数，将repeated uint32类型参数转换为vector<uint32>类型参数，并对vector类型的参数进行赋值。

```
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));   //调用vector类型对象的push_back函数为min_size对象赋值
}
for(int i = 0; i < param.offset_size(); ++i)
{
   offset.push_back(param.offset(i));      //调用vector对象的push_back函数为offset对象赋值
}
```

b. 调用SetAttr接口，将vector<float>类型参数min_size赋值给op_dest对象的min_size属性；将vector<uint32>类型参数offset赋值给op_dest对象的offset属性，类型为LIST_INT。

```
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));
```

SetAttr接口的详细描述请参见《GE API参考》。

----结束

9.1.4 推理算子输出张量描述

用户需要根据算子的输入张量描述、算子逻辑及算子属性，推理出算子的输出张量描述，包括张量的形状、数据类型及数据排布格式等信息。这样离线模型转换时就可以为所有的张量静态分配内存，避免动态内存分配带来的开销。

函数声明

函数的声明如下所示：

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

- InferShapeAndTypexx：函数名称，用户自定义，需要保持唯一。
- op：计算节点定义，存储输入张量描述及各种算子属性，ge::Operator类型的介绍请参见《GE API参考》中的“Operator类接口”。
- v_output_desc：存储该计算节点的输出张量描述，包括形状、数据排布格式及数据类型，关于TensorDesc类型的介绍请参见《GE API参考》中的“TensorDesc类接口”。

下面详细讲解不同场景下InferShapeAndTypexx函数的实现。

输出张量与输入张量形状相同的算子

对于输出张量与输入张量形状相同的算子，可以直接将输入张量的描述插入输出张量描述所在的向量空间中。

代码示例如下所示：

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

其中GetInputDesc是Operator类中根据算子Input名称或者Input索引获取输入张量描述的接口，详细的接口介绍请参见《GE API参考》中的“Operator类接口”。

降低维度的算子

对于Reduction、Reduce之类的常见降维操作，需要根据算子输入属性axis等信息计算出输出张量的形状（包含输出张量维度以及每一个维度有多少个元素），然后将输出张量的形状插入到v_output_desc向量中。

代码示例如下所示：

步骤1 获取算子的输入张量描述及输入张量的形状信息。

```
auto tensorDesc = op.GetInputDesc(0);        //输入张量描述，包括的形状、数据排布格式及数据类型
auto shape = tensorDesc.GetShape();          //获取输入张量的形状
```

GetShape接口的详细介绍请参见《GE API参考》中的“TensorDesc类接口”。

步骤2 根据计算逻辑获取算子的属性值，并计算算子输出张量的形状。

例如，对于Mylenet网络中的Reduction算子，由于Reduction的上一层为Softmax，Softmax的实际输出是2维，但在离线模型中会被补齐到4维，所以需要调整axis，将其指向2维的位置，进行reduce操作，并将调整后的Shape值赋给输出张量描述。

- 从operator对象中获取axis属性的键值对，然后从其中获取axis的属性值，并将其从INT类型转换为int64_t类型赋给名称为axis的变量，最后对axis的值进行校验、调整，将其指向轴1的位置，代码示例如下所示：

```
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;
    }
```

- 调整Shape，将轴1及其之后的维度设置为1，例如输入张量的Shape为(2,3,4,5)，则调整之后的Shape为(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);
}
```

- 将调整后的Shape值设置到tensorDesc对象。

```
tensorDesc.SetShape(shape);
```

GetDimNum与SetDim接口的详细介绍请参见《GE API参考》中的“Shape类接口”。

步骤3 设置算子的输出张量描述。

```
v_output_desc.push_back(tensorDesc)
```

将tensorDesc赋给输出张量的描述对象v_output_desc。

----结束

网络存在 Type 相同的多个算子

一个网络中往往存在多层某个Type相同的算子，例如Convolution算子，开发者有时需要对某层算子进行形状的自定义（重定义已经存在的某层算子），此时在进行输出张量描述推理的时候需要根据网络的不同情况，根据num_output、kernel、stride、pad等属性判断是针对哪一层的算子进行的自定义，从而推断出对应算子的张量信息。

代码示例如下所示：

步骤1 将输入张量描述赋给输出张量描述，并获取算子的输入张量描述及输入张量的形状信息。

```
v_output_desc.push_back(op.GetInputDesc(0));  //将输入张量描述信息赋给输出张量描述对象，开发者也可以将后续推理出的shape直接赋给tensorDesc对象，再将tensorDesc对象赋给输出张量描述对象。
auto tensorDesc = op.GetInputDesc(0);        //输入张量描述，包括的形状、数据排布格式及数据类型
auto shape = tensorDesc.GetShape();          //获取输入张量的形状
```

GetShape接口的详细介绍请参见《GE API参考》中的“TensorDesc类接口”。

步骤2 根据计算逻辑获取算子的属性值，并根据算子的属性值、shape匹配算子，并计算算子输出张量的形状。

例如，对于某一网络中的Type为Convolution的算子（网络中存在多层Convolution），匹配num_outputs为128、shape.GetDim(0)为1、shape.GetDim(1)为128、shape.GetDim(2) 为28、shape.GetDim(3) 为28的Convolution，并对其输出描述张量的形状重新赋值。

- 获取num_outputs的属性值

```
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");
    }
```

- 匹配num_outputs为128、shape.GetDim(0)为1、shape.GetDim(1) 为128、shape.GetDim(2) 为28、shape.GetDim(3) 为28的Convolution，并对其输出张量描述的形状重新赋值。

```
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;
}
```

GetDimNum与SetDim接口的详细介绍请参见《GE API参考》中的“Shape类接口”。

说明

若需要使用op_name进行匹配，获取op_name的方法如下所示：

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

其他属性kernel_w、kernel_h、stride_w、stride_h、pad_w、pad_h的获取方法与num_output相同，修改op.GetAttr中的key值即可。

----结束

9.1.5 编译算子

函数声明

算子编译函数的声明如下所示：

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

其中：

- BuildTeBinxx：函数名称，用户自定义，需要保持唯一。
- op：目标算子模型，适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息，Operator类的详细描述请参见《GE API参考》中的“Operator类接口”。
- te_bin_info：存储自定义算子二进制文件路径、算子描述文件路径以及DDK的版本信息。TEBinInfo结构体的详细描述请参见《Framework API参考》中的“TEBinBuildFn函数”。

实现流程

算子编译函数在离线模型生成器进行模型转换时调用，主要完成以下流程：

- 获取算子张量描述信息以及算子属性，模型转换时，这些信息都需要是确定值，用于进行算子的匹配。

  例如，如下所示，模型转换时只匹配axis为1、输入张量Shape的Dim为4的Reduction算子。

```
  //解析算子属性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");
   }

   // 解析算子属性axis ,并调整axis指向Mylenet网络中Reduction算子的上一层Softmax算子的实际输出维度，即轴1的位置。
   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;

    // 解析算子属性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");
    }
    // 获取算子输入张量描述信息
    TensorDesc input_desc      = op.GetInputDesc(0);

    // 解析Input shape并校验算子的Dim是否为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;
    }
```

- 指定算子实现文件、算子实现函数以及内核名。

```
    FilePath   = "project_path/operator/reduction";    //算子实现文件所在的绝对路径+算子py文件名称
    FuncName   = "Reduction";                //算子实现文件中算子实现函数的名称。
    KernelName = "Reduction";                //算子实现文件中算子实现函数中定义的kernel_name，即生成
的二进制文件的名称。
```

- 指定算子编译生成的算子描述文件（*.json）文件的路径，请保持如下固定配置。

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

  模型转换时，会从此路径中读取算子的二进制描述文件，获取算子相关信息。

  执行omg模型转换命令时，会根据FilePath中配置的算子实现的路径拷贝算子编译生成的kernel_meta文件夹到执行omg命令时所在的路径，所以*.json文件相对于执行omg命令所在目录的路径固定为“./kernel_meta”。

- 调用te::BuildCustomop函数来调用算子实现文件的python函数编译算子。

  BuildCustomop函数调用如下所示：

```
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());
```

  其中：

  - te_bin_info.ddk_version：DDK的版本信息，固定参数。后续模型转换时会自动传入DDK版本信息。
  - op.GetName()：获取算子名称，固定参数。
  - FilePath：前面定义的算子文件所在相对路径。
  - FuncName：算子实现文件中的算子实现函数名称。
  - *"(i,i,i,i), s, i, s, f, s"*：代表算子实现文件中实现函数的参数占位符，其中i表示整形，s表示字符串类型，f表示单精度浮点数类型，o表示PyObject*类型。占位符需要和后面紧跟的参数顺序和类型保持一致，同时需要和算子实现文件中的算子实现函数的定义保持一致。BuildCustomop会根据这些参数调用算子实现函数，再通过TVM机制生成内核。

9.1.6 算子注册

框架管理器（Framework）提供REGISTER_CUSTOM_OP宏，按照指定的算子名称完成算子的注册。

自定义算子的注册代码如下所示：

```
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});
```

其中：

- REGISTER_CUSTOM_OP：注册自定义算子，"test_layer"作为离线模型文件中的算子名称，可以任意命名但不能和已有的算子命名冲突。
- FrameworkType：不同框架的算子参数解析逻辑不同，因此对于不同框架模型需要有不同的插件，在插件的注册代码需要表明对应的框架。当前请配置为CAFFE。
- OriginOpType：算子类型，需要与Caffe Prototxt中定义的算子类型保持一致，否则无法正常解析，内置的caffe.proto文件路径为DDK安装路径下的“/include/inc/custom/proto/caffe/caffe.proto”。
- ParseParamsFn：用来注册解析模型的函数，“ParseParamsxx”即**9.1.3 解析算子**中实现的函数，针对Caffe框架开发插件时需要，若自定义的算子为重写昇腾AI处理器已经支持的算子，则此步骤可跳过；若自定义的算子为昇腾AI处理器不支持的算子，则此步骤必选。
- InferShapeAndTypeFn：用来注册形状和类别推断函数，“InferShapeAndTypexx”即**9.1.4 推理算子输出张量描述**中实现的函数。
- TEBinBuildFn：用来注册TE算子编译函数，“BuildTeBinxx”即**9.1.5 编译算子**中实现的函数。
- ImplyType：指定算子的实现方式，ImplyType::TVM表示该算子是TE算子。
- Formats：算子输入数据与输出数据的数据排布格式，其中第一个列表是输入数据格式列表，第二个列表为输出数据格式列表，若输入数据有多个，请在第一个列表中列出每一个输入数据的排布格式。例如若有两个输入数据，分别是NC1HWC0格式，则Formats函数调用方式如下：

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

  详细说明请参见《Framework API参考》中的“Formats函数”。
- WeightFormats：设置算子的权重数据的排布格式，支持的数据格式类型请参见《Framework API参考》中的“WeightFormats函数”。例如Convolution的filter的数据排布格式是fractal_Z，bias的数据排布格式是NC1HWC0。

  如果模型转换时开启量化开关，需要在此接口中增加Framework处理时新增的常量数据的格式，当前Framework支持的量化算子有Conv，FC与Depthwise Conv，若这几个算子在模型转换时开启了量化开关，则需要在WeightFormats接口的参数列表最后增加6个DOMI_TENSOR_NC1HWC0的数据格式（量化时Framework新增了6个数据排布格式为NC1HWC0的常量，此处不详细介绍），开启量化后的Convolution的WeightFormats接口示例如下：

  ```
  .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 插件代码样例

步骤1 修改“customop_te/plugin/caffe_reduction_layer.cpp”代码文件。

- 修改“FilePath”为当前算子文件所在绝对路径+算子py文件名称。

FilePath = "*project_path*/operator/reduction";

- 修改FuncName为reduction.py文件中算子实现的函数名称。

  FuncName ="reduction"

- 修改KernelName为reduction.py文件中设置的算子的KernelName。

  KernelName = "Reduction";

- 修改算子的二进制文件*.o及描述文件.json所在的路径。

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

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

  此路径为相对于当前工程目录的相对路径，例如Reduction.o文件在当前工程目录的operator/kernel_meta目录下。

步骤2 修改后的reduction算子插件的代码样例如下所示：

```
#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
{
// 用户自定义重写ParseParams函数，解析算子参数
Status CaffeReductionParseParams(const Message* op_origin, ge::Operator& op_dest)
{
    // 将op_origin转化为layer对象
    const caffe::LayerParameter* layer =
        dynamic_cast<const caffe::LayerParameter*>(op_origin);

    // 校验输入参数合法性
    if (nullptr == layer)
    {
        printf("Dynamic cast op_src to LayerParameter failed\n");
        return FAILED;
    }
    // 将算子参数中的枚举类型转换为map类型
    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" },
    };
    // 获取当前算子层的句柄
    const caffe::ReductionParameter& param = layer->reduction_param();
    // 将算子中所有参数赋值给离线模型算子数据结构op_dest对象
    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;
```

```
}

// 自定义重写InferShapeAndType函数，获取算子的输出描述
Status CaffeReductionInferShapeAndType(const ge::Operator& op, vector<ge::TensorDesc>& v_output_desc)
{
    // 获取算子输入的TensorDesc对象
    auto tensorDesc      = op.GetInputDesc(0);
    // 获取输入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);
    }
    // 将调整后的shape信息设置到tensorDesc
    tensorDesc.SetShape(shape);
    v_output_desc.push_back(tensorDesc);

    return SUCCESS;

}


// 编译算子，获取算子的二进制文件及算子描述文件
Status CaffeReductionBuildTeBin(const ge::Operator& op, TEBinInfo& te_bin_info)
{
    // 算子相关参数声明
    std::string FilePath   = "";
    std::string FuncName   = "";
    std::string KernelName = "";

    std::string operation  = "";
    int64_t     axis       = -1;
    float       coeff      = 1;
    // 获取算子参数值
    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");
    }
    // 获取算子输入描述信息
    TensorDesc input_desc      = op.GetInputDesc(0);

    // 解析Input shape并校验算子的Dim是否为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;
    }
    // 设置算子文件路径
    FilePath   = "/home/ascend/tools/projects/customop_te/operator/reduction";
    // 设置实现算子的函数名
    FuncName   = "reduction";
    // 设置算子实现文件中定义的算子名称
    KernelName = "Reduction";

    // i => int; s => string; f => dobule; O => bool, and bool value is Py_True or Py_False，调用Tensor Engine
的编译接口编译算子
    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());    //此处的参数顺序需要与算子实现函数中参数顺序保持一致。

    // 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
```

----结束

9.3 编译算子插件

步骤1 以DDK安装用户登录DDK所在服务器。

步骤2 在“projects/customop_te/plugin”目录下修改Makefile文件。

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

vi Makefile

- 修改生成的算子插件名称。

```
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即为生成的算子插件名称，用户可自行修改。

  lib_caffe_parser.so是生成的解析caffe.proto文件的库文件，名称不可修改。若是Caffe算子，请确保同一模型中的所有不支持的自定义算子已在**8 caffe.proto文件算子定义（可选）**完成定义。

- 将“TOPDIR”指定到DDK的安装目录，例如

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

- Makefile中其他内容无需改动，为算子插件编译通用模板。

步骤3 执行编译操作。

在当前plugin目录下执行如下命令进行算子插件的编译。

make

编译完成后会在当前目录下生成算子插件文件**libcaffe_reduction_layer.so**。

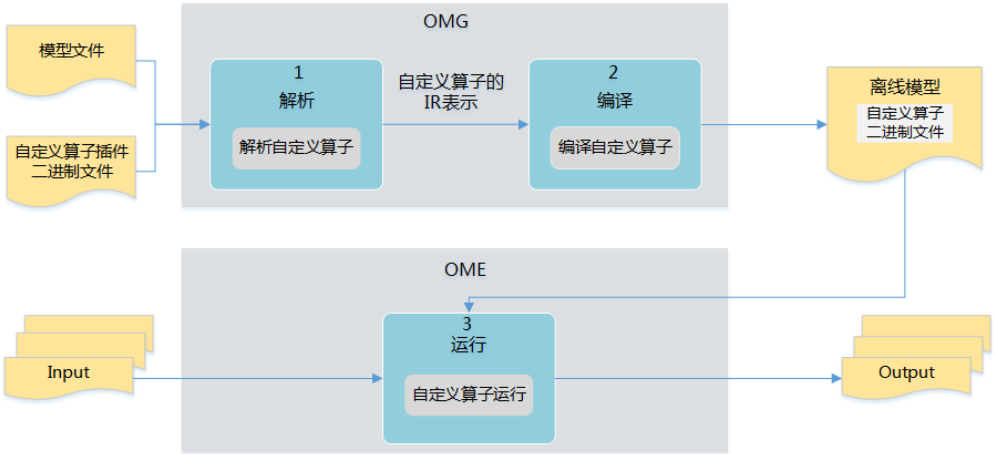
----结束

10 加载插件转换模型

原理说明

加载了用户自定义算子插件的模型转换流程如图10-1所示

图 10-1 加载插件进行模型转换流程



1. OMG（Offline Model Generate）加载模型文件、自定义算子插件，对模型文件中的算子进行解析，并将自定义算子转换为IR（Intermediate representation）表示。
2. OMG根据运行环境对自定义算子进行数据转换、运行内存计算，同时编译生成自定义算子的二进制文件（*.o）文件，并生成离线模型（*.om）文件
3. 应用程序运行时，OME会获取输入数据，加载离线模型文件进行算子的循环调用执行，输出结果数据。

操作流程

步骤1 以DDK安装用户进入自定义算子开发工程根目录。

cd $HOME/tools/projects/customop_te/

步骤2 执行如下命令进行模型转换。

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项的值为mylenet网络的原始模型文件的相对路径。
- --weight项的值为mylenet网络的预训练模型文件的相对路径。
- --framework为原始框架类型。
  - 0：caffe
  - 3：tensorflow
- --plugin项的值为自定义算子插件所在目录。
- --output项的值为输出模型文件的名称，用户可自定义。
- --ddk_version项的值为自定义算子运行需要匹配DDK的版本号，可在“$HOME/tools/che/ddk/ddk/ddk_info”文件中查看DDK版本号。

说明

更多参数说明请参考《模型转换指导》。

----结束

11 算子插件开发函数参考

11.1 ParseParamsFn函数
11.2 InferShapeAndTypeFn函数
11.3 TEBinBuildFn函数
11.4 WeightFormats函数
11.5 算子构建函数注册宏

11.1 ParseParamsFn 函数

函数功能

参数解析函数。

函数原型

OpRegistrationData& ParseParamsFn(ParseParamFunc parseParamFn);

参数说明

参数	输入/输出	说明
parseParamFn	输入	回调函数ParseParamFunc，请参见**回调函数ParseParamFunc**。

回调函数 ParseParamFunc

用户自定义并实现ParseParamFunc类函数，完成caffe模型参数和权值的转换，将结果填到Operator类中。

函数原型

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

参数说明

参数	输入/输出	说明
op_origin	输入	protobuf格式的数据结构（来源于caffe模型的prototxt文件），包含算子参数信息。
op_dest	输出	适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息。 关于Operator类，请参见《GE API参考》中的“Operator类接口”。

11.2 InferShapeAndTypeFn 函数

函数功能

维度推导函数。

函数原型

OpRegistrationData& InferShapeAndTypeFn(InferShapeFunc inferShapeFn);

参数说明

参数	输入/输出	说明
inferShapeFn	输入	回调函数InferShapeFunc，请参见**回调函数InferShapeFunc**。

回调函数 InferShapeFunc

用户自定义并实现InferShapeFunc类函数，用于获取算子的输出描述，包括输出shape信息、数据类型等张量描述信息。

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI处理器的离线模型的算子数据结构。 关于Operator类，请参见《GE API参考》中的“Operator类接口”。

参数	输入/输出	说明
v_output_desc	输出	存储算子的输出描述。 关于TensorDesc类，请参见《GE API参考》中的“TensorDesc类接口”。

11.3 TEBinBuildFn 函数

函数功能

build回调函数。

函数原型

OpRegistrationData& TEBinBuildFn(BuildTeBinFunc buildTeBinFn);

参数说明

参数	输入/输出	说明
buildTeBinFn	输入	回调函数BuildTeBinFunc，参见**回调函数BuildTeBinFunc**。

回调函数 BuildTeBinFunc

用户自定义并实现BuildTeBinFunc类函数，用于构建算子二进制文件。

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息。 关于Operator类，请参见《GE API参考》中的“Operator类接口”。

参数	输入/输出	说明

teBinInfo	输出	存储自定义算子二进制文件路径和ddk描述的数据。
		struct TEBinInfo
		{
		std::string bin_file_path; //自动从json文件binFileName字段获取。为了兼容以前用户编写的用例，字段不删除。
		std::string json_file_path;
		std::string ddk_version;
		};

11.4 WeightFormats 函数

函数功能

设置算子权重支持的数据格式。

函数原型

```
OpRegistrationData& WeightFormats(
const std::initializer_list<domi::tagDomiTensorFormat>& weight_formats);
```

参数说明

参数	输入/输出	说明
weight_formats	输入	权重支持的数据格式列表： 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 算子构建函数注册宏

宏功能

调用DOMI_REGISTER_OP完成算子注册。

宏原型

REGISTER_CUSTOM_OP(name)

参数说明

参数	输入/输出	说明
name	输入	算子在离线模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。

12 附录

12.1 修订记录

12.1 修订记录

文档版本	发布日期	修改说明
01	2020-05-09	第一次正式发布。