CANN
6.3.RC2

Ascend Graph 开发指南

文档版本 01

发布日期 2023-07-26

华为技术有限公司

商标声明


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暗示的声明或保证。

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

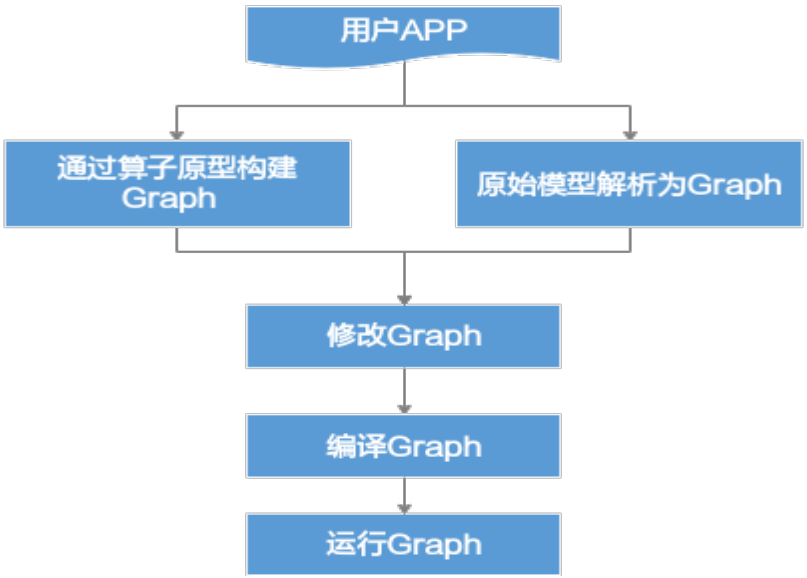
1 概述

简介

本文档用于指导开发者使用AscendCL接口进行计算图的构建、修改、编译、执行。

不局限于Caffe、TensorFlow等深度学习框架，用户可以通过开放的AscendCL编程接口，基于算子原型进行构图。同时，也可以通过框架解析功能将主流的模型格式转换成CANN模型格式，从而隔离上层框架的差异，当前仅支持对Caffe/TensorFlow/ONNX原始框架模型的解析。

通过算子原型构建Graph或者将原始模型解析为Graph后，可以对图进行修改、编译、执行操作，从而实现模型的表达和高层次的编译优化。



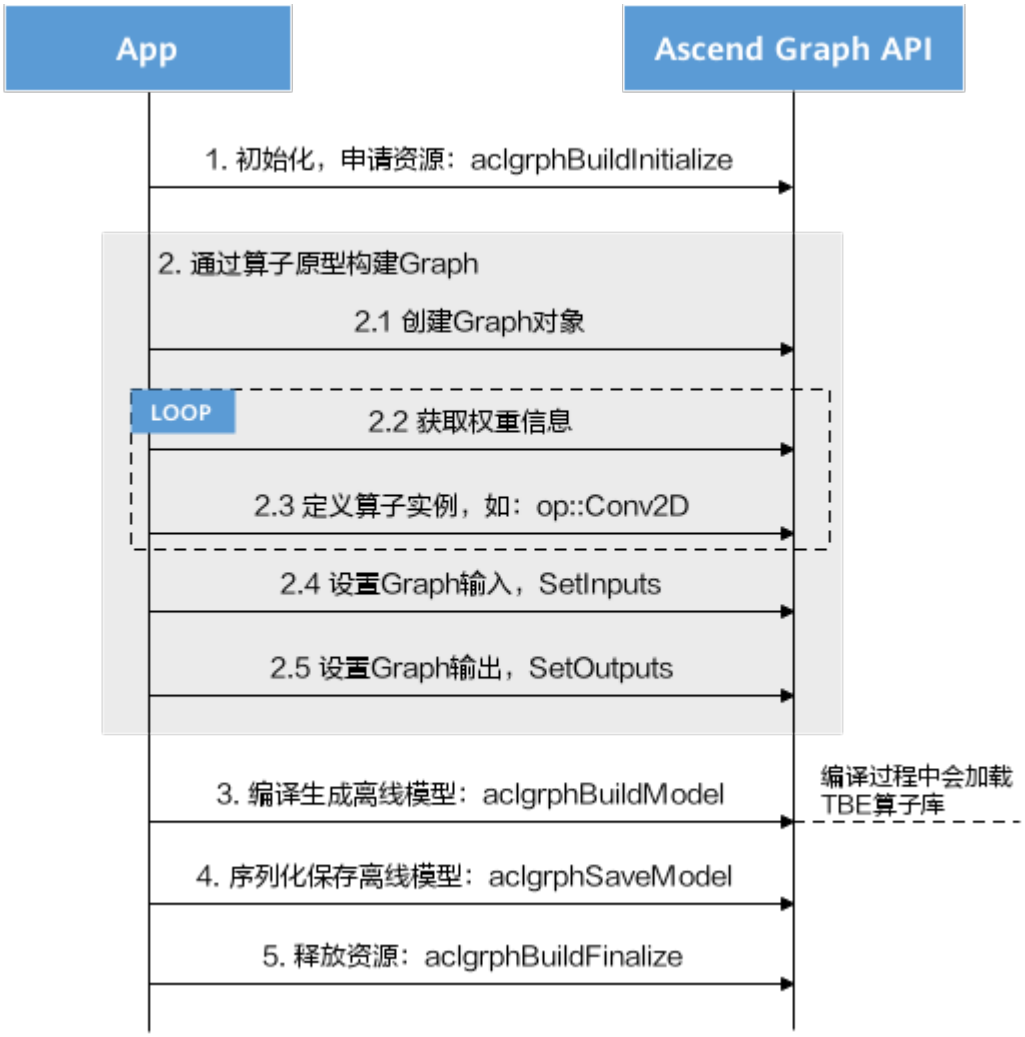
典型业务流程

网络模型定义格式大同小异，主要由Tensor、Node（Operator）、Graph组合而成。

- Tensor包括Tensor描述及数据两部分，Tensor描述包括了该Tensor的name、dtype、shape、format信息。
- Operator包括算子的name、type、输入、属性信息。
- Graph包括网络的name、算子列表、输入算子、输出算子。

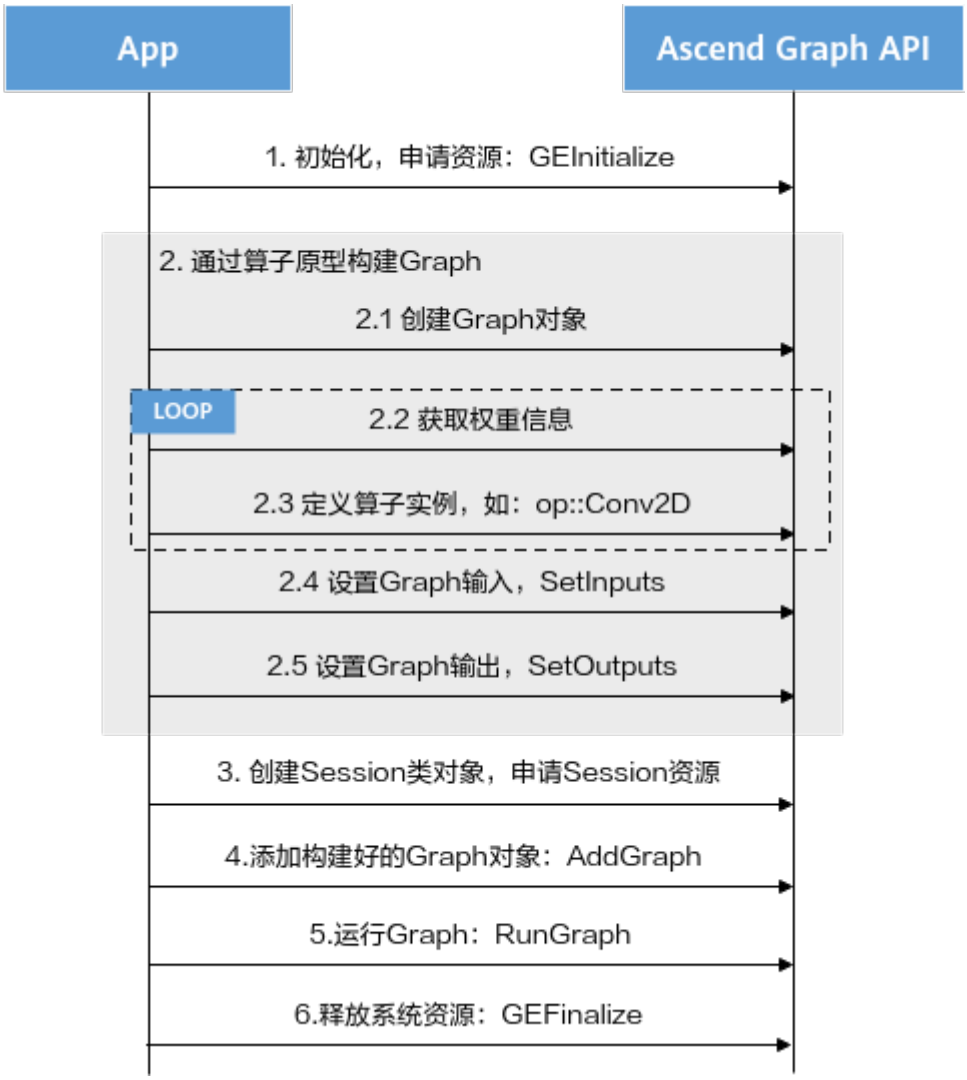
如果您仅需要把Graph编译并保存为适配昇腾AI处理器的离线模型，编译生成的离线模型可以通过aclmdlLoadFromFile接口加载，并通过aclmdlExecute接口执行推理模型，调用AscendCL接口完成构图及图编译的具体流程如图1-1所示。

图 1-1 模型构建流程

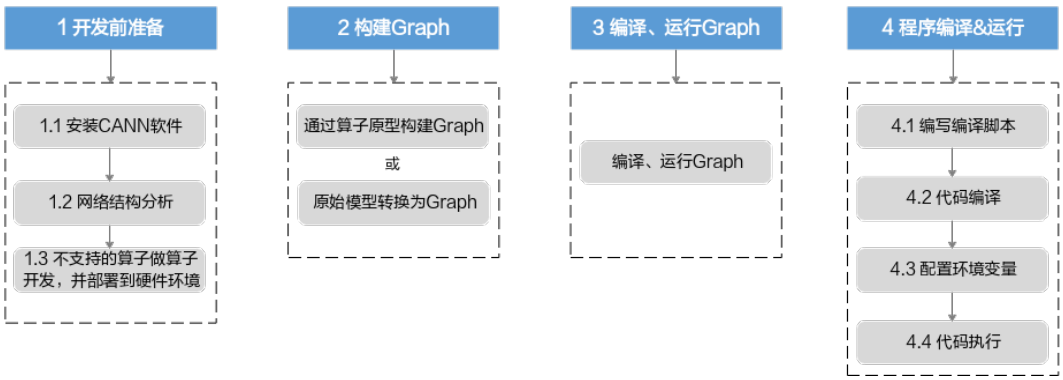


如果您需要编译并直接运行Graph，调用AscendCL接口完成构图及图编译运行的具体流程如图1-2所示。

图 1-2 编译并直接运行流程图



整体开发流程



2 开发前准备

环境准备

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行. **${install_path}/set_env.sh**命令设置环境变量。

- “Ascend-cann-toolkit安装目录/ascend-toolkit/latest/oppvi g/built-in/op_proto/inc”下提供了AICore和AICPU算子原型定义，用于通过算子原型构建Graph。
- “Ascend-cann-toolkit安装目录/ascend-toolkit/latest/include/graph”下提供了Graph构建接口。
- “Ascend-cann-toolkit安装目录/ascend-toolkit/latest/include/ge”下提供了Graph运行接口。
- “Ascend-cann-toolkit安装目录/ascend-toolkit/latest/compiler/lib64/stub”下为相关依赖库。

网络分析

- 如果通过算子原型构建Graph，需要根据原始网络，明确如下信息：
  a. 网络中包含哪些算子，以及这些算子的输入、输出、属性等信息。
  b. 网络中算子之间的关联关系。
  c. 确认原始网络中的算子在昇腾AI处理器是否支持，当前支持的算子请参考《**算子清单**》。如果不支持或不满足实际需要，请参考《**TBE&AI CPU算子开发指南**》自定义算子，并将算子部署到硬件环境。
- 如果是将原始模型解析为Graph，需要确认原始网络中的算子在昇腾AI处理器是否支持，当前支持的算子请参考《**算子清单**》。如果不支持或不满足实际需要，请参考《**TBE&AI CPU算子开发指南**》自定义算子，并将算子部署到硬件环境。

3 通过算子原型构建 Graph

3.1 什么是算子原型

什么是算子原型

算子原型规定了在昇腾AI处理器上可运行算子的约束，主要体现算子的数学含义，包含定义算子输入、输出、属性。

在算子开发阶段，会使用REG_OP宏，以“.”链接INPUT、OUTPUT、ATTR等接口注册算子的输入、输出和属性信息，最终以OP_END_FACTORY_REG接口结束，完成算子的注册。注册代码实现如下所示：

```
namespace ge{
    REG_OP(OpType) //算子类型名称
    .INPUT(x1, TensorType({ DT_FLOAT, DT_INT32 }))
    .INPUT(x2, TensorType({ DT_FLOAT, DT_INT32 }))
    // .DYNAMIC_INPUT(x, TensorType{DT_FLOAT, DT_INT32})
    // .OPTIONAL_INPUT(b, TensorType{DT_FLOAT})
    .OUTPUT(y, TensorType({ DT_FLOAT, DT_INT32 }))
    // .DYNAMIC_OUTPUT(y, TensorType{DT_FLOAT, DT_INT32})
    .ATTR(x, Type, DefaultValue)
    // .REQUIRED_ATTR(x, Type)
    .OP_END_FACTORY_REG(OpType)
}
```

- REG_OP(*OpType*)

  OpType：注册到昇腾AI处理器的自定义算子库的算子类型。

- INPUT(*x1,* TensorType({ *DT_FLOAT,DT_UINT8,...* }))

  注册算子的输入信息。

  - x：宏参数，算子的输入名称。
  - TensorType({ *DT_FLOAT,DT_UINT8,...* })：“{ }”中为此输入支持的数据类型的列表。

  若算子有多个输入，每个输入需要使用一条INPUT(*x,* TensorType({ *DT_FLOAT,DT_UINT8,...* }))语句进行描述。

- DYNAMIC_INPUT(*x*, TensorType{*DT_FLOAT, DT_INT32*, ...})

  算子为动态多输入场景下的输入信息注册。

  - x：宏参数，算子的输入名称，图运行时，会根据输入的个数自动生成x0、x1、x2……，序号依次递增。
  - TensorType({ *DT_FLOAT,DT_UINT8,...* })：“{ }”中为此输入支持的数据类型的列表。

- OPTIONAL_INPUT(*x*, TensorType{*DT_FLOAT, ...*})

  若算子输入为可选输入，可使用此接口进行算子输入的注册。

  - x：宏参数，算子输入的名称。
  - TensorType{*DT_FLOAT, ...*}：“{ }”中为此输入支持的数据类型的列表。

- OUTPUT(y, TensorType({ *DT_FLOAT,DT_UINT8,...* }))

  注册算子的输出信息。

  - y：宏参数，算子的输出名称。
  - TensorType({ *DT_FLOAT,DT_UINT8,...* })：“{ }”中为此输出支持的数据类型的列表。

  若算子有多个输出，每个输出需要使用一条OUTPUT(*x,* TensorType({ *DT_FLOAT,DT_UINT8,...* }))语句进行注册。

- DYNAMIC_OUTPUT(y, TensorType{DT_FLOAT, DT_INT32})

  算子为动态多输出场景下的输出信息注册。

  - y：宏参数，算子的输出名称，图运行时，会根据输出的个数自动生成y0、y1、y2……，序号依次递增。
  - TensorType({ *DT_FLOAT,DT_UINT8,...* })：“{ }”中为此输出支持的数据类型的列表。

- ATTR(x, Type，*DefaultValue*)

  注册算子的属性，包括算子的属性名称，属性类型以及属性值的默认值，当开发者不设置算子对象的属性值时使用默认值。

  例如：ATTR(mode, Int, 1)，注册属性mode，属性类型为int64_t，默认值为1。

  若算子有多个属性，每个属性需要使用一条ATTR(x, Type，*DefaultValue*)语句或者REQUIRED_ATTR(x, Type)语句进行注册。

- REQUIRED_ATTR(x, Type)

  注册算子的属性，包括算子的属性名称与属性类型，无默认值，开发者必须设置算子对象的属性值。

  若算子有多个属性，每个属性需要使用一条ATTR(x, Type，*DefaultValue*)语句或者REQUIRED_ATTR(x, Type)语句进行注册。

- OP_END_FACTORY_REG(*OpType*)：结束算子注册。*OpType*与REG_OP(*OpType*)中的*OpType*保持一致。

说明

*DT_FLOAT,DT_UINT8*等数据类型对应关系请参见**10.1.2.10.2 DataType**。

如何获取算子原型

在模型构建时，用户需要了解算子原型，包括输入、输出和属性信息，从而创建算子实例，构建自己的Graph。

- 对于用户自定义算子，请自行获取自定义算子的算子原型定义头文件，了解算子的原型定义。

  从“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/opp/built-in/op_proto/custom/inc”算子原型定义的头文件中获取，例如：

```
REG_OP(SubMConv3dCube)
    .INPUT(x, TensorType({DT_FLOAT16}))
    .INPUT(filter, TensorType({DT_FLOAT16}))
   .OUTPUT(y, TensorType({DT_FLOAT16}))
   .ATTR(is_first, Bool, false)
   .OP_END_FACTORY_REG(SubMConv3dCube)
```

- 对于内置算子，用户可以通过如下两种方式获取算子原型：
  - 从《**算子清单**》获取，如**图3-1**所示。

    图 3-1 查看算子原型信息

    SoftmaxV2 ( )

    Applies the Softmax function to an n-dimensional input Tensor rescaling them
    so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

    Inputs:
    One input: x: A mutable Tensor. Must be one of the following types: float16, float32, double. Should be a Variable Tensor.

    Attributes:
    axes: A list of ints. The dimension softmax would be performed on.

    Outputs:
    y: A Tensor. Has the same dimensionality and shape as the "x" with values in the range [0, 1]. Must be one of the following types: float16, float32, int32.

    Third-party framework compatibility
    Compatible with the TensorFlow operator BatchMatmul.

  - 从“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/opp/built-in/op_proto/inc”算子原型定义的头文件中获取，例如：

```
REG_OP(SoftmaxV2)
    .INPUT(x, TensorType({DT_DOUBLE, DT_FLOAT16, DT_FLOAT}))
    .OUTPUT(y, TensorType({DT_DOUBLE, DT_FLOAT16, DT_FLOAT}))
    .ATTR(axes, ListInt, {-1})
    .OP_END_FACTORY_REG(SoftmaxV2)
```

3.2 如何通过算子原型构建 Graph

功能介绍

使用REG_OP宏将算子原型注册成功后，会自动生成对应的衍生接口（参见**10.1.1.2 衍生接口说明**），用户可以通过这些接口在Graph中定义算子，然后创建一个Graph实例，并在Graph中设置输入算子、输出算子，从而完成Graph构建。

使用算子原型衍生接口定义算子

下面仅介绍使用算子原型衍生接口定义算子的通用过程，具体各类算子（例如数据节点、计算节点）的定义示例，请参考**3.3 各类算子表达**。

1. 包含的头文件。
   ```
   #include "all_ops.h"
   ```
   - 对于内置算子，需要包括all_ops.h头文件，定义内置算子类型，可使用内置算子类型相关的接口，头文件所在路径为“CANN软件安装目录/opp/built-in/op_proto/inc/all_ops.h”。
   - 对于自定义算子，需要包括自定义算子的原型定义头文件，头文件所在路径为“CANN软件安装目录/opp/op_proto/custom/inc/xx.h”。
2. 创建算子实例。

   使用REG_OP宏注册算子类型后，会自动生成算子类型构造函数explicit OpType(const char* name)，相当于定义了一个op::xxx的Class，开发者include该原型头文件，实例化该Class构建Graph。在构建Graph时可直接传算子名称作为入参，例如：
   ```
   auto softmax = op::SoftmaxV2("softmax")
   ```
   注意：图中的算子名称**必须唯一**。
3. 设置算子输入。

   算子原型中定义了算子的输入名称、输入类型，以及算子支持的数据类型。根据输入类型，可将算子输入分为可选输入、必选输入和动态输入。

   对于可选输入、必选输入和动态输入，需要通过不同的接口设置。
   - 设置算子**必选输入和可选输入**：通过“set_input_*输入名称*”设置，例如：
     ```
     auto softmax = op::SoftmaxV2("softmax")        //创建SoftmaxV2算子实例
       .set_input_x(bias_add_3);                    //设置SoftmaxV2算子输入为bias_add_3
     ```
   - 设置算子**动态输入**：通过“create_dynamic_input_*输入名称*”创建动态输入、“set_dynamic_input_*输入名称*”设置动态输入，示例请参考**3.3.5 定义动态多输入算子（AddN）**。
4. 设置算子属性。

   算子原型中定义了算子的属性名称、属性类型、属性支持的数据类型、属性的默认值及取值范围。根据属性类型，可将算子属性分为：必选属性

（REQUIRED_ATTR，一定要在算子定义时设置属性值）和可选属性（ATTR，不设置算子对象的属性值时使用默认值）。

对于**必选属性和可选属性**，都可以通过“set_attr_*属性名称*”接口设置，例如：

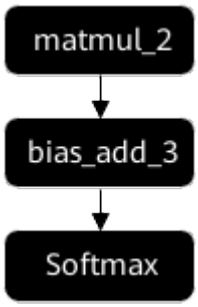
```
auto maxpool1 = op::MaxPool("MaxPool1")
.set_input_x(tanh1)
.set_attr_ksize({1, 1, 2, 1})              //设置ksize属性值
.set_attr_strides({1, 1, 2, 1})            //设置strides属性值
.set_attr_padding("SAME");                 //设置padding属性值
```

算子连接边表达

算子之间的连边分为数据边和控制边。数据边用于指定算子的输入，控制边用于控制算子的执行顺序。

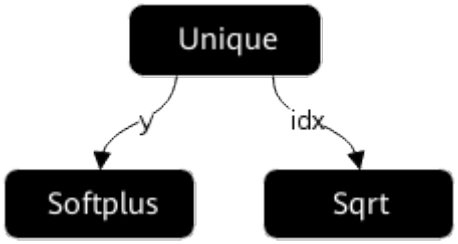
1. 数据边表达。

   对于前一个算子只有一个输出，可以通过“set_input_输入名称”接口传入前一个算子的名称。


   ```
   auto bias_add_3 = op::BiasAdd("bias_add_3")
     .set_input_x(matmul_2)
     .set_input_bias(bias_add_const_3)
     .set_attr_data_format("NCHW");
   auto softmax = op::SoftmaxV2("Softmax")
     .set_input_x(bias_add_3);
   ```

   如果前一个算子有多个输出，则需要传入上一个算子的名称和输出名称，或者传入上一个算子的名称和输出索引。

   传入上一个算子的名称和输出名称：

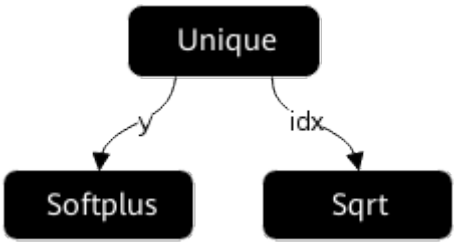
   ```
   auto unique= op::Unique("unique").set_input_x(data)
   auto softplus = op::Softplus("softplus").set_input_x(unique, "y");  //创建softplus算子，设置输入为unique算子的y输出
   auto sqrt = op::Sqrt("sqrt").set_input_x(unique, "idx");  //创建sqrt算子，设置输入为unique算子的idx输出
   ```

   传入上一个算子的名称和输出索引：

   ```
   auto unique= op::Unique("unique").set_input_x(data)
   auto softplus = op::Softplus("softplus").set_input_x(unique, 0);  //创建softplus算子，设置输入为unique算子的第一个输出
   auto sqrt = op::Sqrt("sqrt").set_input_x(unique, 1);  //创建sqrt算子，设置输入为unique算子的第二个输出
   ```

2. 控制边表达。

如果图中某个算子的执行依赖于图中其他算子执行完，如下图所示，如果需要控制先执行Sqrt再执行Softplus，则需要调用**10.1.2.3.2 AddControlInput**接口，对Softplus算子增加控制边。



代码示例：

```
auto unique= op::Unique("unique").set_input_x(data)
auto sqrt = op::Sqrt("sqrt").set_input_x(unique, "idx");
auto softplus = op::Softplus("softplus").set_input_x(unique, "y").AddControlInput(sqrt);
```

创建 Graph 实例

完成算子定义后，需要创建Graph实例，并在Graph中设置输入算子、输出算子，主要过程为：

1. 包含所需的头文件。
   ```
   #include "graph.h"
   ```
2. 创建Graph对象。
   ```
   Graph graph("IrGraph");
   ```
   相关接口请参考**10.1.3.3.1 Graph构造函数和析构函数**。
3. 设置Graph输入和输出算子，使用到的主要接口为：
   - 设置Graph内的输入算子：**10.1.3.3.2 SetInputs**
   - 设置Graph内的输出算子：**10.1.3.3.3 SetOutputs**

   例如设置Graph的输入为Data算子，输出为Softmax算子：

   ```
   std::vector<Operator> inputs{data};
   std::vector<Operator> outputs{softmax};
   graph.SetInputs(inputs).SetOutputs(outputs);
   ```

   如果输入为多个Data算子，需要保证inputs入参顺序和Data算子index属性指定的顺序保持一致，否则后面生成模型时会报错。例如：

   ```
   // 准备第一个输入数据
   auto shape_data0 = vector<int64_t>({1,17,2,2});
   TensorDesc desc_data0(ge::Shape(shape_data0), FORMAT_ND, DT_FLOAT);
   auto data0 = op::Data("data0").set_attr_index(0) ;  //创建data0算子，index属性为0
   data0.update_input_desc_x(desc_data0);         //设置算子输入描述
   data0.update_output_desc_y(desc_data0);        //设置算子输出描述
   // 准备第二个输入数据
   auto shape_data1 = vector<int64_t>({1,5,2,2});
   TensorDesc desc_data1(ge::Shape(shape_data1), FORMAT_ND, DT_FLOAT);
   auto data1 = op::Data("data1").set_attr_index(1) ;  //创建data1算子，index属性为1
   data1.update_input_desc_x(desc_data1);         //设置算子输入描述
   data1.update_output_desc_y(desc_data1);        //设置算子输出描述

   //设置Graph输入算子
   std::vector<Operator> inputs{data0, data1};
   ```

3.3 各类算子表达

3.3.1 定义数据节点（Data）

Graph的输入节点，也就是数据节点，使用Data算子实现。

Data算子原型定义：

```
REG_OP(Data)
    .INPUT(x, TensorType::ALL())
    .OUTPUT(y, TensorType::ALL())
    .ATTR(index, Int, 0)
    .OP_END_FACTORY_REG(Data)
```

根据Data算子原型定义创建Data算子实例，名称为data，初始化参数为desc_data。同时通过“update_input_desc_输入名称”和“update_output_desc_输出名称”接口设置Shape、Format和Dtype，和用户需要处理的数据信息保持一致。

```
auto shape_data = vector<int64_t>({1,17,2,2});
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
auto data = op::Data("data");                 //创建data算子
data.update_input_desc_x(desc_data);          //设置算子输入描述
data.update_output_desc_y(desc_data);         //设置算子输出描述
```

需要注意的是，定义数据节点时，**必须**通过“update_input_desc_输入名称”和“update_output_desc_输出名称”接口设置Shape、Format和Dtype。

3.3.2 定义数据节点（Const）

权值、偏置等信息为常量Tensor，可以使用Const算子实现。

Const算子原型定义：

```
REG_OP(Const)
    .OUTPUT(y, TensorType({DT_FLOAT, DT_FLOAT16, DT_INT8, DT_INT16, DT_UINT16, \
        DT_UINT8, DT_INT32, DT_INT64, DT_UINT32, DT_UINT64, DT_BOOL, DT_DOUBLE}))
    .ATTR(value, Tensor, Tensor())
    .OP_END_FACTORY_REG(Const)
```

直接构造权重数据

根据Const算子原型定义创建Const算子实例，初始值（即属性value的值）为weighttensor1。

```
//构造weighttensor1
TensorDesc weight_desc(ge::Shape({1,3,3,3}), FORMAT_NCHW, DT_INT8);
int bs_size_weight=27;
int8_t * bs_inputData_weight = nullptr;
bs_inputData_weight = new int8_t[bs_size_weight];
for(int i=0; i<bs_size_weight; ++i) {
        *(bs_inputData_weight+i) = 1;
    }
Tensor weighttensor1(weight_desc, (uint8_t*)bs_inputData_weight, bs_size_weight*sizeof(int8_t));

//创建Const算子，初始值（即属性value的值）为weighttensor1
auto weight1 = op::Const().set_attr_value(weighttensor1);
```

从文件读入权重数据

除了直接构造权重数据外，也可以直接从bin文件读入权重数据。

```
//构造weight_tensor
auto weight_shape = ge::Shape({ 5,17,1,1 });
TensorDesc desc_weight_1(weight_shape, FORMAT_NCHW, DT_INT8);
Tensor weight_tensor(desc_weight_1);
uint32_t weight_1_len = weight_shape.GetShapeSize();
```



```
bool res = GetConstTensorFromBin(PATH+"const_0.bin", weight_tensor, weight_l_len*sizeof(int8_t));

//创建Const算子，初始值（即属性value的值）为weight_tensor
auto conv_weight = op::Const("const_0").set_attr_value(weight_tensor);
```

GetConstTensorFromBin函数实现：

```
bool GetConstTensorFromBin(string path, Tensor &weight, uint32_t len) {
    ifstream in_file(path.c_str(), std::ios::in | std::ios::binary);
    if(!in_file.is_open()) {
        std::cout << "failed to open" << path.c_str() << '\n';
        return false;
    }
    in_file.seekg(0, ios_base::end);
    istream::pos_type file_size = in_file.tellg();
    in_file.seekg(0, ios_base::beg);

    if(len != file_size) {
        cout << "Invalid Param.len:" << len << " is not equal with binary size（" << file_size << ")\n";
        in_file.close();
        return false;
    }
    char* pdata = new(std::nothrow) char[len];
    if(pdata == nullptr) {
        cout << "Invalid Param.len:" << len << " is not equal with binary size（" << file_size << ")\n";
        in_file.close();
        return false;
    }
    in_file.read(reinterpret_cast<char*>(pdata), len);
    auto status = weight.SetData(reinterpret_cast<uint8_t*>(pdata), len);
    if(status != ge::GRAPH_SUCCESS) {
        cout << "Set Tensor Data Failed"<< "\n";
        in_file.close();
        return false;
    }
    in_file.close();
    return true;
}
```

GetConstTensorFromBin函数参数说明：

- path：入参，指定权重文件路径，用于到固定目录例如“../data/weight/”下查找权重文件xx.bin，用户需要自行将权重文件解析为bin文件。
- weight：出参，从权重文件中读取的Tensor类型的权重数据。
- len：入参，指定权重数据大小。

3.3.3 定义简单计算算子（SoftmaxV2）

下面以一个简单的SoftmaxV2为例，介绍如何进行算子定义。

SoftmaxV2算子原型定义：

```
REG_OP(SoftmaxV2)
    .INPUT(x, TensorType({DT_DOUBLE, DT_FLOAT16, DT_FLOAT}))
    .OUTPUT(y, TensorType({DT_DOUBLE, DT_FLOAT16, DT_FLOAT}))
    .ATTR(axes, ListInt, {-1})
    .OP_END_FACTORY_REG(SoftmaxV2)
```

从SoftmaxV2算子原型可以看到，SoftmaxV2算子有一个必选输入，输入名称为x。创建SoftmaxV2算子实例：

```
auto softmax = op::SoftmaxV2("Softmax")     //创建算子实例，传算子名称（例如Softmax）作为入参
    .set_input_x(matmul2);                  //设置算子输入为matmul2
```


3.3.4 定义复杂计算算子（Conv2D）

下面以一个较复杂的Conv2D为例，介绍如何进行算子定义。

Conv2D算子原型定义：（**当前版本暂不支持BF16数据类型**）

```
REG_OP(Conv2D)
    .INPUT(x, TensorType({DT_FLOAT16, DT_FLOAT, DT_INT8, DT_BF16}))
    .INPUT(filter, TensorType({DT_FLOAT16, DT_FLOAT, DT_INT8, DT_BF16}))
    .OPTIONAL_INPUT(bias, TensorType({DT_FLOAT16, DT_FLOAT, DT_INT32}))
    .OPTIONAL_INPUT(offset_w, TensorType({DT_INT8}))
    .OUTPUT(y, TensorType({DT_FLOAT16, DT_FLOAT, DT_INT32, DT_BF16}))
    .REQUIRED_ATTR(strides, ListInt)
    .REQUIRED_ATTR(pads, ListInt)
    .ATTR(dilations, ListInt, {1, 1, 1, 1})
    .ATTR(groups, Int, 1)
    .ATTR(data_format, String, "NHWC")
    .ATTR(offset_x, Int, 0)
    .OP_END_FACTORY_REG(Conv2D)
```

从Conv2D算子原型定义可以看到，Conv2D算子包括：两个必选输入x和filter，两个可选输入bias和offset_w，两个必选属性strides、pads，四个可选属性dilations、groups、data_format、offset_x。则Conv2D算子定义的代码为：

```
auto conv2d = op::Conv2D("Conv2d")
    .set_input_x(quant)
    .set_input_filter(conv_weight)
    .set_input_bias(conv_bias)
    .set_attr_strides({ 1, 1, 1, 1 })
    .set_attr_pads({ 0, 0, 0, 0 })
    .set_attr_dilations({ 1, 1, 1, 1 });

TensorDesc conv2d_input_desc_x(ge::Shape(), FORMAT_NCHW, DT_INT8);
TensorDesc conv2d_input_desc_filter(ge::Shape(), FORMAT_NCHW, DT_INT8);
TensorDesc conv2d_input_desc_bias(ge::Shape(), FORMAT_NCHW, DT_INT32);
TensorDesc conv2d_output_desc_y(ge::Shape(), FORMAT_NCHW, DT_INT32);
conv2d.update_input_desc_x(conv2d_input_desc_x);
conv2d.update_input_desc_filter(conv2d_input_desc_filter);
conv2d.update_input_desc_bias(conv2d_input_desc_bias);
conv2d.update_output_desc_y(conv2d_output_desc_y);
```

主要过程为：

1. 调用算子类型构造函数，例如“Conv2D(const char* name)”创建算子实例，并传算子名称（例如Conv2d）作为入参。
   ```
   auto conv2d1 = op::Conv2D("Conv2d")
   ```
2. 调用“set_input_*输入名称*”接口设置算子的输入。
   ```
   .set_input_x(data)
   .set_input_filter(conv_weight)
   .set_input_bias(conv_bias)
   ```
   data为整个graph的输入节点，通过Data算子构造，具体请参考**3.3.1 定义数据节点（Data）**。

   conv_weight为常量数据，通过Const算子构造，具体请参考**3.3.2 定义数据节点（Const）**。

   conv_bias为常量数据，通过Const算子构造，具体请参考**3.3.2 定义数据节点（Const）**。
3. 调用“set_attr_*属性名称*”接口设置算子的属性。
   ```
   .set_attr_strides({1, 1, 1, 1})       //设置strides属性值
   .set_attr_pads({0, 0, 0, 0})          //设置pads属性值
   .set_attr_dilations({1, 1, 1, 1});    //设置dilations属性值
   ```
4. 对于Conv2D等卷积类或对C轴处理敏感的算子，建议通过“update_input_desc_输入名称”接口将Format信息设置为NCHW或者NHWC等，具体和用户需要处理的Format格式保持一致。


```
TensorDesc conv2d_input_desc_x(ge::Shape(), FORMAT_NCHW, DT_INT8);
TensorDesc conv2d_input_desc_filter(ge::Shape(), FORMAT_NCHW, DT_INT8);
TensorDesc conv2d_input_desc_bias(ge::Shape(), FORMAT_NCHW, DT_INT32);
TensorDesc conv2d_output_desc_y(ge::Shape(), FORMAT_NCHW, DT_INT32);
conv2d.update_input_desc_x(conv2d_input_desc_x);
conv2d.update_input_desc_filter(conv2d_input_desc_filter);
conv2d.update_input_desc_bias(conv2d_input_desc_bias);
conv2d.update_output_desc_y(conv2d_output_desc_y);
```

IR构图不支持输入以下FORMAT：

```
NC1HWC0
FRACTAL_Z
NC1C0HWPAD
NHWC1C0
FRACTAL_DECONV
C1HWNC0
FRACTAL_DECONV_TRANSPOSE
FRACTAL_DECONV_SP_STRIDE_TRANS
NC1HWC0_C04
FRACTAL_Z_C04
FRACTAL_DECONV_SP_STRIDE8_TRANS
NC1KHKWHWC0
C1HWNCoC0
FRACTAL_ZZ
FRACTAL_NZ
NDC1HWC0
FORMAT_FRACTAL_Z_3D
FORMAT_FRACTAL_Z_3D_TRANSPOSE
FORMAT_FRACTAL_ZN_LSTM
FORMAT_FRACTAL_Z_G
FORMAT_ND_RNN_BIAS
FORMAT_FRACTAL_ZN_RNN
FORMAT_NYUV
FORMAT_NYUV_A
```

3.3.5 定义动态多输入算子（AddN）

某些算子的输入个数不固定，为动态多输入算子，例如AddN，下面介绍如何定义这类算子。

AddN算子原型定义：

```
REG_OP(AddN)
    .DYNAMIC_INPUT(x, TensorType::NumberType(), DT_VARIANT)
    .OUTPUT(y, TensorType::NumberType(), DT_VARIANT)
    .REQUIRED_ATTR(N, Int)
    .OP_END_FACTORY_REG(AddN)
```

通过AddN算子原型定义可以看到，该算子为动态多输入算子，我们通过“create_dynamic_input_*输入名称*”创建动态输入，通过“set_dynamic_input_*输入名称*”设置动态输入。

```
auto data = op::Data().set_attr_index(0);
auto addn = op::AddN("addn")
 .create_dynamic_input_x(2)    //创建动态输入x，包括2个输入，并且把这两个输入所为算子最后的输入
 .set_dynamic_input_x(0,data)     //设置第1个输入，0表示输入索引，默认从0开始，data表示输入value
 .set_dynamic_input_x(1,data)     //设置第2个输入，1表示输入索引，默认从0开始，data表示输入value
 .set_attr_N(2);                  //设置属性N的值为2，表示该算子有2个输入
```

也可以通过“create_dynamic_input_byindex_*输入名称*”设置动态输入，但是和“create_dynamic_input_*输入名称*”不能同时使用，两者的区别是：
“create_dynamic_input_*输入名称*”默认把创建的动态输入作为算子最后的输入，而“create_dynamic_input_byindex_*输入名称*”可以指定动态输入的索引位置，例如：

```
auto concatv2 = op::ConcatV2("concatv2")
 .create_dynamic_input_byindex_x(2,0)    //创建动态输入x，包括2个输入，并且把这两个输入插入到索引0和索引1的位置
```



```
.set_dynamic_input_x(0,data1)     //设置第1个输入，0表示输入索引，默认从0开始，data1表示输入value
.set_dynamic_input_x(1,data2)     //设置第2个输入，1表示输入索引，默认从0开始，data2表示输入value
.set_input_concat_dim(data3)     //设置第3个输入,data3表示输入value
.set_attr_N(2);                  //设置属性N的值为2，表示该算子有2个输入
```

3.3.6 定义动态多输出算子（Split）

某些算子的输出个数不固定，为动态多输出算子，例如Split，下面介绍如何定义这类算子。

Split算子原型定义：

```
REG_OP(Split)
    .INPUT(split_dim, TensorType({DT_INT32}))
    .INPUT(x, TensorType::BasicType())
    .DYNAMIC_OUTPUT(y, TensorType::BasicType())
    .REQUIRED_ATTR(num_split, Int)
    .OP_END_FACTORY_REG(Split)
```

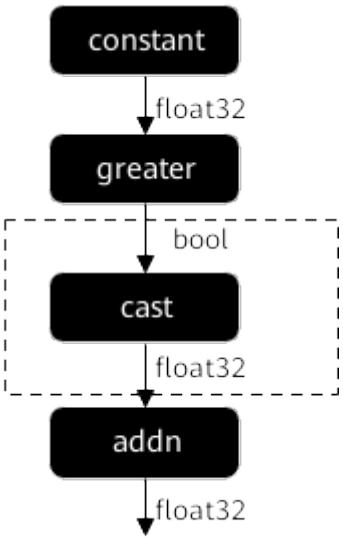
Split算子原型定义可以看到，该算子为动态多输出算子，我们通过“create_dynamic_output_*输出名称*”创建动态输出。

```
auto split = op::Split("split")
  .set_input_x(data)                    //如何构造Data算子，请参考3.3.1 定义数据节点（Data）
  .set_input_split_dim(const)             //如何构造Const算子，请参考3.3.2 定义数据节点（Const）
  .set_attr_num_split(2)
  .create_dynamic_output_y(2);          //创建split算子的动态输出y，包括2个输出

auto addn = op::AddN("addn")
  .create_dynamic_input_x(1)              //创建动态输入x，包括1个输入
  .set_dynamic_input_x(0, split, "y0")      //设置addn算子的第1个输入，split表示输入算子，y0表示split算子的第1个输出
  .set_attr_N(1);                       //设置属性N的值为1，表示该算子有1个输入

auto softplus = op::Softplus("softplus")
  .set_input_x(split, "y1");          //设置softplus算子的输入，split表示输入算子，y1表示split算子的第2个输出
```

3.3.7 定义数据类型转换算子（Cast）

通过算子原型构建Graph时，要求前后算子的dtype必须一致，上一个算子的输出dtype如果和下一层算子的输入dtype不匹配时需要插入Cast算子。

例如下面示例中，addn算子要求输入float32，但是greater算子的输出是bool类型，在数据类型发生变换的情况下，需要通过插入cast算子进行数据类型转换。



Cast算子原型定义：（**当前版本暂不支持BF16数据类型**）

```
REG_OP(Cast)
    .INPUT(x, TensorType({DT_BOOL, DT_FLOAT16, DT_FLOAT, DT_INT8, DT_INT32, DT_UINT32, DT_UINT8,
                          DT_INT64, DT_UINT64, DT_INT16, DT_UINT16, DT_DOUBLE, DT_COMPLEX64,
                          DT_COMPLEX128, DT_QINT8, DT_QUINT8, DT_QINT16, DT_QUINT16, DT_QINT32,
DT_BF16}))
    .OUTPUT(y, TensorType({DT_BOOL, DT_FLOAT16, DT_FLOAT, DT_INT8, DT_INT32, DT_UINT32, DT_UINT8,
                           DT_INT64, DT_UINT64, DT_INT16, DT_UINT16, DT_DOUBLE, DT_COMPLEX64,
                           DT_COMPLEX128, DT_QINT8, DT_QUINT8, DT_QINT16, DT_QUINT16, DT_QINT32,
DT_BF16}))
    .REQUIRED_ATTR(dst_type, Int)
    .OP_END_FACTORY_REG(Cast)
```

Cast算子原型定义可以看到，有一个必选属性dst_type，表示转换后的数据类型，设置为0表示转换后的数据类型为float32，值和数据类型对应关系请参见**10.1.2.10.2 DataType**。

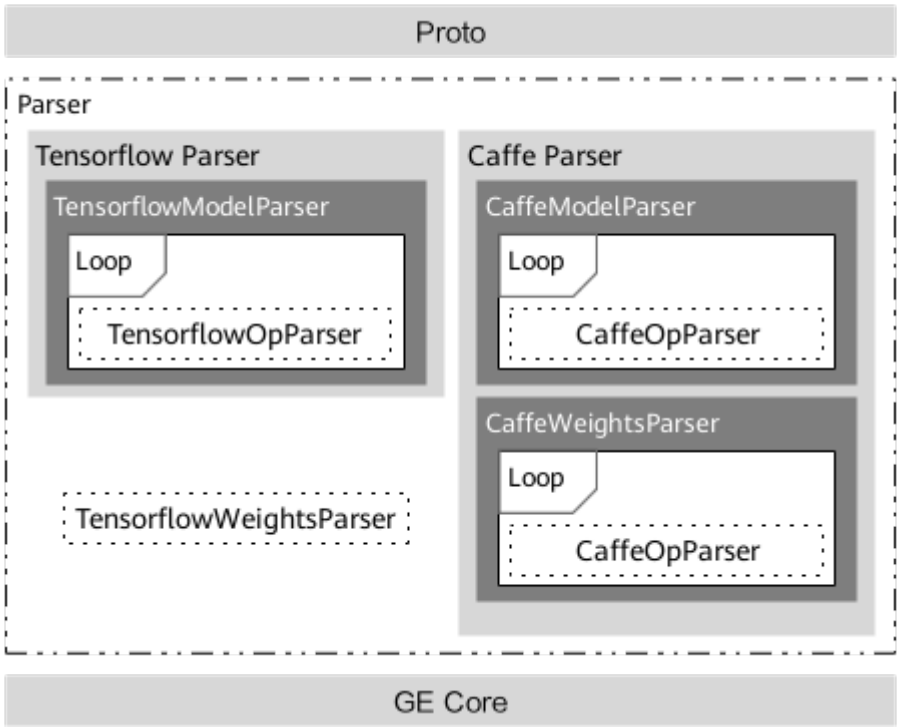
```
auto greater = op::Greater("greater").set_input_x1(const1).set_input_x2(const2);
auto cast = op::Cast("cast").set_input_x(greater)
                            .set_attr_dst_type(0);
auto addn = op::AddN("addn").create_dynamic_input_x(3)
                            .set_dynamic_input_x(0,cast)
                            .set_dynamic_input_x(1,data).set_dynamic_input_x(2,data)
                            .set_attr_N(3);
```


4 原始模型转换为 Graph

功能介绍

除了可以将算子原型直接构图外，CANN还提供了框架Parser功能，将主流的模型格式转换成CANN模型格式。

目前业界开源的深度学习框架（例如TensorFlow，PyTorch、Caffe等），定义模型的格式各有不同，例如TensorFlow通过自定义pb描述静态图和模型，PyTorch通过ONNX规范描述，需要通过统一的框架解析功能隔离上层框架差异，通过Parser模块完成解析并转换成昇腾AI处理器支持的CANN模型格式。



涉及的主要接口为：

- 解析TensorFlow模型：**10.1.3.2.1 aclgrphParseTensorFlow**
- 解析Caffe模型：**10.1.3.2.2 aclgrphParseCaffe**
- 解析ONNX原始模型：**10.1.3.2.3 aclgrphParseONNX**

- 解析加载至内存的ONNX模型：**10.1.3.2.4 aclgrphParseONNXFromMem**

Parser层目前为用户开放了自定义OpParser和自定义TensorFlow Scope融合规则的能力，如果用户在Parser解析时需要对框架进行更灵活的适配，则可以自定义OpParser或自定义开发TensorFlow Scope融合规则。

- 自定义OpParser：如果用户需要将原始框架中算子直接映射到CANN中已实现的TBE算子，可直接进行第三方框架的适配，具体请参考《**TBE&AI CPU算子开发指南**》中的“算子适配插件实现”章节。
- 自定义TensorFlow Scope融合规则：基于TensorFlow构建的神经网络计算图通常由大量的小算子组成，为了实现高性能的计算，往往需要对子图中的小算子进行融合，使得融合后的大算子可以充分利用硬件加速资源。具体请参考《**TensorFlow Parser Scope融合规则开发指南**》。

基于 TensorFlow 模型解析

包含的头文件：

```
#include "tensorflow_parser.h"
#include "all_ops.h"
```

通过**10.1.3.2.1 aclgrphParseTensorFlow**接口将TensorFlow原始模型转换为Graph，此时Graph保存在内存缓冲区中。

```
std::string tfPath = "../data/tf_test.pb";
auto tfStatus = ge::aclgrphParseTensorFlow(tfPath.c_str(),graph1);
```

同时，支持用户指定**parser_params**：

```
std::string tfPath = "../data/tf_test.pb";
std::map<ge::AscendString, ge::AscendString> parser_params= {
        {ge::AscendString(ge::ir_option::LOG_LEVEL), ge::AscendString("debug")},
        {ge::AscendString(ge::ir_option::INPUT_FORMAT), ge::AscendString("NHWC")}};
auto tfStatus = ge::aclgrphParseTensorFlow(tfPath.c_str(), parser_params, graph1);
```

基于 Caffe 模型解析

包含的头文件：

```
#include "caffe_parser.h"
#include "all_ops.h"
```

通过**10.1.3.2.2 aclgrphParseCaffe**接口将Caffe原始模型转换为Graph，此时Graph保存在内存缓冲区中。

```
std::string caffePath = "../data/caffe_test.prototxt";
std::string weight = "../data/caffe_test.caffemodel";
auto caffeStatus = ge::aclgrphParseCaffe(caffePath.c_str(), weight.c_str(), graph1);
```

同时，支持用户指定**parser_params**：

```
std::string caffePath = "../data/caffe_test.prototxt";
std::string weight = "../data/caffe_test.caffemodel";
std::map<ge::AscendString, ge::AscendString> parser_params= {
        {ge::AscendString(ge::ir_option::LOG_LEVEL), ge::AscendString("debug")},
        {ge::AscendString(ge::ir_option::INPUT_FORMAT), ge::AscendString("NHWC")}};
auto caffeStatus = ge::aclgrphParseCaffe(caffePath.c_str(), weight.c_str(), parser_params, graph1);
```

基于 ONNX 模型解析

包含的头文件：

```
#include "onnx_parser.h"
#include "all_ops.h"
```


通过**10.1.3.2.4 aclgrphParseONNXFromMem**接口将加载至内存的ONNX模型转换为Graph，此时Graph保存在内存缓冲区中。同时，支持用户指定**parser_params**：

```
/* read file in binary format */
FILE *pFile = fopen("./onnx/resnet101.onnx", "rb" );
if(pFile==NULL)
{
    fputs("File error",stderr);
    exit(1);
}

/* get the size of the file */
fseek(pFile, 0, SEEK_END);
long lSize = ftell(pFile);
rewind(pFile);

/* assign memory buffer for the file*/
char *buffer =(char*) malloc(sizeof(char)*lSize);
if(buffer == NULL)
{
    fputs("Memory error", stderr);
    exit(2);
}
/* copy the file to buffer */
size_t result = fread(buffer, 1, lSize, pFile);
if(result != lSize)
{
    fputs("Reading error", stderr);
    exit(3);
}
std::map<ge::AscendString, ge::AscendString> parser_params= {
            {ge::AscendString(ge::ir_option::LOG_LEVEL), ge::AscendString("debug")},
            {ge::AscendString(ge::ir_option::INPUT_FORMAT), ge::AscendString("NCHW")}};
auto onnxStatus = ge::aclgrphParseONNXFromMem(buffer, result, parser_params, graph1);
```

通过**10.1.3.2.3 aclgrphParseONNX**接口将ONNX原始模型转换为Graph，此时Graph保存在内存缓冲区中。同时，支持用户指定**parser_params**：

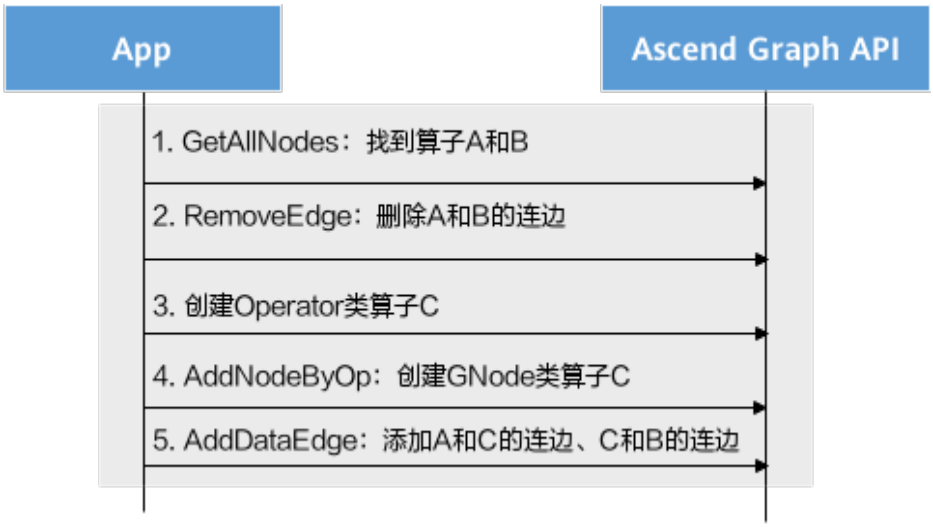
```
std::string onnxPath = "../data/onnx_test.onnx";
std::map<ge::AscendString, ge::AscendString> parser_params= {
            {ge::AscendString(ge::ir_option::LOG_LEVEL), ge::AscendString("debug")},
            {ge::AscendString(ge::ir_option::INPUT_FORMAT), ge::AscendString("NCHW")}};
auto onnxStatus = ge::aclgrphParseONNX(onnxPath .c_str(), parser_params, graph1);
```


5 修改 Graph

功能介绍

如果用户想要直接优化图的结构，比如将某些特定子图替换成一个大算子，以减少计算步骤、外存访问、调度时间等，或者在某些算子中间添加一个算子，此时可以通过本节内容直接将图直接修改成期望的结构。

例如，在算子A和算子B之间添加算子C，涉及的主要接口为：



开发示例

1. 包含的头文件。

```
#include "graph.h"
#include "ascend_string.h"
#include "ge_ir_build.h"
#include "gnode.h"
```

2. 修改图之前，可以先调用**10.1.3.9.1 aclgrphDumpGraph**把Graph dump到本地，查看Graph信息。

   此步骤为可选，需要注意的是，aclgrphDumpGraph接口必须在SetInputs接口和SetOutputs接口之后调用，例如：

```
string op_name = "tc_ge_openpass_0001";
Graph graph(op_name);
auto data = op::Data("data").set_attr_index(0);
TensorDesc data_desc2(ge::Shape({3, 3, 3, 3}), FORMAT_NHWC, DT_FLOAT);
```



```
data.update_input_desc_x(data_desc2);
data.update_output_desc_y(data_desc2);
auto matrixinverse = op::MatrixInverse("MatrixInverse").set_input_x(data);
auto square1 = op::Square("square1").set_input_x(matrixinverse);
std::vector<Operator> inputs{data};
std::vector<Operator> outputs{data,square1};
graph.SetInputs(inputs).SetOutputs(outputs);
std::map<std::string, std::string> init_options = {
    {ge::ir_option::SOC_VERSION,"xxx"}
};
auto ret_init = aclgrphBuildInitialize(init_options);
EXPECT_EQ(ret_init, GRAPH_SUCCESS);
std::cout << "BuildInitialize before infershape Success." << endl;
size_t filesize =24;
const char* file = "tc_ge_openpass_0001_dump";
auto ret = ge::aclgrphDumpGraph(graph,file,filesize);
if(ret != GRAPH_SUCCESS) {
   std::cout<<"dump graph faied."<<std::endl;
}
ACL_LOG("AclgrphDumpGraph，size[%d]",filesize);
int result = AclgrphBuildModel(graph,op_name);
```

3. 在算子A和算子B之间增加算子C，比如在const和add算子之间插入abs。

```
GNode src_node;
GNode dst_node;
std::vector<GNode> nodes = graph.GetAllNodes();
for(auto &node : nodes) {
  ge::AscendString name;
  node.GetName(name);
  std::string node_name(name.GetString());
  if(node_name == CONST) { src_node = node;}
  else if(node_name == ADD) { dst_node = node;}
}
graph.RemoveEdge(src_node, 0, dst_node, 0);
auto abs = op::Abs("input3_abs");
GNode node_abs = graph.AddNodeByOp(abs);
TensorDesc output_tensor_desc;
src_node.GetOutputDesc(0, output_tensor_desc);
abs.UpdateInputDesc(0,  output_tensor_desc);
abs.UpdateOutputDesc(0, output_tensor_desc);
graph.AddDataEdge(src_node, 0, node_abs, 0);
graph.AddDataEdge(node_abs, 0, dst_node, 0);
```

   a. 调用 **GetAllNodes**找到const算子和add算子。
   b. 调用 **RemoveEdge**删除const算子和add算子的连边（数据边或控制边）。
   c. 参考**使用算子原型衍生接口定义算子**，创建Operator类算子abs（也可以调用OperatorFactory::CreateOperator创建算子）。
   d. 调用 **AddNodeByOp**创建GNode类算子abs。

      创建完算子后，可以根据需要更新该算子的input和output TensorDesc，一般根据源节点的Output TensorDesc更新算子abs的Input TensorDesc和Output TensorDesc。如果不更新，系统会设置默认值，在模型编译时对Tensor Shape，type进行推导。
   e. 调用 **AddDataEdge**添加const算子和abs算子，abs算子和add算子之间的连边。如果有控制边，再调用 **AddControlEdge**添加控制边。

   如果在A与B插入多个算子, 比如，A->C->D->B，参考以上步骤，分别执行操作A->C, C->D, D->B。

4. 删除算子A和算子B之间的C算子，比如删除算子const和add之间的abs。

```
graph.RemoveNode(node_abs);
graph.AddDataEdge(src_node, 0, dst_node, 0);
```

   a. 调用 **RemoveNode**删除abs算子。
   b. 调用 **AddDataEdge**添加const和add算子之间的连边。如果有控制边，再调用 **AddControlEdge**添加控制边。

5. 此外，如果需要查询GNode的信息，可以参考**10.1.3.4.2 GNode类**提供的方法。

6 编译 Graph 为离线模型

6.1 功能介绍

构建完Graph之后，如果您仅是需要把Graph编译并保存为适配昇腾AI处理器的离线模型，编译生成的离线模型可以通过aclmdlLoadFromFile接口加载，并通过aclmdlExecute接口执行推理模型，可以参考本节内容编译Graph。如果您需要编译并直接运行Graph，可以参考**7 编译并运行Graph**。

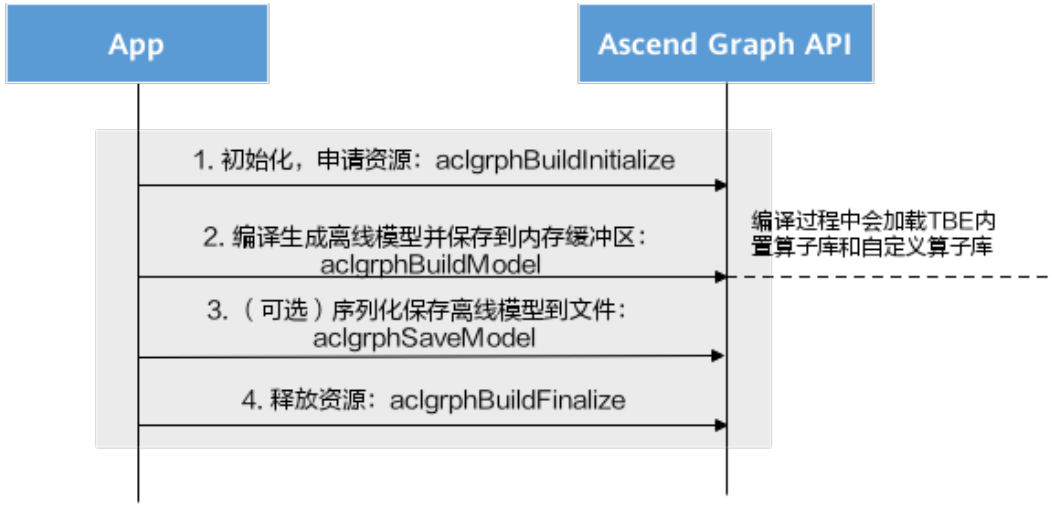
说明

除了aclmdlLoadFromFile和aclmdlExecute接口加载和执行离线模型外，还有其他更丰富的接口适配用户的不同场景，相关接口详细介绍请参考《**应用软件开发指南（C&C++）**》。

6.2 编译 Graph 生成 om 离线模型

功能介绍

将Graph编译为离线模型涉及的主要接口为：



1. Graph定义完成后，调用**10.1.3.5.1 aclgrphBuildInitialize**进行系统初始化，申请系统资源。
2. 调用**10.1.3.5.2 aclgrphBuildModel**将Graph编译为适配昇腾AI处理器的离线模型，编译过程中会加载TBE内置算子库和自定义算子库。此时模型保存在内存缓冲区中。
3. 如果希望将内存缓冲区中的模型保存为离线模型文件xx.om，可以调用**10.1.3.5.3 aclgrphSaveModel**，序列化保存离线模型到文件中。
4. 调用**10.1.3.5.4 aclgrphBuildFinalize**结束进程，释放系统资源。

当前支持在一个进程中连续调用模型编译和模型文件保存接口，用于编译和保存多个离线模型。

开发示例

1. 包含的头文件。
```
#include "ge_ir_build.h"
#include "ge_api_types.h"
```
2. 申请资源。

   创建完Graph以后，通过**10.1.3.5.1 aclgrphBuildInitialize**接口进行系统初始化，并申请资源。示例代码如下：
```
std::map<AscendString, AscendString> global_options = {
        {ge::ir_option::SOC_VERSION, "${soc_version}"},
    };
auto status = aclgrphBuildInitialize(global_options);
```
   可以通过传入**global_options**参数配置离线模型编译初始化信息，当前支持的配置参数请参见**10.1.3.11.2 aclgrphBuildInitialize支持的配置参数**。

   其中SOC_VERSION为必选配置，用于指定目标芯片版本，值${soc_version}需要根据实际情况替换。其他参数请用户根据实际需要可选配置。
3. 通过**10.1.3.5.2 aclgrphBuildModel**接口将Graph编译为离线模型。示例代码如下：
```
ModelBufferData model;
std::map<AscendString, AscendString> options;
PrepareOptions(options);

status = aclgrphBuildModel(graph, options, model);
if(status == GRAPH_SUCCESS) {
    cout << "Build Model SUCCESS!" << endl;
}
else {
    cout << "Build Model Failed!" << endl;
}
```
   可以通过传入**options**参数配置离线模型编译配置信息，当前支持的配置参数请参见**10.1.3.11.3 aclgrphBuildModel支持的配置参数**。配置举例：
```
void PrepareOptions(std::map<AscendString, AscendString>& options) {
    options.insert({
        {ge::ir_option::EXEC_DISABLE_REUSED_MEMORY, "1"} // close resue memory
        });
}
```
4. （可选）也可以通过**10.1.3.5.3 aclgrphSaveModel**将内存缓冲区中的模型保存为离线模型文件。示例代码如下：
```
status = aclgrphSaveModel("ir_build_sample", model);
if(status == GRAPH_SUCCESS) {
        cout << "Save Offline Model SUCCESS!" << endl;
}
else {
        cout << "Save Offline Model Failed!" << endl;
}
```

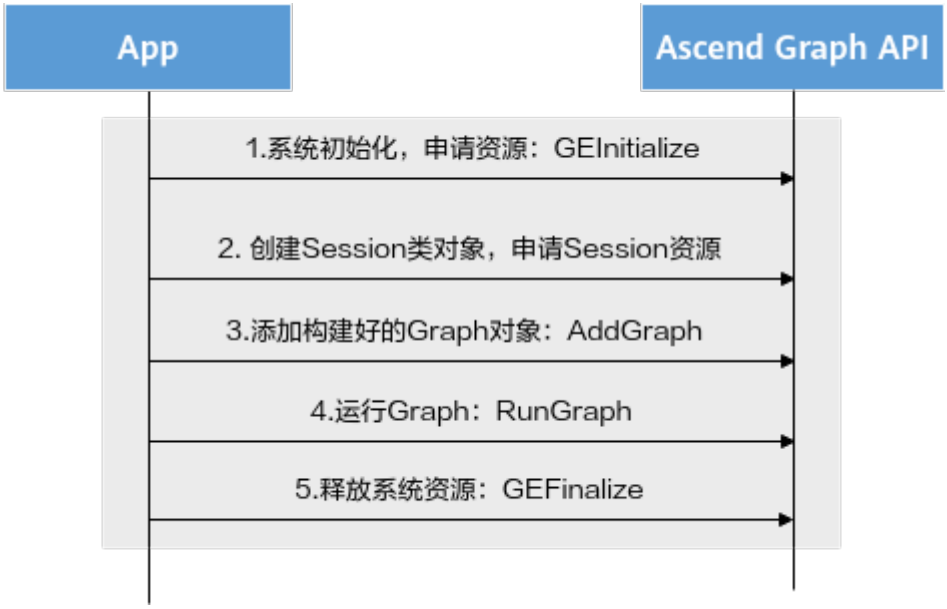

5. 构图进程结束时，通过**10.1.3.5.4 aclgrphBuildFinalize**接口释放资源。示例代码如下：

```
aclgrphBuildFinalize();
```


7 编译并运行 Graph

功能介绍

构建完Graph之后，如果您希望直接编译并运行Graph，得到图的执行结果，可以参考本节内容。涉及的主要接口为：



1. 调用**10.1.3.6.1 GEInitialize**进行系统初始化（也可在Graph构建前调用），申请系统资源。
2. 调用**Session构造函数**创建Session类对象，申请session资源。
3. 调用**10.1.3.6.4 AddGraph**在Session类对象中添加定义好的图。
4. 调用**10.1.3.6.7 RunGraph**运行图。
5. 调用**10.1.3.6.2 GEFinalize**，释放系统资源。

开发示例

1. 包含的头文件。
```
#include "ge_api.h"
```
2. 申请系统资源。

Graph定义完成后，调用GEInitialize进行系统初始化（也可在Graph定义前调用），申请系统资源。示例代码如下：

```
std::map<AscendString, AscendString>config = {{"ge.exec.deviceId", "0"},
                                               {"ge.graphRunMode", "1"}};
Status ret = ge::GEInitialize(config);
```

可以通过config配置传入ge运行的初始化信息，当前必选的配置参数为示例代码中的ge.exec.deviceId和ge.graphRunMode，分别用于指定GE实例运行设备，图执行模式（在线推理请配置为0，训练请配置为1）。更多配置请参考**表10-5**。

3. 添加Graph对象并运行Graph。

   若想使定义好的Graph运行起来，首先，要创建一个session对象，然后调用AddGraph接口添加图，再调用RunGraph接口执行图。示例代码如下：

   ```
   std::map <AscendString, AscendString> options;
   ge::Session *session = new Session(options);
   if(session == nullptr) {
   std::cout << "Create session failed." << std::endl;
   return FAILED;
   }
   Status ret = session->AddGraph(conv_graph_id, conv_graph);
   if(ret != SUCCESS) {
   return FAILED;
   }
   ret = session->RunGraph(conv_graph_id, input_cov, output_cov);
   if(ret != SUCCESS) {

   return FAILED;
   }
   ```

   用户可以通过传入options配置图运行相关配置信息，相关配置请参考**Session构造函数**。

   其中图运行完之后的数据保存在Tensor output_cov中。

4. 图运行完之后，通过GEFinalize释放资源。

   ```
   ret = ge::GEFinalize();
   ```


8 完整样例参考

样例获取和使用

单击Gitee或Github，进入Ascend samples开源仓，参见README中的“版本说明”下载配套版本的sample包，从“samples/cplusplus/level1_single_api/3_ir/IRBuild”目录中获取样例，该样例介绍如何构建Graph并编译成离线模型。

单击Gitee或Github，进入Ascend samples开源仓，参见README中的“版本说明”下载配套版本的sample包，从“samples/cplusplus/level1_single_api/8_graphrun/graph_run”目录中获取样例，该样例介绍如何构建Graph并直接编译并运行Graph。

编译脚本编写注意事项

您可以参考上述样例编写编译脚本，其中需要根据实际情况进行修改的部分为：

- ASCEND_PATH：指定到“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/”路径。
- INCLUDES：需要包含的头文件，当需要添加头文件时，在示例下方直接增加行即可，注意不要删除原有项目。如果网络中有自定义算子，请增加自定义算子的原型定义头文文件。
- LIBS：需要链接的库，当需要添加链接库时，在示例下方直接增加行即可，注意不要删除原有项目。

须知

禁止链接软件包中的其他so，否则后续升级可能会导致兼容性问题。

程序执行前依赖的环境变量

安装CANN软件后，使用CANN运行用户编译、运行时，需要以CANN运行用户登录环境，执行. **${install_path}/set_env.sh**命令设置环境变量。

说明

上述仅列出了程序执行必配的环境变量，更多环境变量说明请参考**10.2 环境变量参考**。

9 专题

9.1 量化

9.1.1 手工量化

功能介绍

量化是指对模型的参数和数据进行低比特处理，让最终生成的网络模型更加轻量化，从而达到节省网络模型存储空间、降低传输时延、提高计算效率达到性能提升与优化的目标。

用户可通过自己的框架和工具完成量化，并将这些量化参数（$scale_d$、$scale_w$、$offset_d$）在模型构建时注入到模型中。

须知

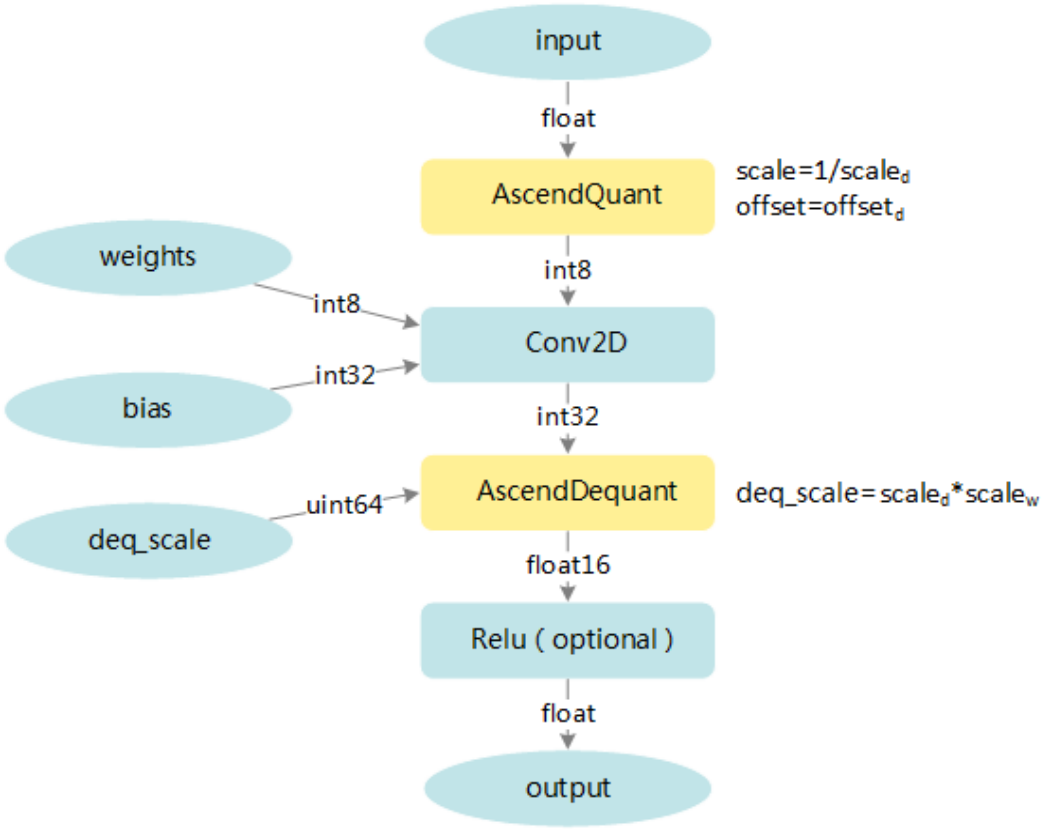
- 当前仅Conv2D/DepthwiseConv2D/FullyConnection算子支持量化。
- 网络中Conv2D/DepthwiseConv2D/FullyConnection算子输入数据的Channel维度小于等于16时，由于Padding补齐，INT8量化无性能受益。因此量化要求Conv2D/DepthwiseConv2D/FullyConnection算子输入数据的Channel维度大于16，否则无法进行量化。

以Conv2D算子为例，通过在Conv2D算子前插入AscendQuant量化算子，在Conv2D算子后插入AscendDequant反量化算子实现模型量化，如图9-1所示。

AscendQuant量化算子的作用是将float类型的数据转换为int8类型，即：$data_{int8}=round[(data_{float}*scale)+offset]$，其中$scale=1/scale_d$，$offset=offset_d$。此处的round算法类似于C语言rint取整模式中的FE_TONEAREST模式。

AscendDequant反量化算子的作用是将int32类型的数据转换为float16类型，即：$data_{float}=data_{int32}*deq\_scale$，其中$deq_scale=scale_d*scale_w$。

图 9-1 量化示意图



在 Conv2D 算子前插入 AscendQuant 算子

AscendQuant算子原型定义：

```
REG_OP(AscendQuant)
    .INPUT(x, TensorType({DT_FLOAT16, DT_FLOAT32}))
    .OUTPUT(y, TensorType({DT_INT8, DT_INT4}))
    .REQUIRED_ATTR(scale, Float)
    .REQUIRED_ATTR(offset, Float)
    .ATTR(sqrt_mode, Bool, false)
    .ATTR(round_mode, String, "Round")
    .OP_END_FACTORY_REG(AscendQuant)
```

可以看到AscendQuant算子有1个输入x，两个必选属性scale和offset，两个可选属性sqrt_mode和round_mode，参数含义如下：

- x：指定AscendQuant算子输入，Tensor类型，支持的数据类型为float16和float32。

- scale：指定量化系数scale=1/$scale_d$，支持float32和float16数据类型，建议在float16数据类型表达范围内，如果超出了float16表达范围，请配置sqrt_mode为True。
- offset：指定量化偏移量offset=1/$offset_d$，支持float32和float16数据类型。
- sqrt_mode：是否对scale进行开平方处理，取值为False和True，建议保持默认False。如果scale超出了float16表达范围，为了不损失精度，请配置sqrt_mode为True，系统会对scale做开平方处理。
- round_mode：float类型到int类型的转换方式，默认为Round，支持配置为：Round, Floor, Ceiling, Truncate。

根据AscendQuant算子原型定义创建AscendQuant算子实例：

```
auto quant = op::AscendQuant("quant")
  .set_input_x(data)
  .set_attr_scale(1.00049043)      //指定量化系数
  .set_attr_offset(-128.0);        //指定偏移量
```

Conv2D 算子

Conv2D算子的输入为AscendQuant，并设置输出type为int32。

```
// const op: conv2d weight
auto weight_shape = ge::Shape({ 5,17,1,1 });
TensorDesc desc_weight_1(weight_shape, FORMAT_NCHW, DT_INT8);
Tensor weight_tensor(desc_weight_1);
uint32_t weight_1_len = weight_shape.GetShapeSize();
bool res = GetConstTensorFromBin(PATH+"const_0.bin", weight_tensor, weight_1_len);
if(!res) {
    std::cout << "GetConstTensorFromBin Failed!" << std::endl;
    return -1;
}
auto conv_weight = op::Const("const_0")
    .set_attr_value(weight_tensor);

// const op: conv2d bias
auto bias_shape = ge::Shape({ 5 });
TensorDesc desc_bias(bias_shape, FORMAT_NCHW, DT_INT32);
Tensor bias_tensor(desc_bias);
uint32_t bias_len = bias_shape.GetShapeSize() * sizeof(int32_t);
res = GetConstTensorFromBin(PATH + "const_1.bin", bias_tensor, bias_len);
if(!res) {
    std::cout << "GetConstTensorFromBin Failed!" << std::endl;
    return -1;
}
auto conv_bias = op::Const("const_1")
    .set_attr_value(bias_tensor);

// conv2d op
auto conv2d = op::Conv2D("Conv2d")
    .set_input_x(quant)                                    //AscendQuant作为Conv2D算子的输入
    .set_input_filter(conv_weight)
    .set_input_bias(conv_bias)
    .set_attr_strides({ 1, 1, 1, 1 })
    .set_attr_pads({ 0, 0, 0, 0 })
    .set_attr_dilations({ 1, 1, 1, 1 });

TensorDesc conv2d_input_desc_x(ge::Shape(), FORMAT_NCHW, DT_INT8);       //量化后，输入x的type为INT8
TensorDesc conv2d_input_desc_filter(ge::Shape(), FORMAT_NCHW, DT_INT8);
TensorDesc conv2d_input_desc_bias(ge::Shape(), FORMAT_NCHW, DT_INT32);
TensorDesc conv2d_output_desc_y(ge::Shape(), FORMAT_NCHW, DT_INT32);
conv2d.update_input_desc_x(conv2d_input_desc_x);
conv2d.update_input_desc_filter(conv2d_input_desc_filter);
conv2d.update_input_desc_bias(conv2d_input_desc_bias);
conv2d.update_output_desc_y(conv2d_output_desc_y);
```


在 Conv2D 算子后插入 AscendDequant 算子

AscendDequant算子原型定义：

```
REG_OP(AscendDequant)
    .INPUT(x, TensorType({DT_INT32}))
    .INPUT(deq_scale, TensorType({DT_FLOAT16, DT_UINT64}))
    .OUTPUT(y, TensorType({DT_FLOAT16, DT_FLOAT}))
    .ATTR(sqrt_mode, Bool, false)
    .ATTR(relu_flag, Bool, false)
    .ATTR(dtype, Int, false, DT_FLOAT)
    .OP_END_FACTORY_REG(AscendDequant)
```

可以看到AscendDequant算子有2个输入x和deq_scale，两个可选属性sqrt_mode和relu_flag，参数含义如下：

- x：指定AscendDequant算子输入，Tensor类型，支持的数据类型为int32。
- deq_scale：指定反量化系数deq_scale=$scale_d$*$scale_w$，Tensor类型，支持的数据类型为uint64。shape可以为1，或者和Conv2D输出数据的Channel维度保持一致。

  须知

  要求用户把$scale_d$和$scale_w$相乘得到的float32数据类型转换为uint64类型，并填入到deq_scale的低32位中，高32位要求全部为0。

  ```
  import numpy as np
  def trans_float32_scale_deq_to_uint64(scale_deq):
      float32_scale_deq = np.array(scale_deq, np.float32)
      uint32_scale_deq = np.frombuffer(float32_scale_deq, np.uint32)
      uint64_result = np.zeros(float32_scale_deq.shape, np.uint64)
      uint64_result |= np.uint64(uint32_scale_deq)
      return uint64_result
  ```

- sqrt_mode：是否对deq_scale进行开平方处理，取值为False和True，建议保持默认False。如果deq_scale超出了float16表达范围，为了不损失精度，请配置sqrt_mode为True，同时将deq_scale开平方后填入。
- relu_flag：是否执行Relu，取值为False和True，默认值为False。

根据AscendDequant算子原型定义创建AscendDequant算子实例：

```
//构造dequant_scale
TensorDesc desc_dequant_shape(ge::Shape({ 5 }), FORMAT_NCHW, DT_UINT64);
Tensor dequant_tensor(desc_dequant_shape);
uint32_t dequant_scale_len = 5 * sizeof(uint64_t);
res = GetConstTensorFromBin(PATH + "const_2.bin", dequant_tensor, dequant_scale_len);
if(!res) {
    std::cout << "GetConstTensorFromBin Failed!" << std::endl;
    return -1;
}
auto dequant_scale = op::Const("dequant_scale")
    .set_attr_value(dequant_tensor);

//定义AscendDequant算子
auto dequant = op::AscendDequant("dequant")
  .set_input_x(conv2d)                          //Conv2D作为AscendDequant算子的输入
  .set_input_deq_scale(dequant_scale);
```

将AscendDequant的输出作为其他算子的输入，或者作为整个graph的输出。

```
auto bias_add_1 = op::BiasAdd("bias_add_1")
   .set_input_x(dequant)
```



```
    .set_input_bias(bias_weight_1)
    .set_attr_data_format("NCHW");
```

9.1.2 自动量化

9.1.2.1 功能介绍

功能介绍

量化是指对模型的权重（weight）和数据（activation）进行低比特处理，让最终生成的网络模型更加轻量化，从而节省网络模型存储空间、降低传输时延、提高计算效率，达到性能提升与优化的目标。您可以通过调用**10.1.3.7.1 aclgrphCalibration**接口对非量化的Graph进行自动量化。

量化前会对模型中的某些结构做算子融合，融合后图结构得到优化，从而提升网络推理性能。具体支持的融合功能请参见**9.1.2.4 支持的融合功能**。

使用约束

支持量化的层及约束请参考**9.1.2.2 支持量化的层及约束**。

调用示例

步骤1 添加头文件。

```
#include "amct/acl_graph_calibration.h"
#include "amct/acl_calibration_configs.h"
#include "amct/amct_error_code.h"
```

步骤2 通过解析或手动构造方式创建Graph对象。

```
#include "parser/caffe_parser.h"

ge::Graph graph("origin");
std::map<ge::AscendString, ge::AscendString> parseOptions;
auto ret = ge::aclgrphParseCaffe("./path/to/resnet50.prototxt",
    "./path/to/resnet50.caffemodel", parseOptions, graph);
```

步骤3 定义aclgrphCalibration接口所需配置参数。

```
std::map<ge::AscendString, ge::AscendString> quantizeConfigs = {};
quantizeConfigs[ge::AscendString(amct::aclCaliConfigs::INPUT_DATA_DIR)] = ge::AscendString("./path/to/
cali_data.bin");
quantizeConfigs[ge::AscendString(amct::aclCaliConfigs::INPUT_SHAPE)] = ge::AscendString("input:
16,224,224,3");
quantizeConfigs[ge::AscendString(amct::aclCaliConfigs::SOC_VERSION)] =
ge::AscendString("SOC_VERSION"); // SOC_VERSION请配置为实际的AI处理器型号
```

如果需要自行控制量化过程，可以使用CONFIG_FILE参数项传入控制量化过程的简易配置文件。配置文件的示例请参考**9.1.2.3 简易配置文件**。

CONFIG_FILE参数的配置示例如下：

```
quantizeConfigs[ge::AscendString(amct::aclCaliConfigs::CONFIG_FILE)]=ge::AscendString("./calibration.cfg");
```

步骤4 调用接口执行量化改图。

```
ret = ge::aclgrphCalibration(graph, quantizeConfigs);
if (ret != ge::GRAPH_SUCCESS) {
    return FAILED;
}
return SUCCESS;
```

----结束

说明

编写编译上述程序的脚本时需要链接libamctacl.so，并添加链接时动态库的搜索路径${ASCEND_PATH}/compiler/lib64，其中ASCEND_PATH为“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/”路径。

9.1.2.2 支持量化的层及约束

说明

若网络模型输入数据类型或权重数据类型为Float16或混合精度类型（Float32/Float16共存），AMCT会关闭如下算子的量化功能：

AvgPool、Pooling、AvgPoolV2、MaxPool、MaxPoolV3、Pooling、Add、Eltwise。

表 9-1 均匀量化支持的层及约束

框架	支持的层类型	约束	对应Ascend IR定义的层类型
Caffe	InnerProduct：全连接层	transpose属性为false，axis为1	FullyConnection
	Convolution：卷积层	dilation为1、filter维度为4	Conv2D
	Deconvolution: 反卷积层	dilation为1、filter维度为4	Deconvolution
	Pooling	• mode为1，全量化（weight+tensor），global_pooling为false，不支持移位N操作 • mode为0，只做tensor量化	Pooling
	Eltwise	-	Eltwise
TensorFlow	MatMul：全连接层	transpose_a为False, transpose_b为False，adjoint_a为False，adjoint_b为False。	MatMulV2
	Conv2D：卷积层	weight的输入来源不含有placeholder等可动态变化的节点，且weight的节点类型只能是const。	Conv2D
	DepthwiseConv2dNative：Depthwise卷积层	weight的输入来源不含有placeholder等可动态变化的节点，且weight的节点类型只能是const。	DepthwiseConv2D
	Conv2DBackpropInput	dilation为1，weight的输入来源不含有placeholder等可动态变化的节点，且weight的节点类型只能是const。	Conv2DBackpropInput
	BatchMatMulV2	adj_x=False，第二路输入要求为2维const	BatchMatMulV2
	AvgPool	不支持移位N操作	AvgPool

框架	支持的层类型	约束	对应Ascend IR定义的层类型
	Conv3D	dilation_d为1	Conv3D
	MaxPool	-	MaxPool、MaxPoolV3
	Add	-	Add
ONNX	Conv：卷积层	filter维度为5的情况下，要求dilation_d为1	Conv2D、Conv3D
	Gemm：广义矩阵乘	transpose_a=false	MatMulV2
	ConvTranspose：转置卷积	dilation为1、filter维度为4	Conv2DTranspose
	MatMul	第二路输入要求为2维const	BatchMatMulV2
	AveragePool	global_pooling为false，不支持移位N操作	AvgPoolV2
	MaxPool	-	MaxPool、MaxPoolV3
	Add	-	Add

表 9-2 非均匀量化支持的层及约束

框架	支持的层类型	约束	对应Ascend IR定义的层类型
Caffe	Convolution：卷积层	dilation为1、filter维度为4	Conv2D
	InnerProduct：全连接层	transpose属性为false，axis为1	FullyConnection
TensorFlow	Conv2D：卷积层	dilation为1	Conv2D
	MatMul：全连接层	transpose_a为False	MatMulV2
ONNX	Conv：卷积层	-	Conv2D
	Gemm：广义矩阵乘	transpose_a=false	MatMulV2

9.1.2.3 简易配置文件

表 9-3 calibration_config.proto 参数说明

消息	是否必填	类型	字段	说明
AMCTConfig	-	-	-	AMCT训练后量化的简易配置。
	optional	bool	activation_offset	数据量化是否带offset。全局配置参数。 ● 带offset：数据量化使用非对称量化。 ● 不带offset：数据量化使用对称量化。
	optional	bool	joint_quant	是否进行Eltwise联合量化，默认为false，表示关闭联合量化功能。 开启后对部分网络可能会存在性能提升但是精度下降的问题。
	repeated	string	skip_layers	不需要量化层的层名。
	repeated	string	skip_layer_types	不需要量化的层类型。
	optional	int32	version	简易配置文件的版本。
	optional	CalibrationConfig	common_config	通用的量化配置，全局量化配置参数。若某层未被override_layer_types或者override_layer_configs重写，则使用该配置。 参数优先级：override_layer_configs>override_layer_types>common_config
	repeated	OverrideLayerType	override_layer_types	重写某一类型层的量化配置，即对哪些层进行差异化量化。 例如全局量化配置参数配置的量化因子搜索步长为0.01，可以通过该参数对部分层进行差异化量化，可以配置搜索步长为0.02。 参数优先级：override_layer_configs>override_layer_types>common_config

消息	是否必填	类型	字段	说明
	repeated	OverrideLayer	override_layer_configs	重写某一层的量化配置，即对哪些层进行差异化量化。 例如全局量化配置参数配置的量化因子搜索步长为0.01，可以通过该参数对部分层进行差异化量化，可以配置搜索步长为0.02。 参数优先级：override_layer_configs>override_layer_types>common_config
	optional	bool	do_fusion	是否开启BN融合功能，默认为true，表示开启该功能。
	repeated	string	skip_fusion_layers	跳过bn融合的层，配置之后这些层不会进行bn融合。
	repeated	TensorQuantize	tensor_quantize	对网络模型中指定节点的输入Tensor进行训练后量化，来提高数据搬运时的推理性能。 当前仅支持对MaxPool/Add算子做tensor量化。
	optional	bool	enable_auto_nuq	是否开启权重自动非均匀量化功能。默认为false，表示不开启该功能。 开启该功能，不影响用户已强制配置的量化层（通过简易配置文件中override_layer_configs配置的层），只会在剩余的均匀量化层中自动搜索因权重过大导致性能瓶颈的层，对其量化，提高权重的压缩率，从而达到降低带宽、提升性能的目的。
OverrideLayerType	required	string	layer_type	支持量化的层类型的名称。
	required	CalibrationConfig	calibration_config	重置的量化配置。
OverrideLayer	-	-	-	重置某层量化配置。
	required	string	layer_name	被重置层的层名。
	required	CalibrationConfig	calibration_config	重置的量化配置。
TensorQuantize	-	-	-	需要进行训练后量化的输入Tensor配置。

消息	是否必填	类型	字段	说明
	required	string	layer_name	需要对节点输入Tensor进行训练后量化的节点名称， 当前仅支持对MaxPool算子的输入Tensor进行量化。
	required	uint32	input_index	需要对节点输入Tensor进行训练后量化的节点的输入索引。
	-	FMRQuantize	ifmr_quantize	数据量化算法配置。 ifmr_quantize：IFMR量化算法配置。默认为IFMR量化算法。
	-	HFMGQuantize	hfmg_quantize	数据量化算法配置。 hfmg_quantize：HFMG量化算法配置。
CalibrationConfig	-	-	-	Calibration量化的配置。
	-	ARQuantize	arq_quantize	权重量化算法配置。 arq_quantize：ARQ量化算法配置。
	-	FMRQuantize	ifmr_quantize	数据量化算法配置。 ifmr_quantize：IFMR量化算法配置。
	-	NUQuantize	nuq_quantize	权重量化算法配置。 nuq_quantize：非均匀量化算法配置。
ARQuantize	-	-	-	ARQ量化算法配置。
	optional	bool	channel_wise	是否对每个channel采用不同的量化因子。 ● true：每个channel独立量化，量化因子不同。 ● false：所有channel同时量化，共享量化因子。
FMRQuantize	-	-	-	FMR量化算法配置。
	optional	float	search_range_start	量化因子搜索范围左边界。
	optional	float	search_range_end	量化因子搜索范围右边界。
	optional	float	search_step	量化因子搜索步长。
	optional	float	max_percentile	最大值搜索位置。
	optional	float	min_percentile	最小值搜索位置。

消息	是否必填	类型	字段	说明
	optional	bool	asymmetric	是否进行非对称量化。用于控制逐层量化算法的选择。 ● true：非对称量化 ● false：对称量化 如果override_layer_configs、override_layer_types、common_config配置项都配置该参数，或者配置了activation_offset参数，则生效优先级为： override_layer_configs>override_layer_types>common_config>activation_offset
NUQuantize	-	-	-	非均匀权重量化算法配置。
	optional	uint32	num_steps	非均匀量化的台阶数。当前仅支持设置为16和32。
	optional	uint32	num_of_iteration	非均匀量化优化的迭代次数。当前仅支持设置为{0,1,2,3,4,5}，0表示没有迭代。

- 基于该文件构造的**均匀量化简易配置文件***quant*.cfg样例如下所示：*Optype*需要配置为基于Ascend IR定义的算子类型，详细对应关系请参见**9.1.2.2 支持量化的层及约束**。

```
# global quantize parameter
activation_offset : true
joint_quant : false
enable_auto_nuq : false
version : 1
skip_layers : "conv_1"
skip_layer_types:"Optype"
do_fusion: true
skip_fusion_layers : "conv_1"
common_config : {
    arq_quantize : {
        channel_wise : true
    }
    ifmr_quantize : {
        search_range_start : 0.7
        search_range_end : 1.3
        search_step : 0.01
        max_percentile : 0.999999
        min_percentile : 0.999999
        asymmetric : true
    }
}

override_layer_types : {
    layer_type : "Optype"
    calibration_config : {
        arq_quantize : {
            channel_wise : false
        }
        ifmr_quantize : {
```



```
            search_range_start : 0.8
            search_range_end : 1.2
            search_step : 0.02
            max_percentile : 0.999999
            min_percentile : 0.999999
            asymmetric : false
        }
    }
}

override_layer_configs : {
    layer_name : "conv_2"
    calibration_config : {
        arq_quantize : {
            channel_wise : true
        }
        ifmr_quantize : {
            search_range_start : 0.8
            search_range_end : 1.2
            search_step : 0.02
            max_percentile : 0.999999
            min_percentile : 0.999999
            asymmetric : false
        }
    }
}
tensor_quantize {
    layer_name: "max_pooling2d/MaxPool"
    input_index: 0
    ifmr_quantize: {
        search_range_start : 0.7
        search_range_end : 1.3
        search_step : 0.01
        min_percentile : 0.999999
       asymmetric : false
       }
}
tensor_quantize {
    layer_name: "max_pooling2d_1/MaxPool"
    input_index: 0
}
```

- 基于该文件构造的**非均匀量化简易配置文件***quant*.cfg样例如下所示：

```
# global quantize parameter
activation_offset : true
joint_quant : false
enable_auto_nuq : false

common_config : {
    arq_quantize : {
        channel_wise : true
    }
    ifmr_quantize : {
        search_range_start : 0.7
        search_range_end : 1.3
        search_step : 0.01
        max_percentile : 0.999999
        min_percentile : 0.999999
        asymmetric : true
    }
}

override_layer_types : {
    layer_type : "MatMul"
    calibration_config : {
        arq_quantize : {
            channel_wise : false
        }
        ifmr_quantize : {
            search_range_start : 0.7
```



```
            search_range_end : 1.3
            search_step : 0.01
            max_percentile : 0.999999
            min_percentile : 0.999999
            asymmetric : false
        }
    }
}
override_layer_configs : {
    layer_name : "Conv_10"
    calibration_config : {
        nuq_quantize : {
            num_steps : 32
            num_of_iteration : 1
        }
        ifmr_quantize : {
            search_range_start : 0.8
            search_range_end : 1.2
            search_step : 0.02
            max_percentile : 0.999999
            min_percentile : 0.999999
            asymmetric : false
        }
    }
}
tensor_quantize {
    layer_name: "max_pooling2d/MaxPool"
    input_index: 0
    ifmr_quantize: {
        search_range_start : 0.7
        search_range_end : 1.3
        search_step : 0.01
        min_percentile : 0.999999
        asymmetric : false
    }
}
tensor_quantize {
    layer_name: "max_pooling2d_1/MaxPool"
    input_index: 0
}
```

9.1.2.4 支持的融合功能

- Caffe框架：
  - Conv+BN+Scale融合，AMCT在量化前会对模型中的"Convolution+BatchNorm+Scale"结构做Conv+BN+Scale融合，融合后的"BatchNorm"、"Scale"层会被删除。
  - DeConv+BN+Scale融合，AMCT在量化前会对模型中的"Deconvolution+BatchNorm+Scale"结构做DeConv+BN+Scale融合，融合后的"BatchNorm"、"Scale"层会被删除。
- TensorFlow框架：
  - Conv+BN融合：AMCT在量化前会对模型中的"Conv2D/Conv3D+BatchNorm"结构做Conv+BN融合，融合后的"BatchNorm"层会被删除。
  - Depthwise_Conv+BN融合：AMCT在量化前会对模型中的"DepthwiseConv2dNative+BatchNorm"结构做Depthwise_Conv+BN融合，融合后的"BatchNorm"层会被删除。
  - Conv2DBackpropInput+BN融合：AMCT在量化前会对模型中的"Conv2DBackpropInput+BatchNorm"结构做Conv2DBackpropInput+BN融合，融合后的"BatchNorm"层会被删除。
- ONNX框架：

  - Conv+BN融合，AMCT在量化前会对模型中的"Conv +BatchNormalization"结构做Conv+BN融合，融合后的"BatchNorm"层会被删除。
  - ConvTranspose+BN融合，AMCT在量化前会对模型中的"ConvTranspose +BatchNormalization"结构做ConvTranspose+BN融合，融合后的"BatchNorm"层会被删除。

- requant融合场景中的Relu6算子替换成Relu（Relu6无法做Requant融合，需替换为Relu）：由于Relu6算子在Relu的基础上增加了对6以上数值的截断，同时量化过程中也会对输入浮点数进行截断，故AscendDequant+Relu6+AscendQuant与AscendDequant+Relu+AscendQuant存在等价的场景可以进行替换。基于该背景AMCT对量化部署模型中的AscendDequant+Relu6+AscendQuant结构等价替换为AscendDequant+Relu+AscendQuant。

  该场景下需要满足限制条件才可以进行替换，条件为(127-offset)/scale<6，其中scale/offset为quant中取出的量化参数。
- 支持对满足requant融合场景中的relu6算子替换成relu。
- BatchNorm+Mul+Add融合（适用于TensorFlow框架和ONNX框架）

  AMCT在量化前会先对模型中的 “BatchNorm+Mul” 结构做“BN+Mul”融合，融合后的 “Mul” 层会被删除；然后对模型中的 “BatchNorm+Add” 结构做“BN+Add”融合，融合后的 “Add” 层会被删除。
- BN小算子融合为BatchNorm大算子（适用于TensorFlow框架和ONNX框架）。

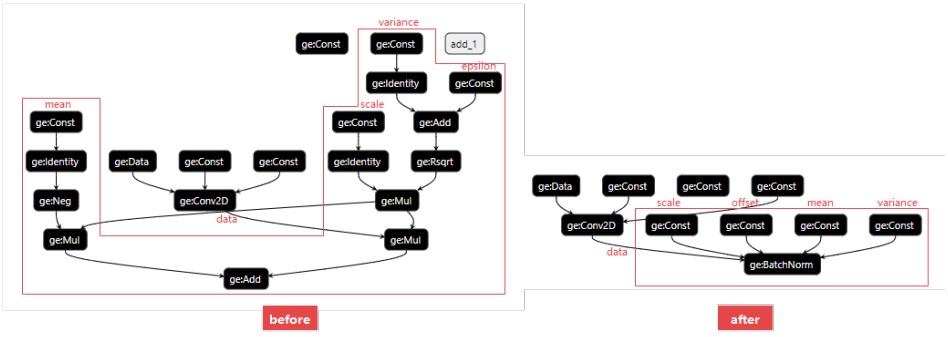
  AMCT对小算子结构的BN进行匹配，并将匹配到的小算子BN结构替换为大算子的BN结构。BN小算子结构融合的前提条件包括：

  a. BN结构中的data节点必须为可量化节点（Conv2D、DepthwiseConv2D、MatMulV2、Conv3D）。

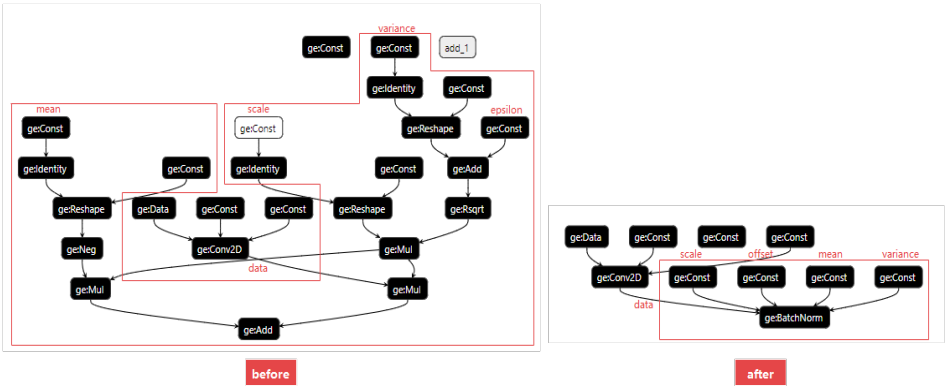
  b. 如果结构中有除data、scale、offset、mean、variance和输出节点之外的节点与该结构外的节点连接，则不做融合。

  对于没有offset的BN结构，融合时会构造一个全0的offset节点；对于没有scale的BN结构，融合时会构造一个全1的scale节点。具体支持的小算子BN结构的场景如下所示：

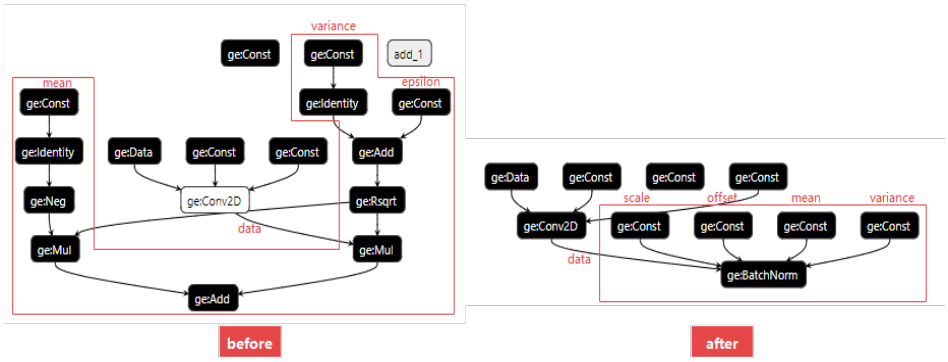
  - 无offset，is_training=False，data_format=NHWC，输入节点为Const类型，融合前后网络结构图为：

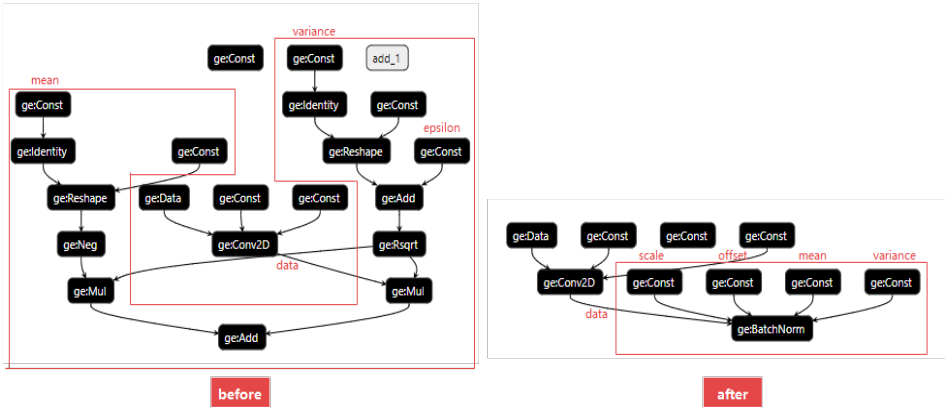
  - 无offset，is_training=False，data_format=NCHW，输入节点为Const类型，融合前后网络结构图为：



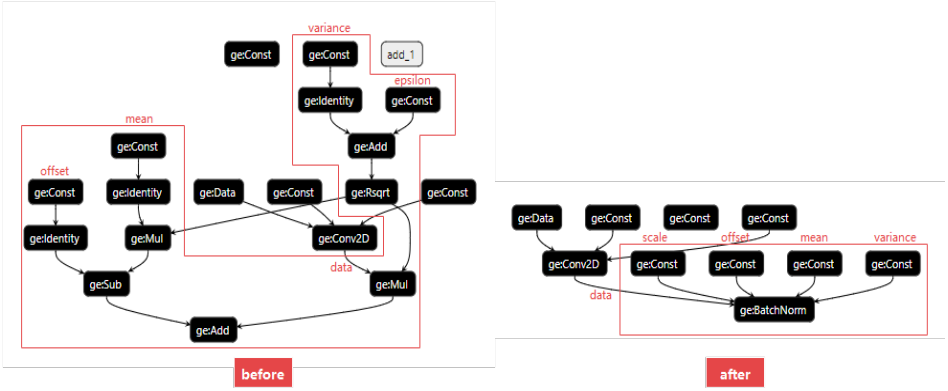
- 无scale和offset，is_training=False，data_format=NHWC，输入节点为Const类型，融合前后网络结构图为：



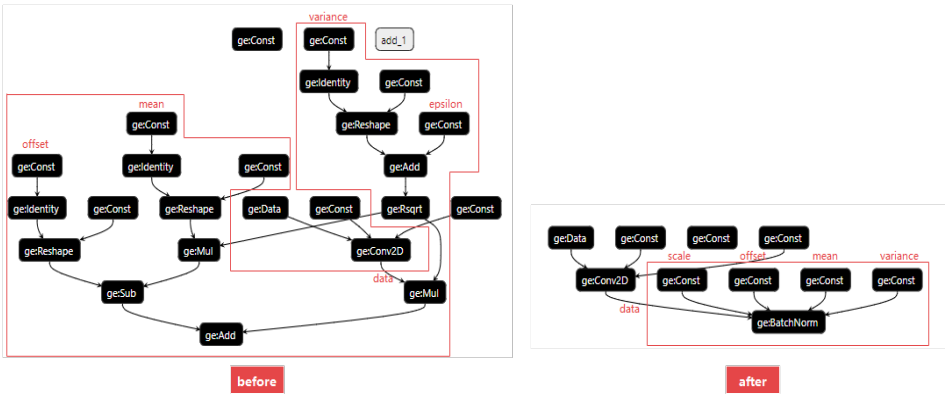
- 无scale和offset，is_training=False，data_format=NCHW，输入节点为Const类型，融合前后网络结构图为：



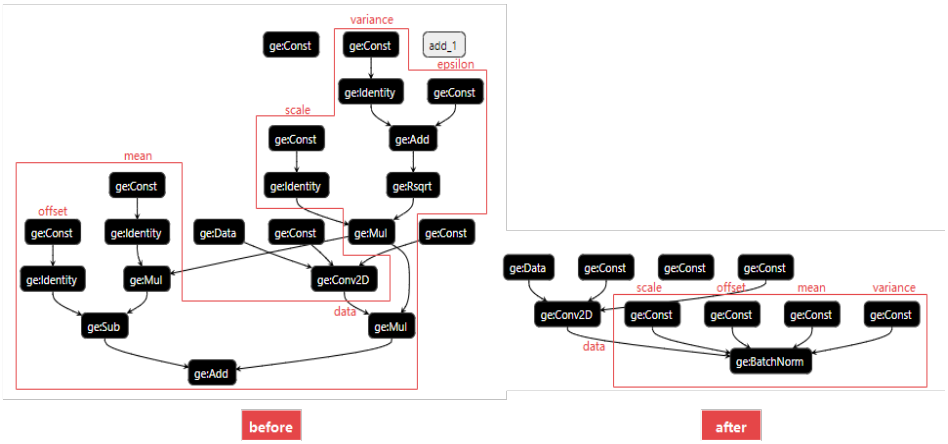
- 无scale，is_training=False，data_format=NHWC，输入节点为Const类型，融合前后网络结构图为：



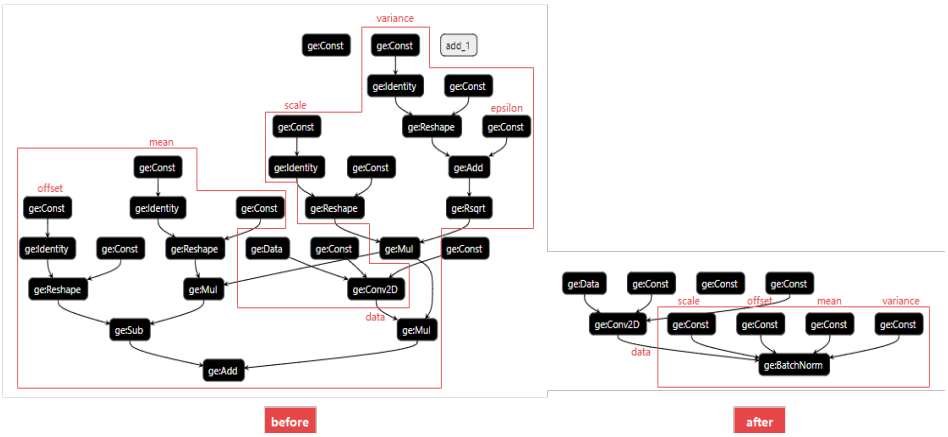
- 无scale，is_training=False，data_format=NCHW，输入节点为Const类型，融合前后网络结构图为：



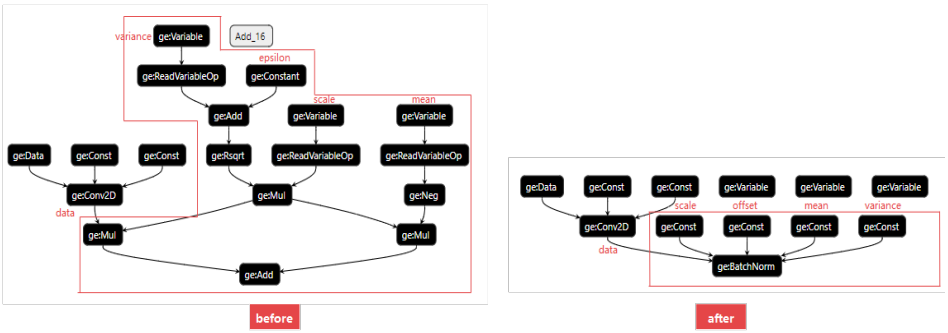
- is_training=False，data_format=NHWC，输入节点为Const类型，融合前后网络结构图为：



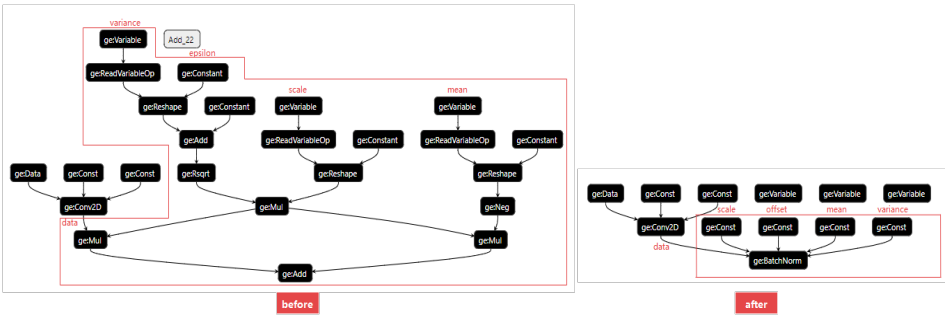
- is_training=False，data_format=NCHW，输入节点为Const类型，融合前后网络结构图为：



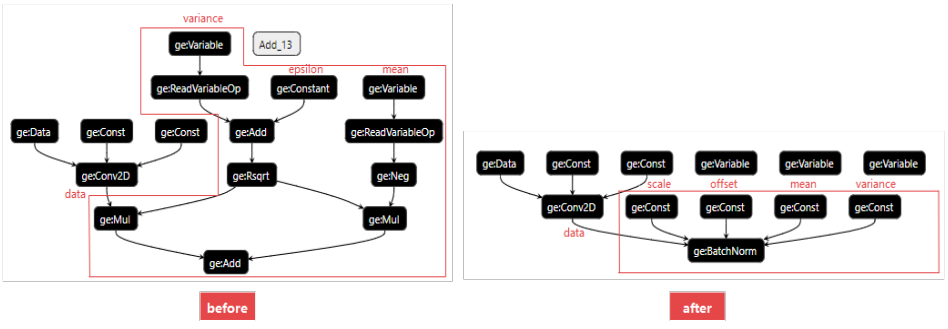
- 无offset，is_training=False，data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：



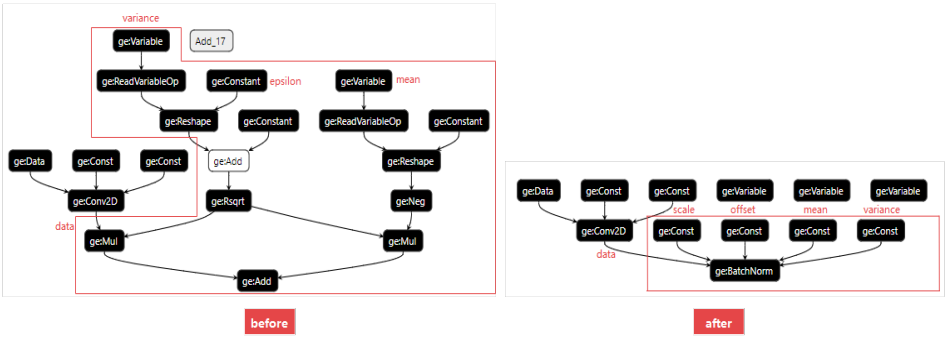
- 无offset，is_training=False，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：



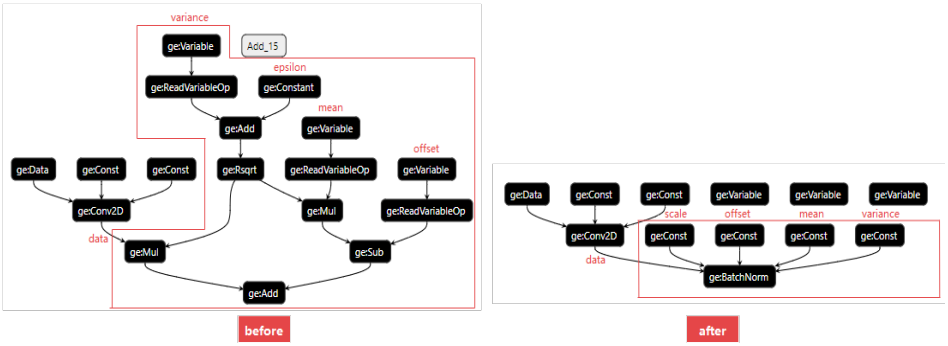
- 无scale和offset，is_training=False，data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：



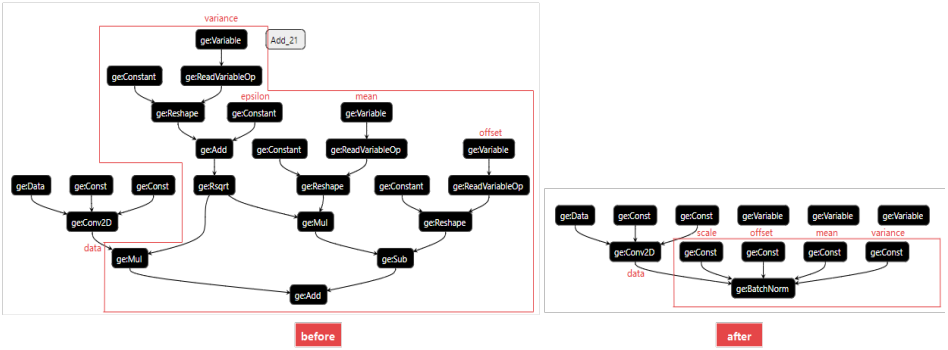
- 无scale和offset，is_training=False，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：



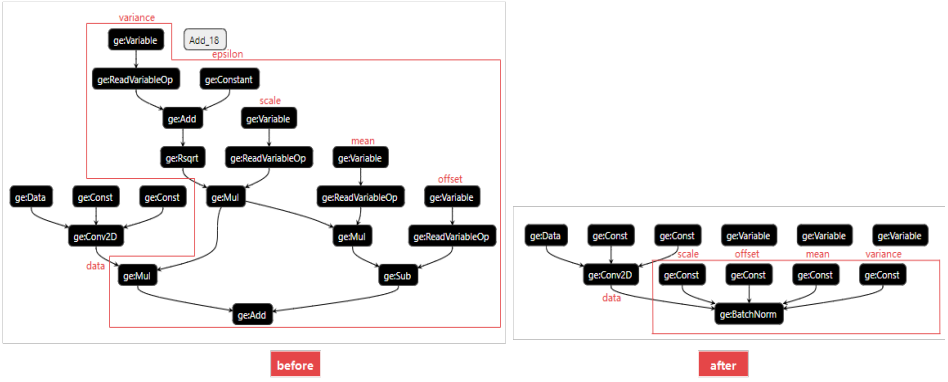
- 无scale，is_training=False，data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：



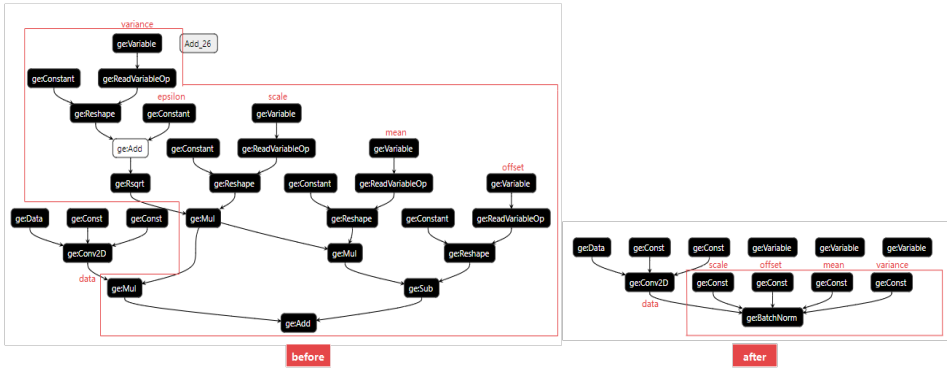
- 无scale，is_training=False，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：



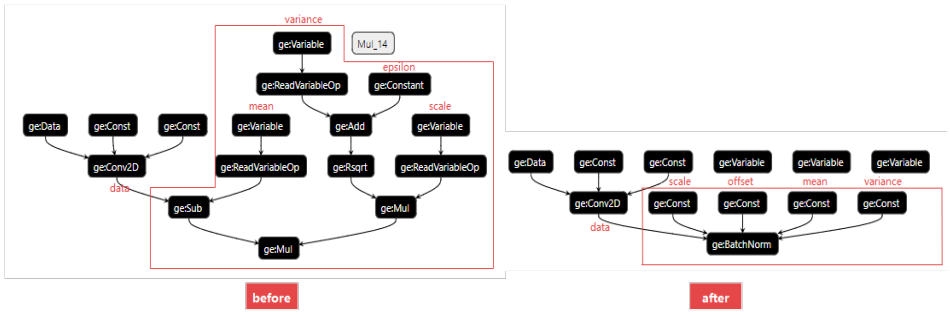
- is_training=False，,data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：



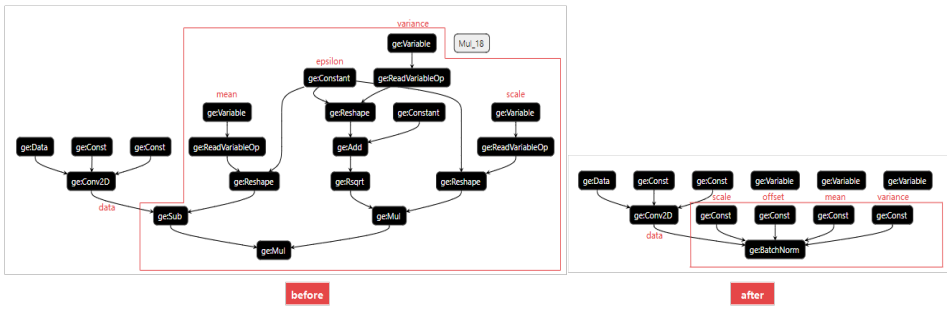
- is_training=False，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：



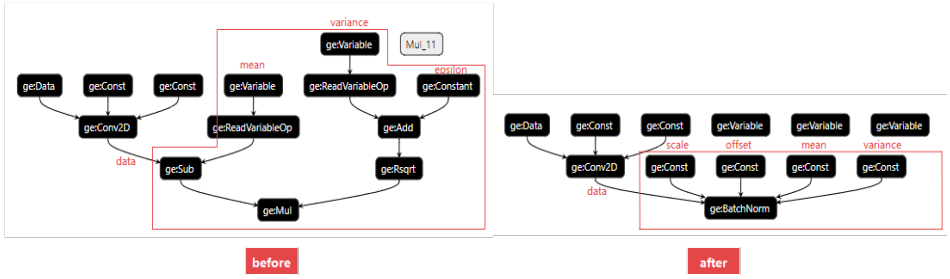
- 一种简化后的BN结构，无offset，data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：



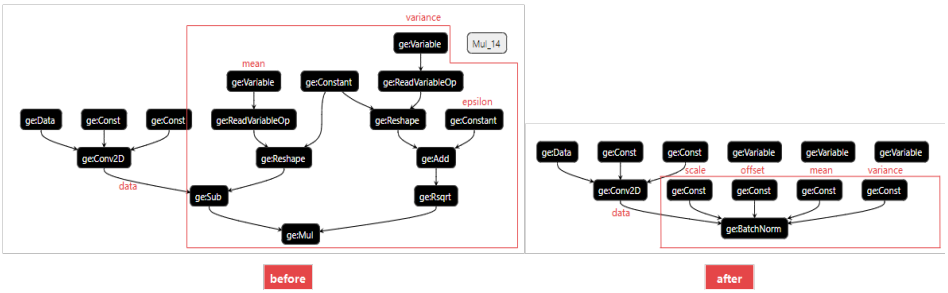
- 一种简化后的BN结构，无offset，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：



- 一种简化后的BN结构，无scale和offset，data_format=NHWC，输入节点为Variable类型，融合前后网络结构图为：

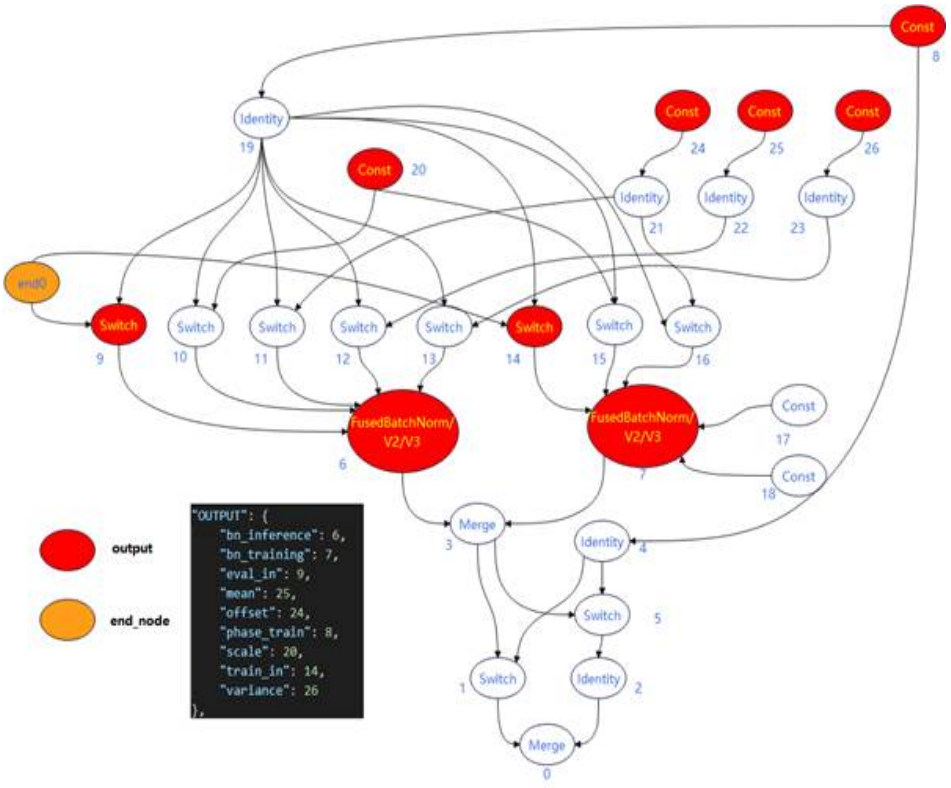


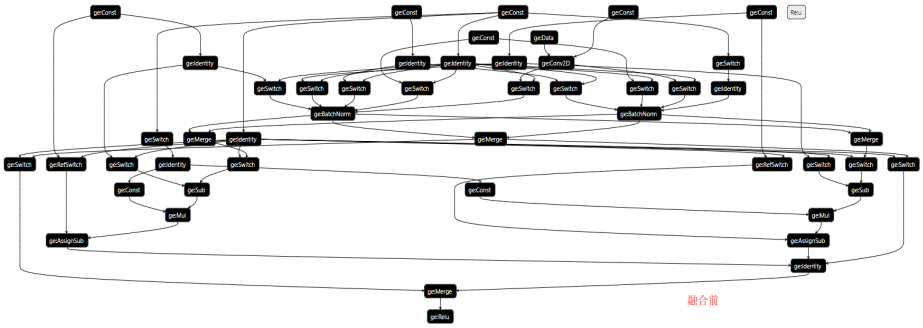
- 一种简化后的BN结构，无scale和offset，data_format=NCHW，输入节点为Variable类型，融合前后网络结构图为：

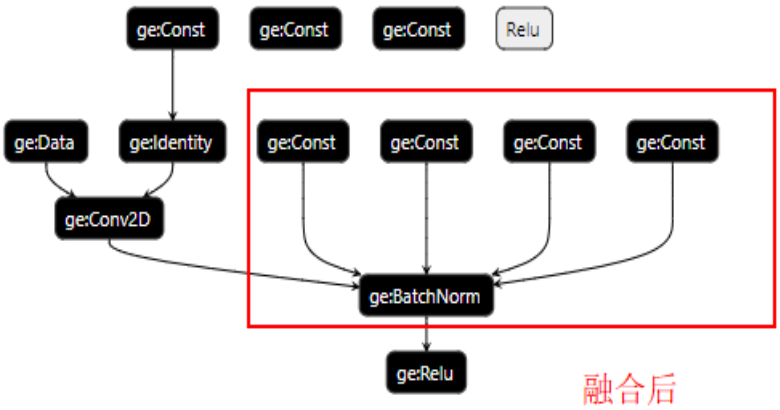


- bn_branch类小算子融合成BN大算子

  AMCT对bn_branch小算子结构的双BN结构进行匹配，并将匹配到的小算子结构替换为大算子的BN结构。如果结构中存在以下三种场景：两个BN分支的输入数据来源不同、结构输出为训练BN输出、推理BN的is_traing参数为True，则不做融合。算子融合过程会保留原结构中的推理BN大算子，并删除多余的Switch等小算子结构的节点，并重新添加保留BN算子的输入输出边，实现小算子融合成大算子。具体支持的小算子BN结构的场景如下所示：







9.2 AIPP

功能介绍

AIPP（Artificial Intelligence Pre-Processing）人工智能预处理，用于在AI Core上完成图像预处理，包括色域转换（转换图像格式）、图像归一化（减均值/乘系数）和抠图（指定抠图起始点，抠出神经网络需要大小的图片）。AIPP区分为静态AIPP和动态AIPP，只能二选一，不能同时支持。

- 静态AIPP：模型编译时设置AIPP模式为静态，同时设置AIPP参数，模型生成后，AIPP参数值被保存在离线模型中，每次模型推理过程采用固定的AIPP预处理参数（无法修改）。
- 动态AIPP：模型编译时设置AIPP模式为动态，每次模型推理前，根据需求，在执行模型前设置动态AIPP参数值，然后在模型执行时可使用不同的AIPP参数。动态AIPP在根据业务要求改变预处理参数的场合下使用（如不同摄像头采用不同的归一化参数，输入图片格式需要兼容YUV420和RGB等）。

关于AIPP功能的详细介绍请参考《**ATC工具使用指南**》。

使用方法

下面介绍如何在模型构建时支持AIPP功能：

1. Data算子输入数据的Channel维度的元素个数和图片实际保持一致。
```
auto shape_data = vector<int64_t>({ 1,10,12,12 });
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_NCHW, DT_FLOAT16);
auto data = op::Data("data");
data.update_input_desc_x(desc_data);
data.update_output_desc_y(desc_data);
```
2. 准备AIPP配置文件，配置说明请参考《**ATC工具使用指南**》，下面给出一些配置示例：

   静态AIPP配置文件示例：
```
aipp_op {
aipp_mode:static         # 表示静态AIPP
input_format:YUV420SP_U8
csc_switch:true
var_reci_chn_0:0.00392157
```



```
var_reci_chn_1:0.00392157
var_reci_chn_2:0.00392157
}
```

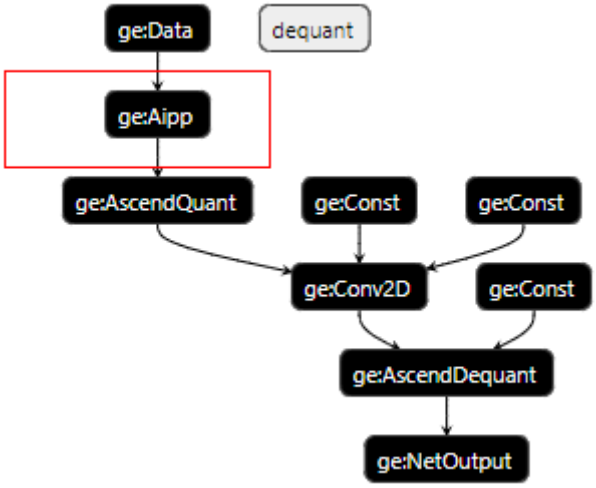
动态AIPP配置文件示例（除了以下字段外，其他无需配置）：

```
aipp_op {
aipp_mode:dynamic         # 表示动态AIPP
related_input_rank: 0     # 可选，标识对模型的第几个输入做AIPP处理，从0开始，默认为0
max_src_image_size:752640 # 输入图像最大的size，动态AIPP必须配置
}
```

3. 模型编译时，在aclgrphBuildModel接口options中设置**INSERT_OP_FILE**和**INPUT_FORMAT**。

```
void PrepareOptions(std::map<AscendString, AscendString>& options) {
    options.insert({
        {ge::ir_option::EXEC_DISABLE_REUSED_MEMORY, "1"},
       {ge::ir_option::INSERT_OP_FILE, PATH + "aipp_nv12_img.cfg"},
       {ge::ir_option::INPUT_FORMAT, "NCHW"}
        });
}
```

4. 编译生成的离线模型自动插入aipp算子。



9.3 动态 BatchSize

功能介绍

BatchSize即每次模型推理处理的图片数，对于每次推理图片数量固定的场景，处理图片数由数据shape的N值决定；对于每次推理图片数量不固定的场景，则可以通过动态BatchSize功能来动态分配每次处理的图片数量。例如用户执行推理业务时需要每次处理2张，4张，8张图片，则可以在模型中配置档位信息2,4,8，申请了档位后，模型推理时会根据实际档位申请内存。

使用方法

下面介绍如何在模型构建时支持动态BatchSize功能：

1. 在Data算子定义时，将数据shape的指定维度设置为-1：

```
auto shape_data = vector<int64_t>({ -1,1,28,28 });
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
auto data = op::Data("data");
data.update_input_desc_data(desc_data);
data.update_output_desc_out(desc_data);
```


2. 模型编译时，在aclgrphBuildModel接口options中设置INPUT_SHAPE/INPUT_FORMAT信息，同时通过DYNAMIC_BATCH_SIZE指定动态batchsize的具体大小。

须知

- INPUT_FORMAT必须设置并且和所有Data算子的format保持一致，且仅支持NCHW和NHWC，否则会导致模型编译失败。
- INPUT_SHAPE可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。

```
void PrepareOptions(std::map<AscendString, AscendString>& options) {
    options.insert({
        {ge::ir_option::INPUT_FORMAT, "NCHW"},
        {ge::ir_option::INPUT_SHAPE, "data:-1,1,28,28"},
        {ge::ir_option::DYNAMIC_BATCH_SIZE, "2,4,8"}     // 设置N的档位
    });
}
```

使用注意事项

- 该功能不能和动态分辨率、动态维度同时使用。
- DYNAMIC_BATCH_SIZE最多支持100档配置，每一档通过英文逗号分隔，每个档位数值限制为：[1~2048]。如果用户设置的档位数值过大或档位过多，可能会导致模型编译失败，此时建议用户减少档位或调低档位数值。
- 如果模型编译时通过该参数设置了动态batch，则使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，增加**aclmdlSetDynamicBatchSize**接口，用于设置真实的BatchSize档位。关于**aclmdlSetDynamicBatchSize**接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。

9.4 动态分辨率

功能介绍

在模型推理时，对于每次处理图片宽和高不固定的场景，用户可以在模型构建时设置不同的图片宽高档位。

使用方法

1. 在Data算子定义时，将数据shape的HW维度设置为-1：

```
auto shape_data = vector<int64_t>({ 8,3,-1,-1 });
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
auto data = op::Data("data");
data.update_input_desc_data(desc_data);
data.update_output_desc_out(desc_data);
```

2. 模型编译时，在aclgrphBuildModel接口options中设置INPUT_SHAPE/INPUT_FORMAT信息，同时通过DYNAMIC_IMAGE_SIZE指定动态分辨率的档位。

须知

- INPUT_FORMAT必须设置并且和所有Data算子的format保持一致，且仅支持NCHW和NHWC，否则会导致模型编译失败。
- INPUT_SHAPE可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。

```
void PrepareOptions(std::map<AscendString, AscendString>& options) {
    options.insert({
        {ge::ir_option::INPUT_FORMAT, "NCHW"},
        {ge::ir_option::INPUT_SHAPE, "data: 8,3,-1,-1"},
        {ge::ir_option::DYNAMIC_IMAGE_SIZE, "416,416;832,832"}  // 设置HW档位，支持处理HW为416,416，或者832,832的图片
    });
}
```

使用注意事项

- 该功能不能和动态Batch、动态维度同时使用。
- DYNAMIC_IMAGE_SIZE最多支持100档配置，每一档通过英文分号分隔。如果用户设置的分辨率数值过大或档位过多，可能会导致模型编译失败，此时建议用户减少档位或调低档位数值。
- 如果模型编译时通过该参数设置了动态分辨率，则使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，增加**aclmdlSetDynamicHWSize**接口，用于设置真实的分辨率，且实际推理时，使用的数据集图片大小需要与具体使用的分辨率相匹配。关于**aclmdlSetDynamicHWSize**接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。

9.5 动态分档

9.5.1 编译 Graph 为离线模型场景

功能介绍

用户可以在模型构建时，设置ND格式下动态维度的档位。适用于执行推理时，每次处理任意维度的场景。

支持的芯片型号

昇腾310 AI处理器

昇腾310P AI处理器

昇腾910 AI处理器

昇腾910B AI处理器

使用方法

1. 在Data算子定义时，将数据shape的动态维度设置为-1：

```
auto shape_data = vector<int64_t>({ 1,-1,-1 });
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
auto data = op::Data("data");
```



```
    data.update_input_desc_data(desc_data);
    data.update_output_desc_out(desc_data);
```

2. 模型编译时，在aclgrphBuildModel接口options中设置INPUT_SHAPE/INPUT_FORMAT信息，同时通过DYNAMIC_DIMS指定档位信息。

须知

- INPUT_FORMAT必须设置并且和所有Data算子的format保持一致，且仅支持ND，否则会导致模型编译失败。
- INPUT_SHAPE可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。

```
void PrepareOptions(std::map<std::string, std::string>& options) {
    options.insert({
        {ge::ir_option::INPUT_FORMAT, "ND"},
        {ge::ir_option::INPUT_SHAPE, "data:1,-1,-1"},
        {ge::ir_option::DYNAMIC_DIMS, "1,2;3,4;5,6;7,8"}  // 模型编译时，支持的data算子的shape为1,1,2; 1,3,4; 1,5,6; 1,7,8
    });
}
```

使用注意事项

- 该功能不能和动态Batch、动态分辨率、AIPP功能同时使用。
- 参数通过"dim1,dim2,dim3;dim4,dim5,dim6;dim7,dim8,dim9"的形式设置，所有档位必须放在双引号中，每档中间使用英文分号分隔，每档中的dim值与INPUT_SHAPE参数中的-1标识的参数依次对应，INPUT_SHAPE参数中有几个-1，则每档必须设置几个维度。例如：

```
void PrepareOptions(std::map<std::string, std::string>& options) {
    options.insert({
        {ge::ir_option::INPUT_FORMAT, "ND"},
        {ge::ir_option::INPUT_SHAPE, "data:1,1,40,-1;label:1,-1;mask:-1,-1"},
        {ge::ir_option::DYNAMIC_DIMS, "20,20,1,1;40,40,2,2;80,60,4,4"}
    });
}
```

  则模型编译时，支持的输入shape为：

  第0档：data(1,1,40,20)+label(1,20)+mask(1,1)

  第1档：data(1,1,40,40)+label(1,40)+mask(2,2)

  第2档：data(1,1,40,80)+label(1,60)+mask(4,4)

- 支持的档位数取值范围为：(1,100]，每档最多支持任意指定4个维度，建议配置为3~4档。
- 如果模型编译时通过该参数设置了动态维度，则使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，增加**aclmdlSetInputDynamicDims**接口，用于设置真实的维度。关于**aclmdlSetInputDynamicDims**接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。

9.5.2 编译并运行 Graph 场景

功能介绍

当前系统支持在用户脚本中指定配置动态档位信息，从而支持动态输入的场景。

编译并运行Graph场景的动态分档当前仅支持整图分档：使用session参数设置分档信息，输入可以为dataset方式、placeholder方式，或者两种混合方式。对于混合输入，当前仅支持其中一种为动态变化的场景。

支持的芯片型号

昇腾310P AI处理器

昇腾910 AI处理器

昇腾910B AI处理器

使用方法

1. 在Data算子定义时，将数据shape的动态维度设置为-1：

```
auto shape_data = vector<int64_t>({ 1,-1,-1 });
TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
auto data = op::Data("data");
data.update_input_desc_data(desc_data);
data.update_output_desc_out(desc_data);
```

2. 编译并运行Graph时，在Session接口和AddGraph接口的options中设置ge.inputShape/ge.dynamicNodeType信息，同时通过ge.dynamicDims指定档位信息。

须知

- ge.inputShape可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。
- 整图分档时，用户在脚本中设置的ge.inputShape的输入顺序要和实际data节点的name字母序保持一致，比如有三个输入：label、data、mask，则ge.inputShape输入顺序应该为data、label、mask：

```
"data:1,1,40,-1;label:1,-1;mask:-1,-1"
```

```
std::map<std::string, std::string> options = {{"ge.inputShape", ""data:1,1,40,-1;label:1,-1;mask:-1,-1""},
                                              {"ge.dynamicDims", "20,20,1,1;40,40,2,2;80,60,4,4"},
                                              {"ge.dynamicNodeType", "1"}};
//分档参数添加到session
ge::Session *session = new ge::Session(options);
//分档参数添加到graph
session->AddGraph(graph_id, graph, options);
```

- ge.inputShape表示网络的输入shape信息，以上配置表示网络中有三个输入，输入的name分别为data，label，mask，各输入的shape分别为（1,1,40,-1）、（1,-1）、（-1,-1），其中-1表示该维度上为动态档位，需要通过ge.dynamicDims设置动态档位参数。
- ge.dynamicDims表示输入的对应维度的档位信息。档位中间使用英文分号分隔，每档中的dim值与ge.inputShape参数中的-1标识的参数依次对应，ge.inputShape参数中有几个-1，则每档必须设置几个维度。结合ge.inputShape信息，ge.dynamicDims配置为"20,20,1,1;40,40,2,2;80,60,4,4"的含义如下：

  有三个“;”，表示输入shape支持三个档位，每个档位中的值对应输入shape中的“-1”的取值：

  - 第0档：data(1,1,40,20)，label(1,20)，mask(1,1)
  - 第1档：data(1,1,40,40)，label(1,40)，mask(2,2)

  - 第2档：data(1,1,40,80)，label(1,60)，mask(4,4)
- dynamic_node_type用于指定动态输入的节点类型。0：dataset输入为动态输入；1：placeholder输入为动态输入。当前不支持dataset和placeholder输入同时为动态输入。

9.6 动态输入 shape range

功能介绍

用户在模型编译时可以指定模型输入数据的shape range，从而编译出支持动态输入的模型。

支持的芯片型号

昇腾910 AI处理器

昇腾310P AI处理器

昇腾910B AI处理器

使用方法

下面介绍如何在模型构建时设置动态输入的shape range：

1. 在Data算子定义时，将数据shape的指定维度设置为-1：
   ```
   auto shape_data = vector<int64_t>({ -1,3,5,-1 });
   TensorDesc desc_data(ge::Shape(shape_data), FORMAT_ND, DT_FLOAT);
   auto data = op::Data("data");
   data.update_input_desc_data(desc_data);
   data.update_output_desc_out(desc_data);
   ```
2. 模型编译时，在aclgrphBuildModel接口options中通过INPUT_SHAPE指定模型输入数据的shape range。
   ```
   void PrepareOptions(std::map<AscendString, AscendString>& options) {
       options.insert({
           {ge::ir_option::INPUT_FORMAT, "NCHW"},
           {ge::ir_option::INPUT_SHAPE, "8~20,3,5,-1"}
       });
   }
   ```
   INPUT_SHAPE参数设置shape范围时的格式要求：
   - 支持按照name设置："input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"，例如："input_name1:8~20,3,5,-1;input_name2:5,3~9,10,-1"。指定的节点必须放在双引号中，节点中间使用英文分号分隔。input_name必须是转换前的网络模型中的节点名称。如果用户知道data节点的name，推荐按照name设置。
   - 支持按照index设置："n1,c1,h1,w1;n2,c2,h2,w2"，例如："8~20,3,5,-1；；5,3~9,10,-1"。可以不指定节点名，节点按照索引顺序排列，节点中间使用英文分号分隔。按照index设置shape范围时，data节点需要设置属性index，说明是第几个输入，index从0开始。
   - 动态维度有shape范围的用波浪号“~”表示，固定维度用固定数字表示，无限定范围的用-1表示。

使用注意事项

如果模型编译时通过该参数设置了动态输入的shape range，则进行模型推理时，需要在模型执行**aclmdlExecute**接口之前调用**aclmdlSetDatasetTensorDesc接口**，用于

设置该输入的tensor描述信息（主要是设置Shape信息）；模型执行之后，调用**aclmdlGetDatasetTensorDesc**接口获取模型动态输出的Tensor描述信息（主要是获取输出内存的大小、输出内存地址等）。接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。

9.7 Profiling 性能数据采集

功能介绍

用户可以在图加载和图执行过程中，采集Profiling性能数据，用于性能分析。使用时有三种方式：

表 9-4 Profiling 性能数据采集方式

序号	采集方式
方式一（不推荐使用）	通过配置如下环境变量，采集Profiling性能数据： • PROFILING_MODE • PROFILING_OPTIONS
方式二	通过**10.1.3.6.1 GEInitialize**传入option参数： • ge.exec.profilingMode • ge.exec.profilingOptions
方式三	调用如下接口，采集Profiling性能数据： • **10.1.3.8.1 aclgrphProfInit** • **10.1.3.8.2 aclgrphProfFinalize** • **10.1.3.8.3 aclgrphProfCreateConfig** • **10.1.3.8.4 aclgrphProfDestroyConfig** • **10.1.3.8.5 aclgrphProfStart** • **10.1.3.8.6 aclgrphProfStop** 如果需要采集迭代轨迹数据，还需要通过**10.1.3.6.1 GEInitialize**传入option参数ge.exec.profilingOptions或通过环境变量PROFILING_OPTIONS传入。传入字段包括training_trace/bp_point/fp_point。

说明

环境变量说明请参考**10.2 环境变量参考**。

支持的芯片型号

昇腾310P AI处理器

昇腾910 AI处理器

采集前配置

进行Profiling采集前，需要从**8 完整样例参考**中获取构建Graph并直接编译并运行Graph样例后，进行如下操作：

1. 在源码文件“main.cpp”开头添加“#include "ge/ge_prof.h"”代码。
2. 在编译脚本“Makefile”内的“LIBS”行下添加“-lmsprofiler”字段或在“CMakeLists.txt”内的“target_link_libraries”行下添加“msprofiler”字段。

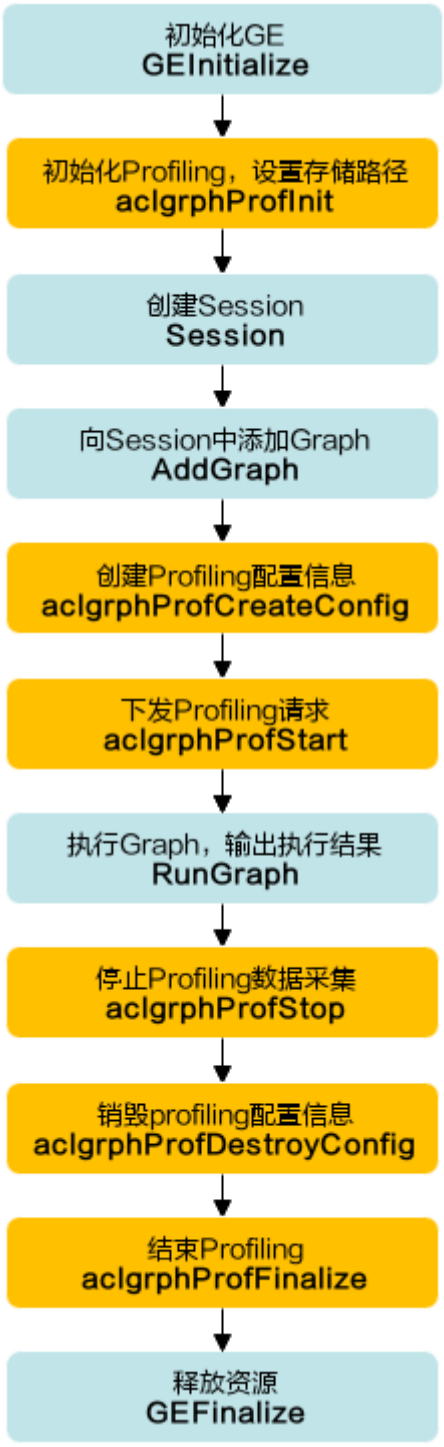
表 9-5 头文件列表

定义接口的头文件	用途	对应的库文件
ge/ge_prof.h	用于定义Profiling配置的接口。	libmsprofiler.so **说明** ge头文件在“CANN软件安装后文件存储路径/include/”目录下，ge库文件在“CANN软件安装后文件存储路径/lib64/”目录下。

通过方式三采集性能数据

该特性为全局特性，不是session级特性，使能后在所有session均配置生效。

建议的接口调用顺序为：



调用示例为：

```
// 构造Graph，该步骤省略
// ......

// init ge
std::map<std::string, std::string> ge_options = {{"ge.socVersion", "xxx"}, {"ge.graphRunMode", "1"}};
ge::GEInitialize(ge_options);

std::string profilerResultPath = "/home/test/prof";       //该路径需要提前创建
uint32_t length = strlen("/home/test/prof");
ret = ge::aclgrphProfInit(profilerResultPath.c_str(), length);
```



```
std::map<string, string> options = {{"a", "b"}, {"c", "d"}};
uint32_t graphId = 0;

ge::Session *session = new Session(options);
ret = session->AddGraph(graphId, graph);

uint32_t deviceid_list[1] = {0};
uint32_t device_nums = 1;
uint64_t data_type_config = ProfDataTypeConfig::kProfTaskTime | ProfDataTypeConfig::kProfAiCoreMetrics
| ProfDataTypeConfig::kProfAicpu | ProfDataTypeConfig::kProfTrainingTrace | ProfDataTypeConfig::kProfHccl
| ProfDataTypeConfig:: kProfL2cache;
ProfAicoreEvents *aicore_events = NULL;
ProfilingAicoreMetrics aicore_metrics = ProfilingAicoreMetrics::kAicoreArithmeticUtilization;
ge::aclgrphProfConfig *pro_config = ge::aclgrphProfCreateConfig(deviceid_list, device_nums,
aicore_metrics, aicore_events, data_type_config);

ge::aclgrphProfStart(pro_config);

session->RunGraph(graphId, inputs_r, outputs_r);

ge::aclgrphProfStop(pro_config);

ge::aclgrphProfDestroyConfig(pro_config);

ge::aclgrphProfFinalize();

delete session;
ge::GEFinalize();
```

9.8 图编译缓存

功能介绍

图编译缓存功能支持将图编译结果进行磁盘持久化，当应用程序重新运行时直接加载磁盘上缓存的编译结果。在编译并运行Graph过程中，为了减少图编译时长，可以开启图编译缓存功能。

使用约束

- 当前不支持带资源类算子的模型。
- 根据options参数中的ge.graph_compiler_cache_dir和ge.graph_key确定缓存文件，缓存文件不存在则保存缓存，缓存文件存在则直接加载缓存。ge.graph_compiler_cache_dir和ge.graph_key同时配置非空时该功能生效。
- ge.graph_compiler_cache_dir指定的缓存目录必须存在，否则会导致编译失败。
- ge.graph_key由用户保证其唯一性，否则，在相同ge.graph_key的缓存已存在的情况下，会直接加载对应的缓存。
- 图发生变化后，原来的缓存文件不可用，用户需要手动删除缓存目录中的缓存文件或者修改ge.graph_key，重新编译生成缓存文件。
- 跨版本的缓存无法保证兼容性，如果版本升级，需要清理缓存目录重新编译生成缓存。

使用方法

```
std::map<ge::AscendString, ge::AscendString> session_options = {{"ge.graph_compiler_cache_dir", "./
build_cache_dir"}};
std::shared_ptr<ge::Session> session = std::make_shared<ge::Session>(session_options);
const auto graph = CreateGraph();
std::map<ge::AscendString, ge::AscendString> graph_options = {{"ge.graph_key", "test_graph_001"}};
```



```
auto = session->AddGraph(0, graph, graph_options);
...
```

缓存文件生成规则

生成文件包括：

- 模型缓存文件
- 索引文件，便于用户通过graph_key快速找到对应的缓存文件，索引文件内容示例如下：

```
{
   "cache_file_list":[
      {
         "cache_file_name":"./cache_dir/graph_$key1_20230117202307.om",
         "graph_key":"graph_$key1",
         "var_desc_file_name":"./cache_dir/graph_$key1_20230117202307.rdcpkt"
      },
      {
         "cache_file_name":"./cache_dir/graph_$key1_20230117203007.om",
         "graph_key":"graph_$key1",
         "var_desc_file_name":"./cache_dir/graph_$key1_20230117203007.rdcpkt"
      }
   ]
}
```

- 变量格式文件，仅在图中存在变量时生成。用于框架匹配模型缓存文件，如果graph_key对应的图内变量格式发生变更，则之前缓存的缓存文件将无法直接恢复使用，该场景下会重新触发编译流程重新生成缓存文件。

文件名生成规则：

- 当ge.graph_key配置值只包含大小写字母（A-Z，a-z）、数字（0-9）、下划线（_）、中划线（-）并且长度不超过128时
  - 索引文件命名为：ge.graph_key + “.idx”。
  - 模型缓存文件命名为: ge.graph_key + 时间戳 + “.om”。
  - 变量格式文件命名为: ge.graph_key + 时间戳 + “.rdcpkt”。
- 不满足上面条件时，缓存文件会根据ge.graph_key的hashcode生成
  - 索引文件命名为："hash_" + hashcode + “.idx”，比如hash_17242059865018222533.idx。
  - 模型缓存文件命名为: "hash_" + hashcode + 时间戳 + “.om”。
  - 变量格式文件命名为: "hash_" + hashcode + 时间戳 + “.rdcpkt”。

10 参考

10.1 接口参考

10.1.1 原型定义接口

10.1.1.1 原型定义接口（REG_OP）

函数原型

函数原型定义示例如下：

```
REG_OP(xxx)
    .INPUT(x1, type)
    .OPTIONAL_INPUT(x2, type)
    .DYNAMIC_INPUT(x3, type)
    .OUTPUT(y1, type)
    .DYNAMIC_OUTPUT(y3, type)
    .REQUIRED_ATTR(a, type)
    .ATTR(b, type, default_value)
    .GRAPH(z1)
    .DYNAMIC_GRAPH(z2)
    .OP_END_FACTORY_REG(xxx)
```

功能说明

定义算子的原型，包括算子的输入、输出、属性以及对应的数据类型。

进行如上算子原型定义后，即相当于向GE注册了该算子的原型，告知GE对应类型的算子应该具备哪些输入、输出与属性；同时相当于定义了一个op::xxx的Class，开发者可以include该原型头文件，然后实例化该Class进行IR模型构建，如下所示：

```
conv = op::Conv2D()
conv.set_input_x(feature_map_data)
conv.set_input_filter(weight_data)
```

具体的模型构建可以参考《**Ascend Graph开发指南**》。

接口说明

接口名称	接口说明	衍生接口（可用于IR模型构建）
REG_OP(x)	定义一个算子原型，算子类型为x。	REG_OP
.INPUT(x, type)	定义输入名称（x）和类型(type)。 类型为TensorType类型，例如： • TensorType{DT_FLOAT} • TensorType({DT_FLOAT, DT_INT8}) • TensorType::ALL() 关于TensorType类，请参见TensorType类说明。	INPUT
.OPTIONAL_INPUT(x, type)	定义可选输入的名称（x）和类型（type）。 类型为TensorType类型，例如： • TensorType{DT_FLOAT} • TensorType({DT_FLOAT, DT_INT8}) • TensorType::ALL() 关于TensorType类，请参见TensorType类说明。	OPTIONAL_INPUT
.DYNAMIC_INPUT(x, type)	定义动态输入的名称（x）和类型（type）。 类型为TensorType类型，例如： • TensorType{DT_FLOAT} • TensorType({DT_FLOAT, DT_INT8}) • TensorType::ALL() 关于TensorType类，请参见TensorType类说明。	DYNAMIC_INPUT

接口名称	接口说明	衍生接口（可用于IR模型构建）
.OUTPUT(x, type)	定义输出的名称（x）和类型（type）。 类型为TensorType类型，例如： • TensorType{DT_FLOAT} • TensorType({DT_FLOAT, DT_INT8}) • TensorType::ALL() 关于TensorType类，请参见**TensorType类说明**。	**OUTPUT**
.DYNAMIC_OUTPUT(x, type)	定义动态输出的名称（x）和类型（type）。 类型为TensorType类型，例如： • TensorType{DT_FLOAT} • TensorType({DT_FLOAT, DT_INT8}) • TensorType::ALL() 关于TensorType类，请参见**TensorType类说明**。	**DYNAMIC_OUTPUT**

接口名称	接口说明	衍生接口（可用于IR模型构建）
.REQUIRED_ATTR(x, type)	定义必备属性的名称（x）和类型（type）。 type的可选值包括： • Int，属性类型为int64_t • Float, 属性类型为float • String，属性类型为string • Bool，属性类型为bool • Tensor，属性类型为Tensor • Type，属性为Type枚举定义 • NamedAttrs，属性类型为NamedAttrs • AscendString，属性类型为AscendString • ListInt，属性类型为vector<int64_t>，int64_t列表 • ListFloat, 属性类型为vector<float>，float列表 • ListString，属性类型为vector<string>，string列表 • ListBool，属性类型为vector<bool>，bool列表 • ListTensor，属性类型为vector<Tensor>，Tensor列表 • Bytes，属性类型为Buffer • ListType，属性类型为vector<Type>，Type列表 • ListListInt，属性类型为	REQUIRED_ATTR

接口名称	接口说明	衍生接口（可用于IR模型构建）
	vector<vector<int64_t>>，2维列表 ● ListAscendString，属性类型为vector<AscendString>，AscendString列表 ● ListNamedAttrs，属性类型为vector<NamedAttrs>，NamedAttrs列表	

接口名称	接口说明	衍生接口（可用于IR模型构建）
.ATTR(x, type, default_value)	定义可选属性的名称、类型以及默认值。 当用户不设置算子对象的属性时，会使用此处设置的默认值。 type的可选值包括： • Int，属性类型为int64_t • Float, 属性类型为float • String，属性类型为string • Bool，属性类型为bool • Tensor，属性类型为Tensor • Type，属性为Type枚举定义 • NamedAttrs，属性类型为NamedAttrs • AscendString，属性类型为AscendString • ListInt，属性类型为vector<int64_t>, int64_t列表 • ListFloat, 属性类型为vector<float>, float列表 • ListString，属性类型为vector<string>，string列表 • ListBool，属性类型为vector<bool>，bool列表 • ListTensor，属性类型为vector<Tensor>，Tensor列表 • Bytes，属性类型为Buffer • ListType，属性类型为vector<Type>，Type列表	ATTR

接口名称	接口说明	衍生接口（可用于IR模型构建）
	• ListListInt，属性类型为vector<vector<int64_t>>，2维列表 • ListAscendString，属性类型为vector<AscendString>，AscendString列表 • ListNamedAttrs，属性类型为vector<NamedAttrs>，NamedAttrs列表 定义示例： • .ATTR(mode, Int, 1) • .ATTR(pad, ListInt, {0, 0, 0, 0})	
.GRAPH(z1)	注册算子中包含的子图信息，输入z1为子图名称。 例如If算子注册的子图为： .GRAPH(then_branch) .GRAPH(else_branch) 对于同一个算子，注册的算子子图名称需要保持唯一。	GRAPH
.DYNAMIC_GRAPH(z2)	注册动态算子子图信息，输入z2为子图名称。 例如Case算子注册的子图为： .DYNAMIC_GRAPH(branches) 对于同一个算子，注册的算子子图名称需要保持唯一。	DYNAMIC_GRAPH
.INFER_SHAPE_AND_TYPE()	该接口为历史遗留兼容性接口，当前版本用户无需使用。	-

接口名称	接口说明	衍生接口（可用于IR模型构建）
.OP_END_FACTORY_REG(x)	与REG_OP配对，结束算子原型定义。 算子类型（x）与REG_OP(x)中的类型相同。	-

说明

OpReg类中的OpReg &N()接口的功能是为了用户进行算子注册的时候，使用.**的方式调用OpReg类的接口，例如.INPUT(x, type)、.OUTPUT(x, type)，无其他含义。

返回值

无

约束和限制说明

- REG_OP的算子类型必须全局唯一。
- 同一个算子的输入名称之间不能重复。
- 同一个算子的输出名称之间不能重复。
- 同一个算子的属性名称之间不能重复。

调用示例和相关 API

动态输入的算子原型定义示例：

```
REG_OP(AddN)
    .DYNAMIC_INPUT(x, TensorType::NumberType(), DT_VARIANT)
    .OUTPUT(y, TensorType::NumberType())
    .REQUIRED_ATTR(N, Int)
    .OP_END_FACTORY_REG(AddN)
```

多输入的算子原型定义示例：

```
REG_OP(GreaterEqual)
    .INPUT(x1, TensorType::RealNumberType())
    .INPUT(x2, TensorType::RealNumberType())
    .OUTPUT(y, TensorType({DT_BOOL}))
    .OP_END_FACTORY_REG(GreaterEqual)
```

注册子图的算子原型定义示例：

```
REG_OP(If)
    .INPUT(cond, TensorType::ALL())
    .DYNAMIC_INPUT(input, TensorType::ALL())
    .DYNAMIC_OUTPUT(output, TensorType::ALL())
    .GRAPH(then_branch)
    .GRAPH(else_branch)
    .OP_END_FACTORY_REG(If)
```


10.1.1.2 衍生接口说明

算子原型定义的相关接口会自动生成对应的衍生接口，可用于IR模型构建，以下接口的详细使用方法可参见《**Ascend Graph开发指南**》。

REG_OP

注册算子类型后，会自动生成算子类型的两个构造函数。

例如，注册算子的类型名称Conv2D，可调用REG_OP(Conv2D)接口，调用该接口后，定义了算子的类型名称Conv2D，同时产生Conv2D的两个构造函数，其中，**Conv2D(const AscendString& name)**需指定算子名称，Conv2D()使用默认算子名称。

```
class Conv2D : public Operator {
    typedef Conv2D _THIS_TYPE;
public:
    explicit Conv2D(const char *name);
    explicit Conv2D();
}
```

INPUT

注册算子输入信息成功后，自动生成算子输入的相关接口，用于获取算子输入的名称、设置算子输入的对应描述等。

例如，注册算子输入x，算子输入支持的数据类型为TensorType{DT_FLOAT}，可调用**INPUT(x, TensorType{DT_FLOAT})**接口，注册算子输入成功后，自动生成以下相关接口：

```
static const string name_in_x(); // 返回输入的名称，即“x”
_THIS_TYPE &set_input_x(Operator& v, const string& srcName); // 指定输入x与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE &set_input_x_by_name(Operator& v, const char *srcName);  // 指定输入x与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE &set_input_x(Operator &v, uint32_t index); // 指定输入x与算子对象v的索引为index的输出存在连接关系，返回算子对象本身
_THIS_TYPE &set_input_x(Operator& v); // 指定输入x与算子对象v的索引0的输出存在连接关系，返回算子对象本身
TensorDesc get_input_desc_x(); // 返回输入x对应的描述
graphStatus update_input_desc_x(const TensorDesc& tensorDesc);// 设置输入x对应的描述，包括Shape、DataType、Format等信息，graphStatus即uint32_t类型，返回非0表示出错
```

OPTIONAL_INPUT

注册可选算子输入信息成功后，自动生成算子输入的相关接口，用于获取算子输入的名称、设置算子输入的对应描述等。

例如，注册算子输入b，算子输入支持的数据类型为TensorType{DT_FLOAT}，可调用**OPTIONAL_INPUT(b, TensorType{DT_FLOAT})**接口，注册算子输入成功后，自动生成以下相关接口：

```
static const string name_in_b(); // 返回输入的名称，即“b”
_THIS_TYPE& set_input_b(Operator& v, const string& srcName);// 指定输入b与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE& set_input_b_by_name(Operator& v, const char *srcName);// 指定输入b与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE& set_input_b(Operator& v); // 指定输入b与算子对象v的索引0的输出存在连接关系，返回算子对象本身
TensorDesc get_input_desc_b(); // 返回输入b对应的描述
graphStatus update_input_desc_b(const TensorDesc& tensorDesc);// 设置输入b对应的描述，包括Shape、DataType、Format等信息
```


DYNAMIC_INPUT

注册动态算子输入信息成功后，自动生成算子输入的相关接口，用于创建动态输入、设置算子输入的对应描述等。

例如，注册算子的动态输入d，算子输入支持的数据类型为TensorType{DT_FLOAT}，可调用**DYNAMIC_INPUT(d, TensorType{DT_FLOAT})**接口，注册算子的动态输入成功后，自动生成以下相关接口：

```
_THIS_TYPE& create_dynamic_input_d(unsigned int num); // 创建动态输入d，包括num个输入，并且把这个输入做为算子最后的输入
_THIS_TYPE &create_dynamic_input_byindex_d(unsigned int num, size_t index) //创建动态输入d，包括num个输入，插入到索引为index的位置，和create_dynamic_input_d不能同时使用
TensorDesc get_dynamic_input_desc_d(unsigned int index);// 返回动态输入d第index个描述，包括Shape、DataType、Format等信息
graphStatus update_dynamic_input_desc_d(unsigned int index, const TensorDesc& tensorDesc);// 更新动态输入d的第index个描述
_THIS_TYPE& set_dynamic_input_d(unsigned int dstIndex, Operator &v); // 设置输入d的第dstIndex个输入与算子对象v的索引0的输出存在连接关系，返回算子对象本身
_THIS_TYPE& set_dynamic_input_d(unsigned int dstIndex, Operator &v, const string &srcName); //指定动态输入d的第dstIndex个输入与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE& set_dynamic_input_d(unsigned int dstIndex, Operator &v, const char *srcName); //指定动态输入d的第dstIndex个输入与算子对象v的输出srcName存在连接关系，返回算子对象本身
```

OUTPUT

注册算子输出信息成功后，自动生成算子输出的相关接口，用户获取算子输出的名称、获取算子输出的描述、设置算子输出的描述。

例如，注册算子输出y，算子输出支持的数据类型为TensorType{DT_FLOAT}，可调用**OUTPUT(y, TensorType{DT_FLOAT})**接口，注册算子输出成功后，自动生成以下相关接口

```
static const string name_out_y();// 返回输出的名称，即“y”
TensorDesc get_output_desc_y();// 返回输出y对应的描述
graphStatus update_output_desc_y(const TensorDesc& tensorDesc); // 设置输出y对应的描述，包括Shape、DataType、Format等信息
```

DYNAMIC_OUTPUT

注册动态算子输出信息成功后，自动生成动态算子输出的相关接口，包括用于创建动态输出、设置算子输出的对应描述等。

例如，注册动态算子输出d，算子输出支持的数据类型为TensorType{DT_FLOAT}，可调用**DYNAMIC_OUTPUT(d, TensorType{DT_FLOAT})**接口，注册动态算子输出成功后，自动生成以下相关接口

```
_THIS_TYPE& create_dynamic_output_d(unsigned int num); // 创建动态输出d，包括num个输出
TensorDesc get_dynamic_output_desc_d(unsigned int index);// 返回动态输出d第index个描述，包括Shape、DataType、Format等信息
graphStatus update_dynamic_output_desc_d(unsigned int index, const TensorDesc& tensorDesc);// 更新动态输出d的第index个描述
```

REQUIRED_ATTR

注册算子属性成功后，自动生成算子属性的3个对外接口，用于获取属性的名称、获取属性的值、设置属性的值。

例如，注册类型为int64_t的属性mode，可调用**REQUIRED_ATTR(mode, Int)**接口，注册算子属性成功后，会自动生成如下接口：

```
static const string name_attr_mode(); // 返回属性的名称，即“mode”
static const void name_attr_mode(AscendString &attr_name);// 出参获取属性的名称，即“mode”
```



```
OpInt get_attr_mode() const; // 返回mode属性的值，OpInt即int64_t
_THIS_TYPE& set_attr_mode(const OpInt& v); // 设置mode属性的值，返回算子对象本身
```

ATTR

注册算子属性成功后，自动生成算子属性的3个对外接口，用于获取属性的名称、获取属性的值、设置属性的值。

下面以注册类型为int64_t的属性、类型为int64_t列表两种场景为例，说明所生成的算子属性接口：

- 调用ATTR(mode, Int, 1)接口，注册属性mode，属性类型为int64_t，默认值为1。

  注册属性成功后，自动生成以下接口：

  ```
  static const string name_attr_mode(); // 返回属性的名称，即“mode”
  static const void name_attr_mode(AscendString &attr_name);// 出参获取属性的名称，即“mode”
  OpInt get_attr_mode() const; // 返回mode属性的值，OpInt即int64_t
  _THIS_TYPE& set_attr_mode(const OpInt& v); // 设置mode属性的值，返回算子对象本身
  ```

- 调用ATTR(pad, ListInt, {0, 0, 0, 0})接口，注册属性pad，属性类型为int64_t列表，默认值为{0,0,0,0}。

  注册属性成功后，自动生成以下接口：

  ```
  static const string name_attr_pad(); // 返回属性的名称，即“pad”
  static const void name_attr_pad(AscendString &attr_name);// 出参获取属性的名称，即“pad”
  OpListInt get_attr_pad() const; ; // 返回属性pad的值，OpListInt即vector<int64_t>
  _THIS_TYPE& set_attr_pad(const OpListInt& v); // 设置属性pad的值，返回算子对象本身
  ```

下面以注册类型为string属性场景为例，说明所生成的算子属性接口：

调用ATTR(data_format, String, "NHWC")接口，注册属性data_format，属性类型为string。

```
static const string name_attr_data_format(); // 返回属性的名称，即“data_format”
static const void name_attr_data_format(AscendString &attr_name);// 出参获取属性的名称，即“data_format”
OpString get_attr_data_format() const; // 返回data_format属性的值，OpString即string
graphStatus get_attr_data_format(AscendString &val);//出参返回data_format属性的值
_THIS_TYPE& set_attr_data_format(const string& v); // 设置data_format属性的值，返回算子对象本身
_THIS_TYPE& set_attr_data_format(const char* v); // 设置data_format属性的值，返回算子对象本身
```

GRAPH

注册算子子图信息成功后，自动生成算子子图的相关接口，用户获取算子子图的名称、获取算子子图的描述、设置算子子图的描述。

例如，注册算子子图y，可调用**GRAPH(y)**接口，注册算子子图成功后，自动生成以下相关接口：

```
static const string name_graph_y();// 返回算子子图的名称，即“y”
SubgraphBuilder get_subgraph_builder_y() const;// 返回子图y对应的构建函数对象
_THIS_TYPE &set_subgraph_builder_y(const SubgraphBuilder &v);// 设置子图y对应的构建函数对象
Graph get_subgraph_y() const;// 获取子图y对应的graph对象
```

DYNAMIC_GRAPH

注册动态算子子图信息成功后，自动生成动态算子子图的相关接口，包括用于创建动态子图、设置算子子图的对应描述等。

例如，注册动态算子子图branches，可调用**DYNAMIC_GRAPH(branches)**接口，注册动态算子子图成功后，自动生成以下相关接口：


```
_THIS_TYPE& create_dynamic_subgraph_branches(unsigned int num); // 创建动态子图branches，包括num个子图
SubgraphBuilder get_dynamic_subgraph_builder_branches(unsigned int index) ;// 返回动态输子图第index个子图构建函数对象
Graph get_dynamic_subgraph_branches(unsigned int index) ;// 返回动态输子图第index个子图对象
_THIS_TYPE &set_dynamic_subgraph_builder_branches(unsigned int index,const SubgraphBuilder &v);// 设置动态子图branches的第index个子图构建函数对象
```

10.1.2 Operator 接口

10.1.2.1 简介

本文档主要描述Operator相关接口，您可以在CANN软件安装后文件存储路径下的“include/graph”路径下查看对应接口的头文件。

接口分类	头文件
AscendString类	ascend_string.h
Operator类	operator.h
Tensor类	tensor.h
TensorDesc类	tensor.h
Shape类	tensor.h
AttrValue类	attr_value.h
Memblock类	ge_allocator.h
Allocator类	ge_allocator.h
数据类型和枚举值	types.h

10.1.2.2 AscendString 类

10.1.2.2.1 AscendString 构造函数和析构函数

函数功能

AscendString构造函数和析构函数。

函数原型

AscendString() = default;

~AscendString() = default;

explicit AscendString(const char* name);

参数说明

参数名	输入/输出	描述
name	输入	字符串名称。

返回值

AscendString构造函数返回AscendString类型的对象。

异常处理

无。

约束说明

无。

10.1.2.2.2 GetString

函数功能

获取字符串地址。

函数原型

const char* GetString() const

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	char	字符串地址。

10.1.2.2.3 关系符重载

对于AscendString对象大小比较的使用场景（例如map数据结构的key进行排序），通过重载以下关系符实现。

```
bool operator<(const AscendString& d) const;
bool operator>(const AscendString& d) const;
bool operator<=(const AscendString& d) const;
bool operator>=(const AscendString& d) const;
```



```
bool operator==(const AscendString& d) const;
bool operator!=(const AscendString& d) const;
```

10.1.2.3 Operator 类

10.1.2.3.1 Operator 构造函数和析构函数

函数功能

Operator构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator();

explicit Operator(const string& type)

explicit Operator(const char *type);

explicit Operator(const string& name, const string& type)

Operator(const ge::AscendString &name, const ge::AscendString &type);

Operator(const char *name, const char *type);

virtual ~Operator()

参数说明

参数名	输入/输出	描述
type	输入	算子类型。
name	输入	算子名称。

返回值

Operator构造函数返回Operator类型的对象。

异常处理

无。

约束说明

无。

10.1.2.3.2 AddControlInput

函数功能

添加算子的控制边，控制边目前只是控制算子的执行顺序。

函数原型

Operator& AddControlInput(const Operator& src_oprt)

参数说明

参数名	输入/输出	描述
src_oprt	输入	控制边对应的源算子。

返回值

参数名	类型	描述
-	Operator&	算子对象本身。

异常处理

无。

约束说明

无。

10.1.2.3.3 BreakConnect

函数功能

删除当前算子与前一个算子之间的所有连接关系，删除当前算子与下一个算子之间的所有连接关系。

函数原型

void BreakConnect() const

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.3.4 IsEmpty

函数功能

判断operator对象是否为空，空表示不可用。

函数原型

bool IsEmpty() const

参数说明

无。

返回值

参数名	类型	描述
-	const	• True：非空。 • False：为空。

异常处理

无。

约束说明

无。

10.1.2.3.5 InferShapeAndType

函数功能

推导Operator输出的shape和DataType。

关于DataType数据类型的定义，请参见**10.1.2.10.2 DataType**。

函数原型

graphStatus InferShapeAndType()

参数说明

无。

返回值

参数名	类型	描述
-	graphStatus	推导成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.6 GetAttr

函数功能

根据属性名称获取对应的属性值。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

```
graphStatus GetAttr(const string& name, int64_t& attr_value) const;
graphStatus GetAttr(const char *name, int64_t &attr_value) const;
graphStatus GetAttr(const string& name, int32_t& attr_value) const;
graphStatus GetAttr(const char *name, int32_t &attr_value) const;
graphStatus GetAttr(const string& name, uint32_t& attr_value) const;
graphStatus GetAttr(const char *name, uint32_t &attr_value) const;
graphStatus GetAttr(const string& name, std::vector<int64_t>& attr_value) const;
graphStatus GetAttr(const char *name, std::vector<int64_t> &attr_value) const;
graphStatus GetAttr(const string& name, std::vector<int32_t>& attr_value) const;
```


graphStatus GetAttr(const char *name, std::vector<int32_t> &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<uint32_t>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetAttr(const string& name, float& attr_value) const;

graphStatus GetAttr(const char *name, float &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<float>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<float> &attr_value) const;

graphStatus GetAttr(const string& name, AttrValue& attr_value) const;

graphStatus GetAttr(const char *name, AttrValue &attr_value) const;

graphStatus GetAttr(const string& name, string& attr_value) const;

graphStatus GetAttr(const char *name, AscendString &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<string>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<AscendString> &attr_values) const;

graphStatus GetAttr(const string& name, bool& attr_value) const;

graphStatus GetAttr(const char *name, bool &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<bool>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<bool> &attr_value) const;

graphStatus GetAttr(const string& name, Tensor& attr_value) const;

graphStatus GetAttr(const char *name, Tensor &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<Tensor>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<Tensor>& attr_value) const;

graphStatus GetAttr(const string& name, OpBytes& attr_value) const;

graphStatus GetAttr(const char *name, OpBytes &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<std::vector<int64_t>>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<std::vector<int64_t>> &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<ge::DataType>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<ge::DataType> &attr_value) const;

graphStatus GetAttr(const string& name, ge::DataType& attr_value) const;

graphStatus GetAttr(const char *name, ge::DataType &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<ge::NamedAttrs>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<ge::NamedAttrs> &attr_value) const;

graphStatus GetAttr(const string& name, ge::NamedAttrs& attr_value) const;

graphStatus GetAttr(const char *name, ge::NamedAttrs &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
attr_value	输出	返回的int64_t表示的整型类型属性值。
attr_value	输出	返回的int32_t表示的整型类型属性值。
attr_value	输出	返回的uint32_t表示的整型类型属性值。
attr_value	输出	返回的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	返回的浮点类型的属性值。
attr_value	输出	返回的浮点列表类型的属性值。
attr_value	输出	返回的AttrValue类型的属性值。
attr_value	输出	返回的布尔类型的属性值。
attr_value	输出	返回的布尔列表类型的属性值。
attr_value	输出	返回的字符串类型的属性值。
attr_value	输出	返回的字符串列表类型的属性值。
attr_value	输出	返回的Tensor类型的属性值。
attr_value	输出	返回的Tensor列表类型的属性值。
attr_value	输出	返回的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。

参数名	输入/输出	描述
attr_value	输出	返回的量化数据的属性值。
attr_value	输出	返回的vector<vector<int64_t>>表示的整型二维列表类型属性值。
attr_value	输出	返回的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输出	返回的DataType类型的属性值。
attr_value	输出	返回的vector<ge::NamedAttrs>表示的NamedAttrs列表类型属性值。
attr_value	输出	返回的NamedAttrs类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	找到对应name，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.7 GetAllAttrNamesAndTypes

函数功能

获取该算子所有的属性名称和属性类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::map<std::string, std::string> GetAllAttrNamesAndTypes() const

graphStatus GetAllAttrNamesAndTypes(std::map<ge::AscendString, ge::AscendString> &attr_name_types) const

参数说明

参数名	输入/输出	描述
attr_name_types	输出	所有的属性名称和属性类型。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

10.1.2.3.8 GetDynamicInputNum

函数功能

获取算子的动态Input的实际个数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

int GetDynamicInputNum(const string& name) const

int GetDynamicInputNum(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	算子的动态Input名。

返回值

参数名	类型	描述
-	int	实际动态Input的个数。 当name非法，或者算子无动态Input时，返回-1。

约束说明

无。

10.1.2.3.9 GetDynamicInputDesc

函数功能

根据name和index的组合获取算子动态Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetDynamicInputDesc(const string& name,uint32_t index) const

TensorDesc GetDynamicInputDesc(const char *name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子动态Input的名称。
index	输入	算子动态Input编号，编号从1开始。

返回值

参数名	类型	描述
-	TensorDesc	获取TensorDesc成功，则返回算子动态Input的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

10.1.2.3.10 GetDynamicOutputNum

函数功能

获取算子的动态Output的实际个数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

int GetDynamicOutputNum(const string& name) const

int GetDynamicOutputNum(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	算子的动态Output名。

返回值

参数名	类型	描述
-	int	实际动态Output的个数。 当name非法，或者算子无动态Output时，返回0。

约束说明

无。

10.1.2.3.11 GetDynamicOutputDesc

函数功能

根据name和index的组合获取算子动态Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetDynamicOutputDesc(const char *name, uint32_t index) const

TensorDesc GetDynamicOutputDesc(const string& name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子动态Output的名称。
index	输入	算子动态Output编号，编号从1开始。

返回值

参数名	类型	描述
-	TensorDesc	获取TensorDesc成功，则返回算子动态Output的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

10.1.2.3.12 GetDynamicSubgraph

函数功能

根据子图名称和子图索引获取算子对应的动态输入子图。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Graph GetDynamicSubgraph(const string &name, uint32_t index) const

Graph GetDynamicSubgraph(const char *name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	子图名。
index	输入	同名子图的索引。

返回值

Graph对象。

异常处理

无。

约束说明

无。

10.1.2.3.13 GetDynamicSubgraphBuilder

函数功能

根据子图名称和子图索引获取算子对应的动态输入子图的构建函数对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

SubgraphBuilder GetDynamicSubgraphBuilder(const string &name, uint32_t index) const

SubgraphBuilder GetDynamicSubgraphBuilder(const char *name, uint32_t index) const

参数说明

参数	输入/输出	描述
name	输入	子图名。
index	输出	同名子图的索引。

返回值

SubgraphBuilder对象。

异常处理

无。

约束说明

无。

10.1.2.3.14 GetInferenceContext

函数功能

获取当前算子传递infershape推导所需要的关联信息，比如前面算子的shape和DataType信息。

函数原型

InferenceContextPtr GetInferenceContext() const

参数说明

无。

返回值

参数名	类型	描述
-	InferenceContextPtr	返回当前operator的推理上下文。 InferenceContextPtr是指向InferenceContext类的指针的别名： `using InferenceContextPtr = std::shared_ptr<InferenceContext>;`

异常处理

无。

约束说明

无。

10.1.2.3.15 GetInputConstData

函数功能

如果指定算子Input对应的节点为Const节点，可调用该接口获取Const节点的数据。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus GetInputConstData(const string& dst_name, Tensor& data) const

graphStatus GetInputConstData(const char *dst_name, Tensor &data) const

参数说明

参数名	输入/输出	描述
dst_name	输入	输入名称。
data	输出	返回Const节点的数据Tensor。

返回值

参数名	类型	描述
-	graphStatus	如果指定算子Input对应的节点为Const节点且获取数据成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.16 GetInputsSize

函数功能

获取当前算子Input个数。

函数原型

size_t GetInputsSize() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回当前算子Input个数。

异常处理

无。

约束说明

无。

10.1.2.3.17 GetInputDesc

函数功能

根据算子Input名称或Input索引获取算子Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetInputDesc(const string& name) const;

TensorDesc GetInputDescByName(const char *name) const;

TensorDesc GetInputDesc(uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子Input名称。 当无此算子Input名称时，则返回TensorDesc默认构造的对象，其中，主要设置10.1.2.10.2 DataType为DT_FLOAT（表示float类型），10.1.2.10.1 Format为FORMAT_NCHW（表示NCHW）。
index	输入	算子Input索引。 当无此算子Input索引时，则返回TensorDesc默认构造的对象，其中，主要设置10.1.2.10.2 DataType为DT_FLOAT（表示float类型），10.1.2.10.1 FormatFORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述
-	TensorDesc	算子Input的TensorDesc。

异常处理

无。

约束说明

无。

10.1.2.3.18 GetName

函数功能

获取算子名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetName() const

graphStatus GetName(ge::AscendString &name) const

参数说明

参数名	输入/输出	描述
name	输出	算子名称。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

10.1.2.3.19 GetSubgraph

函数功能

根据子图名称获取算子对应的子图。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Graph GetSubgraph(const string &name) const

Graph GetSubgraph(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	子图名称。

返回值

Graph对象。

异常处理

无。

约束说明

无。

10.1.2.3.20 GetSubgraphBuilder

函数功能

根据子图名称获取算子对应的子图构建的函数对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

SubgraphBuilder GetSubgraphBuilder(const string &name) const

SubgraphBuilder GetSubgraphBuilder(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	子图名称。

返回值

SubgraphBuilder对象。

异常处理

无。

约束说明

无。

10.1.2.3.21 GetSubgraphNamesCount

函数功能

获取一个算子的子图个数。

函数原型

size_t Operator::GetSubgraphNamesCount() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回当前算子子图个数。

异常处理

无。

约束说明

无。

10.1.2.3.22 GetSubgraphNames

函数功能

获取一个算子的子图名称列表。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::vector<std::string> GetSubgraphNames() const

graphStatus GetSubgraphNames(std::vector<ge::AscendString> &names) const

参数说明

参数名	输入/输出	描述
names	输出	获取一个算子的子图名称列表。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED: 失败。 GRAPH_SUCCESS: 成功。

异常处理

无。

约束说明

无。

10.1.2.3.23 GetOpType

函数功能

获取算子类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetOpType() const

graphStatus GetOpType(ge::AscendString &type) const

参数说明

参数名	输入/输出	描述
type	输出	算子类型。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

10.1.2.3.24 GetOutputDesc

函数功能

根据算子Output名称或Output索引获取算子Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetOutputDesc(const string& name) const

TensorDesc GetOutputDescByName(const char *name) const

TensorDesc GetOutputDesc(uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子Output名称。 当无此算子Output名称时，返回TensorDesc默认构造的对象，其中，主要设置10.1.2.10.2 DataType为DT_FLOAT（表示float类型），10.1.2.10.1 Format为FORMAT_NCHW（表示NCHW）。
index	输入	算子Output索引。 当无此算子Output索引时，则返回TensorDesc默认构造的对象，其中，主要设置10.1.2.10.2 DataType为DT_FLOAT（表示float类型），10.1.2.10.1 Format为FORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述
-	TensorDesc	算子Output的TensorDesc。

异常处理

无。

约束说明

无。

10.1.2.3.25 GetOutputsSize

函数功能

获取算子所有Output的个数。

函数原型

size_t GetOutputsSize() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回当前算子的Output个数。

约束说明

无。

10.1.2.3.26 SetAttr

函数功能

设置算子属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值

  使用SetAttr(const string& name, int64_t attrValue)设置属性值，以GetAttr(const string& name, int32_t& attrValue) 、GetAttr(const string& name, uint32_t& attrValue) 取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 字符串列表：std::vector<string>
- 布尔：bool
- 布尔列表：std::vector<bool>
- Tensor：Tensor
- Tensor列表：std::vector<Tensor>
- Bytes：字节数组，SetAttr接受通过OpBytes（即vector<uint8_t>），和（const uint8_t* data, size_t size）表示的字节数组
- 量化数据：UsrQuantizeFactorParams
- AttrValue类型
- 整型二维列表类型：std::vector<std::vector<int64_t>
- DataType列表类型：vector<ge::DataType>
- DataType类型：DataType
- NamedAttrs类型： NamedAttrs
- NamedAttrs列表类型： vector<<NamedAttrs>>

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator &SetAttr(const string& name, int64_t attr_value);

Operator &SetAttr(const char *name, int64_t attr_value);

Operator &SetAttr(const string& name, int32_t attr_value);

Operator &SetAttr(const char *name, int32_t attr_value);

Operator &SetAttr(const string& name, uint32_t attr_value);

Operator &SetAttr(const char *name, uint32_t attr_value);

Operator &SetAttr(const string& name, const std::vector<int64_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<int64_t> &attr_value);

Operator &SetAttr(const string& name, const std::vector<int32_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<int32_t> &attr_value);

Operator &SetAttr(const string& name, const std::vector<uint32_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<uint32_t> &attr_value);

Operator &SetAttr(const string& name, std::initializer_list<int64_t>&& attr_value);

Operator &SetAttr(const char *name, std::initializer_list<int64_t> &&attr_value);

Operator &SetAttr(const string& name, float attr_value);

Operator &SetAttr(const char *name, float attr_value);

Operator &SetAttr(const string& name, const std::vector<float>& attr_value);

Operator &SetAttr(const char *name, const std::vector<float> &attr_value);

Operator &SetAttr(const string& name, AttrValue&& attr_value);

Operator &SetAttr(const char *name, AttrValue &&attr_value);

Operator &SetAttr(const string& name, const string& attr_value);

Operator &SetAttr(const char *name, const char *attr_value);

Operator &SetAttr(const char *name, const AscendString &attr_value);

Operator &SetAttr(const string& name, const std::vector<string>& attr_value);

Operator &SetAttr(const char *name, const std::vector<AscendString> &attr_values);

Operator &SetAttr(const string& name, bool attr_value);

Operator &SetAttr(const char *name, bool attr_value);

Operator &SetAttr(const string& name, const std::vector<bool>& attr_value);

Operator &SetAttr(const char *name, const std::vector<bool> &attr_value);

Operator &SetAttr(const string& name, const Tensor& attr_value);

Operator &SetAttr(const char *name, const Tensor &attr_value);

Operator &SetAttr(const string& name, const std::vector<Tensor>& attr_value);

Operator &SetAttr(const char *name, const std::vector<Tensor> &attr_value);

Operator &SetAttr(const string& name, const OpBytes& attr_value);

Operator &SetAttr(const char *name, const OpBytes &attr_value);

Operator &SetAttr(const string& name, const std::vector<std::vector<int64_t>>& attr_value);

Operator &SetAttr(const char *name, const std::vector<std::vector<int64_t>> &attr_value);

Operator &SetAttr(const string& name, const std::vector<ge::DataType>& attr_value);

Operator &SetAttr(const char *name, const std::vector<ge::DataType> &attr_value);

Operator &SetAttr(const string& name, const ge::DataType& attr_value);

Operator &SetAttr(const char *name, const ge::DataType &attr_value);

Operator &SetAttr(const string& name, const ge::NamedAttrs &attr_value);

Operator &SetAttr(const char *name, const ge::NamedAttrs &attr_value);

Operator &SetAttr(const string& name, const std::vector<ge::NamedAttrs> &attr_value);

Operator &SetAttr(const char *name, const std::vector<ge::NamedAttrs> &attr_value);

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。

参数名↵	输入/输出↵	描述↵
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的std::initializer_list<int64_t>&&表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的AttrValue类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。
attr_value	输入	需设置的Tensor类型的属性值。
attr_value	输入	需设置的Tensor列表类型的属性值。
attr_value	输入	需设置的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。
data	输入	需设置的Bytes，即字节数组类型的属性值，指定了字节流的首地址。
size	输入	需设置的Bytes，即字节数组类型的属性值，指定了字节流的长度。
attr_value	输入	需设置的量化数据的属性值。
attr_value	输入	需设置的vector<vector<int64_t>>表示的整型二维列表类型属性值。
attr_value	输入	需设置的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输入	需设置的DataType类型的属性值。
attr_value	输入	需设置的NamedAttrs类型的属性值。
attr_value	输入	需设置的vector<ge::NamedAttrs>表示的NamedAttrs列表类型的属性值。

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

约束说明

无。

10.1.2.3.27 SetInput

函数功能

设置算子Input，即由哪个算子的输出连到本算子。

有如下几种SetInput方法：

如果指定srcOprt第0个Output为当前算子Input，使用第一个函数原型设置当前算子Input，不需要指定srcOprt的Output名称。

如果指定srcOprt的其它Output为当前算子Input，使用第二个函数原型设置当前算子Input，需要指定srcOprt的Output名称。

如果指定srcOprt的其它Output为当前算子Input，使用第三个函数原型设置当前算子Input，需要指定srcOprt的第index个Output。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator& SetInput(const string& dst_name, const Operator& src_oprt);

Operator &SetInput(const char *dst_name, const Operator &src_oprt);

Operator& SetInput(const string& dst_name, const Operator& src_oprt, const string &name)

Operator &SetInput(const char *dst_name, const Operator &src_oprt, const char *name);

Operator& SetInput(const string& dst_name, const Operator& src_oprt, uint32_t index)

Operator &SetInput(const char *dst_name, const Operator &src_oprt, uint32_t index);

参数说明

参数名	输入/输出	描述
dst_name	输入	当前算子Input名称。
src_oprt	输入	Input名称为dstName的输入算子对象。
name	输入	srcOprt的Output名称。
index	输入	srcOprt的第index个Output。

返回值

参数名	类型	描述
-	Operator&	当前调度者本身。

异常处理

无。

约束说明

无。

10.1.2.3.28 SetInferenceContext

函数功能

向当前算子传递infershape推导所需要的关联信息，比如前面算子的shape和DataType信息。

函数原型

void SetInferenceContext(const InferenceContextPtr &inference_context)

参数说明

参数名	输入/输出	描述
inference_context	输入	当前operator的推理上下文。 InferenceContextPtr是指向InferenceContext类的指针的别名： `using InferenceContextPtr = std::shared_ptr<InferenceContext>;`

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.3.29 SetInputAttr

函数功能

设置算子输入Tensor属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值

  以int64_t为例，使用:

  SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);

  SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);

  设置属性值，以:

  GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;

  GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const

  取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 布尔：bool
- 布尔列表：std::vector<bool>

函数原型

Operator &SetInputAttr(const int32_t index, const char_t *name, const char_t *attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const char_t *attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const AscendString &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const AscendString &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, int32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, int32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, uint32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, uint32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, bool attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, bool attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, float32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, float32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<float32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<float32_t> &attr_value);

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输入索引
dst_name	输入	输入边名称
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

约束说明

无。

10.1.2.3.30 SetOutputAttr

函数功能

设置算子输出Tensor属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值
  以int64_t为例，使用:
  SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);
  SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);
  设置属性值，以:
  GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;
  GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const
  取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 布尔：bool
- 布尔列表：std::vector<bool>

函数原型

Operator &SetOutputAttr(const int32_t index, const char_t *name, const char_t *attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const char_t *attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const AscendString &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const AscendString &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, int64_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, int32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, int32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, uint32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, uint32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, bool attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, bool attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, float32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, float32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<float32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<float32_t> &attr_value);

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输出索引
dst_name	输入	输出边名称
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

约束说明

无。

10.1.2.3.31 GetInputAttr

函数功能

根据属性名称获取算子输入Tensor对应的属性值。

函数原型

graphStatus GetInputAttr(const int32_t index, const char_t *name, AscendString &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, AscendString &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, int32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, int32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, uint32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, uint32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, bool &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, bool &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, float32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, float32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<float32_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<float32_t> &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输入索引。
dst_name	输入	输入边名称。
attr_value	输出	获取到的int64_t表示的整型类型属性值。
attr_value	输出	获取到的int32_t表示的整型类型属性值。
attr_value	输出	获取到的uint32_t表示的整型类型属性值。
attr_value	输出	获取到的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的浮点类型的属性值。
attr_value	输出	获取到的浮点列表类型的属性值。
attr_value	输出	获取到的布尔类型的属性值。
attr_value	输出	获取到的布尔列表类型的属性值。
attr_value	输出	获取到的字符串类型的属性值。
attr_value	输出	获取到的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	找到对应属性，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.32 GetOutputAttr

函数功能

根据属性名称获取算子输出Tensor对应的属性值。

函数原型

graphStatus GetOutputAttr(const int32_t index, const char_t *name, AscendString &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, AscendString &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, int32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, int32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, uint32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, uint32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, bool &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, bool &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, float32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, float32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<float32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<float32_t> &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输出索引。
dst_name	输入	输出边名称。
attr_value	输出	获取到的int64_t表示的整型类型属性值。
attr_value	输出	获取到的int32_t表示的整型类型属性值。
attr_value	输出	获取到的uint32_t表示的整型类型属性值。

参数名	输入/输出	描述
attr_value	输出	获取到的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的浮点类型的属性值。
attr_value	输出	获取到的浮点列表类型的属性值。
attr_value	输出	获取到的布尔类型的属性值。
attr_value	输出	获取到的布尔列表类型的属性值。
attr_value	输出	获取到的字符串类型的属性值。
attr_value	输出	获取到的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	找到对应属性，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.33 TryGetInputDesc

函数功能

根据算子Input名称获取算子Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus TryGetInputDesc(const string& name, TensorDesc& tensor_desc) const

graphStatus TryGetInputDesc(const char *name, TensorDesc &tensor_desc) const

参数说明

参数名	输入/输出	描述
name	输入	算子的Input名。
tensor_desc	输出	返回算子端口的当前设置格式，为TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	True：有此端口，获取TensorDesc成功。 False：无此端口，出参为空，获取TensorDesc失败。

异常处理

异常场景	说明
无对应name输入	返回False。

约束说明

无。

10.1.2.3.34 UpdateInputDesc

函数功能

根据算子Input名称更新Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateInputDesc(const string& name, const TensorDesc& tensor_desc);

graphStatus UpdateInputDesc(const char *name, const TensorDesc &tensor_desc);

参数说明

参数名	输入/输出	描述
name	输入	算子Input名称。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

异常场景	说明
无对应name 输入	函数提前结束，返回GRAPH_FAILED。

约束说明

无。

10.1.2.3.35 UpdateOutputDesc

函数功能

根据算子Output名称更新Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateOutputDesc(const string& name, const TensorDesc& tensor_desc)

graphStatus UpdateOutputDesc(const char *name, const TensorDesc &tensor_desc)

参数说明

参数名	输入/输出	描述
name	输入	算子Output名称。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.36 UpdateDynamicInputDesc

函数功能

根据name和index的组合更新算子动态Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateDynamicInputDesc(const string& name, uint32_t index, const TensorDesc& tensor_desc)

graphStatus UpdateDynamicInputDesc(const char *name, uint32_t index, const TensorDesc &tensor_desc)

参数说明

参数名	输入/输出	描述
name	输入	算子动态Input的名称。
index	输入	算子动态Input编号，编号从1开始。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新动态Input成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.37 UpdateDynamicOutputDesc

函数功能

根据name和index的组合更新算子动态Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateDynamicOutputDesc(const string& name, uint32_t index, const TensorDesc& tensor_desc)

graphStatus UpdateDynamicOutputDesc(const char *name, uint32_t index, const TensorDesc &tensor_desc);

参数说明

参数名	输入/输出	描述
name	输入	算子动态Output的名称。
index	输入	算子动态Output编号。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新动态Output成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.3.38 VerifyAllAttr

函数功能

根据disableCommonVerifier值，校验Operator中的属性是否有效，校验Operator的输入输出是否有效。

函数原型

graphStatus VerifyAllAttr(bool disable_common_verifier = false)

参数说明

参数名	输入/输出	描述
disable_common_verifier	输入	当false时，只校验属性有效性，当true时，增加校验Operator所有输入输出有效性。 默认值为false。

返回值

参数名	类型	描述
-	graphStatus	推导成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.4 Tensor 类

Tensor的拷贝构造和赋值操作均共享Tensor信息，Tensor复制需使用Clone方法。

10.1.2.4.1 Tensor 构造函数和析构函数

函数功能

Tensor构造函数和析构函数。

函数原型

```
Tensor();
explicit Tensor(const TensorDesc &tensorDesc);
Tensor(const TensorDesc &tensorDesc, const std::vector<uint8_t> &data);
Tensor(const TensorDesc &tensorDesc, const uint8_t *data, size_t size);
Tensor(TensorDesc &&tensorDesc, std::vector<uint8_t> &&data);
~Tensor();
```

参数说明

参数名	输入/输出	描述
tensorDesc	输入	TensorDesc对象，需设置的Tensor描述符。
data	输入	需设置的数据。
size	输入	数据的长度，单位为字节。

返回值

Tensor构造函数返回Tensor类型的对象。

异常处理

无。

约束说明

无。

10.1.2.4.2 Clone

函数功能

拷贝Tensor。

函数原型

Tensor Clone() const

参数说明

无。

返回值

参数名	类型	描述
-	Tensor	返回拷贝的Tensor对象。

异常处理

无。

约束说明

无。

10.1.2.4.3 IsValid

函数功能

判断Tensor对象是否有效。

若实际Tensor数据的大小与TensorDesc所描述的Tensor数据大小一致，则有效。

函数原型

graphStatus IsValid()

参数说明

无。

返回值

参数名	类型	描述
-	graphStatus	如果Tensor对象有效，则返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.4.4 GetData

函数功能

获取Tensor中的数据。

const uint8_t* GetData() const返回的数据不可修改，uint8_t* GetData()返回的数据可修改。

函数原型

const uint8_t* GetData() const;

uint8_t* GetData()

参数说明

无。

返回值

参数名	类型	描述
-	const Buffer	Tensor中所存放的数据。

异常处理

无。

约束说明

无。

10.1.2.4.5 GetTensorDesc

函数功能

获取Tensor的描述符。

函数原型

TensorDesc GetTensorDesc() const

参数说明

无。

返回值

参数名	类型	描述
-	TensorDesc	返回当前Tensor的描述符。

异常处理

无。

约束说明

修改返回的TensorDesc信息，不影响Tensor对象中已有的TensorDesc信息。

10.1.2.4.6 GetSize

函数功能

获取Tensor中的数据的大小。

函数原型

size_t GetSize() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	Tensor中存放的数据的大小，单位为字节。

异常处理

无。

约束说明

无。

10.1.2.4.7 SetData

函数功能

向Tensor中设置数据。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus SetData(std::vector<uint8_t> &&data);

graphStatus SetData(const std::vector<uint8_t> &data);

graphStatus SetData(const uint8_t *data, size_t size);

graphStatus SetData(const std::string &data);

graphStatus SetData(const char *data);

graphStatus SetData(const std::vector<std::string> &data);

graphStatus SetData(const std::vector<ge::AscendString> &datas);

graphStatus SetData(uint8_t *data, size_t size, const Tensor::DeleteFunc &deleter_func);

参数说明

参数名	输入/输出	描述
data	输入	需设置的数据。
size	输入	数据的长度，单位为字节。

参数名	输入/输出	描述
deleter_func	输入	用于释放data数据。 using DeleteFunc = std::function<void(uint8_t *)>;

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.4.8 SetTensorDesc

函数功能

设置Tensor的描述符（TensorDesc）。

函数原型

graphStatus SetTensorDesc(const TensorDesc &tensorDesc)

参数说明

参数名	输入/输出	描述
tensorDesc	输入	需设置的Tensor描述符。

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.5 TensorDesc 类

TensorDesc的拷贝构造和赋值操作均为值拷贝，不共享TensorDesc信息。

TensorDesc的Move构造和Move赋值，会将原有TensorDesc信息移动到新的TensorDesc对象中。

10.1.2.5.1 TensorDesc 构造函数和析构函数

函数功能

TensorDesc构造函数和析构函数。

函数原型

TensorDesc();

explicit TensorDesc(Shape shape, Format format = FORMAT_ND, DataType dt = DT_FLOAT);

TensorDesc(const TensorDesc& desc);

TensorDesc(TensorDesc&& desc);

TensorDesc &operator=(const TensorDesc &desc);

TensorDesc &operator=(TensorDesc &&desc);

~TensorDesc()

参数说明

参数名	输入/输出	描述
shape	输入	Shape对象。
format	输入	Format对象，默认取值FORMAT_ND。 关于Format数据类型的定义，请参见**10.1.2.10.1 Format**。
dt	输入	DataType对象，默认取值DT_FLOAT。 关于DataType数据类型的定义，请参见**10.1.2.10.2 DataType**。

返回值

TensorDesc构造函数返回TensorDesc类型的对象。

异常处理

无。

约束说明

无。

10.1.2.5.2 GetDataType

函数功能

获取TensorDesc所描述Tensor的数据类型。

函数原型

DataType GetDataType() const

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	DataType	TensorDesc所描述的Tensor的数据类型。

异常处理

无。

约束说明

由于返回的DataType信息为值拷贝，因此修改返回的DataType信息，不影响TensorDesc中已有的DataType信息。

10.1.2.5.3 GetFormat

函数功能

获取TensorDesc所描述的Tensor的Format。

函数原型

Format GetFormat() const

参数说明

无。

返回值

参数名	类型	描述
-	Format	TensorDesc所描述的Tensor的format信息。

异常处理

无。

约束说明

由于返回的Format信息为值拷贝，因此修改返回的Format信息，不影响TensorDesc中已有的Format信息。

10.1.2.5.4 GetName

函数功能

获取TensorDesc所描述Tensor的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::string GetName() const;

graphStatus GetName(ge::AscendString &name);

参数说明

参数名	输入/输出	描述
name	输出	算子名称。

返回值

参数名	类型	描述
-	graphStatus	获取name成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.5.5 GetOriginFormat

函数功能

获取TensorDesc所描述Tensor的原始Format。

函数原型

Format GetOriginFormat() const

参数说明

无。

返回值

参数名	类型	描述
-	Format	TensorDesc所描述的Tensor的originFormat信息。 关于Format数据类型的定义，请参见Format。

异常处理

无。

约束说明

无。

10.1.2.5.6 GetOriginShape

函数功能

获取TensorDesc所描述Tensor的原始Shape。

函数原型

Shape GetOriginShape() const

参数说明

无。

返回值

参数名	类型	描述
-	Shape	TensorDesc描述的originShape。

异常处理

无。

约束说明

无。

10.1.2.5.7 GetPlacement

函数功能

获取Tensor中数据地址的类型。

函数原型

Placement GetPlacement() const

参数说明

无

返回值

参数名	输入/输出	描述
无	**Placement**	获取Tensor中数据地址的类型。

异常处理

无。

约束说明

无。

10.1.2.5.8 GetRealDimCnt

函数功能

获取TensorDesc所描述Tensor的实际维度个数。

函数原型

int64_t GetRealDimCnt() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	TensorDesc所描述的实际维度个数。

异常处理

无。

约束说明

无。

10.1.2.5.9 GetShape

函数功能

获取TensorDesc所描述Tensor的Shape。

函数原型

Shape GetShape() const

参数说明

无。

返回值

参数名	类型	描述
-	Shape	TensorDesc描述的shape。

异常处理

无。

约束说明

由于返回的Shape信息为值拷贝，因此修改返回的Shape信息，不影响TensorDesc中已有的Shape信息。

10.1.2.5.10 GetShapeRange

函数功能

获取设置的shape变化范围。

函数原型

graphStatus GetShapeRange(std::vector<std::pair<int64_t,int64_t>> &range) const

参数说明

参数名	输入/输出	描述
range	输出	设置过的shape变化范围。

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。若成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

约束说明

无。

10.1.2.5.11 GetSize

函数功能

获取TensorDesc所描述Tensor的数据大小。

函数原型

int64_t GetSize() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	TensorDesc所描述的Tensor的数据大小信息。

异常处理

无。

约束说明

无。

10.1.2.5.12 SetDataType

函数功能

向TensorDesc中设置Tensor的数据类型。

函数原型

void SetDataType(DataType dt)

参数说明

参数名	输入/输出	描述
dt	输入	需设置的TensorDesc所描述的Tensor的数据类型信息。 关于DataType类型，请参见**DataType**。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.13 SetFormat

函数功能

向TensorDesc中设置Tensor的Format。

函数原型

void SetFormat(Format format)

参数说明

参数名	输入/输出	描述
format	输入	需设置的format信息。 关于Format类型，请参见10.1.2.10.1 Format。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.14 SetName

函数功能

向TensorDesc中设置Tensor的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetName(const std::string &name)

void SetName(const char *name)

参数说明

参数名	输入/输出	描述
name	输入	需设置的Tensor的名称。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.15 SetOriginFormat

函数功能

向TensorDesc中设置Tensor的原始Format。

该Format是指原始网络模型的Format。

函数原型

void SetOriginFormat(Format originFormat)

参数说明

参数名	输入/输出	描述
originFormat	输入	需设置的原始Format信息。 关于Format数据类型的定义，请参见Format。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.16 SetOriginShape

函数功能

向TensorDesc中设置Tensor的原始Shape。

该Shape是指原始网络模型的Shape。

函数原型

void SetOriginShape(const Shape &originShape)

参数说明

参数名	输入/输出	描述
originShape	输入	向TensorDesc设置原始的originShape对象。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.17 SetPlacement

函数功能

设置Tensor中数据地址的类型。

函数原型

void SetPlacement(Placement placement)

参数说明

参数名	输入/输出	描述

placement	输入	Tensor数据中地址的种类。
		枚举值定义如下：
		enum Placement {
		kPlacementHost = 0, // host data addr
		kPlacementDevice = 1, // device data addr
		};

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.18 SetRealDimCnt

函数功能

向TensorDesc中设置Tensor的实际维度数目。

通过**GetShape**接口返回的Shape的维度可能存在补1的场景，因此可以通过该接口设置Shape的实际维度个数。

函数原型

void SetRealDimCnt(const int64_t realDimCnt)

参数说明

参数名	输入/输出	描述
realDimCnt	输入	需设置的TensorDesc的实际数据维度数目信息。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.19 SetSize

函数功能

向TensorDesc中设置Tensor的数据大小。

函数原型

void SetSize(int64_t size)

参数说明

参数名	输入/输出	描述
size	输入	需设置的Tensor的数据大小信息。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.20 SetShape

函数功能

向TensorDesc中设置Tensor的Shape。

函数原型

void SetShape(const Shape &shape)

参数说明

参数名	输入/输出	描述
shape	输入	需向TensorDesc设置的shape对象。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.5.21 SetShapeRange

函数功能

设置shape的变化范围。

函数原型

graphStatus SetShapeRange(const std::vector<std::pair<int64_t,int64_t>> &range)

参数说明

参数名	输入/输出	描述
range	输入	shape代表的变化范围。vector中的每一个元素为一个pair，pair的第一个值为该维度上的dim最小值，第二个值为该维度上dim的最大值。举例如下： 该tensor的shape为{1，1，-1，2}，第三个轴的最大值为100，则range可设置为{ {1,1},{1,1},{1,100},{2,2}}

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。若成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

约束说明

无。

10.1.2.5.22 SetUnknownDimNumShape

函数功能

设置tensor的shape为{-2}，用来表示tensor是完全未知的。

函数原型

graphStatus SetUnknownDimNumShape()

参数说明

无。

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。执行成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

约束说明

无。

10.1.2.5.23 Update

函数功能

更新TensorDesc的shape、format、datatype属性。

函数原型

void Update(const Shape &shape, Format format = FORMAT_ND, DataType dt = DT_FLOAT)

参数说明

参数名	输入/输出	描述
shape	输入	需刷新的shape对象。

参数名	输入/输出	描述
format	输入	需刷新的format对象，默认取值FORMAT_ND。
dt	输入	需刷新的datatype对象，默认取值DT_FLOAT。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.6 Shape 类

10.1.2.6.1 Shape 构造函数和析构函数

函数功能

Shape构造函数和析构函数。

函数原型

Shape();

~Shape();

explicit Shape(const std::vector<int64_t>& dims)

参数说明

参数名	输入/输出	描述
dims	输入	Shape的取值内容。 Shape表征张量数据的维度大小，用std::vector<int64_t>表征每一个维度的具体大小。

返回值

Shape构造函数返回Shape类型的对象。

异常处理

无。

约束说明

无。

10.1.2.6.2 GetDim

函数功能

获取Shape第idx维的长度。

函数原型

int64_t GetDim(size_t idx) const

参数说明

参数名	输入/输出	描述
idx	输入	维度索引，索引从0开始。

返回值

参数名	类型	描述
-	int64_t	第idx维的长度。

异常处理

无。

约束说明

无。

10.1.2.6.3 GetDims

函数功能

获取Shape所有维度组成的向量。

函数原型

std::vector<int64_t> GetDims() const

参数说明

无。

返回值

参数名	类型	描述
-	std::vector<int64_t>	Shape的所有维度组成的向量。

异常处理

无。

约束说明

无。

10.1.2.6.4 GetDimNum

函数功能

获取Shape的维度个数。

函数原型

size_t GetDimNum() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	Tensor Shape的维度个数。

异常处理

无。

约束说明

无。

10.1.2.6.5 GetShapeSize

函数功能

获取Shape中所有dim的累乘结果。

函数原型

int64_t GetShapeSize() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	返回所有dim的累乘结果。

异常处理

无。

约束说明

无。

10.1.2.6.6 SetDim

函数功能

将Shape中第idx维度的值设置为value。

函数原型

graphStatus SetDim(size_t idx, int64_t value)

参数说明

参数名	输入/输出	描述
idx	输入	Shape维度的索引，索引从0开始。
value	输入	需设置的值。

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

使用SetDim接口前，只能使用Shape(const std::vector<int64_t>& dims)构造shape对象。如果使用Shape()构造shape对象，使用SetDim接口将返回失败。

10.1.2.7 AttrValue 类

10.1.2.7.1 AttrValue 构造函数和析构函数

函数功能

AttrValue构造函数和析构函数。

函数原型

AttrValue();

~AttrValue()

参数说明

无。

返回值

AttrValue构造函数返回AttrValue类型的对象。

异常处理

无。

约束说明

无。

10.1.2.7.2 CreateFrom

函数功能

将传入的DT类型（支持int64_t、float、std::string类型）的参数转换为对应T类型（支持INT、FLOAT、STR类型）的参数。

- 支持将int64_t类型转换为INT类型
- 支持将float类型转换为FLOAT类型
- 支持将std::string类型转换为STR类型

函数原型

template<typename T, typename DT>

static T CreateFrom(DT&& val)

参数说明

参数名	输入/输出	描述
val	输入	DT类型的参数。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	• INT • FLOAT • STR	T类型的参数。

异常处理

无。

约束说明

无。

10.1.2.7.3 GetValue

函数功能

获取属性key-value键值对中的value值，并将value值从T类型转换为DT类型。

- 支持将INT类型转换为int64_t类型
- 支持将FLOAT类型转换为float类型

- 支持将STR类型转换为std::string类型

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

template<typename T, typename DT>

graphStatus GetValue(DT& val) const

graphStatus GetValue(ge::AscendString &val)

参数说明

参数名	输入/输出	描述
val	输出	DT类型的参数。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	数据类型转换成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

10.1.2.8 Memblock 类

10.1.2.8.1 MemBlock 构造函数和析构函数

函数功能

MemBlock构造函数和析构函数。

函数原型

MemBlock(Allocator &allocator, void *addr, size_t block_size)


```
: allocator_(allocator), addr_(addr), count_(1U), block_size_(block_size) {}
virtual ~MemBlock() = default;
```

参数说明

参数名	输入/输出	描述
allocator	输入	用户根据10.1.2.9 Allocator类派生的类的引用。
addr	输入	device内存地址。
block_size	输入	device内存addr的大小。

返回值

MemBlock构造函数返回MemBlock类型的对象。

异常处理

无。

约束说明

用户继承10.1.2.9 Allocator类后，申请内存需要返回MemBlock类型指针，用户只需按构造函数构造MemBlock对象即可，析构函数根据用户需求可以自定义，避免内存泄露。

10.1.2.8.2 GetAddr

函数功能

获取只读的device内存地址。

函数原型

```
const void *GetAddr() const
```

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	void*	只读的device内存地址。

异常处理

无。

约束说明

无。

10.1.2.8.3 GetAddr

函数功能

获取可读写的device内存地址。

函数原型

void *GetAddr()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	void*	可读写的device内存地址。

异常处理

无。

约束说明

无。

10.1.2.8.4 GetSize

函数功能

获取device内存对应的大小。

函数原型

size_t GetSize() const

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	device内存大小。

异常处理

无。

约束说明

无。

10.1.2.8.5 SetSize

函数功能

设置device内存地址大小。

函数原型

void SetSize(const size_t mem_size)

参数说明

参数名	输入/输出	描述
mem_size	输入	device内存大小。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.8.6 Free

函数功能

MemBlock的引用计数减为0时，释放MemBlock到内存池。

函数原型

void Free()

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无。

10.1.2.8.7 AddCount

函数功能

MemBlock引用计数加1。

函数原型

size_t AddCount()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回加1后的引用计数。

异常处理

无。

约束说明

无。

10.1.2.8.8 SubCount

函数功能

MemBlock引用计数减1。

函数原型

size_t SubCount()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回减1后的引用计数。

异常处理

无。

约束说明

无。

10.1.2.8.9 GetCount

函数功能

获取MemBlock的引用计数。

函数原型

size_t GetCount() const

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回引用计数。

异常处理

无。

约束说明

无。

10.1.2.9 Allocator 类

10.1.2.9.1 Allocator 构造函数和析构函数

函数功能

Allocator构造函数和析构函数。

函数原型

```
Allocator() = default;
virtual~Allocator() = default;
```

参数说明

无。

返回值

无。

异常处理

无。

约束说明

纯虚类需要用户派生。

10.1.2.9.2 Malloc

函数功能

在用户内存池中根据指定size大小申请device内存。

函数原型

```
virtual MemBlock *Malloc(size_t size) = 0;
```


参数说明

参数名	输入/输出	描述
size	输入	指定需要申请内存大小。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	MemBlock*	返回10.1.2.8 Memblock类指针。

异常处理

无。

约束说明

纯虚函数用户必须实现。

10.1.2.9.3 Free

函数功能

根据指定的MemBlock释放内存到内存池。

函数原型

virtual void Free(MemBlock *block) = 0;

参数说明

参数名	输入/输出	描述
block	输入	内存block指针。

返回值

无。

异常处理

无。

约束说明

纯虚函数用户必须实现。

10.1.2.9.4 MallocAdvise

函数功能

在用户内存池中根据指定size大小申请device内存，建议申请的内存地址为addr。

函数原型

virtual MemBlock *MallocAdvise(size_t size, void *addr)

参数说明

参数名	输入/输出	描述
size	输入	指定需要申请内存大小。
addr	输入	建议申请的内存地址为addr。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	MemBlock*	返回10.1.2.8 Memblock类指针。

异常处理

无。

约束说明

虚函数需要用户实现，如若未实现，默认同10.1.2.9.2 Malloc功能相同。

10.1.2.10 数据类型

10.1.2.10.1 Format

```
enum Format {
  FORMAT_NCHW = 0,   // NCHW
  FORMAT_NHWC,       // NHWC
  FORMAT_ND,         // Nd Tensor
  FORMAT_NC1HWC0,    // NC1HWC0
  FORMAT_FRACTAL_Z,  // FRACTAL_Z
  FORMAT_NC1C0HWPAD,
  FORMAT_NHWC1C0,
  FORMAT_FSR_NCHW,
  FORMAT_FRACTAL_DECONV,
  FORMAT_C1HWNC0,
  FORMAT_FRACTAL_DECONV_TRANSPOSE,
  FORMAT_FRACTAL_DECONV_SP_STRIDE_TRANS,
  FORMAT_NC1HWC0_C04,    // NC1HWC0, C0 =4
  FORMAT_FRACTAL_Z_C04,  // FRACZ, C0 =4
  FORMAT_CHWN,
```



```
  FORMAT_FRACTAL_DECONV_SP_STRIDE8_TRANS,
  FORMAT_HWCN,
  FORMAT_NC1KHKWHWC0,  // KH,KW kernel h& kernel w maxpooling max output format
  FORMAT_BN_WEIGHT,
  FORMAT_FILTER_HWCK,  // filter input tensor format
  FORMAT_HASHTABLE_LOOKUP_LOOKUPS = 20,
  FORMAT_HASHTABLE_LOOKUP_KEYS,
  FORMAT_HASHTABLE_LOOKUP_VALUE,
  FORMAT_HASHTABLE_LOOKUP_OUTPUT,
  FORMAT_HASHTABLE_LOOKUP_HITS = 24,
  FORMAT_C1HWNCoC0,
  FORMAT_MD,
  FORMAT_NDHWC,
  FORMAT_FRACTAL_ZZ,
  FORMAT_FRACTAL_NZ,
  FORMAT_NCDHW,
  FORMAT_DHWCN,  // 3D filter input tensor format
  FORMAT_NDC1HWC0,
  FORMAT_FRACTAL_Z_3D,
  FORMAT_CN,
  FORMAT_NC,
  FORMAT_DHWNC,
  FORMAT_FRACTAL_Z_3D_TRANSPOSE, // 3D filter(transpose) input tensor format
  FORMAT_FRACTAL_ZN_LSTM,
  FORMAT_FRACTAL_Z_G,
  FORMAT_RESERVED,
  FORMAT_ALL,
  FORMAT_NULL,
  FORMAT_ND_RNN_BIAS,
  FORMAT_FRACTAL_ZN_RNN,
  FORMAT_NYUV,
  FORMAT_NYUV_A,
  FORMAT_END,
  FORMAT_MAX = 0xff
};
```

IR构图不支持输入以下FORMAT：

```
NC1HWC0
FRACTAL_Z
NC1C0HWPAD
NHWC1C0
FRACTAL_DECONV
C1HWNC0
FRACTAL_DECONV_TRANSPOSE
FRACTAL_DECONV_SP_STRIDE_TRANS
NC1HWC0_C04
FRACTAL_Z_C04
FRACTAL_DECONV_SP_STRIDE8_TRANS
NC1KHKWHWC0
C1HWNCoC0
FRACTAL_ZZ
FRACTAL_NZ
NDC1HWC0
FORMAT_FRACTAL_Z_3D
FORMAT_FRACTAL_Z_3D_TRANSPOSE
FORMAT_FRACTAL_ZN_LSTM
FORMAT_FRACTAL_Z_G
FORMAT_ND_RNN_BIAS
FORMAT_FRACTAL_ZN_RNN
FORMAT_NYUV
FORMAT_NYUV_A
```

10.1.2.10.2 DataType

```
  enum DataType {
    DT_FLOAT = 0,          // float type
    DT_FLOAT16 = 1,        // fp16 type
    DT_INT8 = 2,           // int8 type
```



```
    DT_INT16 = 6,     // int16 type
    DT_UINT16 = 7,      // uint16 type
    DT_UINT8 = 4,           // uint8 type
    DT_INT32 = 3,           //
    DT_INT64 = 9,           // int64 type
    DT_UINT32 = 8,          // unsigned int32
    DT_UINT64 = 10,          // unsigned int64
    DT_BOOL = 12,           // bool type
    DT_DOUBLE = 11,          // double type
    DT_STRING = 13,           // string type
    DT_DUAL_SUB_INT8 = 14,    /**< dual output int8 type */
    DT_DUAL_SUB_UINT8 = 15,    /**< dual output uint8 type */
    DT_COMPLEX64 = 16,         // complex64 type
    DT_COMPLEX128 = 17,        // complex128 type
    DT_QINT8 = 18,             // qint8 type
    DT_QINT16 = 19,            // qint16 type
    DT_QINT32 = 20,            // qint32 type
    DT_QUINT8 = 21,            // quint8 type
    DT_QUINT16 = 22,           // quint16 type
    DT_RESOURCE = 23,          // resource type
    DT_STRING_REF = 24,        // string ref type
    DT_DUAL = 25,              // dual output type
    DT_VARIANT = 26,           // dt_variant type
    DT_BF16 = 27,              // bf16 type,the current version does not support this enumerated value.
    DT_UNDEFINED = 28,         // Used to indicate a DataType field has not been set.
    DT_INT4 = 29,              // int4 type
    DT_MAX                     // Mark the boundaries of data types
  };
```

10.1.2.10.3 TensorType

TensorType类用以定义输入或者输出支持的数据类型，TensorType提供以下接口指定支持的数据类型：

```
struct TensorType {
  explicit TensorType(DataType dt);

  TensorType(const std::initializer_list<DataType> &types);

  static TensorType ALL() {
    return TensorType{DT_BOOL,   DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,
DT_FLOAT16, DT_INT16,
                      DT_INT32,  DT_INT64,      DT_INT8,      DT_QINT16, DT_QINT32, DT_QINT8,   DT_QUINT16,
                      DT_QUINT8, DT_RESOURCE,   DT_STRING,    DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8};
  }

  static TensorType QuantifiedType() { return TensorType{DT_QINT16, DT_QINT32, DT_QINT8, DT_QUINT16,
DT_QUINT8}; }

  static TensorType OrdinaryType() {
    return TensorType{DT_BOOL,  DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16,
DT_INT16,
                      DT_INT32, DT_INT64,      DT_INT8,      DT_UINT16, DT_UINT32, DT_UINT64,  DT_UINT8};
  }

  static TensorType BasicType() {
    return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,
                      DT_INT32,      DT_INT64,     DT_INT8,   DT_QINT16, DT_QINT32,  DT_QINT8,
                      DT_QUINT16,    DT_QUINT8,    DT_UINT16, DT_UINT32, DT_UINT64,  DT_UINT8};
  }

  static TensorType NumberType() {
    return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,
DT_INT32,  DT_INT64,
                      DT_INT8,       DT_QINT32,    DT_QINT8,  DT_QUINT8, DT_UINT16,  DT_UINT32, DT_UINT64,
DT_UINT8};
  }
```

```
  static TensorType RealNumberType() {
    return TensorType{DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,  DT_INT32, DT_INT64,
                      DT_INT8,   DT_UINT16, DT_UINT32,  DT_UINT64, DT_UINT8};
  }

  static TensorType ComplexDataType() { return TensorType{DT_COMPLEX128, DT_COMPLEX64}; }

  static TensorType IntegerDataType() {
    return TensorType{DT_INT16, DT_INT32, DT_INT64, DT_INT8, DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8};
  }

  static TensorType SignedDataType() { return TensorType{DT_INT16, DT_INT32, DT_INT64, DT_INT8}; }

  static TensorType UnsignedDataType() { return TensorType{DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8}; }

  static TensorType FloatingDataType() { return TensorType{DT_DOUBLE, DT_FLOAT, DT_FLOAT16}; }

  static TensorType IndexNumberType() { return TensorType{DT_INT32, DT_INT64}; }

  static TensorType UnaryDataType() { return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE,
DT_FLOAT, DT_FLOAT16}; }

  static TensorType FLOAT() { return TensorType{DT_FLOAT, DT_FLOAT16}; }

  std::shared_ptr<TensorTypeImpl> tensor_type_impl_;
};
```

10.1.2.10.4 UsrQuantizeFactor

```
struct UsrQuantizeFactor
{
public:
    //QuantizeScaleMode scale_mode;
    UsrQuantizeScaleMode scale_mode{USR_NORMAL_MODE};
    std::vector<uint8_t> scale_value;
    int64_t scale_offset{0};
    std::vector<uint8_t> offset_data_value;
    int64_t offset_data_offset{0};
    std::vector<uint8_t> offset_weight_value;
    int64_t offset_weight_offset{0};
    std::vector<uint8_t> offset_pad_value;
    int64_t offset_pad_offset{0};

    USR_TYPE_DEC(UsrQuantizeScaleMode, scale_mode);
    USR_TYPE_BYTES_DEC(scale_value);


    USR_TYPE_DEC(int64_t, scale_offset);
    USR_TYPE_BYTES_DEC(offset_data_value);
    USR_TYPE_DEC(int64_t, offset_data_offset);


    USR_TYPE_BYTES_DEC(offset_weight_value);
    USR_TYPE_DEC(int64_t, offset_weight_offset);
    USR_TYPE_BYTES_DEC(offset_pad_value);
    USR_TYPE_DEC(int64_t, offset_pad_offset);


};
```

10.1.2.10.5 TensorDescInfo

```
  struct TensorDescInfo {
      Format format_ = FORMAT_RESERVED;       /* tbe op register support format */
      DataType dataType_ = DT_UNDEFINED;      /* tbe op register support datatype */
  };
```


Format为枚举类型，定义请参考**10.1.2.10.1 Format**。

DataType为枚举类型，定义请参考**10.1.2.10.2 DataType**。

10.1.2.10.6 GetSizeByDataType

函数功能

根据传入的data_type，获取该data_type所占用的内存大小。

函数原型

int GetSizeByDataType(DataType data_type)

参数说明

参数	输入/输出	说明
data_type	输入	数据类型，请参见**10.1.2.10.2 DataType**。

返回值

该data_type所占用的内存大小（单位为bytes），如果传入非法值或不支持的数据类型，返回-1。

异常处理

无。

约束说明

无。

10.1.2.10.7 GetFormatName

函数功能

根据传入的format类型，获取format的字符串描述。

函数原型

const char *GetFormatName(Format format)

参数说明

参数	输入/输出	说明
format	输入	format枚举值。

返回值

该format所对应的字符串描述，若format不合法或不被识别，则返回nullptr。

异常处理

无。

约束说明

返回的字符串不可被修改。

10.1.2.10.8 GetFormatFromSub

函数功能

根据传入的主format和子format信息得到实际的format。

实际format为4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息，如下：

```
/*
* ----------------------------------------------------
* | 4 bits | 4bits | 2 bytes | 1 byte |
* |------------|-------------|----------------|--------|
* | reserved | c0 -format | sub-format | format |
* ----------------------------------------------------
*/
```

函数原型

int32_t GetFormatFromSub(int32_t primary_format, int32_t sub_format)

参数说明

参数	输入/输出	说明
primary_format	输入	主format信息，值不超过0xff。
sub_format	输入	子format信息，值不超过0xffff。

返回值

指定的主format和子format对应的实际format。

异常处理

无。

约束说明

无。

10.1.2.10.9 GetPrimaryFormat

函数功能

从实际format中解析出主format信息。

函数原型

int32_t GetPrimaryFormat(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

实际format中包含的主format。

异常处理

无。

约束说明

无。

10.1.2.10.10 GetSubFormat

函数功能

从实际format中解析出子format信息。

函数原型

int32_t GetSubFormat(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat段，第2-3字节为子format信息，第4字节为主format信息）。

返回值

实际format中包含的子format。

异常处理

无。

约束说明

无。

10.1.2.10.11 HasSubFormat

函数功能

判断实际format中是否包含子format。

函数原型

bool HasSubFormat(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

true：实际format中包含子format；false：实际format中不包含子format。

异常处理

无。

约束说明

无。

10.1.2.10.12 HasC0Format

函数功能

判断实际format中是否包含C0 format。

函数原型

bool HasC0Format(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

- true：实际format中包含c0 format。
- false：实际format中不包含c0 format。

异常处理

无。

约束说明

无。

10.1.2.10.13 GetC0Value

函数功能

从实际format中解析出c0 format信息。

函数原型

int64_t GetC0Value(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

- 如果包含c0 format，返回实际format中包含的c0 format。
- 如果不包含c0 format，返回-1。

异常处理

无。

约束说明

设置实际format格式时，第一个字节低四位的c0 format的范围只支持x=(0001~1111)，实际获取的c0 value为2^x-1。

10.1.3 构图接口

10.1.3.1 简介

本文档主要描述给应用用户或者上层框架提供了安全易用的构图接口集合，开发者可以调用这些接口构建网络模型，设置模型所包含的图、图内的算子以及算子的属性信息（包括Input、Output及其他属性信息）。

您可以在CANN软件包安装路径查看对应接口的头文件。

接口分类	头文件路径
Parser解析接口	/include/parser/tensorflow_parser.h /include/parser/caffe_parser.h /include/parser/onnx_parser.h
Graph构建接口	/include/graph/graph.h
Graph修改接口	/include/graph/graph.h /include/graph/gnode.h
Graph编译接口	/include/ge/ge_ir_build.h
Graph运行接口	/include/ge/ge_api.h
维测接口	/include/ge/ge_prof.h

接口分类	头文件路径
其他接口	/include/ge/ge_ir_build.h /include/ge/ge_api_error_codes.h
数据类型	/include/ge/ge_api_types.h

10.1.3.2 Parser 解析接口

10.1.3.2.1 aclgrphParseTensorFlow

函数功能

将TensorFlow模型解析为图。

函数原型

graphStatus aclgrphParseTensorFlow(const char *model_file, ge::Graph &graph);

graphStatus aclgrphParseTensorFlow(const char *model_file, const std::map<ge::AscendString, ge::AscendString> &parser_params, ge::Graph &graph);

约束说明

无。

参数说明

参数名	输入/输出	描述
model_file	输入	TensorFlow原始模型文件路径。
parser_params	输入	配置参数map映射表，key为参数类型，value为参数值，均为AscendString格式，用于描述原始模型解析参数。 map中支持的配置参数请参见**10.1.3.11.4 Parser解析接口支持的配置参数**。
graph	输出	解析后生成的图。

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.2.2 aclgrphParseCaffe

函数功能

将Caffe模型解析为图。

函数原型

graphStatus aclgrphParseCaffe(const char *model_file, const char *weights_file, ge::Graph &graph);

graphStatus aclgrphParseCaffe(const char *model_file, const char *weights_file, const std::map<ge::AscendString, ge::AscendString> &parser_params, ge::Graph &graph);

约束说明

无。

参数说明

参数名	输入/输出	描述
model_file	输入	Caffe原始模型文件路径。
weights_file	输入	Caffe权重文件路径。
parser_params	输入	配置参数map映射表，key为参数类型，value为参数值，均为AscendString格式，用于描述原始模型解析参数。 map中支持的配置参数请参见**10.1.3.11.4 Parser解析接口支持的配置参数**。
graph	输出	解析后生成的图。

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.2.3 aclgrphParseONNX

函数功能

将ONNX模型解析为图。

函数原型

graphStatus aclgrphParseONNX(const char *model_file, const std::map<ge::AscendString, ge::AscendString> &parser_params, ge::Graph &graph);

约束说明

无。

参数说明

参数名	输入/输出	描述
model_file	输入	Onnx原始模型文件路径。
parser_params	输入	配置参数map映射表，key为参数类型，value为参数值，均为AscendString格式，用于描述原始模型解析参数。 map中支持的配置参数请参见**10.1.3.11.4 Parser解析接口支持的配置参数**。
graph	输出	解析后生成的图。

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.2.4 aclgrphParseONNXFromMem

函数功能

将加载至内存的ONNX模型解析为图。

函数原型

graphStatus aclgrphParseONNXFromMem(const char *buffer, size_t size, const std::map<ge::AscendString, ge::AscendString> &parser_params, ge::Graph &graph);

约束说明

无。

参数说明

参数名	输入/输出	描述
buffer	输入	onnx模型以二进制load到内存缓冲区的指针，尚未反序列化。
size	输入	内存缓冲区的大小。
parser_params	输入	配置参数map映射表，key为参数类型，value为参数值，均为AscendString格式，用于描述原始模型解析参数。 map中支持的配置参数请参见**10.1.3.11.4 Parser解析接口支持的配置参数**。
graph	输出	解析后生成的图。

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.3 Graph 构建接口

10.1.3.3.1 Graph 构造函数和析构函数

函数功能

Graph构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

```
explicit Graph(const std::string& name);
explicit Graph(const char *name);
Graph();
~Graph()
```


参数说明

参数名	输入/输出	描述
name	输入	Graph名称，按照指定的名称构造Graph。

返回值

Graph构造函数返回Graph类型的对象。

异常处理

无。

约束说明

无。

10.1.3.3.2 SetInputs

函数功能

设置Graph内的输入算子。

函数原型

Graph& SetInputs(const std::vector<Operator> &inputs)

参数说明

参数名	输入/输出	描述
inputs	输入	Graph内的输入算子。

返回值

参数名	类型	描述
-	Graph&	返回调用者本身。

约束说明

无。

10.1.3.3.3 SetOutputs

函数功能

设置Graph关联的输出算子。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Graph& SetOutputs(const std::vector<Operator>& outputs);

Graph& SetOutputs(const std::vector<std::pair<Operator, std::vector<size_t>>> &output_indexs);

Graph& SetOutputs(const std::vector<std::pair<ge::Operator, std::string> > &outputs);

Graph &SetOutputs(const std::vector<std::pair<ge::Operator, AscendString>> &outputs);

参数说明

参数名	输入/输出	描述
outputs	输入	输出节点列表，如果输出节点中有多个Output，则表示所有Output均返回结果，结果返回顺序按照此接口入参算子列表顺序，多个Output按照端口index从小到大排序。
output_indexs	输入	按照用户指定的算子及其Output（端口index描述，size_t类型）列表返回计算结果。顺序与此接口入参顺序一致。
outputs	输入	按照用户指定的算子及其Output（端口名字描述，string类型）列表返回计算结果。顺序与此接口入参顺序一致。

返回值

参数名	类型	描述
-	Graph&	返回调用者本身

异常处理

无。

约束说明

无。

10.1.3.3.4 SetTargets

函数功能

设置Graph的结束节点列表。

在该列表中的算子需要被执行到，但它的输出不用返回给用户

函数原型

Graph& SetTargets(const std::vector<Operator> &targets)

参数说明

参数名	输入/输出	描述
targets	输入	结束节点列表。

返回值

参数名	类型	描述
-	Graph&	返回调用者本身

异常处理

无。

约束说明

无。

10.1.3.3.5 IsValid

函数功能

判断Graph对象是否有效，即是否可被运行。

函数原型

bool IsValid() const

参数说明

无。

返回值

参数名	类型	描述
-	bool	• True，构建Graph对象有效，可被执行。 • False，构建Graph对象无效，不可被执行。

约束说明

无。

10.1.3.3.6 SetNeedIteration

函数功能

标记Graph是否需要循环执行。

函数原型

void SetNeedIteration(bool need_iteration)

参数说明

参数名	输入/输出	描述
need_iteration	输入	标记图是否需要循环执行。取值： • true • false

返回值

无。

异常处理

无。

约束说明

需要与npu_runconfig/iterations_per_loop、npu_runconfig/loop_cond、npu_runconfig/one、npu_runconfig/zero等搭配使用，用户需要先构造带有npu_runconfig/iterations_per_loop、npu_runconfig/loop_cond、npu_runconfig/one、npu_runconfig/zero名字的variable算子节点。

10.1.3.3.7 AddOp

函数功能

用户算子缓存接口，通过此接口可以将不带连接关系的算子缓存在图中，用于查询和对象获取。

函数原型

graphStatus AddOp(const ge::Operator& op)

参数说明

参数名	输入/输出	描述
op	输入	需增加的算子

返回值

参数名	类型	描述
-	graphStatus	SUCCESS：操作成功 FAILED： 操作失败

约束说明

此接口为非必须接口。主要用于用户在多个函数间切换时，operator对象可能因为是局部变量而无法被多个函数获取，此时operator可以注册在graph中，通过graph获取先前创建的operator对象，但注册到graph后，此operator对象除非graph销毁，否则一直存在。

10.1.3.3.8 FindOpByName

函数功能

基于算子名称，获取缓存在Graph中的op对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus FindOpByName(const string& name, ge::Operator& op) const;

graphStatus FindOpByName(const char *name, ge::Operator &op) const;

参数说明

参数名	输入/输出	描述
name	输入	需要获取的算子名称。
op	输出	返回用户所需要的op对象。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS： 成功获取算子实例。 FAILED：此名字没有在Grap中注册op对象。

约束说明

此接口为非必须接口，与10.1.3.3.7 AddOp搭配使用。

10.1.3.3.9 GetAllOpName

函数功能

获取Graph中已注册的所有缓存算子的名称列表。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus GetAllOpName(std::vector<string>& op_name) const;

graphStatus GetAllOpName(std::vector<AscendString> &names) const;

参数说明

参数名	输入/输出	描述
opName	输出	返回Graph中的所有算子的名称。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS：操作成功 FAILED： 操作失败

异常处理

无。

约束说明

此接口为非必须接口，与10.1.3.3.7 AddOp搭配使用。

10.1.3.3.10 SaveToFile

函数功能

将序列化后的Graph结构保存到文件中。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus SaveToFile(const string& file_name) const;

graphStatus SaveToFile(const char *file_name) const;

参数说明

参数名	输入/输出	描述
file_name	输入	需要保存的文件路径和文件名。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS：成功 FAILED： 失败

约束说明

无。

10.1.3.3.11 LoadFromFile

函数功能

从文件中读取Graph。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus LoadFromFile(const string& file_name);

graphStatus LoadFromFile(const char *file_name);

参数说明

参数名	输入/输出	描述
file_name	输入	文件路径和文件名。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS：成功 FAILED： 失败

异常处理

无。

约束说明

无。

10.1.3.3.12 FindOpByType

函数功能

基于算子类型，获取缓存在Graph中的所有指定类型的op对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus FindOpByType(const string &type, std::vector<ge::Operator> &ops) const;

graphStatus FindOpByType(const char *type, std::vector<ge::Operator> &ops) const;

参数说明

参数名	输入/输出	描述
type	输入	需要获取的算子类型。
ops	输出	返回用户所需要的op对象。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS： 成功获取算子实例。

10.1.3.3.13 GetName

函数功能

获取当前图的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::string &Graph::GetName() const;

graphStatus GetName(AscendString &name) const;

参数说明

参数名	输入/输出	描述
name	输出	需要获取的算子名字。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS： 成功获取算子名字。

10.1.3.3.14 CopyFrom

函数功能

拷贝源图生成目标图。目标图的graph_name优先使用用户指定的，未指定则使用原图的graph_name。graph_id/session归属，拷贝后需要用户自行添加。

函数原型

graphStatus Graph::CopyFrom(const Graph &src_graph)

约束说明

无。

参数说明

参数名	输入/输出	描述
src_graph	输入	待拷贝的源图

返回值

返回0表示成功，返回其它值表示失败。

10.1.3.4 Graph 修改接口

10.1.3.4.1 Graph 类

GetAllNodes

函数功能

获取图中的所有节点，包含本图包含子图的节点。

函数原型

std::vector<GNode> GetAllNodes() const;

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	std::vector<GNode>	图中的所有的节点，按照系统拓扑排序的结果顺序进行输出。

GetDirectNode

函数功能

获取本图中的所有节点，不包含本图内子图的节点。

函数原型

std::vector<GNode> GetDirectNode() const;

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	std::vector<GNode>	图中的所有的节点，按照系统拓扑排序的结果顺序进行输出。

RemoveNode

函数功能

删除图中的指定节点，并删除节点之间的连边。

函数原型

graphStatus RemoveNode(GNode &node);

graphStatus RemoveNode(GNode &node, bool contain_subgraph)

约束说明

无

参数说明

参数名	输入/输出	描述
node	输入	删除图中的指定节点，如果是子图中的节点，则需要指定contain_subgraph为true。
contain_subgraph	输入	指示图中是否包含子图

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

RemoveEdge

函数功能

删除图中的指定连接边。

函数原型

graphStatus RemoveEdge(GNode &src_node, const int32_t src_port_index, GNode &dst_node, const int32_t dst_port_index);

约束说明

无

参数说明

参数名	输入/输出	描述
src_node	输入	连接边的源节点
src_port_index	输入	源节点的输出端口号(-1表示控制边)
dst_node	输入	连接边的目的节点
dst_port_index	输入	目的节点的输入端口号（-1表示控制边）

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

AddNodeByOp

函数功能

新增一个节点到图上。

函数原型

GNode AddNodeByOp(const Operator &op);

约束说明

此时算子未连接边，该node未生效，需要显示补充连边关系。

参数说明

参数名	输入/输出	描述
op	输入	新增节点的原型op对象

返回值

参数名	类型	描述
-	GNode	返回新增的节点

AddDataEdge

函数功能

新增一条数据边。

函数原型

graphStatus AddDataEdge(GNode &src_node, const int32_t src_port_index, GNode &dst_node, const int32_t dst_port_index);

约束说明

无

参数说明

参数名	输入/输出	描述
src_node	输入	连接边的源节点
src_port_index	输入	源节点的输出端口号
dst_node	输入	连接边的目的节点
dst_port_index	输入	目的节点的输入端口号

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

AddControlEdge

函数功能

新增一个控制边。

函数原型

graphStatus AddControlEdge(GNode &src_node, GNode &dst_node);

约束说明

无

参数说明

参数名	输入/输出	描述
src_node	输入	连接边的源节点
dst_node	输入	连接边的目的节点

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

ConstructFromInputs

函数功能

支持基于用户构造的operator对象生成一个graph对象，功能与10.1.3.3.2 SetInputs一致。

SetInputs未来会逐步消亡，统一使用此接口。

函数原型

static GraphPtr ConstructFromInputs(const std::vector<Operator> &inputs, const ge::AscendString &name);

参数说明

参数名	输入/输出	描述
inputs	输入	整图输入的operator
name	输入	Graph的名字

返回值

参数名	类型	描述
-	GraphPtr	图指针，返回新构造的图

调用示例

```
GraphPtr graph;
graph =  Graph::ConstructFromInputs(inputs, graph_name);
graph->SetOutputs(outputs);
```


10.1.3.4.2 GNode 类

GNode 构造函数和析构函数

GNode();

~ GNode() = default;

GetType

函数功能

获取算子type。

函数原型

graphStatus GetType(ge::AscendString &type) const;

约束说明

无

参数说明

参数名	输入/输出	描述
type	输出	获取算子类型

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetName

函数功能

获取算子名字。

函数原型

graphStatus GetName(ge::AscendString &name) const;

约束说明

无

参数说明

参数名	输入/输出	描述
name	输出	获取算子名字

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetInDataNodesAndPortIndexs

函数功能

获取算子指定输入端口对应的对端算子以及对端算子的输出端口号。

函数原型

std::pair<NodePtr, int32_t > GetInDataNodesAndPortIndexs(const int32_t index) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	算子的输入端口号

返回值

参数名	类型	描述
-	std::pair<GNodePtr, int32_t >	对端输入算子以及对端输入算子的输出端口号，空表示没有对端算子

GetInControlNodes

函数功能

获取算子的控制输入节点。

函数原型

vector<GNodePtr> GetInControlNodes() const;

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	vector<GNodePtr>	算子的控制输入节点列表，空表示没有控制算子，算子的控制输入可能有多个

GetOutDataNodesAndPortIndexs

函数功能

获取算子指定输出端口对应的对端算子以及对端算子的输入端口号。

函数原型

vector<std::pair<GNodePtr, int32_t >> GetOutDataNodesAndPortIndexs(const int32_t index) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	算子的输出端口号

返回值

参数名	类型	描述
-	vector<std::pair<GNodePtr, int32_t>>	获取算子指定输出端口对应的对端算子以及对端算子的输入端口号

GetOutControlNodes

函数功能

获取算子的控制输出节点。

函数原型

vector<GNodePtr> GetOutControlNodes() const;

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	vector<GNodePtr>	算子的控制输出节点列表，空表示没有控制算子

GetInputConstData

函数功能

如果算子的输入是const节点，会返回const节点的值，否则返回失败。支持跨子图获取输入是否是const及const下的value。

函数原型

graphStatus GetInputConstData(const int32_t index, Tensor &data) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	算子的输入端口号
data	输出	输入const的值

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS表示成功，GRAPH_NO_CONST输入非const，其他表示失败

GetInputIndexByName

函数功能

通过算子的端口名获取输入端口index。

函数原型

graphStatus GetInputIndexByName(const ge::AscendString &name, int32_t &index)) const;

约束说明

无

参数说明

参数名	输入/输出	描述
name	输入	算子的端口名。对于动态输入，需要传入端口名和编号
index	输出	算子的输入端口号

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetOutputIndexByName

函数功能

通过算子的端口名获取输出端口index

函数原型

graphStatus GetOutputIndexByName(const ge::AscendString &name, int32_t &index);

约束说明

无

参数说明

参数名	输入/输出	描述
name	输入	算子的端口名
index	输出	算子的输出端口号

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetInputsSize

函数功能

返回节点的有效输入个数，即算子的实际输入个数。

函数原型

size_t GetInputsSize() const;

约束说明

无

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回输入个数

GetOutputsSize

函数功能

返回有效输出个数，即算子定义时的输出个数。

函数原型

size_t GetOutputsSize() const;

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	size_t	返回输出个数

GetInputDesc

函数功能

获取指定输入端口的tensor格式。

函数原型

graphStatus GetInputDesc(const int32_t index, TensorDesc &tensor_desc) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	输入的index
tensor_desc	输出	获取到的tensor格式

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

UpdateInputDesc

函数功能

更新指定输入端口的tensor格式。

函数原型

graphStatus UpdateInputDesc(const int32_t index, const TensorDesc &tensor_desc);

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	指定更新的输入端口
tensor_desc	输入	需要更新的tensor格式

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetOutputDesc

函数功能

获取指定输出端口的tensor格式

函数原型

graphStatus GetOutputDesc(const int32_t index, TensorDesc &tensor_desc) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	算子的输出端口号
tensor_desc	输出	获取到的tensor格式

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

UpdateOutputDesc

函数功能

更新指定输出端口的tensor格式。

函数原型

graphStatus UpdateOutputDesc(const int32_t index, const TensorDesc &tensor_desc)

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	算子的输出端口号
tensor_desc	输入	需要更新的tensor格式

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

SetAttr

函数功能

设置Node属性的属性值。

Node可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值

  使用SetAttr(const AscendString& name, int64_t attrValue)设置属性值，以GetAttr(const AscendString& name, int32_t& attrValue) 、GetAttr(const AscendString& name, uint32_t& attrValue) 取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：AscendString
- 字符串列表：std::vector<AscendString>
- 布尔：bool
- 布尔列表：std::vector<bool>
- Tensor：Tensor
- Tensor列表：std::vector<Tensor>
- OpBytes：字节数组，vector<uint8_t>
- AttrValue类型
- 整型二维列表类型：std::vector<std::vector<int64_t>
- DataType列表类型：vector<ge::DataType>
- DataType类型：DataType

函数原型

graphStatus SetAttr(const ge::AscendString &name, int64_t &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, int32_t &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, uint32_t &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, float &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, ge::AscendString &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, bool &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, Tensor &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<int64_t> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<int32_t> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<uint32_t> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<float> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<ge::AscendString> &attr_values) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<bool> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<Tensor> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, OpBytes &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<std::vector<int64_t>> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, std::vector<ge::DataType> &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, ge::DataType &attr_value) const;

graphStatus SetAttr(const ge::AscendString &name, AttrValue &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。
attr_value	输入	需设置的浮点类型float的属性值。
attr_value	输入	需设置的字符串类型AscendString的属性值。
attr_value	输入	需设置的布尔类型bool的属性值。
attr_value	输入	需设置的Tensor类型的属性值。
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的浮点列表类型vector<float>的属性值。
attr_value	输入	需设置的字符串列表类型vector<ge::AscendString>的属性值。
attr_value	输入	需设置的布尔列表类型vector<bool>的属性值。
attr_value	输入	需设置的Tensor列表类型vector<Tensor>的属性值。
attr_value	输入	需设置的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。
attr_value	输入	需设置的vector<vector<int64_t>>表示的整型二维列表类型属性值。
attr_value	输入	需设置的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输入	需设置的DataType类型的属性值。
attr_value	输入	需设置的AttrValue类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

约束说明

无。

GetAttr

函数功能

获取指定属性名字的值。

函数原型

graphStatus GetAttr(const ge::AscendString &name, int64_t &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, int32_t &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, uint32_t &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, float &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, ge::AscendString &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, bool &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, Tensor &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<int64_t> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<int32_t> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<uint32_t> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<float> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<ge::AscendString> &attr_values) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<bool> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<Tensor> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, OpBytes &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<std::vector<int64_t>> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, std::vector<ge::DataType> &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, ge::DataType &attr_value) const;

graphStatus GetAttr(const ge::AscendString &name, AttrValue &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称
attr_value	输出	返回的int64_t表示的整型类型属性值。
attr_value	输出	返回的int32_t表示的整型类型属性值。
attr_value	输出	返回的uint32_t表示的整型类型属性值。
attr_value	输出	返回的浮点类型float的属性值。
attr_value	输出	返回的字符串类型AscendString的属性值。
attr_value	输出	返回的布尔类型bool的属性值。
attr_value	输出	返回的Tensor类型的属性值。
attr_value	输出	返回的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	返回的浮点列表类型vector<float>的属性值。
attr_value	输出	返回的字符串列表类型vector<ge::AscendString>的属性值。
attr_value	输出	返回的布尔列表类型vector<bool>的属性值。
attr_value	输出	返回的Tensor列表类型vector<Tensor>的属性值。
attr_value	输出	返回的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。
attr_value	输出	返回的vector<vector<int64_t>>表示的整型二维列表类型属性值。

参数名	输入/输出	描述
attr_value	输出	返回的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输出	返回的DataType类型的属性值。
attr_value	输出	返回的AttrValue类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

约束说明

无。

HasAttr

函数功能

查询属性是否存在

函数原型

bool HasAttr(const ge::AscendString &name);

约束说明

无

参数说明

参数名	输入/输出	描述
name	输入	指定的属性名

返回值

参数名	类型	描述
-	bool	True表示存在，false表示不存在

GetSubgraph

函数功能

获取当前节点的第index个子图对象。

函数原型

graphStatus GetSubgraph(uint32_t index, graphPtr &graph) const;

约束说明

无

参数说明

参数名	输入/输出	描述
index	输入	子图的编号
graph	输出	子图的指针，空表示无对应子图

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

GetALLSubgraphs

函数功能

获取当前节点根图的所有子图对象。

函数原型

graphStatus GetALLSubgraphs(vector<graphPtr> &graph_list) const;

约束说明

无

参数说明

参数名	输入/输出	描述
graph_list	输出	子图的指针，空表示无对应子图

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

10.1.3.5 Graph 编译接口

10.1.3.5.1 aclgrphBuildInitialize

函数功能

模型构建的初始化函数，用于申请资源。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus aclgrphBuildInitialize(std::map<std::string, std::string> &global_options);

graphStatus aclgrphBuildInitialize(std::map<AscendString, AscendString> &global_options);

约束说明

一个进程内只能调用一次aclgrphBuildInitialize接口。

调用该接口后，可以在同一进程中连续调用多次aclgrphBuildModel接口。

对于aclgrphBuildModel和aclgrphBuildInitialize中重复的编译配置参数，建议不要重复配置，如果设置重复，则以aclgrphBuildModel传入的为准。

参数说明

参数名	输入/输出	描述
global_options	输入	配置参数map映射表，key为参数类型，value为参数值，均为字符串格式，用于描述离线模型编译初始化信息。 map中支持的配置参数请参见**10.1.3.11.2 aclgrphBuildInitialize支持的配置参数**。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS：成功。 其他值：失败。

10.1.3.5.2 aclgrphBuildModel

函数功能

将输入的Graph编译为适配昇腾AI处理器的离线模型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus aclgrphBuildModel(const ge::Graph &graph, const std::map<std::string, std::string> &build_options, ModelBufferData& model);

graphStatus aclgrphBuildModel(const ge::Graph &graph, const std::map<AscendString, AscendString> &build_options, ModelBufferData &model);

约束说明

- 对于aclgrphBuildModel和aclgrphBuildInitialize中重复的编译配置参数，建议不要重复配置，如果设置重复，则以aclgrphBuildModel传入的为准。

参数说明

参数名	输入/输出	描述
graph/graphs	输入	待编译的Graph。
build_options	输入	配置参数map映射表，key为参数类型，value为参数值，均为字符串格式，用于描述离线模型编译配置信息。 map中的配置参数请参见**10.1.3.11.3 aclgrphBuildModel支持的配置参数**。

参数名	输入/输出	描述

model	输出	编译生成的离线模型缓存。
		struct ModelBufferData
		{
		std::shared_ptr<uint8_t> data = nullptr;
		uint64_t length;
		};
		其中data指向生成的模型数据，length代表该模型的实际大小。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS：成功。 其他值：失败。

10.1.3.5.3 aclgrphSaveModel

函数功能

将离线模型序列化并保存到指定文件中。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus aclgrphSaveModel(const string &output_file, const ModelBufferData& model);

graphStatus aclgrphSaveModel(const char *output_file, const ModelBufferData &model);

约束说明

若生成的om模型文件名中含操作系统及架构，但操作系统及其架构与模型运行环境不一致时，需要与OPTION_HOST_ENV_OS、OPTION_HOST_ENV_CPU参数配合使用，设置模型运行环境的操作系统类型及架构。参数具体介绍请参见10.1.3.11.2 aclgrphBuildInitialize支持的配置参数。

参数说明

参数名	输入/输出	描述

model	输入	离线模型缓冲数据。
		struct ModelBufferData
		{
		std::shared_ptr<uint8_t> data = nullptr;
		uint32_t length;
		};
		其中data指向生成的模型数据，length代表该模型的实际大小。

output_file	输入	保存离线模型的文件名。生成的离线模型文件名会自动以.om后缀结尾，例如*ir_build_sample.om*或者 *ir_build_sample_linux_x86_64*.om，若om文件名中包含操作系统及架构，则该om文件只能在该操作系统及架构的运行环境中使用。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS：成功。 其他值：失败。

10.1.3.5.4 aclgrphBuildFinalize

函数功能

系统完成模型构建后，通过该接口释放资源。

函数原型

void aclgrphBuildFinalize()

约束说明

该接口在aclgrphBuildInitialize之后调用，且仅能在进程退出时调用一次。

参数说明

无。

返回值

无。

10.1.3.6 Graph 运行接口

10.1.3.6.1 GEInitialize

函数功能

初始化GE，完成运行准备。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Status GEInitialize(const std::map<std::string, std::string>& options)

Status GEInitialize(const std::map<AscendString, AscendString> &options)

参数说明

参数名	输入/输出	描述
options	输入	map表，key为参数类型，value为参数值，均为字符串格式，描述初始化相关的配置信息，当前支持的配置信息请参考表10-5。

返回值

参数名	类型	描述
-	Status	0: 初始化成功; 其他：初始化失败。

约束说明

- GE不支持多实例运行，一次只能初始化一个。
- 多次调用该接口而没有调用GEFinalize，运行不可预期。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.2 GEFinalize

函数功能

GE退出，释放GE相关资源。

在该接口内，默认增加2000ms延时（实际最大延时可达2000ms），用于Device业务日志回传，保证ERROR级别和EVENT级别日志不丢失，您可以将ASCEND_LOG_DEVICE_FLUSH_TIMEOUT环境变量设置为0（命令示例：export ASCEND_LOG_DEVICE_FLUSH_TIMEOUT=0），去除该默认延时。关于ASCEND_LOG_DEVICE_FLUSH_TIMEOUT环境变量的详细描述请参见《日志参考》。

函数原型

Status GEFinalize()

参数说明

无。

返回值

无。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.3 Session 构造函数和析构函数

函数功能

Session构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

explicit Session(const std::map<std::string, std::string> &options);

explicit Session(const std::map<AscendString, AscendString> &options);

~Session()

参数说明

参数名	输入/输出	描述
options	输入	map表，key为参数类型，value为参数值，均为字符串格式，描述Session的配置信息。 一般情况下可不填，与GEInitialize传入的全局option保持一致。 如需单独配置当前session的配置信息时，可以通过此参数配置，支持的配置项请参见**10.1.3.11.9 options参数说明**。

返回值

无。

约束说明

Session暂不支持并发执行，同时session会独占资源，多个session同时创建时，session可能因为资源不足而创建失败。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.4 AddGraph

函数功能

向Session中添加Graph，Session内会生成唯一的Graph ID。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Status AddGraph(uint32_t graphId, const Graph& graph)

Status AddGraph(uint32_t graphId, const Graph& graph, const std::map<std::string, std::string>& options)

Status AddGraph(uint32_t graphId, const Graph &graph, const std::map<AscendString, AscendString> &options)

参数说明

参数名	输入/输出	描述
graphId	输入	Graph对应的id。
graph	输入	需要加载到Session内的Graph。
options	输入	map表，key为参数类型，value为参数值，均为字符串格式，描述Graph配置信息。 一般情况下可不填，与GEInitialize传入的全局option保持一致。 如需单独配置当前Graph的配置信息时，可以通过此参数配置，支持的配置项请参见**10.1.3.11.9 options参数说明**。

返回值

参数名	类型	描述
-	Status	GE_CLI_GE_NOT_INITIALIZED：GE未初始化。 SUCCESS：图添加成功。 FAILED： 图添加失败。

约束说明

- 相同对象的graph调用此接口注册，会导致不同的graphId实际共享同一个graph对象，导致后续操作相互影响而出错。
- 不同的graph对象请不要使用相同的graphId来添加，该情况下，只保留第一次添加的graph对象，后续的graph对象不会添加成功。
- 使用该接口，session会直接修改添加的graph对象。如果AddGraph后需要保持原有的graph对象不受影响，应使用**AddGraphWithCopy**接口，**AddGraphWithCopy**会在session中拷贝一份graph对象，仅对graph对象的拷贝进行修改。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.5 AddGraphWithCopy

函数功能

向Session中添加Graph，Session内会生成唯一的Graph ID。

注意相比于AddGraph接口，此接口传入graph对象后，会产生graph对象的拷贝。session中保存的图是graph对象的一个备份，后续对该graph的修改不影响session内原有graph，同时session内的图的任何修改也不会影响graph对象。

函数原型

Status AddGraphWithCopy(uint32_t graphId, const Graph& graph)

Status AddGraphWithCopy(uint32_t graphId, const Graph& graph, const std::map<AscendString, AscendString>& options)

参数说明

参数名	输入/输出	描述
graphId	输入	Graph对应的id。
graph	输入	需要加载到Session内的Graph。
options	输入	map表，key为参数类型，value为参数值，均为字符串格式，描述Graph配置信息。 一般情况下可不填，与GEInitialize传入的全局option保持一致。 如需单独配置当前Graph的配置信息时，可以通过此参数配置，支持的配置项范围和GEInitialize一致。

返回值

参数名	类型	描述
-	Status	GE_CLI_GE_NOT_INITIALIZED：GE未初始化。 SUCCESS：图添加成功。 FAILED： 图添加失败。

约束说明

相同对象的graph调用此接口注册，会导致不同的graphId对应不同的备份，两个不同graphId对应的备份不共享。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.6 RemoveGraph

函数功能

在当前Session中删除指定ID对应的Graph。

函数原型

Status RemoveGraph(uint32_t graphId)

参数说明

参数名	输入/输出	描述
graphId	输入	要删除的Graph所对应的id。

返回值

参数名	类型	描述
-	Status	GE_CLI_SESS_REMOVE_FAILED：删除子图时序列化转换失败。 SUCCESS：删除子图成功。 FAILED：删除子图失败。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.7 RunGraph

函数功能

同步执行指定id对应的Graph图，输出执行结果。

函数原型

Status RunGraph(uint32_t graph_id, const std::vector<Tensor>& inputs, std::vector<Tensor>& outputs)

参数说明

参数名	输入/输出	描述
graph_id	输入	要执行图对应的id。
inputs	输入	计算图输入Tensor，为Host上分配的内存空间。
outputs	输出	计算图输出Tensor，用户无需分配内存空间，执行完成后GE会分配内存并赋值。

返回值

参数名	类型	描述
-	Status	GE_CLI_SESS_RUN_FAILED：执行子图时序列化转换失败。 SUCCESS：执行子图成功。 FAILED：执行子图失败。

约束说明

inputs与图中的data节点相对应，data节点的index属性表征inputs列表中对应数据的位置。即用户需要保证，可以按照data节点的index属性从inputs中获取对应的数据，否则返回错误。如果图中没有data节点，也可以输入的inputs为空。输出的outputs与用户指定的输出节点及输出端口个数与顺序相一致。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.8 BuildGraph

函数功能

同步编译指定id对应的Graph图，生成可用于执行的模型。

函数原型

Status BuildGraph(uint32_t graphId, const std::vector<InputTensorInfo> &inputs);

Status BuildGraph(uint32_t graphId, const std::vector<ge::Tensor> &inputs);

参数说明

参数名	输入/输出	描述
graphId	输入	子图对应的id。
inputs	输入	当前子图对应的输入数据。 InputTensorInfo结构体定义请参见**10.1.3.11.5 InputTensorInfo**。

返回值

参数名	类型	描述
-	Status	SUCCESS：编译子图成功。 FAILED：编译子图失败。

约束说明

无

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.9 RunGraphAsync

函数功能

异步执行指定id对应的Graph图，输出执行结果。

函数原型

Status RunGraphAsync(uint32_t graph_id, const std::vector<ge::Tensor> &inputs, RunAsyncCallback callback)

Status RunGraphAsync(uint32_t graph_id, const ContinuousTensorList &inputs, RunAsyncCallback callback)

带**ContinuousTensorList入参的原型接口，**仅在HS场景使用。

带**ContinuousTensorList入参的原型接口**，仅在昇腾310P AI处理器场景使用。

参数说明

参数名	输入/输出	描述
graphId	输入	子图对应的id。
inputs	输入	当前子图对应的输入数据。 昇腾310P昇腾AI处理器： 使用ContinuousTensorList类型的inputs，在非HS场景下，会转换成std::vector<Tensor>的输入来保证用户代码的兼容性。 std::vector<Tensor>和ContinuousTensorList类型的输入仅构造方式不同，建议使用std::vector<Tensor>类型的输入。

callback	输出	当前子图对应的回调函数。
		`using Status = uint32_t;`
		`using RunAsyncCallback = std::function<void(Status, std::vector<ge::Tensor> &)>;`

返回值

参数名	类型	描述
-	Status	GE_CLI_GE_NOT_INITIALIZED：GE未初始化。 SUCCESS：异步执行图成功。 FAILED：异步执行图失败。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

调用示例

此函数与RunGraph均为执行指定id对应的图，并输出结果，区别于10.1.3.6.7 RunGraph的是，该接口：

- 异步执行。
- 承载数据的数据格式不一样，但含义相同。
- 用户通过回调函数RunAsyncCallback获取图计算结果，即用户自行定义函数RunAsyncCallback。该回调函数，当status为success时，即可处理数据。

使用示例：

1. 用户自定义RunAsyncCallback，来决定如何处理数据, 例如：
```
void CallBack(Status result，std::vector<ge::Tensor> &out_tensor) {
if ( result == ge::SUCCESS) {
// 读取out_tensor数据， 用户根据需求处理数据；
for(auto &tensor : out_tensor) {
auto data = tensor.data;
int64_t length = tensor.length;
}
}
}
```
2. 定义好指定图的输入数据const std::vector<ge::Tensor> &inputs；
3. 调用接口RunGraphAsync(inputs, CallBack);

10.1.3.6.10 RunGraphWithStreamAsync

函数功能

异步执行指定id对应的Graph图，输出执行结果。

函数原型

Status RunGraphWithStreamAsync(uint32_t graph_id, void *stream, const std::vector<Tensor> &inputs,std::vector<Tensor> &outputs);

参数说明

参数名	输入/输出	描述
graphId	输入	子图对应的id。
stream	输入	指定图在哪个Stream上执行。
inputs	输入	当前子图对应的输入数据，为Device上的内存空间。
outputs	输出	当前子图对应的输出数据，为Device上的内存空间。

返回值

参数名	类型	描述
-	Status	GE_CLI_GE_NOT_INITIALIZED：GE未初始化。 SUCCESS：异步执行图成功。 FAILED：异步执行图失败。

约束说明

- 调用该接口前，需要确定好Device上分配的内存大小。
- 调用该接口前，需要通过aclrtCreateStream创建Stream。
- 得到输出执行结果前，需要通过aclrtSynchronizeStream接口保证Stream上的任务已经执行完。

  aclrtSynchronizeStream接口说明请参见《**应用软件开发指南（C&C++）**》>“AscendCL API参考”>同步等待。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

调用示例

此函数与RunGraph均为执行指定id对应的图，并输出结果，区别于**10.1.3.6.7 RunGraph**的是，该接口：

- 异步执行。
- inputs和outputs均为Device上的内存空间，且需要在执行前由用户分配内存大小。

10.1.3.6.11 RegisterCallBackFunc

函数功能

注册回调函数。

注册用户指定的summary、checkpoint回调接口。当用户下发给GE的图中带有summary、checkpoint算子时，GE会调用该回调函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Status RegisterCallBackFunc(const std::string &key, const pCallBackFunc &callback)

Status RegisterCallBackFunc(const char *key, const session::pCallBackFunc &callback)

参数说明

参数名	输入/输出	描述
key	输入	注册回调函数对应的关键字，字符串格式，表示回调函数类型，如Summary、Save。
callback	输入	回调函数返回的对应信息。 typedef uint32_t(*pCallBackFunc)(uint32_t graph_id, const std::map<std::string, ge::Tensor> &params_list); typedef uint32_t session::(*pCallBackFunc)(uint32_t graph_id, const std::map<AscendString, ge::Tensor> &params_list);

返回值

参数名	类型	描述
-	Status	GE_SESSION_MANAGER_NOT_INIT：session管理未初始化。 SUCCESS：注册回调函数成功。 FAILED：注册回调函数失败。

约束说明

- 回调函数类型仅支持Summary、Save。
- 如无注册则且下发summary、checkpoint算子会报错。
- 目前暂时只支持图执行完后一次性调用回调函数。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.12 IsGraphNeedRebuild

函数功能

图是否需要重新编译。

函数原型

bool IsGraphNeedRebuild(uint32_t graphId)

参数说明

参数名	输入/输出	描述
graphId	输入	子图对应的id。

返回值

参数名	类型	描述
-	bool	• False：图可以正常调用RunGraphAsync。 • True：需要RemoveGraph后，重新AddGraph和BuildGraph。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.13 GetVariables

函数功能

变量查询接口，获取Session内所有variable算子或指定variable算子的tensor内容。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Status GetVariables(const std::vector<std::string> &var_names, std::vector<Tensor> &var_values)

Status GetVariables(const std::vector<AscendString> &var_names, std::vector<Tensor> &var_values)

参数说明

参数名	输入/输出	描述
var_names	输入	variable算子的名字，为空时返回所有variable类型算子的值。
var_values	输出	variable算子的值，按照tensor格式返回；与var_names保持一致序列。

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 其他：不存在该name的variable算子；variable未执行初始化等场景

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.14 GetSessionId

函数功能

获取Session的ID。

函数原型

unit64_t GetSessionId() const;

参数说明

无。

返回值

参数名	类型	描述
-	unit64_t	Session的ID。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.15 CompileGraph

函数功能

同步编译指定id对应的Graph图。与**10.1.3.6.8 BuildGraph**相比，该接口仅包含图编译功能，不生成可用于执行的模型，BuildGraph包含了图编译过程，并在编译完成后进行模型所需内存资源的初始化，生成可用于执行的模型。

该接口不包含模型所需内存资源管理功能，而是将这部分管理内存的工作开放给用户。您可以配合编译后Graph资源占用查询接口、内存的基地址刷新接口来使用，达到自行管理模型内存、获得更多灵活性的目的。

您可以在调用该接口后，调用**10.1.3.6.16 GetCompiledGraphSummary**获取图编译结果的概要信息（比如模型执行所需的内存资源大小及内存是否可刷新、复用等），根据查询到的内存大小，自行申请并管理内存；然后通过**10.1.3.6.18 SetGraphConstMemoryBase**、**10.1.3.6.17 UpdateGraphFeatureMemoryBase**对内存基址进行设置和刷新。

函数原型

Status CompileGraph(uint32_t graph_id);

参数说明

参数名	输入/输出	描述
graph_id	输入	子图对应的id。

返回值

参数名	类型	描述
-	Status	SUCCESS：编译子图成功。 FAILED：编译子图失败。

约束说明

- 该接口只能与10.1.3.6.10 RunGraphWithStreamAsync接口配合使用，不支持与10.1.3.6.7 RunGraph、10.1.3.6.9 RunGraphAsync接口配合使用。
- 包含Variable算子的图不支持使用该接口。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.16 GetCompiledGraphSummary

函数功能

查询图编译结果的概要信息。包括Feature内存大小、Const内存大小、Stream、Event数目及内存是否可刷新等信息。您可以根据该信息，自行申请内存，再通过10.1.3.6.17 UpdateGraphFeatureMemoryBase、10.1.3.6.18 SetGraphConstMemoryBase接口设置Feature内存、Const内存基址。

函数原型

CompiledGraphSummaryPtr GetCompiledGraphSummary(uint32_t graph_id);

参数说明

参数名	输入/输出	描述
graphId	输入	子图对应的id。

返回值

参数名	类型	描述

-	**CompiledGraphSummaryPtr**	图编译结果的概要信息CompiledGraphSummary的share_ptr。
		CompiledGraphSummary具体结构如下所示：
		class GE_FUNC_VISIBILITY CompiledGraphSummary {
		public:
		class Builder;
		class SummaryData;
		CompiledGraphSummary();
		CompiledGraphSummary &operator=(const CompiledGraphSummary &) & = delete;
		CompiledGraphSummary(const CompiledGraphSummary &) = delete;
		// 返回graph是否是静态编译的,只有在静态编译的情况下，才可以使用后续的获取Summary信息的接口
		bool **IsStatic**() const;
		// 获取编译后的Const内存大小
		Status **GetConstMemorySize**(size_t &size) const;
		// 获取编译后的Feature内存大小
		Status **GetFeatureMemorySize**(size_t &size) const;
		// 获取Feature基址是否可刷新
		Status **GetFeatureMemoryBaseRefreshable** (bool &v) const;
		
		// 获取Stream数目
		Status **GetStreamNum**(size_t &num) const;
		// 获取Event数目
		Status **GetEventNum**(size_t &num) const;
		// 获取图编译的输出shape，输出shape可用于计算outputs所占内存大小
		Status **GetOutputShapes**(std::vector<ge::Shape> &shapes) const;
		private:
		CompiledGraphSummary() = default;
		std::shared_ptr<SummaryData> data_{nullptr};
		};

约束说明

在调用本接口前，必须先调用**10.1.3.6.15 CompileGraph**接口进行图编译。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.17 UpdateGraphFeatureMemoryBase

函数功能

用于更新Graph的Feature内存基址。Featrure内存指的是模型执行过程中所需要的中间内存（比如中间节点的输入输出等内存）。

函数原型

Status UpdateGraphFeatureMemoryBase(uint32_t graph_id, const void *const memory, size_t size);

参数说明

参数名	输入/输出	描述
graph_id	输入	子图对应的id。
memory	输入	设置的feature内存基地址
size	输入	设置的feature内存大小

返回值

参数名	类型	描述
-	Status	SUCCESS：设置成功。 FAILED：设置失败。

约束说明

- 在调用本接口前，必须先调用**10.1.3.6.15 CompileGraph**接口进行图编译。
- 调用**10.1.3.6.10 RunGraphWithStreamAsync**完成图执行后，只有feature地址可刷新的图才支持继续调用本接口刷新feature地址，您可以通过**10.1.3.6.16 GetCompiledGraphSummary**接口获取feature地址是否可刷新。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.18 SetGraphConstMemoryBase

函数功能

设置Graph的Const内存基址。

函数原型

Status SetGraphConstMemoryBase(uint32_t graph_id, const void *const memory, size_t size);

参数说明

参数名	输入/输出	描述
graph_id	输入	子图对应的id。
memory	输入	设置的const内存基地址
size	输入	设置的const内存大小

返回值

参数名	类型	描述
-	Status	SUCCESS：设置成功。 FAILED：设置失败。

约束说明

- 在调用本接口前，必须先调用10.1.3.6.15 CompileGraph接口进行图编译。
- 每个graph只支持设置一次，不支持刷新。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.19 RegisterExternalAllocator

函数功能

用户将自己的Allocator注册给GE，适用于静态shape使用用户的内存池场景。

函数原型

Status RegisterExternalAllocator(const void *const stream, AllocatorPtr allocator);

参数说明

参数名	输入/输出	描述
stream	输入	指定allocator注册在哪个Stream上。
allocator	输入	用户allocator对象的智能指针。allocator基于10.1.2.9 Allocator类派生。

返回值

参数名	类型	描述
-	Status	SUCCESS：设置成功。 FAILED：设置失败。

约束说明

- 此接口需要配合10.1.3.6.10 RunGraphWithStreamAsync接口使用，并且需要在10.1.3.6.10 RunGraphWithStreamAsync接口调用前注册。
- 对于同一条流，多次调用本接口，以最后一次注册为准。
- 对于不同流，如果用户使用同一个Allocator，不可以多条流并发执行，在执行下一条Stream前，需要对上一Stream做流同步。
- 将Allocator中的内存释放给操作系统前，需要先调用aclrtSynchronizeStream接口执行流同步，确保Stream中的任务已执行完成。

  aclrtSynchronizeStream接口说明请参见《应用软件开发指南（C&C++）》>“AscendCL API参考”>同步等待。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.20 UnregisterExternalAllocator

函数功能

将用户基于stream注册的Allocator销毁，适用于静态shape使用用户的内存池场景。

函数原型

Status UnregisterExternalAllocator(const void *const stream);

参数说明

参数名	输入/输出	描述
stream	输入	指定需要销毁的用户allocator在哪个流上。

返回值

参数名	类型	描述
-	Status	SUCCESS：设置成功。 FAILED：设置失败。

约束说明

- 用户销毁Allocator前，调用本接口取消注册。
- 待取消注册的Stream不存在，或多次调用本接口取消注册，本接口内部不做任何操作，返回成功。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.21 GEGetErrorMsg

函数功能

执行报错时，调用该接口获取详细的错误信息。

函数原型

std::string GEGetErrorMsg();

参数说明

无。

返回值

参数名	类型	描述
-	std::string	执行过程中的报错信息。

约束说明

建议在执行报错时，调用该接口，获取错误信息以辅助定位问题。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.6.22 GEGetWarningMsg

函数功能

程序执行结束需要查看告警信息时，调用该接口打印告警信息。

函数原型

std::string GEGetWarningMsg();

参数说明

无。

返回值

参数名	类型	描述
-	std::string	执行过程中的告警信息。

约束说明

无。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.7 量化接口

10.1.3.7.1 aclgrphCalibration

函数功能

将非量化Graph自动修改为量化后的Graph。具体的样例请参考**9.1.2 自动量化**。

函数原型

graphStatus aclgrphCalibration(ge::Graph &graph, const std::map<AscendString, AscendString> &quantizeConfigs)

参数说明

参数名	输入/输出	描述
graph	输入	待修改的用户原始Graph。

参数名	输入/输出	描述
quantizeConfigs	输入	map表，key为参数类型，value为参数值，均为字符串格式，描述执行接口所需要的配置选项。具体配置参数如下所示: ● INPUT_DATA_DIR：必填，用于计算量化因子的数据bin文件路径，建议传入不少于一个batch的数据。如果模型有多个输入，则输入数据文件以英文逗号分隔。 ● INPUT_SHAPE：必填，输入数据的shape。例如："input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"。指定的节点必须放在双引号中，节点中间使用英文分号分隔。input_name必须是Graph中的节点名称。 ● SOC_VERSION：必填，执行的soc型号。 昇腾310 AI处理器参数值：Ascend310 昇腾310P AI处理器参数值：Ascend310P* 昇腾910 AI处理器参数值：Ascend910* ● INPUT_FORMAT：选填，输入数据的排布格式，支持"NCHW", "NHWC", "ND"。 ● INPUT_FP16_NODES：选填，指定输入数据类型为FP16的输入节点名称。 ● CONFIG_FILE：选填，用于配置高级选项的配置文件路径。配置文件的示例请参考**9.1.2.3 简易配置文件**。 ● LOG_LEVEL：选填，设置训练后量化时的日志等级，该参数只控制训练后量化过程中显示的日志级别，默认显示info级别： – debug：输出debug/info/warning/error/event级别的日志信息。 – info：输出info/warning/error/event级别的日志信息。 – warning：输出warning/error/event级别的日志信息。 – error：输出error/event级别的日志信息。 此外，训练后量化过程中的日志打屏以及日志落盘信息由**AMCT_LOG_DUMP**环境变量进行控制： – **export AMCT_LOG_DUMP=1**：表示日志打印到屏幕。 – **export AMCT_LOG_DUMP=2**：将日志落盘到当前路径的“amct_log_{timestamp}/

参数名	输入/输出	描述

		amct_acl.log”文件中，同时在“amct_log_{timestamp}”目录下保存量化因子record文件*record.txt*。
		– **export AMCT_LOG_DUMP=3**：将日志落盘到当前路径的“amct_log_{timestamp}/amct_acl.log”文件中，同时在“amct_log_{timestamp}”目录下保存量化因子record文件*record.txt*和包含量化过程中各阶段的图描述信息的graph文件。
		● OUT_NODES：选填，用户Graph的输出节点名。
		● DEVICE_ID：选填，指定设备ID，默认为0。
		– 昇腾310 AI处理器，支持配置该参数。
		– 昇腾910 AI处理器，支持该配置参数。
		– 昇腾310P AI处理器，支持该配置参数。
		● IP_ADDR：指定NCS所在服务器的IP地址。
		– 昇腾310 AI处理器，不支持配置该参数。
		– 昇腾910 AI处理器，不支持该配置参数。
		– 昇腾310P AI处理器，不支持该配置参数。
		● PORT：指定NCS所在服务器端口。
		– 昇腾310 AI处理器，不支持配置该参数。
		– 昇腾910 AI处理器，不支持该配置参数。
		– 昇腾310P AI处理器，不支持该配置参数。
		● INSERT_OP_CONF：插入算子的配置文件路径与文件名，例如aipp预处理算子。
		配置文件路径和文件名：支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符；文件后缀不局限于.cfg格式，但是配置文件中的内容需要满足prototxt格式。
		配置文件的内容示例如下：
		aipp_op {
		aipp_mode:static
		input_format:YUV420SP_U8
		csc_switch:true
		var_reci_chn_0:0.00392157
		var_reci_chn_1:0.00392157
		var_reci_chn_2:0.00392157
		}
		说明
		关于配置文件详细配置以及参数说明，请参见《**ATC工具使用指南**》>高级功能>AIPP使能章节。

参数名	输入/输出	描述
		– 昇腾310 AI处理器，支持该参数。 – 昇腾910 AI处理器，支持该参数。 – 昇腾310P AI处理器，支持该参数。 – 昇腾910B AI处理器，支持该参数。

返回值

参数名	类型	描述
-	graphStatus	SUCCESS：图修改成功。 其他：修改失败

约束说明

- 对于已经被插入量化算子的Graph不支持进行量化。
- 支持量化的层及约束请参考**9.1.2.2 支持量化的层及约束**。

芯片支持情况

昇腾310 AI处理器，支持该接口。

昇腾910 AI处理器，支持该接口。

昇腾910B AI处理器，不支持该接口。

昇腾310P AI处理器，支持该接口。

10.1.3.8 维测接口

10.1.3.8.1 aclgrphProfInit

函数功能

初始化Profiling，设置Profiling参数（目前供用户设置保存性能数据文件的路径）。

函数原型

Status aclgrphProfInit(const char *profiler_path, uint32_t length)

参数说明

参数名	输入/输出	描述
profiler_path	输入	指定保存性能数据的文件的路径，此路径为绝对路径。
length	输入	profiler_path的长度，单位为字节。最大长度不超过4096字节。

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 PATH_INVALID：路径错误 GE_PROF_MODE_CONFLICT：已通过环境变量或者option参数开启Profiling时，aclgrphProfInit将不再生效。

约束说明

- 不支持多次重复调用aclgrphProfInit， 并且该接口需和aclgrphProfFinalize配对使用，先调用aclgrphProfInit接口再调用aclgrphProfFinalize接口。
- 建议该接口在GEInitialize之后，AddGraph之前被调用，可采集到AddGraph时的Profiling数据。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.8.2 aclgrphProfFinalize

函数功能

结束Profiling。

函数原型

Status aclgrphProfFinalize()

参数说明

无

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 其他：失败

约束说明

该接口需和aclgrphProfInit配对使用，先调用aclgrphProfInit接口再调用aclgrphProfFinalize接口，且只需要被调用一次。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.8.3 aclgrphProfCreateConfig

函数功能

创建Profiling配置信息。

函数原型

aclgrphProfConfig *aclgrphProfCreateConfig(uint32_t *deviceid_list, uint32_t device_nums, ProfilingAicoreMetrics aicore_metrics, ProfAicoreEvents *aicore_events, uint64_t data_type_config)

参数说明

参数名	输入/输出	描述
deviceid_list	输入	需要采集数据的Device ID列表。
device_nums	输入	Device个数。需要保证和deviceid_list中的Device数目一致。
aicore_metrics	输入	AI Core metrics，枚举值请参见**10.1.3.11.8 ProfilingAicoreMetrics**。
aicore_events	输入	AI Core events，预留项。

参数名	输入/输出	描述
data_type_config	输入	指定需要采集的Profiling数据内容范围，具体请参见10.1.3.11.7 ProfDataTypeConfig。 数值可按bit位或的方式组合表征，指定输出多类数据信息。

返回值

参数名	类型	描述
aclgrphProfConfig	aclgrphProfConfig	Profiling配置信息结构体指针

约束说明

- aclgrphProfConfig类型数据可以只创建一次，多处使用，用户需要保证数据的一致性和准确性。
- 与aclgrphProfDestroyConfig接口配对使用，先调用aclgrphProfCreateConfig接口再调用aclgrphProfDestroyConfig接口。
- 用户需保证程序结束时调用aclgrphProfDestroyConfig销毁所有创建的profiling配置信息，否则可能会导致内存泄露。
- 如果用户想在不同的Device上指定不同的Profiling配置信息，则可创建不同的aclgrphProfConfig类型数据，并依次调用aclgrphProfStart接口将不同的配置信息下发到不同的Device上。同时注意Device信息不能有重复。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.8.4 aclgrphProfDestroyConfig

函数功能

销毁profiling配置信息。

函数原型

Status aclgrphProfDestroyConfig(aclgrphProfConfig *profiler_config)

参数说明

参数名	输入/输出	描述
profiler_config	输入	profiling配置信息结构指针

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 其他：失败

约束说明

- 与aclgrphProfCreateConfig接口配对使用，先调用aclgrphProfCreateConfig接口再调用aclgrphProfDestroyConfig接口。
- 需要销毁的profiling配置信息必须是由aclgrphProfCreateConfig创建。
- 销毁profiling配置信息失败，请检查配置信息参数是否合理。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.8.5 aclgrphProfStart

函数功能

下发Profiling请求，使能对应数据的采集。

函数原型

Status aclgrphProfStart(const aclgrphProfConfig *profiler_config)

参数说明

参数名	输入/输出	描述
profiler_config	输入	profiling配置信息结构指针

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 其他：失败

约束说明

- 该接口在RunGraph之前调用，若在模型执行过程中调用，Profiling采集到的数据为调用aclgrphProfStart接口之后的数据，可能导致数据不完整。
- 该接口和aclgrphProfStop配对使用，先调用aclgrphProfStart接口再调用aclgrphProfStop接口。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.8.6 aclgrphProfStop

函数功能

停止Profiling数据采集。

函数原型

Status aclgrphProfStop(const aclgrphProfConfig *profiler_config)

参数说明

参数名	输入/输出	描述
profiler_config	输入	profiling配置信息结构指针

返回值

参数名	类型	描述
-	Status	SUCCESS: 成功 其他：失败

约束说明

该接口和aclgrphProfStart配对使用，先调用aclgrphProfStart接口再调用aclgrphProfStop接口。

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.9 其他接口

10.1.3.9.1 aclgrphDumpGraph

函数功能

将输入的图导出到文本中。

函数原型

graphStatus aclgrphDumpGraph(const ge::Graph &graph, const char* file, const size_t len)

约束说明

无

参数说明

参数名	输入/输出	描述
graph	输入	输入的图
file	输出	输出文件名
len	输出	输出文件名的字符长度

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS(0)：成功。 其他值：失败。

10.1.3.9.2 aclgrphSetOpAttr

函数功能

用于支持设置node属性方式的改图。

函数原型

graphStatus aclgrphSetOpAttr(Graph &graph, AttrType attr_type, const char *cfg_path)

约束说明

无。

参数说明

参数名	输入/输出	描述
Graph &graph	输入/输出	作为输入时，代表要通过属性方式修改的图； 作为输出时表示已经修改后的图。
attr_type	输入	attr_type的枚举类型定义： `enum aclgrphAttrType { ATTR_TYPE_KEEP_DTYPE = 0, ATTR_TYPE_WEIGHT_COMPRESS };` ATTR_TYPE_KEEP_DTYPE：保持模型编译时个别算子的计算精度不变。 ATTR_TYPE_WEIGHT_COMPRESS ： 模型编译时对个别算子进行weight压缩。

cfg_path	输出	配置文件路径。配置文件举例：
		计算精度不变的文件格式为：
		`Opname1`
		`Opname2`
		`…`
		weight压缩文件格式为：
		`Opname1;Opname2`

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.9.3 aclgrphGetIRVersion

函数功能

获取模型构建相关接口的版本号。

函数原型

graphStatus aclgrphGetIRVersion(int *major_version, int *minor_version, int *patch_version)

约束说明

入参建议采用整型引用的方式调用接口，如传入整型空指针，则会验空返回失败。

参数说明

参数名	输入/输出	描述
major_version	输出	主版本号
minor_version	输出	中间版本号
patch_version	输出	bug修复版本号

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.9.4 aclgrphGenerateForOp

函数功能

根据单算子信息或单算子json文件构建Graph。

函数原型

graphStatus aclgrphGenerateForOp(const AscendString &op_type, const std::vector<TensorDesc> &inputs,

const std::vector<TensorDesc> &outputs, Graph &graph);

graphStatus aclgrphGenerateForOp(const AscendString &json_path, std::vector<Graph> &graphs)

约束说明

无

参数说明

参数名	输入/输出	描述
op_type	输入	算子类型
inputs	输入	算子的输入Tensor格式列表
outputs	输入	算子的输出Tensor格式列表
json_path	输入	单算子的json文件路径
graph	输出	生成的graph
graphs	输出	生成的graph列表

返回值

参数名	类型	描述
-	graphStatus	0：成功。 其他值：失败。

10.1.3.9.5 GE_ERRORNO

错误码及描述注册宏，该宏对外提供了SUCCESS和FAILED两个错误，供用户使用。

GE_ERRORNO(0, 0, 0, 0, 0, SUCCESS, 0, "success");

GE_ERRORNO(0b11, 0b11, 0b111, 0xFF, 0b11111, FAILED, 0xFFF, "failed");

10.1.3.10 内部关联接口

ge_api_error_codes.h中的如下接口是内部关联接口，开发者不需要直接调用：

表 10-1 内部关联接口

类名	作用	备注
ErrorNoRegisterar类	错误码注册类，用于注册具体的错误码及其描述。	包含以下成员函数： ● 2个构造函数： ErrorNoRegisterar(uint32_t err, const std::string &desc) noexcept; ErrorNoRegisterar(const uint32_t err, const char *const desc) noexcept; 其中，err：错误码；desc: 错误码描述 ● 1个默认析构函数： ~ErrorNoRegisterar() = default;
StatusFactory类	单例状态工厂类，管路注册的错误码。	包含以下成员函数： ● static StatusFactory *Instance() 单例返回StatusFactory 类对象，全局唯一。 ● void RegisterErrorNo(uint32_t err, const std::string &desc) void RegisterErrorNo(const uint32_t err, const char *const desc) 用于错误码注册。 ● std::string GetErrDesc(uint32_t err) 根据错误码值获取错误码描述。

10.1.3.11 数据类型

10.1.3.11.1 ModelBufferData

```
struct ModelBufferData
{
  std::shared_ptr<uint8_t> data = nullptr;
  uint64_t length;
};
```

其中data指向生成的模型数据，length代表该模型的实际大小。

10.1.3.11.2 aclgrphBuildInitialize 支持的配置参数

表 10-2 aclgrphBuildInitialize 支持的配置参数

参数	说明
CORE_TYPE	设置网络模型使用的Core类型，若网络模型中包括Cube算子，则只能使用AiCore。 ● VectorCore ● AiCore，默认为AiCore。 芯片支持情况： 昇腾310P AI处理器：支持 昇腾310 AI处理器，不支持 昇腾910 AI处理器，不支持 昇腾910B AI处理器，不支持
SOC_VERSION	该参数为必选配置。 昇腾310 AI处理器参数值：Ascend310 昇腾310P AI处理器参数值：Ascend310P* 昇腾910 AI处理器参数值：Ascend910* 昇腾910B AI处理器参数值：Ascend910B* 其中：*可能根据芯片性能提升等级、芯片核数使用等级等因素会有不同的取值。 **说明** 如果无法确定当前设备的*<soc_version>*，则在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的*<soc_version>*值为Ascend*xxxyy*。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
BUFFER_OPTIMIZE	数据缓存优化开关。 ● l1_optimize：表示开启l1优化。当前版本该参数无效，等同于off_optimize。 ● l2_optimize：表示开启l2优化。默认为l2_optimize。 ● off_optimize：表示关闭数据缓存优化。 **注意：当参数值设置为l1_optimize，则不能与VIRTUAL_TYPE参数同时使用，同时使用会出现报错，表示虚拟化场景不做l1融合，防止算子过大导致调度异常。** 芯片支持情况： 昇腾310 AI处理器 昇腾910 AI处理器 昇腾310P AI处理器 昇腾910B AI处理器
ENABLE_COMPRESS_WEIGHT	使能全局weight压缩。 AICore支持Weight压缩功能，通过使能该参数，可以对Weight进行数据压缩，在进行算子计算时，对Weight进行解压缩，从而达到减少带宽、提高性能的目的。 该参数使能全局weight压缩，不能与COMPRESS_WEIGHT_CONF同时使用。 ● true：表示使能 ● false：表示关闭 芯片支持情况： 昇腾310 AI处理器：不支持 昇腾910 AI处理器：不支持 昇腾310P AI处理器：不支持 昇腾910B AI处理器：不支持
COMPRESS_WEIGHT_CONF	要压缩的node节点列表配置文件路径，node节点主要为conv算子、fc算子。 路径支持大小写字母、数字，下划线；文件名支持大小写字母、数字，下划线和点(.)。 该参数不能与ENABLE_COMPRESS_WEIGHT参数同时使用。 Weight压缩配置文件由AMCT输出，文件内容即为node名称列表，node名称之间以“;”间隔开。 例如，compress_weight_nodes.cfg文件内容为：conv1; fc1; conv2_2/x1; fc2; conv5_32/x2;fc6。 芯片支持情况： 昇腾310 AI处理器：不支持 昇腾910 AI处理器：不支持 昇腾310P AI处理器：不支持 昇腾910B AI处理器：不支持

参数	说明
PRECISION_MODE	设置网络模型的精度模式。不能与PRECISION_MODE_V2同时使用，建议使用PRECISION_MODE_V2参数。 ● **force_fp32/cube_fp16in_fp32out：** 配置为force_fp32或cube_fp16in_fp32out，效果等同，系统内部都会根据Cube算子或Vector算子，来选择不同的处理方式。cube_fp16in_fp32out为新版本中新增的，对于Cube算子，该选项语义更清晰。 – 对于Cube计算，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32； 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32； 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16； 4. 如果3中的场景不支持，则报错。 – 对于Vector计算，表示网络模型中算子支持float16和float32时，强制选择float32，若原图精度为float16，也会强制转为float32。 如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● **force_fp16：** 表示网络模型中算子支持float16和float32时，强制选择float16。 ● **allow_fp32_to_fp16：** – 对于Cube计算，使用float16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到float16。 ● **must_keep_origin_dtype：** 表示保持原图精度。如果原图中部分算子精度为float16，但NPU中该部分算子的实现不支持float16、仅支持float32，则系统内部会自动采用高精度float32；如果原图中部分算子精度为float32，但NPU中该部分算子的实现不支持float32类型、仅支持float16类型，则不能使用该参数值，系统不支持使用低精度。 ● **allow_mix_precision/allow_mix_precision_fp16**： 配置为allow_mix_precision或allow_mix_precision_fp16，效果等同，均表示混合使用float16和float32数据类型来处理神经网络的过程。 allow_mix_precision_fp16为新版本中新增的，语义更清晰，便于理解。

参数	说明
	针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/<*soc_version*>/aic-<*soc_version*>-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 ● **allow_mix_precision_bf16：当前版本暂不支持。** 表示使用混合使用bfloat16和float32数据类型来处理神经网络的过程。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到bfloat16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/<*soc_version*>/aic-<*soc_version*>-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到bfloat16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到bfloat16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 – 该参数取值仅在昇腾910B AI处理器支持。 ● **allow_fp32_to_bf16：当前版本暂不支持。** – 对于Cube计算，使用bfloat16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网

参数	说明
	络模型中算子不支持float32，则直接降低精度到bfloat16。 – 该参数取值仅在昇腾910B AI处理器支持。 **参数默认值：**force_fp16 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
PRECISION_MODE_V2	设置网络模型的精度模式。不能与PRECISION_MODE同时使用，建议使用PRECISION_MODE_V2参数。 ● **fp16**： 表示网络模型中算子支持float16和float32时，强制选择float16。 ● **origin**： 表示保持原图精度。如果原图中部分算子精度为float16，但NPU中该部分算子实现不支持float16、仅支持float32，则系统内部会自动采用高精度float32；如果原图中部分算子精度为float32，但NPU中该部分算子的实现不支持float32类型、仅支持float16类型，则不能使用该参数值，系统不支持使用低精度。 ● **cube_fp16in_fp32out**： 算子既支持float32又支持float16数据类型时，系统内部根据算子类型不同，选择不同的处理方式。 – 对于Cube计算，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32； 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32； 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16； 4. 如果3中的场景不支持，则报错。 – 对于Vector计算，表示网络模型中算子支持float16和float32时，强制选择float32，若原图精度为float16，也会强制转为float32。 如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● **mixed_float16**： 表示使用混合精度float16和float32数据类型来处理神经网络。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。

参数	说明
	– 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 ● **mixed_bfloat16：当前版本暂不支持。** 表示使用混合精度bfloat16和float32数据类型来处理神经网络。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到bfloat16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到bfloat16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到bfloat16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 **参数默认值：**fp16 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
TUNE_DEVICE_IDS	当前版本暂不支持。

参数	说明
EXEC_DISABLE_REUSED_MEMORY	内存复用开关。 ● 1：关闭内存复用。如果网络模型较大，关闭内存复用开关时可能会造成内存不足，导致模型编译失败。 ● 0：开启内存复用。 默认为0。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
ENABLE_SINGLE_STREAM	是否使能一个模型只能使用一条stream。 ● true：使能 ● false：关闭，默认为false。 芯片支持情况： 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持 昇腾310 AI处理器：不支持 昇腾310P AI处理器：不支持
AICORE_NUM	设置编译时使用的AI Core数目。 芯片支持情况： 昇腾310 AI处理器：不支持 昇腾910 AI处理器：不支持 昇腾310P AI处理器：不支持 昇腾910B AI处理器：不支持

参数	说明

FUSION_SWITCH_FILE	融合开关配置文件路径以及文件名，路径和文件名：支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。
	系统内置了一些图融合和UB融合规则，均为默认开启，可以根据需要关闭指定的融合规则，当前可以关闭的融合规则请参见《**图融合和UB融合规则参考**》，但是由于系统机制，其他融合规则无法关闭。
	配置文件样例*fusion_switch.cfg*，on表示开启，off表示关闭。
	{
	"Switch":{
	"GraphFusion":{
	"RequantFusionPass":"on",
	"ConvToFullyConnectionFusionPass":"off",
	"SoftmaxFusionPass":"on",
	"NotRequantFusionPass":"on",
	"SplitConvConcatFusionPass":"on",
	"ConvConcatFusionPass":"on",
	"MatMulBiasAddFusionPass":"on",
	"PoolingFusionPass":"on",
	"ZConcatv2dFusionPass":"on",
	"ZConcatExt2FusionPass":"on",
	"TfMergeSubFusionPass":"on"
	},
	"UBFusion":{
	"TbePool2dQuantFusionPass":"on"
	}
	}
	}
	同时支持用户一键关闭融合规则：
	{
	"Switch":{
	"GraphFusion":{
	"ALL":"off"
	},
	"UBFusion":{
	"ALL":"off"
	}
	}
	}
	需要注意的是：
	1. 关闭某些融合规则可能会导致功能问题，因此此处的一键式关闭仅关闭系统部分融合规则，而不是全部融合规则。
	2. 一键式关闭融合规则时，可以同时开启部分融合规则：
	{
	"Switch":{
	"GraphFusion":{
	"ALL":"off",
	"SoftmaxFusionPass":"on"
	},
	"UBFusion":{
	"ALL":"off",
	"TbePool2dQuantFusionPass":"on"
	}
	}
	}
	芯片支持情况：
	昇腾310 AI处理器：支持

参数	说明
	昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
ENABLE_SMALL_CHANNEL	是否使能small channel的优化，使能后在channel<=4的卷积层会有性能收益。 建议用户在推理场景下打开此开关。 ● 0：关闭，默认为0。 ● 1：使能。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
OP_SELECT_IMPL_MODE	昇腾AI处理器部分内置算子有高精度和高性能实现方式，用户可以通过该参数配置模型编译时算子选择哪种实现方式。取值包括： ● high_precision：表示算子选择高精度实现。高精度实现算子是指在fp16输入的情况下，通过泰勒展开/牛顿迭代等手段进一步提升算子的精度。 ● high_performance：表示算子选择高性能实现。高性能实现算子是指在fp16输入的情况下，不影响网络精度前提的最优性能实现。默认为high_performance。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OPTYPELIST_FOR_IMPLMODE	设置optype列表中算子的实现模式。 列表中的算子使用OP_SELECT_IMPL_MODE参数指定的模式，当前仅支持指定为high_precision、high_performance两种模式，多个算子使用英文逗号进行分隔。 该参数需要与OP_SELECT_IMPL_MODE参数配合使用，且仅对指定的算子生效，不指定的算子按照默认实现方式选择。例如：OP_SELECT_IMPL_MODE配置为high_precision；OPTYPELIST_FOR_IMPLMODE配置为Pooling、SoftmaxV2。表示对Pooling、SoftmaxV2算子使用统一的高精度模式，未指定算子使用算子的默认实现方式。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OP_COMPILER_CACHE_MODE	用于配置算子编译磁盘缓存模式。默认值为enable。 • enable：启用算子编译缓存功能。启用后，算子编译信息缓存至磁盘，相同编译参数的算子无需重复编译，直接使用缓存内容，从而提升编译速度。 • force：启用算子编译缓存功能，区别于enable模式，force模式下会强制刷新缓存，即先删除已有缓存，再重新编译并加入缓存。比如当用户的python或者依赖库等发生变化时，需要指定为force用于清理已有的缓存。 **说明** 配置为force模式完成编译后，建议后续编译修改为enable模式，以避免每次编译时都强制刷新缓存。 • disable：禁用算子编译缓存功能。 使用说明： • 该参数和OP_COMPILER_CACHE_DIR配合使用。 • 启用算子编译缓存功能时，可以通过如下两种方式来设置缓存文件夹的磁盘空间大小： – 通过配置文件op_cache.ini设置 算子编译完成后，会在OP_COMPILER_CACHE_DIR参数指定路径下自动生成op_cache.ini文件，用户可以通过该配置文件进行缓存磁盘空间大小的配置；若op_cache.ini文件不存在，则需要手动创建。打开该文件，增加如下信息： #配置文件格式，必须包含，自动生成的文件中默认包括如下信息，手动创建时，需要输入 **[op_compiler_cache]** #限制某个芯片下缓存文件夹的磁盘空间的大小，单位为MB **ascend_max_op_cache_size=500** #设置需要保留缓存的空间大小比例，取值范围：[1,100]，单位为百分比；例如80表示缓存空间不足时，删除缓存，保留80% **ascend_remain_cache_size_ratio=80** – 上述文件中的ascend_max_op_cache_size和ascend_remain_cache_size_ratio参数取值都有效时，op_cache.ini文件才会生效。 – 若多个使用者使用相同的缓存路径，建议使用配置文件的方式进行设置，该场景下op_cache.ini文件会影响所有使用者。 – 通过**10.2.7 ASCEND_MAX_OP_CACHE_SIZE**环境变量设置 通过环境变量ASCEND_MAX_OP_CACHE_SIZE来限制某个芯片下缓存文件夹的磁盘空间的大小，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE设置的取值，且需要删除旧的kernel文件时，通过环境变量**10.2.8 ASCEND_REMAIN_CACHE_SIZE_RATIO**设置需要保留缓存的空间大小比例。 **若同时配置了op_cache.ini文件和环境变量，则优先读取op_cache.ini文件中的配置项，若op_cache.ini文件和环境变量都未设置，则读取系统默认值：默认磁盘空间大小500M，默认保留缓存的空间50%。**

参数	说明
	• 由于force选项会先删除已有缓存，所以不建议在程序并行编译时设置，否则可能会导致其他模型因使用的缓存内容被清除而编译失败。 • 建议模型最终发布时设置编译缓存选项为disable或者force。 • 如果算子调优后知识库变更，则需要通过设置为force来刷新缓存，否则无法应用新的调优知识库，从而导致调优应用执行失败。 • 注意，调试开关打开的场景下，即OP_DEBUG_LEVEL非0值或者OP_DEBUG_CONFIG配置非空时，会忽略算子编译磁盘缓存模式的配置，不启用算子编译缓存。主要基于以下两点考虑： – 启用算子编译缓存功能（enable或force模式）后，相同编译参数的算子无需重复编译，编译过程日志无法完整记录。 – 受限于缓存空间大小，对调试场景的编译结果不做缓存。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
OP_COMPILER_CACHE_DIR	用于配置算子编译磁盘缓存的目录。 路径支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 如果参数指定的路径存在且有效，则在指定的路径下自动创建子目录kernel_cache；如果指定的路径不存在但路径有效，则先自动创建目录，然后在该路径下自动创建子目录kernel_cache。 默认值：$HOME/atc_data 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
DEBUG_DIR	用于配置保存算子编译生成的调试相关的过程文件的路径，过程文件包括但不限于算子.o（算子二进制文件）、.json（算子描述文件）、.cce等文件。 默认生成在当前路径下。 如果要自行指定算子编译的过程文件存放路径，需DEBUG_DIR参数与OP_DEBUG_LEVEL参数配合使用，且当OP_DEBUG_LEVEL取值为0时，不能使用DEBUG_DIR参数。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OP_DEBUG_LEVEL	算子编译debug功能开关，取值： ● 0：不开启算子debug功能，在当前路径**不生成**算子编译目录kernel_meta，默认为0。 ● 1：开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）、.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce和python-cce映射文件*_loc.json），用于后续分析AICore Error问题。 ● 2：开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）、.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce和python-cce映射文件*_loc.json），用于后续分析AICore Error问题，同时设置为2，还会关闭编译优化开关、开启ccec调试功能（ccec编译器选项设置为-O0-g）。 ● 3：不开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹中**生成**.o（算子二进制文件）和.json文件（算子描述文件），分析算子问题时可参考。 ● 4：不开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）和.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce）和UB融合计算描述文件（{$kernel_name}_compute.json），可在分析算子问题时进行问题复现、精度比对时使用。 **须知** ● 训练执行时，建议配置为0或3。如果需要进行问题定位，再选择调试开关选项1和2，是因为加入了调试功能会导致网络性能下降。 ● 配置为2，即开启ccec编译选项的场景下，会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要配置为2。 由于算子kernel文件过大而无法编译的日志显示如下： message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx) 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明

MODIFY_MIXLIST	混合精度场景下，配置算子使用混合精度黑白灰名单。配置为路径以及文件名，文件为json格式。
	● 白名单：允许将当前float32类型的算子，降低精度到float16。
	● 黑名单：不允许将当前float32类型的算子，降低精度到float16。
	● 灰名单：当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。
	开启混合精度方式：
	● PRECISION_MODE参数设置为allow_mix_precision、allow_mix_precision_fp16。
	● PRECISION_MODE_V2参数设置为mixed_float16，与PRECISION_MODE参数不能同时配置，建议使用PRECISION_MODE_V2。
	混合精度场景下全网中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用；而使用MODIFY_MIXLIST参数后，用户可以在内置优化策略基础上进行调整，自行指定哪些算子允许降精度，哪些算子不允许降精度。
	配置示例：
	{ge::ir_option::**MODIFY_MIXLIST, "/home/test/ops_info.json**"}
	ops_info.json中可以指定算子类型，多个算子使用英文逗号分隔，样例如下：
	{
	"black-list": { // 黑名单
	"to-remove": [// 黑名单算子转换为灰名单算子
	"Xlog1py"
	],
	"to-add": [// 白名单或灰名单算子转换为黑名单算子
	"Matmul",
	"Cast"
	]
	},
	"white-list": { // 白名单
	"to-remove": [// 白名单算子转换为灰名单算子
	"Conv2D"
	],
	"to-add": [// 黑名单或灰名单算子转换为白名单算子
	"Bias"
	]
	}
	}
	上述配置文件样例中展示的算子仅作为参考，请基于实际硬件环境和具体的算子内置优化策略进行配置。混合精度场景下算子的内置优化策略可在“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/<soc_version>/aic-<soc_version>-ops-info.json”文件中查询，例如：
	"Conv2D":{
	"precision_reduce":{

参数	说明
	"flag":"true" }, true：白名单。false：黑名单。不配置：灰名单。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
SPARSITY	使能全局稀疏特性。 AMCT4选2结构化稀疏后输出的模型，可能存在weight连续4个Cin维度元素中至少有2个为0的场景，模型转换时通过使能全局稀疏开关，将该场景下的元素筛选成2个，从而节省后续推理的计算量，提高推理性能。 由于硬件约束，该参数不能与ENABLE_COMPRESS_WEIGHT、COMPRESS_WEIGHT_CONF同时使用。 **参数值：** ● 1：表示开启4选2结构化稀疏。 ● 0：不开启稀疏特性。 **参数默认值：**0 芯片支持情况： 昇腾310 AI处理器，不支持 昇腾910 AI处理器，不支持 昇腾310P AI处理器，不支持 昇腾910B AI处理器，支持
EXTERNAL_WEIGHT	模型编译时，是否将网络中Const/Constant节点的权重保存在单独的文件中，取值包括： ● 0：不将原网络中的Const/Constant节点的权重保存在单独的文件中，而是直接保存在om模型文件中。默认为0。 ● 1：将原始网络中的Const/Constant节点的权重保存在单独的文件中，并将其类型转换为FileConstant且该文件保存在与om文件同级的weight目录下，权重文件以算子名称命名。 **说明：**当网络中weight占用内存较大且对模型大小有限制时，建议将此配置项设置为“1”。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
DETERMINISTIC	是否开启确定性计算。 默认情况下，不开启确定性计算，算子在相同的硬件和输入下，多次执行的结果可能不同。这个差异的来源，一般是因为在算子实现中，存在异步的多线程执行，会导致浮点数累加的顺序变化。当开启确定性计算功能时，算子在相同的硬件和输入下，多次执行将产生相同的输出。 通常建议不开启确定性计算，因为确定性计算往往会导致算子执行变慢，进而影响性能。当发现模型多次执行结果不同，或者是进行精度调优时，可开启确定性计算，辅助模型调试、调优。 **参数值：** ● 0：默认值，不开启确定性计算。 ● 1：开启确定性计算。 **配置示例：** {ge::ir_option::**DETERMINISTIC, "1"**} 芯片支持情况： 昇腾310P AI处理器，支持 昇腾910 AI处理器，支持 昇腾310 AI处理器，不支持 昇腾910B AI处理器，不支持
OPTION_HOST_ENV_OS	若模型编译环境的操作系统及其架构与模型运行环境不一致时，则需使用本参数设置模型运行环境的操作系统类型。 如果不设置，则默认取模型编译环境的操作系统类型。 与OPTION_HOST_ENV_CPU参数配合使用。 转换后的离线模型文件名称会包含操作系统类型、架构，例如：*xxx_linux_x86_64*.om 参数取值："linux" 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OPTION_HOST_ENV_CPU	若模型编译环境的操作系统及其架构与模型运行环境不一致时，则需使用本参数设置模型运行环境的操作系统架构。 如果不设置，则默认取模型编译环境的操作系统架构。 与OPTION_HOST_ENV_OS参数配合使用。 转换后的离线模型文件名称会包含操作系统类型、架构，例如：*xxx_linux_x86_64*.om 参数取值： ● “aarch64” ● “x86_64” 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
VIRTUAL_TYPE	是否支持离线模型在昇腾虚拟化实例特性生成的虚拟设备上运行。 当前芯片算力比较大，云端用户或者小企业完全不需要使用这么大算力，昇腾虚拟化实例特性支持对芯片的算力进行切分，可满足用户根据自己的业务按需申请算力的诉求。 虚拟设备是指按照指定算力在芯片上申请的虚拟加速资源。 **参数取值：** ● 0：参数默认值，离线模型不在昇腾虚拟化实例特性生成的虚拟设备上运行。 ● 1：离线模型在不同算力的虚拟设备上运行。 **使用约束**： 1. 参数取值为1时，进行模型转换，则转换后离线模型的NPU运行核数（blockdim）可能比实际aicore_num核数大，为aicore_num支持配置范围的最小公倍数：例如aicore_num支持配置范围为{1,2,4,8}，参数取值为1转换后的离线模型，NPU运行核数可能为8。 2. 参数取值为1时，转换后的模型，如果包括如下算子，会默认使用单核，该场景下，将会导致转换后的模型推理性能下降。 ● DynamicRNN ● PadV2D ● SquareSumV2 ● DynamicRNNV2 ● DynamicRNNV3 ● DynamicGRUV 芯片支持情况： 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持 昇腾310 AI处理器：不支持

参数	说明
COMPRESSION_OPTIMIZE_CONF	压缩优化功能配置文件路径以及文件名，通过该参数使能配置文件中指定的压缩优化特性，从而提升网络性能。例如：/home/test/compression_optimize.cfg。 文件内容配置示例如下： enable_first_layer_quantization:true • 当前配置文件仅支持配置**enable_first_layer_quantization**特性，用于控制AIPP首层卷积是否进行优化（AIPP会与量化后模型首层卷积CONV2D前的Quant算子进行融合）。 • 配置文件中冒号前面表示压缩优化特性名称，冒号后面表示是否使能该特性，true表示使能，false表示不使能，默认不使能。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

10.1.3.11.3 aclgrphBuildModel 支持的配置参数

表 10-3 aclgrphBuildModel 支持的配置参数

参数	说明
INPUT_FORMAT	输入数据格式。 支持NCHW、NHWC、ND三种格式。 如果同时开启AIPP，在进行推理业务时，输入图片数据要求为NHWC排布。该场景下INPUT_FORMAT参数指定的数据格式不生效。 **说明** 该参数仅针对动态BatchSize、动态分辨率和动态维度场景。 上述场景下，INPUT_FORMAT必须设置并且和所有Data算子的format保持一致，否则会导致模型编译失败。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
INPUT_SHAPE	模型输入的shape信息。 例如："input_name:n,c,h,w"。指定的节点必须放在双引号中；若模型有多个输入，则不同输入之间使用英文分号分隔，例如，"input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"。input_name必须是转换前的网络模型中的节点名称。 ● 固定shape，例如某网络的输入shape信息，输入1：input_0_0 [16,32,208,208]，输入2：input_1_0 [16,64,208,208]，则INPUT_SHAPE的配置信息为： {ge::ir_option::**INPUT_SHAPE, "input_0_0:16,32,208,208;input_1_0:16,64,208,208"**} ● 若原始模型中输入数据的某个或某些维度值不固定，当前支持通过设置shape分档或设置shape范围两种方式转换模型： – 设置shape分档，包括设置BatchSize档位、设置分辨率档位、设置动态维度档位（最多4维）。 设置INPUT_SHAPE参数时，将对应维度值设置为-1，同时配合使用DYNAMIC_BATCH_SIZE（设置BatchSize档位）或DYNAMIC_IMAGE_SIZE（设置分辨率档位）或DYNAMIC_DIMS（设置动态维度档位）参数。详细用法请参考DYNAMIC_BATCH_SIZE、DYNAMIC_IMAGE_SIZE、DYNAMIC_DIMS参数说明。 – 设置shape范围。 设置INPUT_SHAPE参数时，可将对应维度的值设置为范围。 – 支持按照name设置："input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"，例如："input_name1:8~20,3,5,-1;input_name2:5,3~9,10,-1"。指定的节点必须放在双引号中，节点中间使用英文分号分隔。input_name必须是转换前的网络模型中的节点名称。如果用户知道data节点的name，推荐按照name设置。 – 支持按照index设置："n1,c1,h1,w1;n2,c2,h2,w2"，例如："8~20,3,5,-1；;5,3~9,10,-1"。可以不指定节点名，节点按照索引顺序排列，节点中间使用英文分号分隔。按照index设置shape范围时，data节点需要设置属性index，说明是第几个输入，index从0开始。 如果用户不想指定维度的的范围或具体取值，则可以将其设置为-1，表示此维度可以使用>=1的任意取值。 昇腾310 AI处理器，**不支持设置shape范围**。

参数	说明
	说明 • 上述场景下，INPUT_SHAPE为可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。 • 如果模型转换时通过该参数设置了shape的范围： 使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，调用**aclmdlSetDatasetTensorDesc**接口，用于设置真实的输入Tensor描述信息（输入shape范围）；模型执行之后，调用**aclmdlGetDatasetTensorDesc**接口获取模型动态输出的Tensor描述信息；再进一步调用aclTensorDesc下的操作接口获取输出Tensor数据占用的内存大小、Tensor的Format信息、Tensor的维度信息等。 关于**aclmdlSetDatasetTensorDesc**、**aclmdlGetDatasetTensorDesc**等接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》手册“AscendCL API参考”。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
INPUT_SHAPE_RANGE	**该参数后续版本废弃，请勿使用。**若涉及指定模型输入数据的shape范围，请使用INPUT_SHAPE参数。 指定模型输入数据的shape range。该功能不能与DYNAMIC_BATCH_SIZE、DYNAMIC_IMAGE_SIZE、DYNAMIC_DIMS同时使用。 ● 支持按照name设置："input_name1:[n1,c1,h1,w1];input_name2:[n2,c2,h2,w2]"，例如："input_name1:[8~20,3,5,-1];input_name2:[5,3~9,10,-1]"。指定的节点必须放在双引号中，节点中间使用英文分号分隔。input_name必须是转换前的网络模型中的节点名称，shape range值必须放在[]中。如果用户知道data节点的name，推荐按照name设置INPUT_SHAPE_RANGE。 ● 支持按照index设置："[n1,c1,h1,w1],[n2,c2,h2,w2]"，例如："[8~20,3,5,-1],[5,3~9,10,-1]"。可以不指定节点名，默认第一个中括号为第一个输入节点，节点中间使用英文逗号分隔。按照index设置INPUT_SHAPE_RANGE时，data节点需要设置属性index，说明是第几个输入，index从0开始。 ● 动态维度有shape范围的用波浪号“~”表示，固定维度用固定数字表示，无限定范围的用-1表示； ● 对于标量输入，也需要填入shape范围，表示方法为：[]； ● 对于多输入场景，例如有三个输入时，如果只有第二个第三个输入具有shape范围，第一个输入为固定输入时，仍需要将固定输入shape填入。 芯片支持情况： 昇腾310 AI处理器，不支持。 昇腾910 AI处理器，支持。 昇腾310P AI处理器，支持。 昇腾910B AI处理器，支持。
OP_NAME_MAP	扩展算子（非标准算子）映射配置文件路径和文件名，不同的网络中某扩展算子的功能不同，可以指定该扩展算子到具体网络中实际运行的扩展算子的映射。 路径和文件名：支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 文件内容示例如下： OpA:Network1OpA 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
DYNAMIC_BATCH_SIZE	设置动态batch档位参数，适用于执行推理时，每次处理图片数量不固定的场景。 该参数需要与INPUT_SHAPE配合使用，不能与DYNAMIC_IMAGE_SIZE、DYNAMIC_DIMS同时使用。 最多支持100档配置，每一档通过英文逗号分隔，每个档位数值限制为：[1~2048]。 例如： {ge::ir_option::**INPUT_SHAPE, "data:-1,3,416,416"**}, {ge::ir_option::**DYNAMIC_BATCH_SIZE, "1,2,4,8"**} INPUT_SHAPE中的“-1”表示设置动态batch。 其他使用注意事项： ● 如果通过该参数设置了动态batch，则使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，增加**aclmdlSetDynamicBatchSize**接口，用于设置真实的batch档位。 关于**aclmdlSetDynamicBatchSize**接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。 ● 若用户执行推理业务时，每次处理的图片数量不固定，则可以通过配置该参数来动态分配每次处理的图片数量。例如用户执行推理业务时需要每次处理2张，4张，8张图片，则可以配置为2,4,8，申请了档位后，模型推理时会根据实际档位申请内存。 ● 如果用户设置的档位数值过大或档位过多，可能会导致模型编译失败，此时建议用户减少档位或调低档位数值。 ● 如果用户设置的档位数值过大或档位过多，在运行环境执行推理时，建议执行**swapoff -a**命令关闭swap交换区间作为内存的功能，防止出现由于内存不足，将swap交换空间作为内存继续调用，导致运行环境异常缓慢的情况。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
DYNAMIC_IMAGE_SIZE	设置输入图片的动态分辨率参数。适用于执行推理时，每次处理图片宽和高不固定的场景。 该参数需要与INPUT_SHAPE配合使用，不能与DYNAMIC_BATCH_SIZE、DYNAMIC_DIMS 同时使用。 最多支持100档配置，每一档通过英文分号分隔。 例如："imagesize1_height,imagesize1_width;imagesize2_height,imagesize2_width"，指定的参数必须放在双引号中，每一组参数中间使用英文分号分隔。 具体使用样例如下， INPUT_SHAPE中的“-1”表示设置动态分辨率。 INPUT_SHAPE="data:8,3,-1,-1" ，DYNAMIC_IMAGE_SIZE="416,416;832,832" 其他使用注意事项： ● 如果模型编译时通过该参数设置了动态分辨率，则使用应用工程进行模型推理时，需要在**aclmdlExecute**接口之前，增加**aclmdlSetDynamicHWSize**接口，用于设置真实的分辨率。关于**aclmdlSetDynamicHWSize**接口的具体使用方法，请参见《**应用软件开发指南（C&C++）**》。 ● 如果用户设置的分辨率数值过大或档位过多，可能会导致模型编译失败，此时建议用户减少档位或调低档位数值。 ● 如果用户设置了动态分辨率，实际推理时，使用的数据集图片大小需要与具体使用的分辨率相匹配。 ● 如果用户设置的分辨率数值过大或档位过多，在运行环境执行推理时，建议执行**swapoff -a**命令关闭swap交换区间作为内存的功能，防止出现由于内存不足，将swap交换空间作为内存继续调用，导致运行环境异常缓慢的情况。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明

DYNAMIC_DIMS	设置ND格式下动态维度的档位。适用于执行推理时，每次处理任意维度的场景。
	该参数需要与INPUT_SHAPE配合使用，不能与DYNAMIC_BATCH_SIZE、DYNAMIC_IMAGE_SIZE同时使用。
	参数通过"dim1,dim2,dim3;dim4,dim5,dim6;dim7,dim8,dim9"的形式设置，所有档位必须放在双引号中，每档中间使用英文分号分隔，每档中的dim值与INPUT_SHAPE参数中的-1标识的参数依次对应，INPUT_SHAPE参数中有几个-1，则每档必须设置几个维度。
	支持的档位数取值范围为：(1,100]，每档最多支持任意指定4个维度，建议配置为3~4档。
	例如：
	{ge::ir_option::**INPUT_SHAPE, "data:1,-1"**},
	{ge::ir_option::**DYNAMIC_DIMS, "4;8;16;64"**}
	// 模型编译时，支持的data算子的shape为1,4; 1,8; 1,16;1,64
	{ge::ir_option::**INPUT_SHAPE, "data:1,-1,-1"**},
	{ge::ir_option::**DYNAMIC_DIMS, "1,2;3,4;5,6;7,8"**}
	// 模型编译时，支持的data算子的shape为1,1,2; 1,3,4; 1,5,6; 1,7,8
	芯片支持情况：
	昇腾310 AI处理器：支持
	昇腾310P AI处理器：支持
	昇腾910 AI处理器：支持
	昇腾910B AI处理器：支持

INSERT_OP_FILE	输入预处理算子的配置文件路径，例如aipp算子。
	若配置了该参数，则不能对同一个输入节点同时使用INPUT_FP16_NODES参数。
	配置文件路径：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)
	配置文件的内容示例如下：
	aipp_op {
	aipp_mode:static
	input_format:YUV420SP_U8
	csc_switch:true
	var_reci_chn_0:0.00392157
	var_reci_chn_1:0.00392157
	var_reci_chn_2:0.00392157
	}
	说明
	配置文件详细说明，请参考《**ATC工具使用指南**》。
	芯片支持情况：
	昇腾310 AI处理器：支持
	昇腾310P AI处理器：支持
	昇腾910 AI处理器：支持
	昇腾910B AI处理器：支持

参数	说明
PRECISION_MODE	选择算子精度模式。 ● **force_fp32/cube_fp16in_fp32out：** 配置为force_fp32或cube_fp16in_fp32out，效果等同，系统内部都会根据Cube算子或Vector算子，来选择不同的处理方式。cube_fp16in_fp32out为新版本中新增的，对于Cube算子，该选项语义更清晰。 – 对于Cube计算，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32； 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32； 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16； 4. 如果3中的场景不支持，则报错。 – 对于Vector计算，表示网络模型中算子支持float16和float32时，强制选择float32，若原图精度为float16，也会强制转为float32。 如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● **force_fp16：** 表示网络模型中算子支持float16和float32时，强制选择float16。 ● **allow_fp32_to_fp16：** – 对于Cube计算，使用float16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到float16。 ● **must_keep_origin_dtype：** 表示保持原图精度。如果原图中部分算子精度为float16，但在网络模型中该部分算子的实现不支持float16、仅支持float32，则系统内部会自动采用高精度float32；如果原图中部分算子精度为float32，但在网络模型中该部分算子的实现不支持float32类型、仅支持float16类型，则不能使用该参数值，系统不支持使用低精度。 ● **allow_mix_precision/allow_mix_precision_fp16**： 配置为allow_mix_precision或allow_mix_precision_fp16，效果等同，均表示混合使用float16和float32数据类型来处理神经网络的过程。allow_mix_precision_fp16为新版本中新增的，语义更清晰，便于理解。 针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。

参数	说明
	若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 ● **allow_mix_precision_bf16：当前版本暂不支持**。 表示使用混合使用bfloat16和float32数据类型来处理神经网络的过程。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到bfloat16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到bfloat16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到bfloat16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 – 该参数取值仅在昇腾910B AI处理器支持。 ● **allow_fp32_to_bf16：当前版本暂不支持**。 – 对于Cube计算，使用bfloat16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到bfloat16。 – 该参数取值仅在昇腾910B AI处理器支持。 **参数默认值：**force_fp16 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
PRECISION_MODE_V2	设置网络模型的精度模式。不能与PRECISION_MODE同时使用，建议使用PRECISION_MODE_V2参数。 ● **fp16**： 表示网络模型中算子支持float16和float32时，强制选择float16。 ● **origin**： 表示保持原图精度。如果原图中部分算子精度为float16，但NPU中该部分算子实现不支持float16、仅支持float32，则系统内部会自动采用高精度float32；如果原图中部分算子精度为float32，但NPU中该部分算子的实现不支持float32类型、仅支持float16类型，则不能使用该参数值，系统不支持使用低精度。 ● **cube_fp16in_fp32out**： 算子既支持float32又支持float16数据类型时，系统内部根据算子类型不同，选择不同的处理方式。 – 对于Cube计算，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32； 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32； 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16； 4. 如果3中的场景不支持，则报错。 – 对于Vector计算，表示网络模型中算子支持float16和float32时，强制选择float32，若原图精度为float16，也会强制转为float32。 如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● **mixed_float16**： 表示使用混合精度float16和float32数据类型来处理神经网络。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一

参数	说明
	个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 ● **mixed_bfloat16：当前版本暂不支持。** 表示使用混合精度bfloat16和float32数据类型来处理神经网络。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到bfloat16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到bfloat16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到bfloat16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 **参数默认值：**fp16 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
EXEC_DISABLE_REUSED_MEMORY	内存复用开关。 ● 1：关闭内存复用。如果网络模型较大，关闭内存复用开关时可能会造成内存不足，导致模型编译失败。 ● 0：开启内存复用。 默认为0。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OUTPUT_TYPE	网络输出数据类型： ● FP32：推荐分类网络、检测网络使用。 ● UINT8：图像超分辨率网络，推荐使用，推理性能更好。 ● FP16：推荐分类网络、检测网络使用。 模型编译后，在对应的*.om模型文件中，上述数据类型分别以1、4、2枚举值呈现。 ● 若不指定具体数据类型，则以原始网络模型最后一层输出的算子数据类型为准。 ● 若指定了类型，则以该参数指定的类型为准。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
OUT_NODES	指定输出节点。 如果不指定输出节点（算子名称），则模型的输出默认为最后一层的算子信息，如果指定，则以指定的为准。 某些情况下，用户想要查看某层算子参数是否合适，则需要将该层算子的参数输出，既可以在模型编译时通过该参数指定输出某层算子，模型编译后，在相应.om模型文件最后即可以看到指定输出算子的参数信息，如果通过.om模型文件无法查看，则可以将.om模型文件转换成json格式后查看。 例如："node_name1:0;node_name1:1;node_name2:0"。 指定的输出节点必须放在双引号中，节点中间使用英文分号分隔。node_name必须是模型转换前的网络模型中的节点名称，冒号后的数字表示第几个输出，例如node_name1:0，表示节点名称为node_name1的第0个输出。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
INPUT_FP16_NODES	指定输入数据类型为FP16的输入节点名称。该参数必填。 例如："node_name1;node_name2"，指定的节点必须放在双引号中，节点中间使用英文分号分隔。若配置了该参数，则不能对同一个输入节点同时使用INSERT_OP_FILE参数。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
LOG_LEVEL	设置显示日志的级别。包括如下取值： ● debug：输出debug/info/warning/error/event级别的运行信息。 ● info：输出info/warning/error/event级别的运行信息。 ● warning：输出warning/error/event级别的运行信息。 ● error：输出/error/event级别的运行信息。 ● null：不输出日志。默认为null。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OP_COMPILER_CACHE_MODE	用于配置算子编译磁盘缓存模式。默认值为enable。 ● enable：启用算子编译缓存功能。启用后，算子编译信息缓存至磁盘，相同编译参数的算子无需重复编译，直接使用缓存内容，从而提升编译速度。 ● force：启用算子编译缓存功能，区别于enable模式，force模式下会强制刷新缓存，即先删除已有缓存，再重新编译并加入缓存。比如当用户的python或者依赖库等发生变化时，需要指定为force用于清理已有的缓存。 **说明** 配置为force模式完成编译后，建议后续编译修改为enable模式，以避免每次编译时都强制刷新缓存。 ● disable：禁用算子编译缓存功能。 使用说明： ● 该参数和OP_COMPILER_CACHE_DIR配合使用。 ● 启用算子编译缓存功能时，可以通过如下两种方式来设置缓存文件夹的磁盘空间大小： – 通过配置文件op_cache.ini设置 算子编译完成后，会在OP_COMPILER_CACHE_DIR参数指定路径下自动生成op_cache.ini文件，用户可以通过该配置文件进行缓存磁盘空间大小的配置；若op_cache.ini文件不存在，则需要手动创建。打开该文件，增加如下信息： #配置文件格式，必须包含，自动生成的文件中默认包括如下信息，手动创建时，需要输入 **[op_compiler_cache]** #限制某个芯片下缓存文件夹的磁盘空间的大小，单位为MB **ascend_max_op_cache_size=500** #设置需要保留缓存的空间大小比例，取值范围：[1,100]，单位为百分比；例如80表示缓存空间不足时，删除缓存，保留80% **ascend_remain_cache_size_ratio=80** – 上述文件中的ascend_max_op_cache_size和ascend_remain_cache_size_ratio参数取值都有效时，op_cache.ini文件才会生效。 – 若多个使用者使用相同的缓存路径，建议使用配置文件的方式进行设置，该场景下op_cache.ini文件会影响所有使用者。 – 通过10.2.7 ASCEND_MAX_OP_CACHE_SIZE环境变量设置 通过环境变量ASCEND_MAX_OP_CACHE_SIZE来限制某个芯片下缓存文件夹的磁盘空间的大小，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE设置的取值，且需要删除旧的kernel文件时，通过环境变量10.2.8 ASCEND_REMAIN_CACHE_SIZE_RATIO设置需要保留缓存的空间大小比例。 **若同时配置了op_cache.ini文件和环境变量，则优先读取op_cache.ini文件中的配置项，若op_cache.ini文件和环境变量都未设置，则读取系统默认值：默认磁盘空间大小500M，默认保留缓存的空间50%。**

参数	说明
	● 由于force选项会先删除已有缓存，所以不建议在程序并行编译时设置，否则可能会导致其他模型因使用的缓存内容被清除而编译失败。 ● 建议模型最终发布时设置编译缓存选项为disable或者force。 ● 如果算子调优后知识库变更，则需要通过设置为force来刷新缓存，否则无法应用新的调优知识库，从而导致调优应用执行失败。 ● 注意，调试开关打开的场景下，即OP_DEBUG_LEVEL非0值或者OP_DEBUG_CONFIG配置非空时，会忽略算子编译磁盘缓存模式的配置，不启用算子编译缓存。主要基于以下两点考虑： – 启用算子编译缓存功能（enable或force模式）后，相同编译参数的算子无需重复编译，编译过程日志无法完整记录。 – 受限于缓存空间大小，对调试场景的编译结果不做缓存。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
OP_COMPILER_CACHE_DIR	用于配置算子编译磁盘缓存的目录。 路径支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 如果参数指定的路径存在且有效，则在指定的路径下自动创建子目录kernel_cache；如果指定的路径不存在但路径有效，则先自动创建目录，然后在该路径下自动创建子目录kernel_cache。 默认值：$HOME/atc_data 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
DEBUG_DIR	用于配置保存算子编译生成的调试相关的过程文件的路径，包括算子.o/.json/.cce等文件。 默认生成在当前路径下。 如果要自行指定算子编译的过程文件存放路径，需DEBUG_DIR参数与OP_DEBUG_LEVEL参数配合使用，且当OP_DEBUG_LEVEL取值为0时，不能使用DEBUG_DIR参数。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
OP_DEBUG_LEVEL	算子debug功能开关，取值： ● 0：不开启算子debug功能，在当前路径**不生成**算子编译目录kernel_meta，默认为0。 ● 1：开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）、.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce和python-cce映射文件*_loc.json），用于后续分析AICore Error问题。 ● 2：开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）、.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce和python-cce映射文件*_loc.json），用于后续分析AICore Error问题，同时设置为2，还会关闭编译优化开关、开启ccec调试功能（ccec编译器选项设置为-O0-g）。 ● 3：不开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹中**生成**.o（算子二进制文件）和.json文件（算子描述文件），分析算子问题时可参考。 ● 4：不开启算子debug功能，在当前目录下，生成kernel_meta文件夹，并在该文件夹下**生成**.o（算子二进制文件）和.json文件（算子描述文件）以及TBE指令映射文件（算子cce文件*.cce）和UB融合计算描述文件（{$kernel_name}_compute.json），可在分析算子问题时进行问题复现、精度比对时使用。 **须知** ● 训练执行时，建议配置为0或3。如果需要进行问题定位，再选择调试开关选项1和2，是因为加入了调试功能会导致网络性能下降。 ● 配置为2，即开启ccec编译选项的场景下，会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要配置为2。 由于算子kernel文件过大而无法编译的日志显示如下： message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx) 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
MDL_BANK_PATH	加载模型调优后自定义知识库的路径。 该参数需要与BUFFER_OPTIMIZE参数配合使用，仅在数据缓存优化开关打开的情况下生效，通过利用高速缓存暂存数据的方式，达到提升性能的目的。 **参数值：**模型调优后自定义知识库路径。 **参数值格式：**支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）。 **参数默认值：**${install_path}/compiler/data/fusion_strategy/custom$ 如果默认路径下不存在自定义知识库，则会查找模型的内置知识库，该路径为：${install_path}/compiler/data/fusion_strategy/built-in 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
OP_BANK_PATH	算子调优后自定义知识库路径。 支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）。默认自定义知识库路径${HOME}/Ascend/latest/data/aoe/custom/op 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明

MODIFY_MIXLIST	配置混合精度黑白灰名单，配置为路径以及文件名，文件为json格式。
	● 白名单：允许将当前float32类型的算子，降低精度到float16。
	● 黑名单：不允许将当前float32类型的算子，降低精度到float16。
	● 灰名单：当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。
	allow_mix_precision混合精度模式下，针对全网中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。用户可以在内置优化策略基础上进行调整，自行指定哪些算子允许降精度，哪些算子不允许降精度。
	配置示例：
	{ge::ir_option::**MODIFY_MIXLIST, "/home/test/ops_info.json**"}
	ops_info.json中可以指定算子类型，多个算子使用英文逗号分隔，样例如下：
	{
	"black-list": { // 黑名单
	"to-remove": [// 黑名单算子转换为灰名单算子
	"Xlog1py"
	],
	"to-add": [// 白名单或灰名单算子转换为黑名单算子
	"Matmul",
	"Cast"
	]
	},
	"white-list": { // 白名单
	"to-remove": [// 白名单算子转换为灰名单算子
	"Conv2D"
	],
	"to-add": [// 黑名单或灰名单算子转换为白名单算子
	"Bias"
	]
	}
	}
	上述配置文件样例中展示的算子仅作为参考，请基于实际硬件环境和具体的算子内置优化策略进行配置。混合精度场景下算子的内置优化策略可在“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/soc_version/aic-soc_version-ops-info.json”文件中查询，例如：
	"Conv2D":{
	"precision_reduce":{
	"flag":"true"
	},
	true：白名单。false：黑名单。不配置：灰名单。
	芯片支持情况：
	昇腾310 AI处理器：支持
	昇腾310P AI处理器：支持
	昇腾910 AI处理器：支持

参数	说明
	昇腾910B AI处理器：支持
OP_PRECISION_MODE	设置指定算子内部处理时的精度模式，支持指定一个算子或多个算子。通过该参数传入自定义的精度模式配置文件op_precision.ini，可以为不同的算子设置不同的精度模式。 ini文件中按照算子类型、节点名称设置精度模式，每一行设置一个算子类型或节点名称的精度模式，按节点名称设置精度模式的优先级高于按算子类型。 配置文件中支持设置如下精度模式： ● high_precision：表示高精度。 ● high_performance：表示高性能。 ● support_out_of_bound_index：表示对gather、scatter和segment类算子的indices输入进行越界校验， 校验会降低算子的执行性能。 具体某个算子支持配置的精度/性能模式取值，可以通过CANN软件安装后文件存储路径的opp/built-in/op_impl/ai_core/tbe/impl_mode/all_ops_impl_mode.ini文件查看。 样例如下： [ByOpType] *optype1*=high_precision *optype2*=high_performance *optype3*=support_out_of_bound_index [ByNodeName] *nodename1*=high_precision *nodename2*=high_performance *nodename3*=support_out_of_bound_index 该参数不能与OP_SELECT_IMPL_MODE、OPTYPELIST_FOR_IMPLMODE参数同时使用，若三个参数同时配置，则只有OP_PRECISION_MODE参数指定的模式生效。 该参数不建议配置，若使用高性能或者高精度模式，网络性能或者精度不是最优，则可以使用该参数，通过配置ini文件调整某个具体算子的精度模式。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
SHAPE_GENERALIZED_BUILD_MODE	图编译时Shape的编译方式。**该参数在后续版本废弃、新开发功能请不要使用该参数。** ● shape_generalized：模糊编译，是指对于支持动态Shape的算子，在编译时系统内部对可变维度做了泛化后再进行编译。 该参数使用场景为：用户想编译一次达到多次执行推理的目的时，可以使用模糊编译特性。 ● shape_precise：精确编译，是指按照用户指定的维度信息、在编译时系统内部不做任何转义直接编译。 芯片支持情况： 昇腾310 AI处理器，支持 昇腾910 AI处理器，支持 昇腾310P AI处理器，支持 昇腾910B AI处理器，支持
CUSTOMIZE_DTYES	通过该参数自定义模型编译时算子的计算精度，模型中其他算子以PRECISION_MODE指定的精度模式进行编译。该参数需要配置为配置文件路径及文件名，例如：/home/test/customize_dtypes.cfg。 配置文件中列举需要自定义计算精度的算子名称或算子类型，每个算子单独一行，且算子类型必须为基于IR定义的算子的类型。对于同一个算子，如果同时配置了算子名称和算子类型，编译时以算子名称为准。 配置文件格式要求： # 按照算子名称配置 *Opname1*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* *Opname2*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* # 按照算子类型配置 OpType::*TypeName1*:InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* OpType::*TypeName2*:InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* 配置文件配置示例： # 按照算子名称配置 resnet_v1_50/block1/unit_3/bottleneck_v1/Relu::InputDtype:float16,int8,OutputDtype:float16,int8 # 按照算子类型配置 OpType::Relu:InputDtype:float16,int8,OutputDtype:float16,int8 **说明** ● 算子具体支持的计算精度可以从算子信息库中查看，默认存储路径为CANN软件安装后文件存储路径的：opp/op_impl/custom/ai_core/tbe/config/*${soc_version}*/aic-*${soc_version}*-ops-info.json。 ● 通过该参数指定的优先级高，因此可能会导致精度/性能的下降，如果指定的dtype不支持，会导致编译失败。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
BUILD_INNER_MODEL	当前版本暂不支持。
OP_DEBUG_CONFIG	Global Memory内存检测功能开关。 取值为.cfg配置文件路径，配置文件内多个选项用英文逗号分隔： ● oom：在算子执行过程中，检测Global Memory是否内存越界。 ● dump_bin：算子编译时，在当前执行路径下的kernel_meta文件夹中保留.o和.json文件。 ● dump_cce：算子编译时，在当前执行路径下的kernel_meta文件夹中保留算子cce文件*.cce。 ● dump_loc：算子编译时，在当前执行路径下的kernel_meta文件夹中保留python-cce映射文件*_loc.json。 ● ccec_O0：算子编译时，开启ccec编译器选项-O0，此编译选项针对调试信息不会执行任何优化操作。 ● ccec_g ：算子编译时，开启ccec编译器选项-g，此编译选项相对于-O0，会生成优化调试信息。 配置示例：/root/test0.cfg，其中，test0.cfg文件信息为： op_debug_config = ccec_O0,ccec_g,oom **说明** 开启ccec编译选项的场景下（即ccec_O0、ccec_g选项），会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要开启ccec编译选项。 由于算子kernel文件过大而无法编译的日志显示如下： message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx) 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

参数	说明
EXTERNAL_WEIGHT	是否将网络中Const/Constant节点的权重保存在单独的文件中，取值包括： ● 0：不将原网络中的Const/Constant节点的权重保存在单独的文件中，而是直接保存在om模型文件中。默认为0。 ● 1：将原始网络中的Const/Constant节点的权重保存在单独的文件中，并将其类型转换为FileConstant且该文件保存在与om文件同级的weight目录下，权重文件以算子名称命名。 **说明：**当网络中weight占用内存较大且对模型大小有限制时，建议将此配置项设置为“1”。 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持
EXCLUDE_ENGINES	设置网络模型不使用某个或某些加速引擎。多个以“|”分隔。 NPU集成了多种硬件加速器（也叫加速引擎），比如AiCore/AiVec/AiCpu（按照优先级排列）等，在图编译阶段会按照优先级为算子选择合适的引擎，即当同一个算子被多种引擎支持时，会选择优先级高的那个。 EXCLUDE_ENGINES提供了排除某个引擎的功能，比如在一次训练过程中，为避免数据预处理图和主训练图抢占AiCore，可以给数据预处理图配置不使用AiCore引擎。 取值包括： “AiCore”：AI Core硬件加速引擎 “AiVec”：Vector Core硬件加速引擎 “AiCpu”：AI CPU硬件加速引擎 举例： {ge::ir_option::**EXCLUDE_ENGINES, "AiCore|AiVec"**} 芯片支持情况： 昇腾310 AI处理器：支持 昇腾310P AI处理器：支持 昇腾910 AI处理器：支持 昇腾910B AI处理器：支持

10.1.3.11.4 Parser 解析接口支持的配置参数

表 10-4 Parser 解析接口支持的配置参数

参数	说明
INPUT_FP16_NODES	指定输入数据类型为FP16的输入节点名称。 例如："node_name1;node_name2"，指定的节点必须放在双引号中，节点中间使用英文分号分隔。 示例： {ge::AscendString(ge::ir_option::**INPUT_FP16_NODES**), ge::AscendString("input1;input2")},
IS_INPUT_ADJUST_HW_LAYOUT	用于指定网络输入数据类型是否为FP16，数据格式是否为NC1HWC0。 该参数需要与INPUT_FP16_NODES配合使用。若IS_INPUT_ADJUST_HW_LAYOUT参数设置为true，对应INPUT_FP16_NODES节点的输入数据类型为FP16，输入数据格式为NC1HWC0。 取值范围为false或true，默认值为false。 示例： {ge::AscendString(ge::ir_option::**INPUT_FP16_NODES**), ge::AscendString("input1;input2")}, {ge::AscendString(ge::ir_option::**IS_INPUT_ADJUST_HW_LAYOUT**), ge::AscendString("true,true")},
OUTPUT	指定转图后计算图名称。 示例： {ge::AscendString(ge::ir_option::**OUTPUT**), ge::AscendString("newIssue")},
IS_OUTPUT_ADJUST_HW_LAYOUT	用于指定网络输出的数据类型是否为FP16，数据格式是否为NC1HWC0。 该参数需要与OUT_NODES配合使用。 若IS_OUTPUT_ADJUST_HW_LAYOUT参数设置为true，对应OUT_NODES中输出节点的输出数据类型为FP16，数据格式为NC1HWC0。 取值：false或true，默认为false。 示例： {ge::AscendString(ge::ir_option::**OUT_NODES**), ge::AscendString("add_input:0")}, {ge::AscendString(ge::ir_option::**IS_OUTPUT_ADJUST_HW_LAYOUT**), ge::AscendString("true")},

参数	说明
OUT_NODES	指定输出节点。 如果不指定输出节点（算子名称），则模型的输出默认为最后一层的算子信息，如果指定，则以指定的为准。 某些情况下，用户想要查看某层算子参数是否合适，则需要将该层算子的参数输出，既可以在模型编译时通过该参数指定输出某层算子，模型编译后，在相应.om模型文件最后即可以看到指定输出算子的参数信息，如果通过.om模型文件无法查看，则可以将.om模型文件转换成json格式后查看。 支持三种格式： ● **格式1**："node_name1:0;node_name1:1;node_name2:0"。指定的输出节点必须放在双引号中，节点中间使用英文分号分隔。node_name必须是模型编译前的网络模型中的节点名称，冒号后的数字表示第几个输出，例如node_name1:0，表示节点名称为node_name1的第0个输出。 **示例：** {ge::AscendString(ge::ir_option::**OUT_NODES**), ge::AscendString("add_input:0")}, ● **格式2**："topname1;topname2"。(仅支持**CAFFE**)指定的输出节点必须放在双引号中，节点中间使用英文分号分隔。topname必须是模型编译前的caffe网络模型中的layer的某个top名称；若几个layer有相同的topname，最后一个layer为准。 **示例：** {ge::AscendString(ge::ir_option::**OUT_NODES**), ge::AscendString("res5c_branch2c")}, ● **格式3**："output1;output2;output3"（仅支持**ONNX**）指定网络模型输出的名称（output的name），指定的output的name必须放在双引号中，多个output的name中间使用英文分号分隔。output必须是网络模型的输出。 **示例：** {ge::AscendString(ge::ir_option::**OUT_NODES**), ge::AscendString("output1")},

参数	说明
ENABLE_SCOPE_FUSION_PASSES	指定编译时需要生效的融合规则列表。 融合规则分类如下： • 内置融合规则： – General：各网络通用的scope融合规则；默认生效，不支持用户指定失效。 – Non-General：特定网络适用的scope融合规则；默认不生效，用户可以通过ENABLE_SCOPE_FUSION_PASSES参数指定生效的融合规则列表。 • 用户自定义的融合规则： – General：加载后默认生效，暂不提供用户指定失效的功能。 – Non-General：加载后默认不生效，用户可以通过ENABLE_SCOPE_FUSION_PASSES参数指定生效的融合规则列表。 指定的融合规则必须放在双引号中，规则中间使用英文逗号分隔。 示例： {ge::AscendString(ge::ir_option::**ENABLE_SCOPE_FUSION_PASSES**), ge::AscendString("ScopePass1,ScopePass2")}, **说明** 仅aclgrphParseTensorFlow支持该参数。
INPUT_DATA_NAMES	指定pb文件输入节点的name和index属性的映射关系。系统按照输入name的顺序，设置对应输入节点的index属性。 示例： {ge::AscendString(ge::ir_option::**INPUT_DATA_NAMES**), ge::AscendString("Placeholder,Placeholder_1")}, **说明** 仅aclgrphParseTensorFlow支持该参数。

10.1.3.11.5 InputTensorInfo

```
struct InputTensorInfo {
  uint32_t data_type;              // data type
  std::vector<int64_t> dims;       // shape description
  void *data;                     // tensor data
  int64_t length;                  // tensor length
};
```

10.1.3.11.6 OutputTensorInfo

```
struct OutputTensorInfo {
  uint32_t data_type{};            // data type
  std::vector<int64_t> dims;       // shape description
  std::unique_ptr<uint8_t[]> data;  // tensor data
  int64_t length{};                // tensor length
  OutputTensorInfo() : dims({}), data(nullptr) {}
  OutputTensorInfo(OutputTensorInfo &&out) noexcept = default;
  OutputTensorInfo &operator=(OutputTensorInfo &&out)& noexcept {
    if (this != &out) {
```



```
    data_type = out.data_type;
    dims = out.dims;
    data = std::move(out.data);
    length = out.length;
  }
  return *this;
 }
 OutputTensorInfo(const OutputTensorInfo &) = delete;
 OutputTensorInfo &operator=(const OutputTensorInfo &)& = delete;
};
```

10.1.3.11.7 ProfDataTypeConfig

```
enum ProfDataTypeConfig {
  kProfTaskTime,              // HWTS数据（task level）
  kProfAiCoreMetrics,         // AICore数据
  kProfAicpu,                 // AICPU数据，当前版本暂不支持
  kProfTrainingTrace,         // 迭代轨迹数据
  kProfHccl,                  // 集合通信数据
  kProL2cache,                // L2采样数据
};
```

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.11.8 ProfilingAicoreMetrics

```
enum ProfilingAicoreMetrics {
  kAicoreArithmeticUtilization = 0,  // 各种计算类指标占比统计，包括采集项mac_fp16_ratio、
mac_int8_ratio、vec_fp32_ratio、vec_fp16_ratio、vec_int32_ratio、vec_misc_ratio
  kAicorePipeUtilization = 1,  // 计算单元和搬运单元耗时占比，包括采集项vec_ratio、mac_ratio、
scalar_ratio、mte1_ratio、mte2_ratio、mte3_ratio、icache_miss_rate、fixpipe_ratio
  kAicoreMemory = 2,  // 外部内存读写类指令占比，包括采集项ub_read_bw、ub_write_bw、l1_read_bw、
l1_write_bw、l2_read_bw、l2_write_bw、main_mem_read_bw、main_mem_write_bw
  kAicoreMemoryL0 = 3,  // 内部内存读写类指令占比，包括采集项scalar_ld_ratio、scalar_st_ratio、
l0a_read_bw、l0a_write_bw、l0b_read_bw、l0b_write_bw、l0c_read_bw、l0c_write_bw、
l0c_read_bw_cube、l0c_write_bw_cube
  kAicoreResourceConflictRatio = 4,  // 流水线队列类指令占比，包括采集项vec_bankgroup_cflt_ratio、
vec_bank_cflt_ratio、vec_resc_cflt_ratio、mte1_iq_full_ratio、mte2_iq_full_ratio、mte3_iq_full_ratio、
cube_iq_full_ratio、vec_iq_full_ratio、iq_full_ratio
  kAicoreMemoryUB = 5,  //内部内存读写指令占比，包括采集项ub_read_bw_mte、ub_write_bw_mte、
ub_read_bw_vector、ub_write_bw_vector、ub_read_bw_scalar、ub_write_bw_scalar
  kAicoreL2Cache = 6,  //读写cache命中次数和缺失后重新分配次数， 包括采集项ai*_write_cache_hit、
ai*_write_cache_miss_allocate、ai*_r*_read_cache_hit、ai*_r*_read_cache_miss_allocate
};
```

芯片支持情况

昇腾310P AI处理器，支持

昇腾910 AI处理器，支持

昇腾910B AI处理器，支持

昇腾310 AI处理器，不支持

10.1.3.11.9 options 参数说明

本章节列出GEInitialize、Session构造函数、AddGraph接口传入的配置参数，分别在全局、session、graph生效。

须知

表10-5中仅列出当前版本支持的配置参数，如果表中未列出，表示该参数预留或适用于其他版本的昇腾AI处理器，用户无需关注。

表 10-5 options 配置项

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.graphRunMode	图执行模式，取值： ● 0：在线推理场景下，请配置为0。 ● 1：训练场景下，请配置为1。	必选	all
ge.exec.deviceId	GE实例运行时操作的设备的逻辑ID，限制有限范围为0~N-1，N表示该台Server上的可用昇腾AI处理器个数；	必选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.inputShape	指定模型输入数据的shape。 例如："input_name:n,c,h,w"。指定的节点必须放在双引号中；若模型有多个输入，则不同输入之间使用英文分号分隔，例如，"input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"。input_name必须是转换前的网络模型中的节点名称。 ● 固定shape，例如某网络的输入shape信息，输入1：input_0_0 [16,32,208,208]，输入2：input_1_0 [16,64,208,208]，则INPUT_SHAPE的配置信息为： {"ge.inputShape", "input_0_0:16,32,208,208;input_1_0:16,64,208,208"} ● 若原始模型中输入数据的某个或某些维度值不固定，当前支持通过设置shape分档或设置shape范围两种方式转换模型： – 设置shape分档，包括设置BatchSize档位、设置分辨率档位、设置动态维度档位（最多4维）。 设置INPUT_SHAPE参数时，将对应维度值设置为-1，同时配合使用DYNAMIC_BATCH_SIZE（设置BatchSize档位）或DYNAMIC_IMAGE_SIZE（设置分辨率档位）或DYNAMIC_DIMS（设置动态维度档位）参数。详细用法请参考DYNAMIC_BATCH_SIZE、DYNAMIC_IMAGE_SIZE、DYNAMIC_DIMS参数说明。 – 设置shape范围。 设置INPUT_SHAPE参数时，可将对应维度的值设置为范围。 – 支持按照name设置："input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"，例如："input_name1:8~20,3,5,-1;input_name2:5,3~9,10,-1"。指定的节点必须放在双引号中，节点中间使用英文分号分隔。input_name必须是转换前的网络模型中的节点名称。如果用户知道data节点的name，推荐按照name设置。 如果用户不想指定维度的的范围或具体取值，则可以将其设置为-1，表示此维度可以使用>=1的任意取值。 昇腾310 AI处理器，**不支持设置shape范围**。	可选	session/graph

Options key	Options value	必选/可选	全局/session/graph级别生效
	说明 上述场景下，ge.inputShape为可选设置。如果不设置，系统直接读取对应Data节点的shape信息，如果设置，以此处设置的为准，同时刷新对应Data节点的shape信息。		

ge.dynamicDims	设置ND格式下动态维度的档位。适用于执行推理时，每次处理任意维度的场景。该参数需要与ge.inputShape配合使用。
	参数通过"dim1,dim2,dim3;dim4,dim5,dim6;dim7,dim8,dim9"的形式设置，所有档位必须放在双引号中，每档中间使用英文分号分隔，每档中的dim值与ge.inputShape参数中的-1标识的参数依次对应，inputShape参数中有几个-1，则每档必须设置几个维度。
	支持的档位数取值范围为：(1,100]，每档最多支持任意指定4个维度，建议配置为3~4档。
	使用示例：
	● 若模型只有一个输入：
	{"ge.inputShape", "data:1,-1,-1"}
	{"ge.dynamicDims", "1,2;3,4;5,6;7,8"}
	// graph运行时，支持的data算子的shape为1,1,2; 1,3,4; 1,5,6; 1,7,8
	● 若网络模型有多个输入：
	每档中的dim值与模型输入参数中的-1标识的参数依次对应，模型输入参数中有几个-1，则每档必须设置几个维度。例如网络模型有三个输入，分别为data(1,1,40,T)，label(1,T)，mask(T,T)， 其中T为动态可变。则配置示例为：
	{"ge.inputShape", "data:1,1,40,-1;label:1,-1;mask:-1,-1" }
	{"ge.dynamicDims", "20,20,1,1;40,40,2,2;80,60,4,4"}
	// graph运行时，支持的输入dims组合档数分别为：
	//第0档：data(1,1,40,20)+label(1,20)+mask(1,1)
	//第1档：data(1,1,40,40)+label(1,40)+mask(2,2)
	//第2档：data(1,1,40,80)+label(1,60)+mask(4,4)
		可选	session/graph

ge.dynamicNodeType	用于指定动态输入的节点类型。 ● 0：dataset输入为动态输入。 ● 1：placeholder输入为动态输入。 当前不支持dataset和placeholder输入同时为动态输入。	可选	session/graph

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.precision_mode	算子精度模式，配置要求为string类型。 ● **force_fp32/cube_fp16in_fp32out：** 配置为force_fp32或cube_fp16in_fp32out，效果等同，系统内部都会根据Cube算子或Vector算子，来选择不同的处理方式。cube_fp16in_fp32out为新版本中新增的，对于Cube算子，该选项语义更清晰。 – 对于Cube计算，系统内部会按算子实现的支持情况处理： 1. 优先选择输入数据类型为float16且输出数据类型为float32； 2. 如果1中的场景不支持，则选择输入数据类型为float32且输出数据类型为float32； 3. 如果2中的场景不支持，则选择输入数据类型为float16且输出数据类型为float16； 4. 如果3中的场景不支持，则报错。 – 对于Vector计算，表示网络模型中算子支持float16和float32时，强制选择float32，若原图精度为float16，也会强制转为float32。 如果网络模型中存在部分算子，并且该算子实现不支持float32，比如某算子仅支持float16类型，则该参数不生效，仍然使用支持的float16；如果该算子不支持float32，且又配置了黑名单（precision_reduce = false），则会使用float32的AI CPU算子。 ● **force_fp16：** 表示网络模型中算子支持float16和float32时，强制选择float16。 ● **allow_fp32_to_fp16：** – 对于Cube计算，使用float16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到float16。 ● **must_keep_origin_dtype：** 表示保持原图精度。如果原图中部分算子精度为float16，但在网络模型中该部分算子的实现不支持float16、仅支持float32，则系统内部会自动采用高精度float32；如果原图中	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
	部分算子精度为float32，但在网络模型中该部分算子的实现不支持float32类型、仅支持float16类型，则不能使用该参数值，系统不支持使用低精度。 ● **allow_mix_precision/allow_mix_precision_fp16**：配置为allow_mix_precision或allow_mix_precision_fp16，效果等同，均表示混合使用float16和float32数据类型来处理神经网络的过程。allow_mix_precision_fp16为新版本中新增的，语义更清晰，便于理解。 针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到float16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到float16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 ● **allow_mix_precision_bf16：当前版本暂不支持**。 表示使用混合使用bfloat16和float32数据类型来处理神经网络的过程。针对网络模型中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到bfloat16，从而在精度损失很小的情况下提升系统性能并减少内存使用。 若配置了该种模式，则可以在OPP软件包安装路径${INSTALL_DIR}/opp/built-in/op_impl/ai_core/tbe/config/*<soc_version>*/		

Options key	Options value	必选/可选	全局/session/graph级别生效
	aic-*<soc_version>*-ops-info.json内置优化策略文件中查看“precision_reduce”参数的取值： – 若取值为true（白名单），则表示允许将当前float32类型的算子，降低精度到bfloat16。 – 若取值为false（黑名单），则不允许将当前float32类型的算子降低精度到bfloat16，相应算子仍旧使用float32精度。 – 若网络模型中算子没有配置该参数（灰名单），当前算子的混合精度处理机制和前一个算子保持一致，即如果前一个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。 – 该参数取值仅在昇腾910B AI处理器支持。 ● **allow_fp32_to_bf16：当前版本暂不支持。** – 对于Cube计算，使用bfloat16。 – 对于Vector计算，优先保持原图精度，如果网络模型中算子支持float32，则保留原始精度float32，如果网络模型中算子不支持float32，则直接降低精度到bfloat16。 – 该参数取值仅在昇腾910B AI处理器支持。 **参数默认值：** 训练场景下，针对昇腾910 AI处理器，参数默认值为“allow_fp32_to_fp16”。 训练场景下，针对昇腾910B AI处理器，参数默认值为“must_keep_origin_dtype”。 在线推理场景下，参数默认值为“force_fp16”。		

Options key	Options value	必选/可选	全局/session/graph级别生效

ge.exec.modify_mixlist	配置混合精度黑白灰名单，配置为路径以及文件名，文件为json格式。
	allow_mix_precision混合精度模式下，针对全网中float32数据类型的算子，按照内置的优化策略，自动将部分float32的算子降低精度到float16，从而在精度损失很小的情况下提升系统性能并减少内存使用。用户可以在内置优化策略基础上进行调整，自行指定哪些算子允许降精度，哪些算子不允许降精度。
	配置示例：
	{"ge.exec.modify_mixlist", "/home/test/ops_info.json"};
	ops_info.json中可以指定算子类型，多个算子使用英文逗号分隔，样例如下：
	{
	"black-list": { // 黑名单
	"to-remove": [// 黑名单算子转换为灰名单算子
	"Xlog1py"
	],
	"to-add": [// 白名单或灰名单算子转换为黑名单算子
	"Matmul",
	"Cast"
	]
	},
	"white-list": { // 白名单
	"to-remove": [// 白名单算子转换为灰名单算子
	"Conv2D"
	],
	"to-add": [// 黑名单或灰名单算子转换为白名单算子
	"Bias"
	]
	}
	}
	上述配置文件样例中展示的算子仅作为参考，请基于实际硬件环境和具体的算子内置优化策略进行配置。混合精度场景下算子的内置优化策略可在“OPP安装目录/opp/built-in/op_impl/ai_core/tbe/config/soc_version/aic-soc_version-ops-info.json”文件中查询，例如：
	"Conv2D":{
	"precision_reduce":{
	"flag":"true"
	},
	● true：（白名单）允许将当前float32类型的算子，降低精度到float16。
	● false：（黑名单）不允许将当前float32类型的算子，降低精度到float16。
	● 不配置：（灰名单）当前算子的混合精度处理机制和前一个算子保持一致，即如果前一
		可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
	个算子支持降精度处理，当前算子也支持降精度；如果前一个算子不允许降精度，当前算子也不支持降精度。		
ge.socVersion	指定编译优化模型的昇腾AI处理器版本号。 可从“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/data/platform_config”查看，".ini"文件的文件名即为对应的*${soc_version}*。如果用户根据上述方法仍旧无法确定具体使用的${soc_version}，则： 1. 单击如下手册中的链接并进入该手册，**Ascend-DMI工具使用指导**。 2. 完成“使用工具>使用前准备”，然后进入“使用工具>设备实时状态查询”章节。 3. 使用相关命令查看芯片的详细信息，例如使用**ascend-dmi -i -dt**命令查看芯片的详细信息，返回信息中“Chip Name”对应取值，去掉空格后，即为具体使用的*${soc_version}*。	可选	all
ge.exec.profilingMode	是否开启Profiling功能。 ● 1：开启Profiling功能，从ge.exec.profilingOptions读取Profiling的采集选项。 ● 0：关闭Profiling功能，默认关闭。	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.profilingOptions	Profiling配置选项。 ● output：Profiling采集结果文件保存路径。该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。 – 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。 – 相对路径配置直接以目录名开始，例如：output。 ● storage_limit：指定落盘目录允许存放的最大文件容量。当Profiling数据文件在磁盘中即将占满本参数设置的最大存储空间（剩余空间<=20MB）或剩余磁盘总空间即将被占满时（总空间剩余<=20MB），则将磁盘内最早的文件进行老化删除处理。 单位为MB，有效取值范围为[200, 4294967296]，默认未配置本参数。 参数值配置格式为数值+单位，例如"storage_limit": "200MB"。 未配置本参数时，默认取值为Profiling数据文件存放目录所在磁盘可用空间的90%。 ● training_trace：采集迭代轨迹数据，即训练任务及AI软件栈的软件信息，实现对训练任务的性能分析，重点关注数据增强、前后向计算、梯度聚合更新等相关数据。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。 ● task_trace：采集任务信息数据，即昇腾AI处理器HWTS，分析任务开始、结束等信息。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。 ● hccl：控制hccl数据采集开关，可选on或off，默认为off。 ● aicpu：采集aicpu数据增强的Profiling数据。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。 ● fp_point：training_trace为on时需要配置。指定训练网络迭代轨迹正向算子的开始位置，用于记录前向计算开始时间戳。配置值	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效

	为指定的正向第一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别正向算子的开始位置，例如"fp_point":""。
	• bp_point：training_trace为on时需要配置。指定训练网络迭代轨迹反向算子的结束位置，记录后向计算结束时间戳，BP_POINT和FP_POINT可以计算出正反向时间。配置值为指定的反向最后一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别反向算子的结束位置，例如"bp_point":""。
	• aic_metrics：AI Core的硬件信息，取值如下（单次采集仅支持配置单个参数项）：
	– ArithmeticUtilization：各种计算类指标占比统计
	– PipeUtilization： 计算单元和搬运单元耗时占比，该项为默认值
	– Memory：外部内存读写类指令占比
	– MemoryL0：内部内存读写类指令占比
	– ResourceConflictRatio：流水线队列类指令占比
	• msproftx：控制msproftx用户和上层框架程序输出性能数据的开关，可选on或off，默认值为off。
	Profiling开启msproftx功能之前，需要在程序内调用msproftx相关接口来开启程序的Profiling数据流的输出，详细操作请参见《**应用软件开发指南（C&C++）**》手册“高级功能>Profiling性能数据采集”章节。
	• host_sys_usage：采集Host侧系统及所有进程的CPU和内存数据，可选"cpu"、"mem"或"cpu,mem"。
	• host_sys_usage_freq：配置Host侧系统和所有进程CPU、内存数据的采集频率。取值范围为[1,50]，默认值50，单位hz。
	配置示例：
	`std::map<ge::AscendString, ge::AscendString> ge_options = {{"ge.exec.deviceId", "0"},`
	`{"ge.graphRunMode", "1"},`
	`{"ge.exec.profilingMode", "1"},`
	`{"ge.exec.profilingOptions",`		

Options key	Options value	必选/可选	全局/session/graph级别生效
	R"({"output":"/tmp/profiling","training_trace":"on","fp_point":"resnet_model/conv2d/Conv2Dresnet_model/batch_normalization/FusedBatchNormV3_Reduce","bp_point":"gradients/AddN_70"})"}};		
ge.exec.enableDump	是否开启dump功能，默认关闭。 ● 1：开启dump功能，从dump_path读取dump文件保存路径，dump_path为None时会产生异常。 ● 0：关闭dump功能。	可选	全局/session
ge.exec.dumpPath	Dump文件保存路径。开启dump和溢出检测功能时，该参数必须配置。 该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。 ● 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。 ● 相对路径配置直接以目录名开始，例如：output。	可选	全局/session
ge.exec.dumpStep	指定采集哪些迭代的dump数据。默认值：None，表示所有迭代都会产生dump数据。 多个迭代用“|”分割，例如：0|5|10；也可以用"-"指定迭代范围，例如：0|3-5|10。	可选	全局/session
ge.exec.dumpMode	dump模式，用于指定dump算子输入还是输出数据。取值如下： ● input：仅dump算子输入数据 ● output：仅dump算子输出数据，默认为output ● all：dump算子输入和输出数据	可选	全局/session
ge.exec.dumpData	指定算子dump内容类型，取值： ● tensor: dump算子数据，默认为tensor。 ● stats: dump算子统计数据，保存结果为csv格式。一般情况下dump算子数据量大，用户可以尝试dump算子统计数据。	可选	全局
ge.exec.dumpLayer	指定需要dump的算子。 取值为算子名，多个算子名之间使用空格分隔，例如：“layer1 layer2 layer3”	可选	全局/session

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.enableDumpDebug	是否开启溢出检测功能，默认关闭。 ● 1：开启溢出检测功能，从ge.exec.dumpPath读取Dump文件保存路径，路径配置为None时会产生异常。 ● 0：关闭溢出检测功能。	可选	全局/session
ge.exec.dumpDebugMode	溢出检测模式，取值如下： ● aicore_overflow：AI Core算子溢出检测，检测在算子输入数据正常的情况下，输出是否不正常的极大值（如float16下65500,38400,51200这些值）。一旦检测出这类问题，需要根据网络实际需求和算子逻辑来分析溢出原因并修改算子实现。 ● atomic_overflow：Atomic Add溢出检测，即除了AICore之外，还有其他涉及浮点计算的模块，比如SDMA，检测这些部分出现的溢出问题。 ● all：同时进行AI Core算子溢出检测和Atomic Add溢出检测。	可选	全局/session
ge.exec.disableReuseMemory	内存复用开关; 默认为0。 ● 1：关闭内存复用; ● 0：开启内存复用。	可选	all
ge.graphMemoryMaxSize	**该参数后续版本废弃，请勿使用。** 网络静态内存和最大动态内存，可根据网络大小指定。单位：Byte，取值范围：[0, 256*1024*1024*1024]或[0, 274877906944]。当前受芯片硬件限制，graph_memory_max_size和variable_memory_max_size总和最大支持31G。如果不设置，默认为26GB。	可选	all
ge.variableMemoryMaxSize	**该参数后续版本废弃，请勿使用。** 变量内存，可根据网络大小指定。单位：Byte，取值范围：[0，256*1024*1024*1024]或[0, 274877906944]。当前受芯片硬件限制，graph_memory_max_size和variable_memory_max_size总和最大支持31G。如果不设置，默认为5GB。	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.variable_acc	是否开启变量格式优化。 ● True：开启，默认开启。 ● False：关闭。 为了提高训练效率，在网络执行的变量初始化过程中，将变量转换成更适合在昇腾AI处理器上运行的数据格式。但在用户特殊要求场景下，可以选择关闭该功能开关。	可选	all
ge.exec.rankTableFile	用于描述参与集合通信的集群信息，包括Server，Device，容器等的组织信息，填写ranktable文件路径，包含文件路径和文件名；	可选	all
ge.exec.rankId	rank id，指进程在group中对应的rank标识序号。范围：0~（rank size-1）。对于用户自定义group，rank在本group内从0开始进行重排；对于hccl world group，rank id和world rank id相同。 ● world rank id，指进程在hccl world group中对应的rank标识序号，范围：0~（rank size-1）。 ● local rank id，指group内进程在其所在Server内的rank编号，范围：0~（local rank size-1）。	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.opDebugLevel	算子debug功能开关，取值： ● 0：不开启算子debug功能，默认为0。 ● 1：开启算子debug功能，在当前执行路径下的kernel_meta文件夹中生成TBE指令映射文件（算子cce文件*.cce、python-cce映射文件*_loc.json、.o和.json文件），用于后续工具进行AICore Error问题定位。 ● 2：开启算子debug功能，在当前执行路径下的kernel_meta文件夹中生成TBE指令映射文件（算子cce文件*.cce、python-cce映射文件*_loc.json、.o和.json文件），并关闭编译优化开关（开启ccec编译器选项-O0-g），用于后续工具进行AICore Error问题定位，可定位到出错算子的代码行号。 ● 3：不开启算子debug功能，且在当前执行路径下的kernel_meta文件夹中保留.o和.json文件。 ● 4：不开启算子debug功能，在当前执行路径下的kernel_meta文件夹中**保留**.o（算子二进制文件）和.json文件（算子描述文件），生成TBE指令映射文件（算子cce文件*.cce）和UB融合计算描述文件（{$kernel_name}_compute.json）。 **须知** ● 训练执行时，建议配置为0或3。如果需要进行问题定位，再选择调试开关选项1和2，是因为加入了调试功能会导致网络性能下降。 ● 配置为2，即开启ccec编译选项的场景下，会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要配置为2。由于算子kernel文件过大而无法编译的日志显示如下： message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx)	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效

ge.exec.opDebugConfig	Global Memory内存检测功能开关。
	取值为.cfg配置文件路径，配置文件内多个选项用英文逗号分隔：
	● oom：在算子执行过程中，检测Global Memory是否内存越界
	● dump_bin：算子编译时，在当前执行路径下的kernel_meta文件夹中保留.o和.json文件
	● dump_cce：算子编译时，在当前执行路径下的kernel_meta文件夹中保留算子cce文件*.cce
	● dump_loc：算子编译时，在当前执行路径下的kernel_meta文件夹中保留python-cce映射文件*_loc.json
	● ccec_O0：算子编译时，开启ccec编译器选项-O0，此编译选项针对调试信息不会执行任何优化操作
	● ccec_g ：算子编译时，开启ccec编译器选项-g，此编译选项相对于-O0，会生成优化调试信息
	配置示例：/root/test0.cfg，其中，test0.cfg文件信息为：
	`op_debug_config = ccec_O0,ccec_g,oom`
	说明
	开启ccec编译选项的场景下（即ccec_O0、ccec_g选项），会增大算子Kernel（*.o文件）的大小。动态shape场景下，由于算子编译时会遍历可能存在的所有场景，最终可能会导致由于算子Kernel文件过大而无法进行编译的情况，此种场景下，建议不要开启ccec编译选项。
	由于算子kernel文件过大而无法编译的日志显示如下：
	`message:link error ld.lld: **error: InputSection too large for range extension thunk** ./kernel_meta_xxxxx.o:(xxxx)`
		可选	全局

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.op_compiler_cache_mode	用于配置算子编译磁盘缓存模式。默认值为enable。 ● enable：启用算子编译缓存功能。启用后，算子编译信息缓存至磁盘，相同编译参数的算子无需重复编译，直接使用缓存内容，从而提升编译速度。 ● force：启用算子编译缓存功能，区别于enable模式，force模式下会强制刷新缓存，即先删除已有缓存，再重新编译并加入缓存。比如当用户的python或者依赖库等发生变化时，需要指定为force用于清理已有的缓存。 **说明** 配置为force模式完成编译后，建议后续编译修改为enable模式，以避免每次编译时都强制刷新缓存。 ● disable：禁用算子编译缓存功能。 使用说明： ● 该参数和ge.op_compiler_cache_dir配合使用。 ● 启用算子编译缓存功能时，可以通过如下两种方式来设置缓存文件夹的磁盘空间大小： – 通过配置文件op_cache.ini设置 算子编译完成后，会在OP_COMPILER_CACHE_DIR参数指定路径下自动生成op_cache.ini文件，用户可以通过该配置文件进行缓存磁盘空间大小的配置；若op_cache.ini文件不存在，则需要手动创建。打开该文件，增加如下信息： #配置文件格式，必须包含，自动生成的文件中默认包括如下信息，手动创建时，需要输入 **[op_compiler_cache]** #限制某个芯片下缓存文件夹的磁盘空间的大小，单位为MB **ascend_max_op_cache_size=500** #设置需要保留缓存的空间大小比例，取值范围：[1,100]，单位为百分比；例如80表示缓存空间不足时，删除缓存，保留80% **ascend_remain_cache_size_ratio=80** – 上述文件中的ascend_max_op_cache_size和ascend_remain_cache_size_ratio参数取值都有效时，op_cache.ini文件才会生效。 – 若多个使用者使用相同的缓存路径，建议使用配置文件的方式进行设置，该场	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
	景下op_cache.ini文件会影响所有使用者。 – 通过10.2.7 ASCEND_MAX_OP_CACHE_SIZE环境变量设置 通过环境变量ASCEND_MAX_OP_CACHE_SIZE来限制某个芯片下缓存文件夹的磁盘空间的大小，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE设置的取值，且需要删除旧的kernel文件时，通过环境变量10.2.8 ASCEND_REMAIN_CACHE_SIZE_RATIO设置需要保留缓存的空间大小比例。 **若同时配置了op_cache.ini文件和环境变量，则优先读取op_cache.ini文件中的配置项，若op_cache.ini文件和环境变量都未设置，则读取系统默认值：默认磁盘空间大小500M，默认保留缓存的空间50%。** • 由于force选项会先删除已有缓存，所以不建议在程序并行编译时设置，否则可能会导致其他模型使用的缓存内容被清除而导致失败。 • 建议模型最终发布时设置编译缓存选项为disable或者force。 • 如果算子调优后知识库变更，则需要先通过设置为force来清空缓存，然后再设置为enable重新进行编译，否则无法应用新的调优知识库，从而导致调优应用执行失败。 • 注意，调试开关打开的场景下，即ge.opDebugLevel非0值或者ge.exec.opDebugConfig配置非空时，会忽略算子编译磁盘缓存模式的配置，不启用算子编译缓存。主要基于以下两点考虑： – 启用算子编译缓存功能（enable或force模式）后，相同编译参数的算子无需重复编译，编译过程日志无法完整记录。 – 受限于缓存空间大小，对调试场景的编译结果不做缓存。		

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.op_compiler_cache_dir	用于配置算子编译磁盘缓存的目录。 路径支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）、中文字符。 如果参数指定的路径存在且有效，则在指定的路径下自动创建子目录kernel_cache；如果指定的路径不存在但路径有效，则先自动创建目录，然后在该路径下自动创建子目录kernel_cache。 默认值：$HOME/atc_data	可选	all
ge.debugDir	用于配置保存算子编译生成的调试相关的过程文件的路径，包括算子.o/.json/.cce等文件。 默认生成在当前训练脚本执行路径下。	可选	all
ge.mdl_bank_path	加载模型调优后自定义知识库的路径。 该参数需要与BUFFER_OPTIMIZE参数配合使用，仅在数据缓存优化开关打开的情况下生效，通过利用高速缓存暂存数据的方式，达到提升性能的目的。 **参数值：**模型调优后自定义知识库路径。 **参数值格式：**支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）。 **参数默认值：**${install_path}/compiler/data/fusion_strategy/custom$ 如果默认路径下不存在自定义知识库，则会查找模型的内置知识库，该路径为：${install_path}/compiler/data/fusion_strategy/built-in	可选	all
ge.op_bank_path	算子调优后自定义知识库路径。 支持大小写字母（a-z，A-Z）、数字（0-9）、下划线（_）、中划线（-）、句点（.）。默认自定义知识库路径${HOME}/Ascend/latest/data/aoe/custom/op	可选	all
ge.exec.dynamicGraphExecuteMode	**该参数后续版本废弃，请勿使用。** 对于动态输入场景，需要通过该参数设置执行模式，取值为：dynamic_execute。	可选	graph

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.dataInputsShapeRange	**该参数后续版本废弃，请勿使用。** 动态输入的shape范围。例如全图有2个data输入，配置示例为： std::map<ge::AscendString, ge::AscendString> ge_options = {{"ge.exec.deviceId", "0"}, {"ge.graphRunMode", "1"}, {"ge.exec.dynamicGraphExecuteMode", "dynamic_execute"}, {"ge.exec.dataInputsShapeRange", "[128 ,3~5, 2~128, -1], [128 ,3~5, 2~128, -1]"}}; ● 设置格式为："[n1,c1,h1,w1], [n2,c2,h2,w2]"，例如："[8~20,3,5,-1], [5,3~9,10,-1]"。可以不指定节点名，默认第一个中括号为第一个输入节点，节点中间使用英文逗号分隔。按照index设置INPUT_SHAPE_RANGE时，data节点需要设置属性index，说明是第几个输入，index从0开始。 ● 动态维度有shape范围的用波浪号“~”表示，固定维度用固定数字表示，无限定范围的用-1表示。 ● 对于标量输入，也需要填入shape范围，表示方法为：[]。 ● 对于多输入场景，例如有三个输入时，如果只有第二个第三个输入具有shape范围，第一个输入为固定输入时，仍需要将固定输入shape填入options字段内，例如： {"ge.exec.dataInputsShapeRange", "[3,3,4,10], [-1,3,2~1000,-1],[-1,-1,-1,-1]"}};	可选	graph

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.op_precision_mode	设置指定算子内部处理时的精度模式，支持指定一个算子或多个算子。通过该参数传入自定义的精度模式配置文件op_precision.ini，可以为不同的算子设置不同的精度模式。 ini文件中按照算子类型、节点名称设置精度模式，每一行设置一个算子类型或节点名称的精度模式，按节点名称设置精度模式的优先级高于按算子类型。 配置文件中支持设置如下精度模式： ● high_precision：表示高精度。 ● high_performance：表示高性能。 ● support_out_of_bound_index：表示对gather、scatter和segment类算子的indices输入进行越界校验， 校验会降低算子的执行性能。 具体某个算子支持配置的精度/性能模式取值，可以通过CANN软件安装后文件存储路径的opp/built-in/op_impl/ai_core/tbe/impl_mode/all_ops_impl_mode.ini文件查看。 样例如下： [ByOpType] *optype1*=high_precision *optype2*=high_performance *optype3*=support_out_of_bound_index [ByNodeName] *nodename1*=high_precision *nodename2*=high_performance *nodename3*=support_out_of_bound_index	可选	全局
ge.opSelectImplmode	昇腾AI处理器部分内置算子有高精度和高性能实现方式，用户可以通过该参数配置模型编译时选择哪种算子。取值包括： ● high_precision：表示算子选择高精度实现。高精度实现算子是指在fp16输入的情况下，通过泰勒展开/牛顿迭代等手段进一步提升算子的精度。 ● high_performance：表示算子选择高性能实现。高性能实现算子是指在fp16输入的情况下，不影响网络精度前提的最优性能实现。默认为high_performance。	可选	全局

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.optypelistForImplmode	列举算子optype的列表，该列表中的算子使用ge.opSelectImplmode参数指定的模式，当前支持的算子为Pooling、SoftmaxV2、LRN、ROIAlign。 该参数需要与ge.opSelectImplmode参数配合使用，例如： ge.opSelectImplmode配置为high_precision ge.optypelistForImplmode配置为Pooling。	可选	全局
ge.shape_generalized_build_mode	**该参数在后续版本废弃、请勿使用。**	可选	graph

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.customizeDtypes	通过该参数自定义模型编译时算子的计算精度，模型中其他算子以PRECISION_MODE指定的精度模式进行编译。该参数需要配置为配置文件路径及文件名，例如：/home/test/customize_dtypes.cfg。 配置文件中列举需要自定义计算精度的算子名称或算子类型，每个算子单独一行，且算子类型必须为基于IR定义的算子的类型。对于同一个算子，如果同时配置了算子名称和算子类型，编译时以算子名称为准。 配置文件格式要求： # 按照算子名称配置 *Opname1*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* *Opname2*::InputDtype:*dtype1,dtype2,…*OutputDtype:*dtype1,…* # 按照算子类型配置 OpType::*TypeName1*:InputDtype:*dtype1,dtype2,…* OutputDtype:*dtype1,…* OpType::*TypeName2*:InputDtype:*dtype1,dtype2,…* OutputDtype:*dtype1,…* 配置文件配置示例： # 按照算子名称配置 resnet_v1_50/block1/unit_3/bottleneck_v1/Relu::InputDtype:float16,int8,OutputDtype:float16,int8 # 按照算子类型配置 OpType::Relu:InputDtype:float16,int8,OutputDtype:float16,int8 **说明** • 算子具体支持的计算精度可以从算子信息库中查看，默认存储路径为CANN软件安装后文件存储路径的：opp/op_impl/custom/ai_core/tbe/config/*${soc_version}*/aic-*${soc_version}*-ops-info.json。 • 通过该参数指定的优先级高，因此可能会导致精度/性能的下降，如果指定的dtype不支持，会导致编译失败。 芯片支持情况： 昇腾310 AI处理器，支持该参数。 昇腾910 AI处理器，支持该参数。 昇腾310P AI处理器，支持该参数。 昇腾910B AI处理器，支持该参数。	可选	session

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.atomicCleanPolicy	是否集中清理网络中所有atomic算子占用的内存，取值包括： ● 0：集中清理，默认为0。 ● 1：不集中清理，对网络每一个atomic算子进行单独清零。当网络中内存超限时，可尝试此种清理方式，但可能会导致一定的性能损耗。	可选	session
ge.jit_compile	当前版本暂不支持。	可选	全局/session
ge.build_inner_model	当前版本暂不支持。	可选	NA
ge.externalWeight	是否将网络中Const/Constant节点的权重保存在单独的文件中，取值包括： ● 0：不将原网络中的Const/Constant节点的权重保存在单独的文件中，而是直接保存在om模型文件中。默认为0。 ● 1：将原始网络中的Const/Constant节点的权重保存在单独的文件中，并将其类型转换为FileConstant且该文件保存在与om文件同级的weight目录下，权重文件以算子名称命名。 **说明：**当网络中weight占用内存较大且对模型大小有限制时，建议将此配置项设置为“1”。	可选	session
stream_sync_timeout	图执行时，stream同步等待超时时间，超过配置时间时报同步失败。单位：ms 默认值-1，表示无等待时间，出现同步失败不报错。	可选	全局/session
event_sync_timeout	图执行时，event同步等待超时时间，超过配置时间时报同步失败。单位：ms 默认值-1，表示无等待时间，出现同步失败不报错。	可选	全局/session

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.staticMemoryPolicy	网络运行是否使用内存静态分配方式。 • 0：动态分配内存，即按照实际大小动态分配。 • 1：静态分配内存，即按照允许的最大静态内存与变量内存进行分配。 • 2：动态扩展内存，即当前图所需内存超过前一张图的内存时，直接释放前一张图的内存。 默认值是0。 配置示例： {"ge.exec.staticMemoryPolicy", "1"};	可选	全局
ge.graph_compiler_cache_dir	图编译磁盘缓存目录，和ge.graph_key配合使用，ge.graph_compiler_cache_dir和ge.graph_key同时配置非空时图编译磁盘缓存功能生效。 配置的缓存目录必须存在，否则会导致编译失败。 图发生变化后，原来的缓存文件不可用，用户需要手动删除缓存目录中的缓存文件或者修改ge.graph_key，重新编译生成缓存文件。 其他使用约束和具体的使用方法请参见**9.8 图编译缓存**。	可选	session
ge.graph_key	图唯一标识，建议取值只包含大小写字母（A-Z，a-z）、数字（0-9）、下划线（_）、中划线（-）并且长度不超过128。	可选	graph
ge.featureBaseRefreshable	配置feature内存地址是否可刷新。若用户需要自行管理feature内存并需要多次刷新该地址，则可将该参数配置为可刷新。 取值： 0：feature内存地址不可刷新，默认值为0。 1：支持刷新模型的feature内存地址。	可选	all
ge.constLifecycle	用于在线训练场景下配置常量节点的生命周期。 session：按照session级别存储常量节点，该取值为默认值。 graph：按照graph级别存储常量节点，后续用户可调用**10.1.3.6.18 SetGraphConstMemoryBase**接口按图级别对const内存自行管理。	可选	all

Options key	Options value	必选/可选	全局/session/graph级别生效
ge.exec.inputReuseMemIndexes	用于配置是否开启图的输入节点的内存复用功能，开启后，输入节点的内存可作为模型执行过程中所需要的中间内存再次使用，从而达到降低内存峰值的目的。 参数取值为输入节点的index；如果对多个输入节点都开启内存复用，多个index间使用逗号分隔。输入节点需要设置属性index，说明是第几个输入，index从0开始。 注意： ● 该参数需要与ge.featureBaseRefreshable配合使用。只有在开启模型feature内存地址刷新功能（ge.featureBaseRefreshable配置为1）的情况下，才能开启内存复用功能。 ● 若所配置的输入index大于等于“输入个数”的值，属于非法index，该index配置不生效。 ● 若开启了输入节点的内存复用功能，输入节点内存中的数据会被改写，图执行后，不可以继续使用输入节点内存中的内容。 配置示例： {"ge.exec.inputReuseMemIndexes", "0，1，2"};	可选	graph
ge.exec.outputReuseMemIndexes	用于配置是否开启整图输出的内存复用功能，开启后，整图输出的内存可作为模型执行过程中所需要的中间内存再次使用，从而达到降低内存峰值的目的。 如果开启，配置为整图输出的index；如果对多个输出都开启内存复用，多个index间使用逗号分隔。 注意： ● 输出的index，按照整图输出的顺序标识，index从0开始。 ● 若所配置的输出index大于等于“输出个数”的值，属于非法index，该index配置不生效。 配置示例： {"ge.exec.outputReuseMemIndexes", "0，1，2"};	可选	graph

10.2 环境变量参考

10.2.1 使用说明

本节介绍在程序执行时可配置的环境变量，您可以根据实际需要进行相应配置。

10.2.2 OP_NO_REUSE_MEM

功能描述

在内存复用场景下（默认开启内存复用），支持基于指定算子（节点名称/算子类型）单独分配内存。

通过该环境变量指定要单独分配的一个或多个节点，支持混合配置。配置多个节点时，中间通过逗号(“,”)隔开。

配置示例

- 基于节点名称配置
  ```
  export OP_NO_REUSE_MEM=gradients/logits/semantic/kernel/Regularizer/l2_regularizer_grad/Mul_1,resnet_v1_50/conv1_1/BatchNorm/AssignMovingAvg2
  ```
- 基于算子类型配置
  ```
  export OP_NO_REUSE_MEM=FusedMulAddN,BatchNorm
  ```
- 混合配置
  ```
  export OP_NO_REUSE_MEM=FusedMulAddN, resnet_v1_50/conv1_1/BatchNorm/AssignMovingAvg
  ```

10.2.3 DUMP_GE_GRAPH

功能描述

把整个流程中各个阶段的图描述信息打印到文件中，此环境变量控制dump图的内容多少。取值：

- 1：全量dump。
- 2：不含有权重等数据的基本版dump。
- 3：只显示节点关系的精简版dump。

配置示例

```
export DUMP_GE_GRAPH=1
```

使用约束

NA

10.2.4 DUMP_GRAPH_LEVEL

功能描述

把整个流程中各个阶段的图描述信息打印到文件中，此环境变量可以控制dump图的个数。取值：

- 1：dump所有图。
- 2：dump除子图外的所有图。
- 3：dump最后的生成图。
- 4：dump最早的生成图。

该环境变量只有在DUMP_GE_GRAPH开启时才生效，并且默认为2。

配置示例

```
export DUMP_GRAPH_LEVEL=1
```

10.2.5 DUMP_GRAPH_PATH

功能描述

指定DUMP图文件的保存路径，如果不配置默认保存在脚本执行目录下。

可配置为绝对路径或脚本执行目录的相对路径，配置的路径需要为已存在的目录，且执行用户具有读、写、可执行权限。

配置示例

```
export DUMP_GRAPH_PATH=/home/dumpgraph
```

10.2.6 TE_PARALLEL_COMPILER

功能描述

算子最大并行编译进程数，当大于1时开启并行编译。

网络模型较大时，可通过配置此环境变量开启算子的并行编译功能。最大不超过cpu核数*80%/昇腾AI处理器个数，取值范围1~32，默认值是8。

配置示例

```
export TE_PARALLEL_COMPILER=8
```

10.2.7 ASCEND_MAX_OP_CACHE_SIZE

功能描述

启用算子编译缓存功能时，用于限制某个昇腾AI处理器下缓存文件夹的磁盘空间的大小，默认为500，单位为MB。

配置示例

```
export ASCEND_MAX_OP_CACHE_SIZE=500
```

10.2.8 ASCEND_REMAIN_CACHE_SIZE_RATIO

功能描述

启用算子编译缓存功能时，当编译缓存空间大小达到ASCEND_MAX_OP_CACHE_SIZE而需要删除旧的kernel文件时，需要保留缓存的空间大小比例，默认为50，单位为百分比。

配置示例

```
export ASCEND_REMAIN_CACHE_SIZE_RATIO=50
```


10.2.9 ASCEND_PROCESS_LOG_PATH

功能描述

设置日志落盘路径。

日志存储时如果不存在该目录，会自动创建该目录；如果存在则直接存储。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_PROCESS_LOG_PATH**命令可以查看环境变量设置的路径。
- 若环境变量未配置或配置为空，表示采用日志默认输出方式。

配置示例

```
export ASCEND_PROCESS_LOG_PATH=$HOME/log/
```

可指定日志落盘路径为任意有读写权限的目录。

10.2.10 ASCEND_SLOG_PRINT_TO_STDOUT

功能描述

是否开启日志打屏。开启后，日志将不会保存在log文件中，而是将产生的日志直接打屏显示。

取值为：

- 0：关闭日志打屏，即日志采用默认输出方式，将日志保存在log文件中。
- 1：开启日志打屏。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_SLOG_PRINT_TO_STDOUT**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认输出方式。

配置示例

```
export ASCEND_SLOG_PRINT_TO_STDOUT=1
```

10.2.11 ASCEND_GLOBAL_LOG_LEVEL

功能描述

设置应用类日志的全局日志级别及各模块日志级别。

取值为：

- 0：对应DEBUG级别。

- 1：对应INFO级别。
- 2：对应WARNING级别。
- 3：对应ERROR级别。
- 4：对应NULL级别，不输出日志。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_GLOBAL_LOG_LEVEL**命令可以查看环境变量设置的日志级别。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志配置文件中设置的日志级别。

配置示例

```
export ASCEND_GLOBAL_LOG_LEVEL=1
```

10.2.12 ASCEND_GLOBAL_EVENT_ENABLE

功能描述

设置应用类日志是否开启Event日志。

取值为：

- 0：关闭Event日志。
- 1：开启Event日志。
- 其他值为非法值。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_GLOBAL_EVENT_ENABLE**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志配置文件中设置的日志级别。

配置示例

```
export ASCEND_GLOBAL_EVENT_ENABLE=0
```

10.2.13 ASCEND_LOG_DEVICE_FLUSH_TIMEOUT

功能描述

业务进程退出前，系统有2000ms的默认延时将Device侧应用类日志回传到Host侧，超时后业务进程退出。未回传到Host侧的日志直接在Device侧落盘（路径为/var/log/npu/slog）。

Device侧应用类日志回传到Host侧的延时时间可以通过环境变量**ASCEND_LOG_DEVICE_FLUSH_TIMEOUT**进行设置。

环境变量取值范围为[0, 180000]，单位为ms，默认值为2000。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_LOG_DEVICE_FLUSH_TIMEOUT**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认值。
- 如果业务进程不需要等待所有Device侧应用类日志回传到Host侧，可将环境变量设置为0。
- 针对业务进程退出后仍然有Device侧应用类日志未回传到Host侧这种情况，建议设置更大的延时时间，具体调节的大小可以根据device-app-*pid*的日志内容进行判断。

配置示例

```
export ASCEND_LOG_DEVICE_FLUSH_TIMEOUT=5000
```

10.2.14 ASCEND_HOST_LOG_FILE_NUM

功能描述

EP场景下，设置应用类日志目录（plog和device-*id*）下存储每个进程日志文件的数量。

环境变量取值范围为[1, 1000]，默认值为10。

说明

- 设置的值仅在当前shell下生效。
- 通过执行**echo $ASCEND_HOST_LOG_FILE_NUM**命令可以查看环境变量设置的值。
- 若环境变量未配置或配置为非法值或配置为空，表示采用日志默认值。
- 如果plog和device-*id*日志目录下存储的单个进程的日志文件数量超过设置的值，将会自动删除最早的日志。另外每个plog-*pid*_*.log或device-*pid*_*.log日志文件的大小最大固定为20MB。如果超过该值，会生成新的日志文件。

配置示例

```
export ASCEND_HOST_LOG_FILE_NUM=20
```

10.2.15 PROFILING_MODE

功能描述

是否开启Profiling功能。

- true：开启Profiling功能，从PROFILING_OPTIONS读取Profiling的采集选项。
- false或者不配置：关闭Profiling功能。

配置示例

```
export PROFILING_MODE=true
```


10.2.16 PROFILING_OPTIONS

功能描述

Profiling配置选项。

- output：Profiling采集结果文件保存路径。该参数指定的目录需要在启动训练的环境上（容器或Host侧）提前创建且确保安装时配置的运行用户具有读写权限，支持配置绝对路径或相对路径（相对执行命令行时的当前路径）。
  - 绝对路径配置以“/”开头，例如：/home/HwHiAiUser/output。
  - 相对路径配置直接以目录名开始，例如：output。
- storage_limit：指定落盘目录允许存放的最大文件容量。当Profiling数据文件在磁盘中即将占满本参数设置的最大存储空间（剩余空间<=20MB）或剩余磁盘总空间即将被占满时（总空间剩余<=20MB），则将磁盘内最早的文件进行老化删除处理。

  取值范围为[200, 4294967296]，单位为MB，例如**--storage-limit**=200MB，默认未配置本参数。

  未配置本参数时，默认取值为Profiling数据文件存放目录所在磁盘可用空间的90%。
- training_trace：采集迭代轨迹数据开关，即训练任务及AI软件栈的软件信息，实现对训练任务的性能分析，重点关注前后向计算和梯度聚合更新等相关数据。当采集正向和反向算子数据时该参数必须配置为on。
- task_trace：采集任务信息数据以及Host与Device之间、Device间的同步异步内存复制时延，即昇腾AI处理器HWTS数据，分析任务开始、结束等信息。取值on/off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。当训练profiling mode开启即采集训练Profiling数据时，配置task_trace为on的同时training_trace也必须配置为on。
- hccl：控制hccl数据采集开关，可选on或off，默认为off。
- aicpu：采集AICPU算子的详细信息，如：算子执行时间、数据拷贝时间等。取值on/off，默认为off。如果将该参数配置为“on”和“off”之外的任意值，则按配置为“off”处理。
- fp_point：指定训练网络迭代轨迹正向算子的开始位置，用于记录前向计算开始时间戳。配置值为指定的正向第一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别正向算子的开始位置，例如"fp_point":""。
- bp_point：指定训练网络迭代轨迹反向算子的结束位置，记录后向计算结束时间戳，BP_POINT和FP_POINT可以计算出正反向时间。配置值为指定的反向最后一个算子名字。用户可以在训练脚本中，通过tf.io.write_graph将graph保存成.pbtxt文件，并获取文件中的name名称填入；也可直接配置为空，由系统自动识别反向算子的结束位置，例如"bp_point":""。
- aic_metrics：AI Core和AI Vector Core的硬件信息，取值如下：
  - ArithmeticUtilization：各种计算类指标占比统计；
  - PipeUtilization： 计算单元和搬运单元耗时占比，该项为默认值；
  - Memory：外部内存读写类指令占比；
  - MemoryL0：内部内存读写类指令占比；
  - MemoryUB：内部内存读写指令占比；

  - ResourceConflictRatio：流水线队列类指令占比。
  - L2Cache：读写cache命中次数和缺失后重新分配次数。

  仅昇腾910B AI处理器支持AI Vector Core数据及L2Cache开关。

  说明

  支持自定义需要采集的寄存器，例如："aic_metrics":"**Custom:***0x49,0x8,0x15,0x1b,0x64,0x10*"。

  - Custom字段表示自定义类型，配置为具体的寄存器值，取值范围为[0x1, 0x6E]。
  - 配置的寄存器数最多不能超过8个，寄存器通过“,”区分开。
  - 寄存器的值支持十六进制或十进制。

- l2：控制L2采样数据的开关，可选on或off，默认为off。仅昇腾310P AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持该参数。
- msproftx：控制msproftx用户和上层框架程序输出性能数据的开关，可选on或off，默认值为off。

  Profiling开启msproftx功能之前，需要在程序内调用msproftx相关接口来开启程序的Profiling数据流的输出，详细操作请参见《**应用软件开发指南（C&C++）**》“扩展更多特性>Profiling性能数据采集”章节。
- task_time：控制任务调度耗时以及算子耗时的开关。涉及在ai_stack_time、task_time、op_summary、op_statistic文件中输出相关耗时数据。可选on或off，默认为on。
- runtime_api：控制runtime api性能数据采集开关，可选on或off，默认为off。可采集runtime-api性能数据，包括Host与Device之间、Device间的同步异步内存复制时延等。
- sys_hardware_mem_freq：DDR、HBM带宽采集频率、LLC的读写带宽数据采集频率以及acc_pmu数据和SOC传输带宽信息采集频率，取值范围为[1,100]，默认值50，单位hz。

  说明

  昇腾910B AI处理器不支持DDR采集；仅昇腾910 AI处理器和昇腾910B AI处理器支持HBM采集；仅昇腾910B AI处理器支持acc_pmu数据和SOC传输带宽信息采集。
- llc_profiling：LLC Profiling采集事件，可以设置为：
  - 昇腾310 AI处理器，可选capacity（采集AI CPU和Control CPU的LLC capacity数据）或bandwidth（采集LLC bandwidth），默认为capacity。
  - 昇腾310P AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
  - 昇腾910 AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
  - 昇腾910B AI处理器，可选read（读事件，三级缓存读速率）或write（写事件，三级缓存写速率），默认为read。
- sys_io_sampling_freq：NIC、ROCE采集频率。取值范围为[1,100]，默认值100，单位hz。仅昇腾310 AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持NIC采集频率；仅昇腾910 AI处理器、昇腾910B AI处理器支持ROCE采集频率。
- sys_interconnection_freq：集合通信带宽数据（HCCS）、PCIe数据采集频率以及片间传输带宽信息采集频率。取值范围为[1,50]，默认值50，单位hz。仅昇腾910 AI处理器、昇腾910B AI处理器支持HCCS采集频率；仅昇腾310P AI处理器、昇腾910 AI处理器、昇腾910B AI处理器支持PCIe采集频率；仅昇腾910B AI处理器支持片间传输带宽信息采集频率。

- dvpp_freq：DVPP采集频率。取值范围为[1,100]，默认值50，单位hz。
- instr_profiling_freq：AI Core和AI Vector的带宽和延时采集频率。取值范围[300,30000]，默认值1000，单位cycle。仅昇腾910B AI处理器支持该参数。

  说明

  instr_profiling_freq开关与training_trace、task_trace、hccl、aicpu、fp_point、bp_point、aic_metrics、l2、task_time、runtime_api

  互斥，无法同时执行。
- host_sys：Host侧性能数据采集开关。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开，例如"host_sys": "cpu,mem"。
- host_sys_usage：采集Host侧系统及所有进程的CPU和内存数据。取值包括cpu和mem，可选其中的一项或多项，选多项时用英文逗号隔开。
- host_sys_usage_freq：配置Host侧系统和所有进程CPU、内存数据的采集频率。取值范围为[1,50]，默认值50，单位hz。

说明

- 除动态shape场景外的其他场景，fp_point、bp_point为自动配置项，无需用户手动配置。动态shape场景不支持自动配置fp/bp，需要用户手动设置。
- 在线推理支持task_trace和aicpu，不支持training_trace。

配置示例

```
export PROFILING_OPTIONS='{"output":"/tmp/
profiling","training_trace":"on","task_trace":"on","fp_point":"","bp_point":"","aic_metrics":"PipeUtilization"}'
```


11 附录

开启AI CPU Cast算子自动插入特性

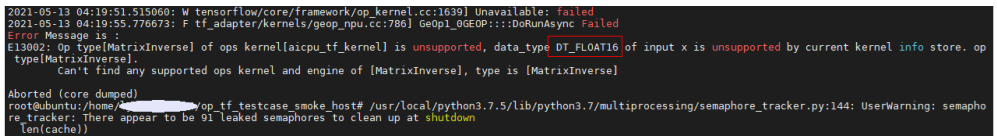
11.1 开启 AI CPU Cast 算子自动插入特性

简介

模型编译时，若遇到AI CPU算子不支持某种数据类型导致编译失败的场景，可通过启用Cast算子自动插入特性快速将输入转换为算子支持的数据类型，从而实现网络的快速打通。

如图11-1，表示MatrixInverse算子的输入x不支持float16的数据类型。

图 11-1 报错示例



此种场景下，即可开启Cast算子自动插入特性，详细操作方法见操作步骤。

操作步骤

步骤1 打开AutoCast开关。

修改“Ascend-cann-toolkit安装目录/ascend-toolkit/latest”目录中“lib64/plugin/opskernel/config/init.conf”文件，将“AutoCastMode”参数的值修改为1，如下所示：

```
...
AutoCastMode = 1
```

步骤2 修改对应的算子信息库（内置算子信息库路径：built-in/op_impl/aicpu/aicpu_kernel/config），在需要修改的算子中插入Cast转换规则。

如下所示，MatrixInverse算子的输入x不支持float16，算子信息库配置如下：

```
"MatrixInverse":{
    "input0":{
```

```
        "name":"x",
        "type":"DT_FLOAT,DT_DOUBLE,DT_COMPLEX128,DT_COMPLEX64"
    },
    "opInfo":{
        "computeCost":"100",
        "engine":"DNN_VM_AICPU",
        "flagAsync":"False",
        "flagPartial":"False",
        "formatAgnostic":"False",
        "opKernelLib":"TFKernel",
        "opsFlag":"OPS_FLAG_OPEN",
        "subTypeOfInferShape":"1"
    },
    "output0":{
        "name":"y",
        "type":"DT_FLOAT,DT_DOUBLE,DT_COMPLEX128,DT_COMPLEX64"
    }
},
```

为了让其支持float16，需要做如下修改：

1. 对输入信息进行修改，增加支持的数据类型，并增加数据类型转换规则。

   例如，对MatrixInverse算子，输入增加对float16类型的支持，并增加cast规则，将float16转换为float32，代表在此输入前会插入一个float16到float32的cast算子。

   ```
   "input0":{
       "name":"x",
       "type":"DT_FLOAT,DT_DOUBLE,DT_COMPLEX128,DT_COMPLEX64,DT_FLOAT16",
       "srcAutoCastType":"DT_FLOAT16",
       "dstAutoCastType":"DT_FLOAT"
   },
   ```

   - 支持的“type”中增加“DT_FLOAT16”数据类型，支持的数据类型可参见对应的算子信息库中Cast算子的定义。
   - 增加配置“srcAutoCastType”，代表输入数据的类型。
   - 增加配置“dstAutoCastType”，代表需要转换成的目标数据类型。
2. 对输出信息进行修改，增加支持的数据类型，并增加数据类型转换规则。

   例如，对MatrixInverse算子，输出增加对float16类型的支持，并增加cast规则，将float32转换为float16，代表在此输出后插入一个float32到float16的cast算子。

   ```
   "output0":{
       "name":"y",
       "type":"DT_FLOAT,DT_DOUBLE,DT_COMPLEX128,DT_COMPLEX64,DT_FLOAT16"
       "srcAutoCastType":"DT_FLOAT",
       "dstAutoCastType":"DT_FLOAT16"
   }
   ```

   - 支持的“type”中增加“DT_FLOAT16”数据类型，支持的数据类型可参见对应的算子信息库中Cast算子的定义。
   - 增加配置“srcAutoCastType”，代表输入数据的类型。
   - 增加配置“dstAutoCastType”，代表需要转换成的目标数据类型。

须知

- 若算子的多个输入、多个输出要求具有相同的数据类型，则每个输入、输出都需要按照上述规则进行修改。
- 由于插入Cast算子，精度会有一定程度的损失，具体损失大小与转换的数据类型有关。

----结束

