TE API Reference

Issue 01

Date 2020-05-30

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Introduction

Tensor Engine, abbreviated as TE, is a development framework for custom operators based on the Tensor Virtual Machine (TVM). The TVM is an open source project of the community. It aims to further abstract the generation rules of operators by dividing the operators into operation primitives and combining the operators when necessary. According to the definition of the computation process of operators, the TVM uses the Schedule technology and the Codegen technology to generate the operators for the specified hardware.

Schedule is used to describe the computation process for implementing an operator on hardware, which requires profound hardware knowledge. To reduce the difficulty in writing operators, the writing of schedules is simplified based on the TVM. The concept of "auto schedule" is used to provide a group of TE APIs to combine the computation of operators. By combination using APIs, you can define the computation process of an operator and hand over the schedule to "auto schedule". This document describes microprocess APIs defined based on the TVM. You can use these APIs to develop operators.

2 Description

To improve the usability of custom operators and the development efficiency, some vector operators are encapsulated in a modular manner. For element-wise operation APIs, you need to use TVM primitives to define the input tensor, call the encapsulated APIs, briefly describe the computation process of the custom operator, and then call the provided **auto schedule** and **build** APIs to compile the custom operator into a binary executable file.

TE provides a group of encapsulated APIs, covering vector calculation. The APIs include the element-wise operation APIs, reduction APIs, broadcast API, index operation APIs, concat API, convolution API, 4D/5D conversion APIs, and matrix multiplication API.

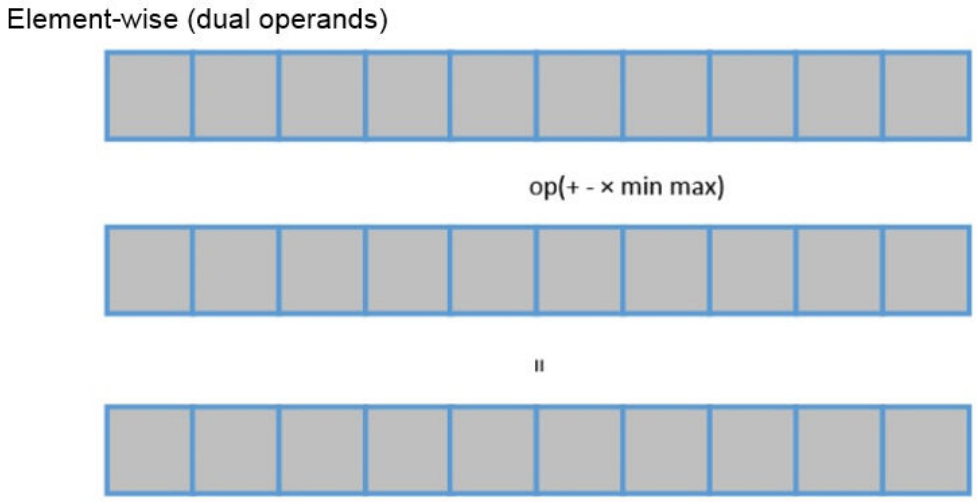
You can view the API definition files in the **ddk/site-packages/te-0.4.0.egg/te/lang/cce/te_compute**, **ddk/site-packages/te-0.4.0.egg/te/lang/cce/te_schedule**, and **ddk/site-packages/topi-0.4.0.egg/topi/generic** directories under the installation directory of the DDK package. If you install Mind Studio and the DDK both in boot installation mode, you can log in to the Mind Studio server as the Mind Studio installation user and view the API definition files under the corresponding directory in **~/tools/che/ddk/ddk/site-packages**. For details about the mapping between the definition files and the APIs, see the following API description. For details about how to use these APIs to develop custom operators and compile and run code, see *TE Custom Operator Development Guide*.

Element-wise Operation APIs

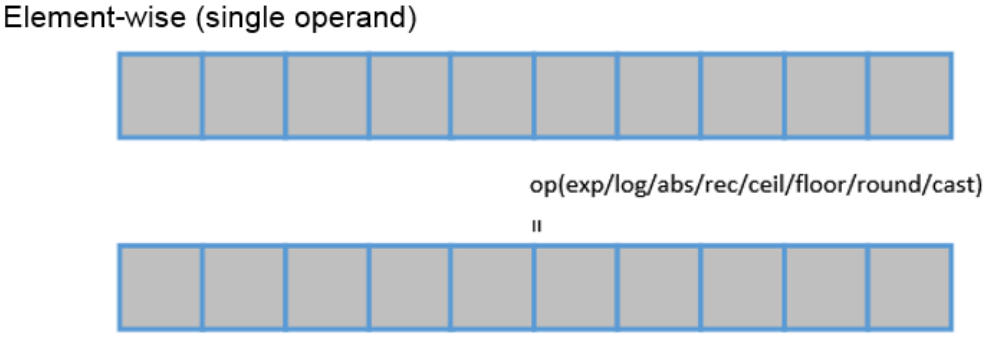
The input data is calculated using different operations by element, and the output usually has the same shape as the input.

- Dual-operand operations. Two tensors are input and calculated by element to obtain the result.
  - Add the corresponding elements of two sensors. For details, see **3.1 te.lang.cce.vadd(lhs, rhs)**.
  - Subtract an element of one tensor from an element of the other tensor. For details, see **3.2 te.lang.cce.vsub(lhs, rhs)**.
  - Multiply an element of one tensor and an element of the other tensor. For details, see **3.3 te.lang.cce.vmul(lhs, rhs)**.
  - Compare the corresponding elements of two sensors and choose the smaller value. For details, see **3.4 te.lang.cce.vmin(lhs, rhs)**.

  - Compare the corresponding elements of two sensors and choose the larger value. For details, see **3.5 te.lang.cce.vmax(lhs, rhs)**.
  - Perform the bitwise OR operation on the corresponding elements of two sensors. For details, see **3.6 te.lang.cce.vor(lhs, rhs)**.
  - Perform the bitwise AND operation on the corresponding elements of two tensors. For details, see **3.7 te.lang.cce.vand(lhs, rhs)**.

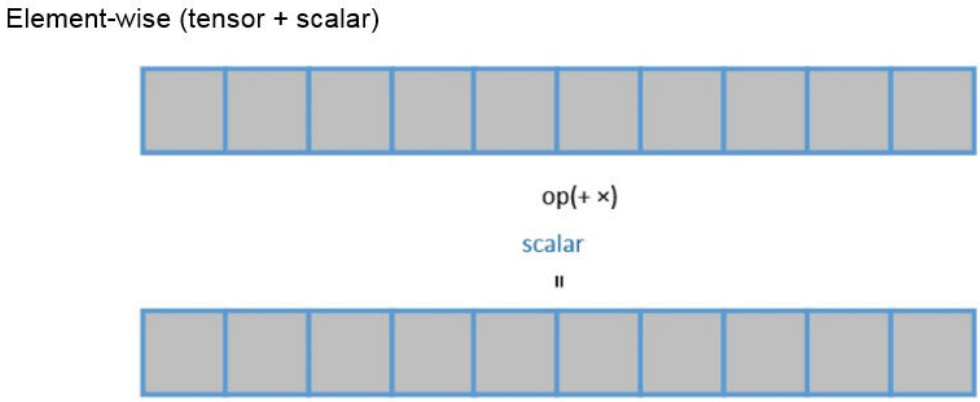


- Single-operand operations. One tensor is input and calculated by element.
  - Natural exponential. For details, see **3.11 te.lang.cce.vexp(raw_tensor)**.
  - Logarithm. For details, see **3.10 te.lang.cce.vlog(raw_tensor)**.
  - Absolute value. For details, see **3.12 te.lang.cce.vabs(raw_tensor)**.
  - Reciprocal. For details, see **3.13 te.lang.cce.vrec(raw_tensor)**.
  - Round up. For details, see **3.21 te.lang.cce.ceil(raw_tensor)**.
  - Round down. For details, see **3.22 te.lang.cce.floor(raw_tensor)**.
  - Banker's rounding. For details, see **3.23 te.lang.cce.round(raw_tensor)**.
  - Data type conversion. For details, see **3.14 te.lang.cce.cast_to(data, dtype, f1628IntegerFlag=False)**.
  - ReLU calculation. For details, see **3.15 te.lang.cce.vrelu(raw_tensor)**.
  - Bitwise NOT. For details, see **3.16 te.lang.cce.vnot(raw_tensor)**.



- Tensor and scalar value operations. Each element of the input tensor is calculated with the same value.

  - Add a scalar to a tensor. For details, see **3.8 te.lang.cce.vadds(raw_tensor, scalar)**.
  - Multiply a tensor by a scalar. For details, see **3.9 te.lang.cce.vmuls(raw_tensor, scalar)**.

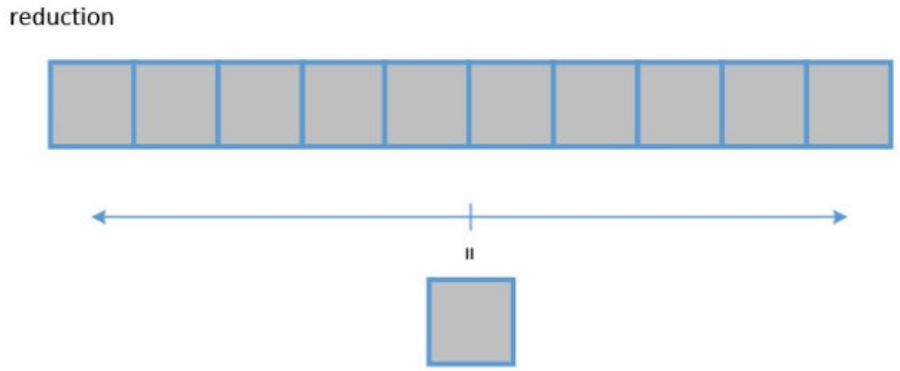


- Three-operand operations. Three tensors are input. The tensors are calculated with scalars by following the preceding calculation rules to obtain the results.
  - Add a scaled tensor to the second tensor. For details, see **3.17 te.lang.cce.vaxpy(lhs, rhs, scalar)**.
  - Calculate three input tensors by following the rule of x * y + z. For details, see **3.18 te.lang.cce.vmla(x, y, z)**.
  - Calculate three input tensors by following the rule of x * z + y. For details, see **3.19 te.lang.cce.vmadd(x, y, z)**.
  - Calculate three input tensors by following the rule of relu(x * z + y). For details, see **3.20 te.lang.cce.vmaddrelu(x, y, z)**.

Reduction APIs

Compress the data of a dimension and perform operations such as accumulation and multiplication on the data in the specified direction. The operation output is one dimension less than the input data.

- Accumulate the values along an axis. For details, see **3.24 te.lang.cce.sum(raw_tensor, axis, keepdims=False)**.
- Calculate the minimum value along an axis. For details, see **3.25 te.lang.cce.reduce_min(raw_tensor, axis, keepdims=False)**.
- Calculate the maximum value along an axis. For details, see **3.26 te.lang.cce.reduce_max(raw_tensor, axis, keepdims=False)**.



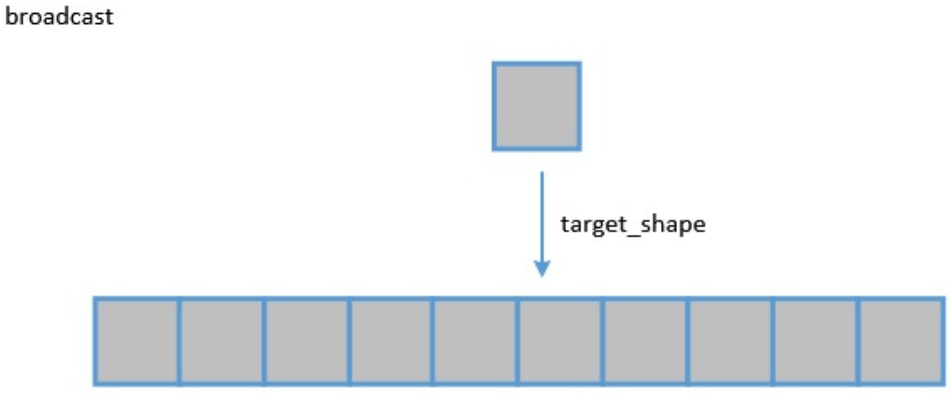
NOTICE

Usage limitation: Due to the data arrangement limitation on the CCE computing platform, the data after the reduction operation needs to be rearranged for subsequent operations. Therefore, when different types of APIs are used, no vector operation can be performed after the reduction operation.

Broadcast API

The broadcast operation is mainly used to process two tensors with different shapes and broadcast an operand with a lower dimension according to a higher-dimension operand. The element-wise calculation is performed after the dimensions of the two operands are the same.

Broadcast a smaller tensor to a larger tensor. For details, see **3.28 te.lang.cce.broadcast(var, shape, output_dtype=None)**.

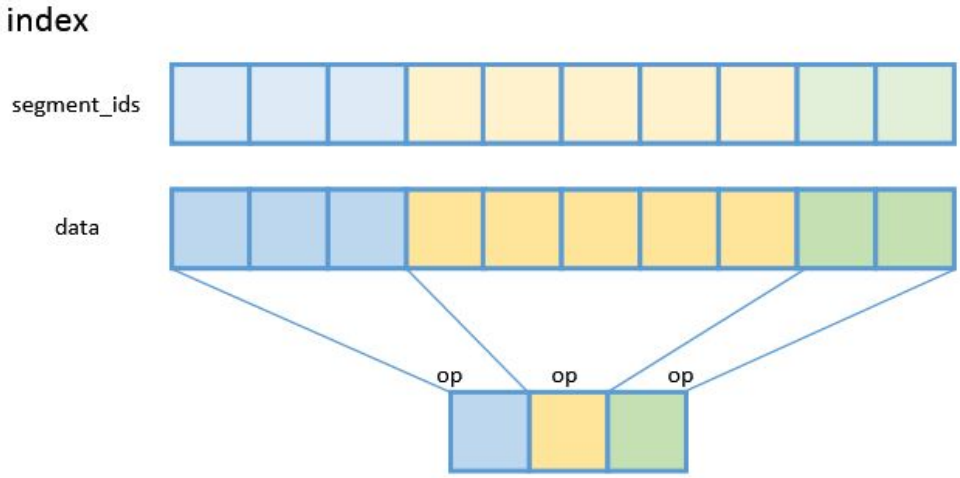


Index Operation APIs

Index operations are used to perform calculations on tensors by segment, such as obtaining the sum value, average value, inner product, maximum value, and minimum value.

- Sum up tensors by segment. For details, see **3.29 te.lang.cce.unsorted_segment_sum(tensor, segment_ids, num_segments, init_value=0)**.

- Calculate the average value of tensors by segment. For details, see **3.30 te.lang.cce.unsorted_segment_mean(tensor, segment_ids, num_segments, init_value=0)**.
- Calculate the inner product of tensors by segment. For details, see **3.31 te.lang.cce.unsorted_segment_prod(tensor, segment_ids, num_segments, init_value=0)**.
- Calculate the minimum value of tensors by segment. For details, see **3.32 te.lang.cce.unsorted_segment_min(tensor, segment_ids, num_segments, init_value=0)**.
- Calculate the maximum value of tensors by segment. For details, see **3.33 te.lang.cce.unsorted_segment_max(tensor, segment_ids, num_segments, init_value=0)**.



Concat API

The concat API is used to concatenate multiple input tensors along an axis.

Concatenate the tensors along an axis. For details, see **3.34 te.lang.cce.concat(raw_tensors, axis)**.

Convolution API

Convolution APIs are used to implement convolutional operators.

Implement convolution. For details, see **3.35 te.lang.cce.conv(*args)**.

4D/5D Conversion APIs

These APIs are used for conversion between 4D NCHW and 5D NC1HWC0.

- Convert from 4D to 5D. For details, see **3.36 te.lang.cce.compute_four2five(input, raw_shape_4D)**.
- Convert from 5D to 4D. For details, see **3.37 te.lang.cce.compute_five2four(input, raw_shape_4D)**.

Matrix Multiplication API

This API is used to implement matrix multiplication.

For details, see **3.38 te.lang.cce.matmul(tensor_a, tensor_b, trans_a=False, trans_b=False, alpha_num=1.0, beta_num=0.0, tensor_c=None)**.

Convergent Operator API

The TE operator convergence function pre-processes information such as the convergence capability and input parameters of a single operator. Therefore, the specified API needs to be called to generate the *.so and *.json files of a single operator.

For details, see **5.1 bool BuildTeCustomOp(std::string ddkVer, std::string opName, std::string opPath, std::string opFuncName, const char *format, ...)**.

3 Compute APIs

3.1 te.lang.cce.vadd(lhs, rhs)

Adds two tensors by element. The data types of the elements must be the same. The supported types are float16, float32, and int32. The data types int8 and uint8 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: lhs + rhs, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
```



```
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vadd(data1, data2)
```

3.2 te.lang.cce.vsub(lhs, rhs)

Subtracts one tensor from the other tensor by element. The data types of the elements must be the same. The supported types are float16, float32, and int32. The data types int8 and uint8 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: lhs – rhs, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vsub(data1, data2)
```

3.3 te.lang.cce.vmul(lhs, rhs)

Multiplies two tensors by element. The data types of the elements must be the same. The supported types are float16, float32, and int32. The data types int8 and uint8 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: lhs x rhs, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmul(data1, data2)
```


3.4 te.lang.cce.vmin(lhs, rhs)

Compares two tensors by element and chooses the smaller value. The data types of the elements must be the same. The supported types are float16, float32, and int32. The data types int8 and uint8 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: result tensor, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmin(data1, data2)
```

3.5 te.lang.cce.vmax(lhs, rhs)

Compares two tensors by element and chooses the larger value. The data types of the elements must be the same. The supported types are float16, float32, and int32. The data types int8 and uint8 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: result tensor, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmax(data1, data2)
```


3.6 te.lang.cce.vor(lhs, rhs)

Performs bitwise OR on two tensor elements. The data types of the elements must be the same. The supported types are int16 and uint16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: bitwise OR between **lhs** and **rhs**, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "int16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vor(data1, data2)
```

3.7 te.lang.cce.vand(lhs, rhs)

Performs bitwise AND on two tensor elements. The data types of the elements must be the same. The supported types are int16 and uint16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type

Return Value

res_tensor: bitwise AND between **lhs** and **rhs**, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "int16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vand(data1, data2)
```

3.8 te.lang.cce.vadds(raw_tensor, scalar)

Adds a scalar to each element in **raw_tensor**. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16. If the data

type of a scalar is different from that of **raw_tensor**, the data type is converted into the corresponding data type.

This API is defined in **elewise_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **scalar**: coefficient to be added to an element in **raw_tensor**, scalar type

Return Value

res_tensor: raw_tensor + scalar, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vadds(data, scalar)
```

3.9 te.lang.cce.vmuls(raw_tensor, scalar)

Multiplies each element in **raw_tensor** by a scalar. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16. If the data type of a scalar is different from that of **raw_tensor**, the data type is converted into the corresponding data type.

This API is defined in **elewise_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **scalar**: coefficient by which the element in **raw_tensor** is multiplied, scalar type

Return Value

res_tensor: raw_tensor x scalar, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vmuls(data, scalar)
```

3.10 te.lang.cce.vlog(raw_tensor)

Performs the logarithmic ln(x) operation on each element in **raw_tensor**. The supported type is float16. The data types int8, uint8, int32, and float32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: ln(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vlog(data)
```

3.11 te.lang.cce.vexp(raw_tensor)

Performs the natural exponential operation e^x on each element in a tensor. The supported type is float16. The data types int8, uint8, int32, and float32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: e^(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vexp(data)
```

3.12 te.lang.cce.vabs(raw_tensor)

Performs the absolute value operation |x| on each element in a tensor. The supported type is float16. The data types int8, uint8, int32, and float32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: |(raw_tensor)|, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vabs(data)
```

3.13 te.lang.cce.vrec(raw_tensor)

Performs the reciprocal operation 1/x on each element in a tensor. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: 1/ raw_tensor, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vrec(data)
```

3.14 te.lang.cce.cast_to(data, dtype, f1628IntegerFlag=False)

Converts a data type, specifically, converts a data type into **dtype**.

This API is defined in **common.py**.

The following data type conversions are supported:

Table 3-1 Supported data type conversions

Source Data Type	Destination Data Type
float32	float16
float32	int8
float32	uint8
float16	float32
float16	int8
float16	uint8
float16	int32

Source Data Type	Destination Data Type
int8	float16
int8	uint8
int32	float16
int32	int8
int32	uint8

Parameter Description

- **data**: input tensor, tvm.tensor type
- **dtype**: destination data type, string type
- **f1628IntegerFlag**: The default value is **False**. If the decimal part of the data before conversion is 0, set **f1628IntegerFlag** to **True**. If the decimal part of the data before conversion is not 0, set **f1628IntegerFlag** to **False**.

Return Value

res_tensor: data after conversion, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.cast_to(data,"float32")
```

3.15 te.lang.cce.vrelu(raw_tensor)

Performs the ReLU operation each element in a tensor. The supported type is float16. The data types int8, uint8, int32, and float32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: relu(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vrelu(data)
```


3.16 te.lang.cce.vnot(raw_tensor)

Performs bitwise NOT on each element in a tensor. The supported types are int16 and uint16.

This API is defined in **elewise_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: bitwise NOT on **raw_tensor**, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "int16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vnot(data)
```

3.17 te.lang.cce.vaxpy(lhs, rhs, scalar)

Multiplies each element in **lhs** by a scalar and adds the corresponding element in **rhs**. The data types of **lhs** and **rhs** must be the same. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

If the data type of the scalar is different from that of the tensor, the data type of the tensor prefers.

This API is defined in **elewise_compute.py**.

Parameter Description

- **lhs**: left tensor, tvm.tensor type
- **rhs**: right tensor, tvm.tensor type
- **scalar**: coefficient by which the element in **lhs** is multiplied, scalar type

Return Value

res_tensor: lhs x scalar + rhs, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vaxpy(data1, data2, scalar)
```


3.18 te.lang.cce.vmla(x, y, z)

Multiplies each element in **x** by a corresponding element in **y**, and then adds a corresponding element in **z**. The data types of the three tensors must be the same. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **x**: tensor, tvm.tensor type
- **y**: tensor, tvm.tensor type
- **z**: tensor, tvm.tensor type

Return Value

res_tensor: x * y + z, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmla(data1, data2, data3)
```

3.19 te.lang.cce.vmadd(x, y, z)

Multiplies each element in **x** by a corresponding element in **z**, and then adds a corresponding element in **y**. The data types of the three tensors must be the same. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **x**: tensor, tvm.tensor type
- **y**: tensor, tvm.tensor type
- **z**: tensor, tvm.tensor type

Return Value

res_tensor: x * z + y, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
```



```
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmadd(data1, data2, data3)
```

3.20 te.lang.cce.vmaddrelu(x, y, z)

Multiplies each element in **x** by a corresponding element in **z**, adds a corresponding element in **y**, and then performs the ReLU operation. The data types of the three tensors must be the same. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

This API is defined in **elewise_compute.py**.

Parameter Description

- **x**: tensor, tvm.tensor type
- **y**: tensor, tvm.tensor type
- **z**: tensor, tvm.tensor type

Return Value

res_tensor: relu(x * z + y), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmaddrelu(data1, data2, data3)
```

3.21 te.lang.cce.ceil(raw_tensor)

Rounds up each element in **raw_tensor**. The supported type is float16. The data type float32 is converted to float16. The output is int32.

This API is defined in **cast_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: ceil(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.ceil(data)
```


3.22 te.lang.cce.floor(raw_tensor)

Rounds down each element in **raw_tensor**. The supported type is float16. The data type float32 is converted to float16. The output is int32.

This API is defined in **cast_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: floor(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.floor(data)
```

3.23 te.lang.cce.round(raw_tensor)

Performs banker's rounding on each element in **raw_tensor**. When the fractional part is 0.5, the element is rounded to the nearest even number. For example, 1.5 rounds up to 2.0 and 2.5 rounds down to 2.0. The supported type is float16. The data type float32 is converted to float16. The output is int32.

This API is defined in **cast_compute.py**.

Parameter Description

raw_tensor: input tensor, tvm.tensor type

Return Value

res_tensor: round(raw_tensor), tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.round(data)
```

3.24 te.lang.cce.sum(raw_tensor, axis, keepdims=False)

Performs the sum operation based on a certain axis to reduce dimensions. The supported types are float16 and float32. The data types int8, uint8, and int32 are converted to float16.

This API is defined in **reduction_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **axis**: axis for the reduce operation. The value range is [–d, d – 1]. The parameter **d** indicates the dimension of **raw_tensor**, int or list type
- **keepdims**: The default value is **False**, indicating that the length of the operation axis is 0 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=False, the shape after reduce is (10, 10). If this parameter is set to **True**, the length of the operation axis is set to 1 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=True, the shape after reduce is (10, 10, 1).

Return Value

res_tensor: tensor after the sum value is obtained, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.sum(data, axis=1)
```

3.25 te.lang.cce.reduce_min(raw_tensor, axis, keepdims=False)

Calculates the minimum value based on a certain axis to reduce dimensions. The supported type is float16. The data types int8, uint8, int32 and float32 are converted to float16.

This API is defined in **reduction_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **axis**: axis for the reduce operation. The value range is [–d, d – 1]. The parameter **d** indicates the dimension of **raw_tensor**, int or list type
- **keepdims**: The default value is **False**, indicating that the length of the operation axis is 0 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=False, the shape after reduce is (10, 10). If this parameter is set to **True**, the length of the operation axis is set to 1 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=True, the shape after reduce is (10, 10, 1).

Return Value

res_tensor: tensor after the minimum value is obtained, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_min(data, axis=1)
```


3.26 te.lang.cce.reduce_max(raw_tensor, axis, keepdims=False)

Calculates the maximum value based on a certain axis to reduce dimensions. The supported type is float16. The data types int8, uint8, int32 and float32 are converted to float16.

This API is defined in **reduction_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **axis**: axis for the reduce operation. The value range is [–d, d – 1]. The parameter **d** indicates the dimension of **raw_tensor**, int or list type
- **keepdims**: The default value is **False**, indicating that the length of the operation axis is 0 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=False, the shape after reduce is (10, 10). If this parameter is set to **True**, the length of the operation axis is set to 1 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=True, the shape after reduce is (10, 10, 1).

Return Value

res_tensor: tensor after the maximum value is obtained, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_max(data, axis=1)
```

3.27 te.lang.cce.reduce_prod(raw_tensor, axis, keepdims=False)

Performs multiplication based on a certain axis to reduce dimensions. The supported type is float16. The data types int8, uint8, int32, and float32 are converted to float16.

This API is defined in **reduction_compute.py**.

Parameter Description

- **raw_tensor**: input tensor, tvm.tensor type
- **axis**: axis for the reduce operation. The value range is [–d, d – 1]. The parameter **d** indicates the dimension of **raw_tensor**, int type
- **keepdims**: The default value is **False**, indicating that the axis length is 0 after the reduce operation. For example, if the original shape is (10, 10, 10) and keepdims=False, the shape after the reduce operation is (10, 10). If this parameter is set to **True**, the axis length is set to 1 after the reduce operation.

For example, if the original shape is (10, 10, 10) and keepdims=True, the shape after the reduce operation is (10, 10, 1).

Return Value

res_tensor: tensor after multiplication is performed based on a certain axis, tvm.tensor type

Calling Example

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_prod(data, axis=1)
```

3.28 te.lang.cce.broadcast(var, shape, output_dtype=None)

Broadcasts **var** to the tensor whose size is **shape**. The data type of the output is specified by **output_dtype**. **var** can be a scalar or tensor. The shape of **var** must have the same length as the second parameter **shape** . The size of each dimension must be either the same as that of **shape**, or be 1. When the size is 1, the dimension is broadcast to be the same as **shape**. For example, if the dimension of **var** is (2, 1, 64) and the value of **shape** is (2, 128, 64), the calculated dimension of **var** is (2, 128, 64). The supported types are float16, float32, and int32.

This API is defined in **broadcast_compute.py**.

Parameter Description

- **var**: data to be broadcast, scalar or tensor type
- **shape**: target shape for the broadcast operation
- **output_dtype**: output data type. The default value is **var.dtype**.

Return Value

res_tensor: tensor obtained after **var** extension. The shape is specified by the parameter **shape**. The data type is **output_dtype**.

Calling Example

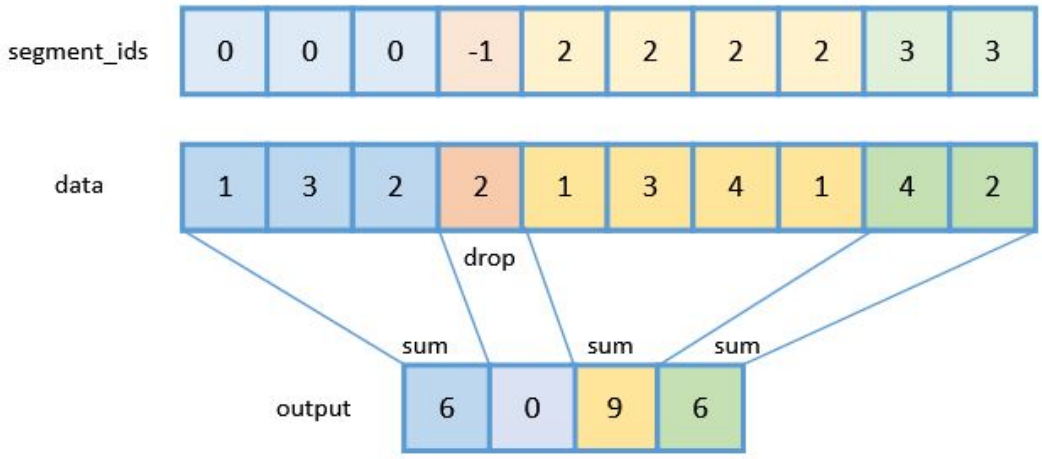
```
outshape = (1024,1024)
shape = (1024,1)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.broadcast(data, outshape)
```

3.29 te.lang.cce.unsorted_segment_sum(tensor, segment_ids, num_segments, init_value=0)

Uses the array **segment_ids** to sum up tensors by segment. Assuming that the input is **data** and the output is **output**, then output[i] = sum(data[j...]), where **j...** is an array, and element **j** meets the following requirement: segment_ids[j] == i.

If a subscript **i** does not appear in **segment_ids**, then output[i] = init_value. For example, in the following figure, if **1** does not appear in **segment_ids**, then output[1] = 0.

If a value in **segment_ids** is a negative number, the value of **data** in the corresponding position is discarded. For example, in the following figure, segment_ids[3] = –1, indicating that the value of **data[3]** is discarded and is not involved in the calculation.



The length of **segment_ids** must be the same as the length of the first dimension of **data**. The value of **num_segments** must be greater than or equal to the value of **segment_ids** plus 1.

The supported types are float16, float32, and int32.

This API is defined in **segment_compute.py**.

Parameter Description

- **tensor**: input tensor, which is of the float16, float32, or int32 type
- **segment_ids**: one-dimensional array. This array is used to segment the input tensor. The length of the array must be the same as the length of the first dimension of the input tensor. The array can be sequenced or unsequenced.
- **num_segments**: length of the first dimension of the output tensor. Its value must be greater than or equal to the value of **segment_ids** plus 1.
- **init_value**: default value of the output when a subscript in **segment_ids** does not exist. It is determined according to the implementation of the operator. The default value is **0**.

Return Value

res_tensor: tensor after calculation

Calling Example

```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
```



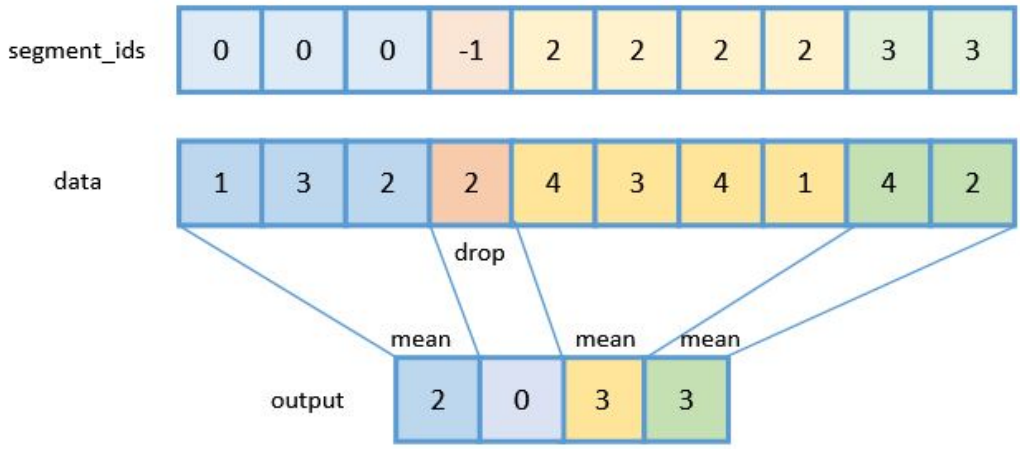
```
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_sum(data, segment_ids, num_segments)
res.shape = (6,1024)
# res[0] = 0
# res[1] = data[0] + data[1]
# res[2] = 0
# res[3] = 0
# res[4] = data[2]
# res[5] = data[3] + data[4]
```

3.30 te.lang.cce.unsorted_segment_mean(tensor, segment_ids, num_segments, init_value=0)

Uses the array **segment_ids** to calculate the average value of tensors by segment. Assuming that the input is **data** and the output is **output**, then output[i] = (1/len(j...)) sum(data[j...]), where **j...** is an array, and element **j** meets the following requirement: segment_ids[j] == i.

If a subscript **i** does not appear in **segment_ids**, then output[i] = init_value. For example, in the following figure, if **1** does not appear in **segment_ids**, then output[1] = 0.

If a value in **segment_ids** is a negative number, the value of **data** in the corresponding position is discarded. For example, in the following figure, segment_ids[3] = –1, indicating that the value of **data[3]** is discarded and is not involved in the calculation.



The length of **segment_ids** must be the same as the length of the first dimension of **data**. The value of **num_segments** must be greater than or equal to the value of **segment_ids** plus 1.

The supported types are float16, float32, and int32.

This API is defined in **segment_compute.py**.

Parameter Description

- **tensor**: input tensor, which is of the float16, float32, or int32 type
- **segment_ids**: one-dimensional array. This array is used to segment the input tensor. The length of the array must be the same as the length of the first dimension of the input tensor. The array can be sequenced or unsequenced.

- **num_segments**: length of the first dimension of the output tensor. Its value must be greater than or equal to the value of **segment_ids** plus 1.
- **init_value**: default value of the output when a subscript in **segment_ids** does not exist. It is determined according to the implementation of the operator. The default value is **0**.

Return Value

res_tensor: tensor after calculation

Calling Example

```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,5,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_mean(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 0
# res[1] = (data[0] + data[1]) / 2
# res[2] = 0
# res[3] = 0
# res[4] = 0
# res[5] = (data[2] + data[3] + data[4]) / 3
```

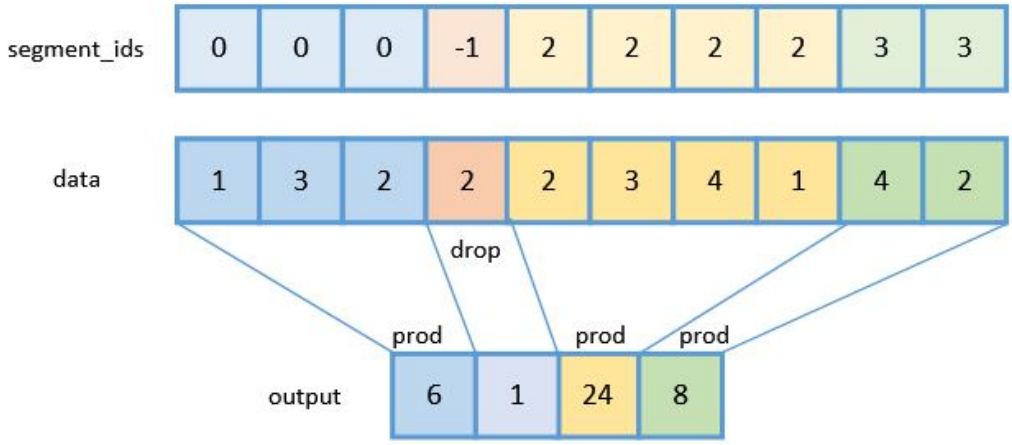
3.31 te.lang.cce.unsorted_segment_prod(tensor, segment_ids, num_segments, init_value=0)

Uses the array **segment_ids** to calculate the inner product of tensors by segment. Assuming that the input is **data** and the output is **output**, then output[i] = product(data[j...]), where **j...** is an array, and element **j** meets the following requirement: segment_ids[j] == i.

> **NOTE**
>
> The parameter **product** indicates the inner product, that is, all elements in **data[j...]** are multiplied by each other.

If a subscript **i** does not appear in **segment_ids**, then output[i] = init_value. For example, in the following figure, if **1** does not appear in **segment_ids**, then output[1] = 0.

If a value in **segment_ids** is a negative number, the value of **data** in the corresponding position is discarded. For example, in the following figure, segment_ids[3] = –1, indicating that the value of **data[3]** is discarded and is not involved in the calculation.



The length of **segment_ids** must be the same as the length of the first dimension of **data**. The value of **num_segments** must be greater than or equal to the value of **segment_ids** plus 1.

The supported types are float16, float32, and int32.

This API is defined in **segment_compute.py**.

Parameter Description

- **tensor**: input tensor, which is of the float16, float32, or int32 type
- **segment_ids**: one-dimensional array. This array is used to segment the input tensor. The length of the array must be the same as the length of the first dimension of the input tensor. The array can be sequenced or unsequenced.
- **num_segments**: length of the first dimension of the output tensor. Its value must be greater than or equal to the value of **segment_ids** plus 1.
- **init_value**: default value of the output when a subscript in **segment_ids** does not exist. It is determined according to the implementation of the operator. The default value is **0**.

Return Value

res_tensor: tensor after calculation

Calling Example

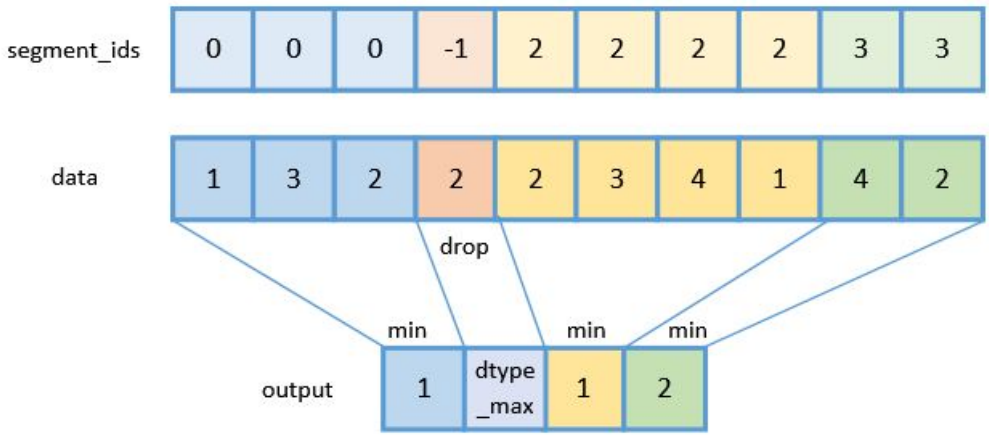
```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_prod(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 1
# res[1] = (data[0] * data[1])
# res[2] = 1
# res[3] = 1
# res[4] = data[2]
# res[5] = (data[3] * data[4])
```


3.32 te.lang.cce.unsorted_segment_min(tensor, segment_ids, num_segments, init_value=0)

Uses the array **segment_ids** to calculate the minimum value of tensors by segment. Assuming that the input is **data** and the output is **output**, then output[i] = min(data[j...]), where **j...** is an array, and element **j** meets the following requirement: segment_ids[j] == i.

If a subscript **i** does not appear in **segment_ids**, then output[i] = init_value. For example, in the following figure, if **1** does not appear in **segment_ids**, then output[1] = 0.

If a value in **segment_ids** is a negative number, the value of **data** in the corresponding position is discarded. For example, in the following figure, segment_ids[3] = –1, indicating that the value of **data[3]** is discarded and is not involved in the calculation.



The length of **segment_ids** must be the same as the length of the first dimension of **data**. The value of **num_segments** must be greater than or equal to the value of **segment_ids** plus 1.

The supported types are float16 and float32.

This API is defined in **segment_compute.py**.

Parameter Description

- **tensor**: input tensor, which is of the float16, float32, or int32 type
- **segment_ids**: one-dimensional array. This array is used to segment the input tensor. The length of the array must be the same as the length of the first dimension of the input tensor. The array can be sequenced or unsequenced.
- **num_segments**: length of the first dimension of the output tensor. Its value must be greater than or equal to the value of **segment_ids** plus 1.
- **init_value**: default value of the output when a subscript in **segment_ids** does not exist. It is determined according to the implementation of the operator. The default value is **0**.

Return Value

res_tensor: tensor after calculation

Calling Example

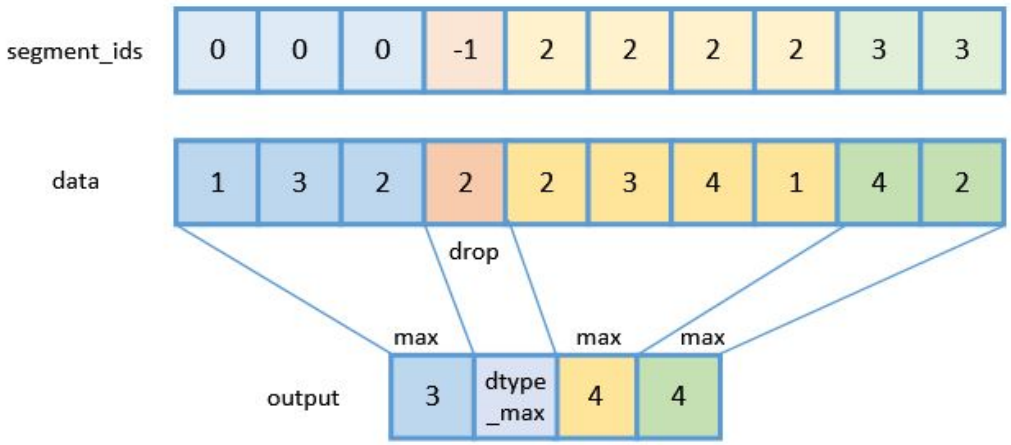
```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_min(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 65504 (maximum value of float16)
# res[1] = min(data[0], data[1])
# res[2] = 65504
# res[3] = 65504
# res[4] = data[2]
# res[5] = min(data[3], data[4])
```

3.33 te.lang.cce.unsorted_segment_max(tensor, segment_ids, num_segments, init_value=0)

Uses the array **segment_ids** to calculate the maximum value of tensors by segment. Assuming that the input is **data** and the output is **output**, then output[i] = max(data[j...]), where **j...** is an array, and element **j** meets the following requirement: segment_ids[j] == i.

If a subscript **i** does not appear in **segment_ids**, then output[i] = init_value. For example, in the following figure, if **1** does not appear in **segment_ids**, then output[1] = 0.

If a value in **segment_ids** is a negative number, the value of **data** in the corresponding position is discarded. For example, in the following figure, segment_ids[3] = –1, indicating that the value of **data[3]** is discarded and is not involved in the calculation.



The length of **segment_ids** must be the same as the length of the first dimension of **data**. The value of **num_segments** must be greater than or equal to the value of **segment_ids** plus 1.

The supported types are float16, float32, and int32.

This API is defined in **segment_compute.py**.

Parameter Description

- **tensor**: input tensor, which is of the float16, float32, or int32 type
- **segment_ids**: one-dimensional array. This array is used to segment the input tensor. The length of the array must be the same as the length of the first dimension of the input tensor. The array can be sequenced or unsequenced.
- **num_segments**: length of the first dimension of the output tensor. Its value must be greater than or equal to the value of **segment_ids** plus 1.
- **init_value**: default value of the output when a subscript in **segment_ids** does not exist. It is determined according to the implementation of the operator. The default value is **0**.

Return Value

res_tensor: tensor after calculation

Calling Example

```
import tvm
import te.lang.cce shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1",type=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_max(data,segment_ids,num_segments)
# res.shape = (6,1024)
# res[0] = 65504 (maximum value of float16)
# res[1] = max(data[0], data[1])
# res[2] = 65504
# res[3] = 65504
# res[4] = data[2]
# res[5] = max(data[3], data[4])
```

3.34 te.lang.cce.concat(raw_tensors, axis)

Reconcatenates multiple input tensors based on the specified axis.

raw_tensors indicates multiple input tensors. The data types are the same.

If raw_tensors[i].shape = [D0, D1, ... Daxis(i), ...Dn], the shape of the output after the concatenation is established based on **axis** is: as follows: [D0, D1, ... Raxis, ...Dn].

Where, Raxis = sum(Daxis(i)).

For input tensors, the dimensions of other axes must be the same except for **axis**.

For example:

```
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
concat([t1, t2], 0)  # [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
concat([t1, t2], 1)  # [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]]

# The shape of tensor t1 is [2, 3].
```



```
# The shape of tensor t2 is [2, 3].
concat([t1, t2], 0).shape  # [4, 3]
concat([t1, t2], 1).shape  # [2, 6]
```

The parameter **axis** can also be a negative number, indicating the **axis** + **len(shape)** axis, which is calculated from the end of the dimension.

For example:

```
t1 = [[[1, 2], [2, 3]], [[4, 4], [5, 3]]]
t2 = [[[7, 4], [8, 4]], [[2, 10], [15, 11]]]
concat([t1, t2], -1)
```

The output is as follows:

```
[[[ 1,  2,  7,  4],
  [ 2,  3,  8,  4]],

 [[ 4,  4,  2, 10],
  [ 5,  3, 15, 11]]]
```

The supported data types are as follows: int8, uint8, int16, int32 float16, and float32.

This API is defined in **concat_compute.py**.

Parameter Description

- **raw_tensors**: tensor list, list type. The element is tvm.tensor, and the last dimension of tensor shape must be 32-byte aligned.
- **axis**: axis based on which the concat operation is performed. The value range is [–d, d–1]. The parameter **d** indicates the dimension of **raw_tensor**.

Return Value

res_tensor: tensor after reconcatenation is implemented, tvm.tensor type

Calling Example

```
import tvm
import te.lang.cce
shape1 = (64,128)
shape1 = (64,128)
input_dtype = "float16"
data1 = tvm.placeholder(shape1, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape2, name="data1", dtype=input_dtype)
data = [data1, data2]
res = te.lang.cce.concat(data, 0)
# res.shape = (128,128)
```

3.35 te.lang.cce.conv(*args)

Computes 2D convolution when the 4D input and **filter** are specified.

The formats of the input tensor and the filter tensor must be NCHW.

The interface supports **bias**, and the supported data type is float16.

This API is defined in **conv_compute.py**.

Parameter Description

***args**: list. The value is variable.

- The parameter list for the bias scenario is as follows: (**hasBias** must be set to **True**)

  A, W, B, res_dtype, padh, padw, strideh, stridew, hasBias

- The parameter list for the non-bias scenario is as follows: (**hasBias** must be set to **False**)

  A, W, res_dtype, padh, padw, strideh, stridew, hasBias

  Where,

  - **A**: input tensor, that is, feature map for convolutional calculation
  - **W**: filter tensor, that is, convolutional core for convolutional calculation
  - **B**: bias tensor, that is, offset of convolution calculation
  - **res_dtype**: data type of the output tensor, that is, data type of the convolution calculation result.
  - **padh**: height of padding, that is, number of padding times in the H direction of the feature map during convolutional calculation
  - **padw**: width of padding, that is, number of padding times in the W direction of the feature map during convolutional calculation
  - **strideh**: stride in the H direction, that is, movement stride of the convolutional calculation filter in the H direction on the feature map
  - **stridew**: stride in the W direction, that is, movement stride of the convolutional calculation filter in the W direction on the feature map
  - **hasBias**: whether **bias** is contained

- Constraints: Assume that the shape of the feature map is (Fn, Fc, Fh, Fw), the shape of **filter** is (Wn, Wc, Wh, Ww), and the output shape of the convolution result is (On, Oc, Oh, Ow). **padh** is **Ph**, **padw** is **Pw**, **strideh** is **Sh**, and **stridew** is **Sw**. These parameters must meet the following requirements:

  - Fc = Wc
  - On = Fn
  - Oc = Wn
  - Oh = ((Fh + 2Ph - Wh) / Sh) + 1
  - Ow = ((Fw + 2Pw - Ww) / Sw) + 1

Return Value

res_tensor: tensor for convolutional calculation, that is, output of convolutional calculation

Calling Example

```
import tvm
import te.lang.cce
shape_in = (64,64)
shape_w = (3,3)
in_dtype = "float16"
A = tvm.placeholder(shape_in, name='A', dtype=in_dtype)
W = tvm.placeholder(shape_w, name='W', dtype=in_dtype)
b_shape = (shape_w[0], )
```



```
B = tvm.placeholder(b_shape, name='B', dtype=res_dtype)
padh = 0
padw = 0
strideh = 1
stridew = 1
res = te.lang.cce.conv(A, W, B, in_dtype, padh, padw, strideh, stridew, True)
# res = A * W + B
```

3.36 te.lang.cce.compute_four2five(input, raw_shape_4D)

Converts the 4D data format **NCHW** to the 5D data format **NC1HWC0**. The supported type is float16.

This API is defined in **dim_conv.py**.

Parameter Description

- **input**: input tensor, 4D (N,C,H,W), tvm.tensor type
- **raw_shape_4D**: dimension of the input tensor

Return Value

res_tensor: tensor after the data format is converted into the 5D format (N,C1,H,W,C0), tvm.tensor type

Calling Example

```
import tvm
import te.lang.cce
raw_shape = (N,C,H,W)
in_dtype = "float16"
input = tvm.placeholder(raw_shape, name='input', dtype=in_dtype)
res = te.lang.cce.compute_four2five(input, raw_shape)
# res.shape = (N,(C+15)//16,H,W,16)
```

3.37 te.lang.cce.compute_five2four(input, raw_shape_4D)

Converts the 5D data format **NC1HWC0** to the 4D data format **NCHW**. The supported type is float16.

This API is defined in **dim_conv.py**.

Parameter Description

- **input**: input tensor, 5D format (N,C1,H,W,C0), tvm.tensor type
- **raw_shape_4D**: dimension of the tensor after conversion

Return Value

res_tensor: tensor after the data format is converted into the 4D format (N,C,H,W), tvm.tensor type

Calling Example

```
import tvm
import te.lang.cce
raw_shape = (N,C,H,W)
input_shape = (N,(C+15)//16,H,W,16)
in_dtype = "float16"
input = tvm.placeholder(input_shape, name='input', dtype=in_dtype)
res = te.lang.cce.compute_five2four(input, raw_shape)
# res.shape = (N,C,H,W)
```

3.38 te.lang.cce.matmul(tensor_a, tensor_b, trans_a=False, trans_b=False, alpha_num=1.0, beta_num=0.0, tensor_c=None)

Performs matrix multiplication as follows: tensor_c=alpha_num * trans_a(tensor_a) * trans_b(tensor_b) + beta_num * tensor_c.

For **tensor_a** and **tensor_b**, the last two dimensions of **shape** (after transposition) must meet the following condition: Matrix multiplication (M, K) * (K, N) = (M, N). Only one batch is supported. The **tensor_a** data layout must comply with the fractal structure of L0A, and the **tensor_b** data layout must comply with the fractal structure of L0B. In mini mode, only the float16 data type is supported.

This API is defined in **mmad_compute.py**.

Parameter Description

- **tensor_a**: matrix A, tvm.tensor type
- **tensor_b**: matrix B, tvm.tensor type
- **trans_a**: whether matrix A is transposed, Boolean type
- **trans_b**: whether matrix B is transposed, Boolean type
- **alpha_num**: matrix A*B coefficient. Only 1.0 is supported.
- **beta_num**: matrix C coefficient. Only 0.0 is supported.
- **tensor_c**: matrix C, tvm.tensor type. Since **beta_num** supports only 0.0, this parameter is used as a reserved interface.

Return Value

tensor_c: tensor calculated based on the relational calculation, tvm.tensor type

Calling Example

```
import tvm
import te.lang.cce
a_shape = (1024, 256)
b_shape = (256, 512)
a_fractal_shape = (a_shape[0] // 16, a_shape[1] // 16, 16, 16)
b_fractal_shape = (b_shape[0] // 16, b_shape[1] // 16, 16, 16)
in_dtype = "float16"
tensor_a = tvm.placeholder(a_fractal_shape, name='tensor_a', dtype=in_dtype)
tensor_b = tvm.placeholder(a_fractal_shape, name='tensor_b', dtype=in_dtype)
res = te.lang.cce.matmul(tensor_a, tensor_b, False, False)
```


4 Build APIs

The build APIs are used to generate the schedule object for the defined computation process and build the CCE operator file.

4.1 te.lang.cce.auto_schedule(outs)

Generates the schedule of DSL. This API is defined in **cce.py**.

Parameter Description

outs: computational graph description of the operator, that is, DSL

Return Value

schedule: computation schedule of the operator

Calling Example

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

shape = (28,28)
dtype = "float16"
# Define input.
data = tvm.placeholder(shape, name="data", dtype=dtype)
# Describe the computation process of the operator.
res = te.lang.cce.vabs(data)
with tvm.target.cce():
    # Generate the schedule object.
      sch = topi.generic.auto_schedule(res)
```

4.2 te.lang.cce.cce_build_code(sch, config_map = {})

Prints lower code or performs build for the schedule. This API is defined in **cce_schedule.py**.

Parameter Description

- **sch**: tvm.schedule, schedule to build or to print lower code
- **config_map**: build parameter configuration, which is a dictionary. The default value is **{}** and is used. The following keys are included:
  - **print_ir**: whether to print lower IR code. The default value is **True**.
  - **need_build**: whether build is performed. The default value is **True**.
  - **name**: operator name. The default value is **cce_op**.
  - **tensor_list**: list of input and output tensors of the operator. The input is the tensor object returned by the placeholder interface. The output is the calculated tensor object. The value is mandatory. Otherwise, an error is reported. In addition, this list determines the sequence of the parameters of the kernel function for operator generation, which is the same as the sequence of the input and output in the list.

Return Value

None

Calling Example

```
import te.lang.cce
from te import tvm
from topi import generic
# Define the input placeholder.
data = tvm.placeholder(shape, name="data", dtype=dtype)
with tvm.target.cce():
    # Describe the computation process of the operator.
    res = te.lang.cce.vabs(data)
    # Generate the schedule object.
    sch = generic.auto_schedule(res)
# Define build parameters.
config = {"print_ir" : True,
          "need_build" : True,
          "name" : "abs_28_28_float16",
          "tensor_list" : [data,res]
    }
# Build an operator.
te.lang.cce.cce_build_code(sch, config)
```


5 Convergence API

The TE operator convergence function pre-processes information such as the convergence capability and input parameters of a single operator. Therefore, a specified API needs to be called to generate a single operator. The following describes the usage and restrictions of the API in detail.

5.1 bool BuildTeCustomOp(std::string ddkVer, std::string opName, std::string opPath, std::string opFuncName, const char *format, ...)

Generates the *.o and *.json files of a single operator. You can view the API definitions in the **ddk/include/inc/custom/custom_op.h** file under the installation directory of the DDK package.

Parameter Description

- **ddkVer**: product version number

  Version number rule: **x.y.z.patch.B*****

  x: corresponding to the VR version

  y: corresponding to the C version

  z: number of releases

  path: patch number (optional)

  B***: internal B version number

  For the specific version number, see the configuration in the **ddk_info** file under the installation directory of the DDK package.
- **opName**: name of a single operator node on the network. The value cannot be empty and can contain only letters, digits, underscores (_), and hyphens (-).
- **opPath**: path of the operator file. The directory contains slashes (/) and file names without filename extensions. The value cannot be empty. The file name can contain only letters, digits, underscores (_), and hyphens (-).

- **opFuncName**: operator function name in the operator file. The function name can contain only letters, digits, underscores (_), and hyphens (-).
- **format**: input parameter of an operator, which has a variable length and must be consistent with the parameter sequence of the operator

Return Value

- **true**: A single operator is successfully generated.
- **false**: A single operator fails to be generated.

Calling Example

```
std::string FilePath   = "topi/cce/caffe_reduction_layer";
std::string FuncName   = "caffe_reduction_layer_cce";
std::string KernelName = "cce_reductionLayer_5_10_5_5_float16__1_SUMSQ_1_0";

// i => int; s => string; f => dobule; O => bool, and bool value is Py_True or Py_False
te::BuildTeCustomOp(1.1.1.patch.B001, "reduction", FilePath, FuncName,"(i,i,i,i), s, i, s, f, s", 5, 5, 5, 5,
"float16", 1, "SUM", 1.0,KernelName.c_str());
```


6 APIs that Compilation Depends on

The following APIs are used only during operator compilation, and are not directly called during operator writing: You can view the API definitions in the **ddk/include/inc/tensor_engine/cce_aicpu_3rd_party.h** file under the installation directory of the DDK package.

- __aicpu__ void *aicpu_malloc(unsigned int size)

  This API is used to allocate memory. The input parameter **size** indicates the size of memory to be allocated. The pointer to memory allocation is returned.
- __aicpu__ void aicpu_free(void * ptr)

  This API is used to release memory. **ptr** is the memory pointer.
- __aicpu__ double log(double x)

  This API is used to obtain the log value of **x** (double type).
- __aicpu__ double exp(double x)

  This API is used to obtain the EXP value of **x** (double type).
- __aicpu__ double round(double x)

  This API is used to round off the value of **x** (double type).
- __aicpu__ double floor(double x)

  This API is used to round down the value of **x** (double type).
- __aicpu__ double ceil(double x);

  This API is used to round up the value of **x** (double type).
- __aicpu__ double trunc(double x);

  This API is used to obtain the truncation value of **x** (double type).
- __aicpu__ double sqrt(double x);

  This API is used to obtain the square value of **x** (double type).
- __aicpu__ float logf(float x)

  This API is used to obtain the log value of **x** (double type).
- __aicpu__ float expf(float x)

  This API is used to obtain the EXP value of **x** (float type).
- __aicpu__ float roundf(float x)

  This API is used to round off the value of **x** (float type).

- __aicpu__ float floorf(float x)
  This API is used to round down the value of **x** (float type).
- __aicpu__ float ceilf(float x);
  This API is used to round up the value of **x** (float type).
- __aicpu__ float truncf(float x);
  This API is used to obtain the truncation value of **x** (float type).
- __aicpu__ float sqrtf(float x);
  This API is used to obtain the square value of **x** (float type).

7 Instructions

7.1 Usage Example

The following example shows how to concatenate the preceding APIs. It implements a simple operator that supports the float16 type to obtain an absolute value.

```
import te.lang.cce
from te import tvm
import topi
shape = (28,28)
dtype = "float16"
# Define input.
data = tvm.placeholder(shape, name="data", dtype=dtype)

# Describe the computation process of the operator.
res = te.lang.cce.vabs(data)

with tvm.target.cce():
    # Generate the schedule object.
     sch = topi.generic.auto_schedule(res)

# Define build parameters.
config = {"print_ir" : True,
          "need_build" : True,
          "name" : "abs_28_28_float16",
          "tensor_list" : [data,res]}
# Build an operator.
te.lang.cce.cce_build_code(sch, config)
```

7.2 Exception Handling

If an exception occurs when an API is executed, the error is usually caused by incorrect input parameters. The following example shows the error information when **tensor_list** is incomplete.

Code example:

```
data = tvm.placeholder(shape, name="data", dtype=inp_dtype)
with tvm.target.cce():
```



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

The following error information is displayed:

```
Traceback (most recent call last):
  File "llt/tensor_engine/ut/testcase_python/tf_abs/test_tf_abs_cce.py", line 71, in test_cce_tf_abs_99991_fp16
    tf_abs_cce((99991,), dtype = "Float16", need_build = False, need_print = False, kernel_name =
"cce_tf_abs")
  File "/home1/repotvm/tensor_engine/topi/python/topi/cce/tf_abs.py", line 68, in tf_abs_cce
    te.lang.cce.cce_build_code(sch, config)
  File "/home1/repotvm/tensor_engine/python/te/lang/cce/te_schedule/cce_schedule.py", line 381, in
cce_build_code
    _build(sch, tensor_list, local_config_map["name"])
  File "/home1/repotvm/tensor_engine/python/te/lang/cce/te_schedule/cce_schedule.py", line 338, in _build
    mod = tvm.build(sch, tensor_list, device, name=name)
  File "/home1/repotvm/tensor_engine/python/te/tvm/build_module.py", line 432, in build
    binds=binds)
  File "/home1/repotvm/tensor_engine/python/te/tvm/build_module.py", line 353, in lower
    stmt = ir_pass.StorageFlatten(stmt, binds, 64)
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/function.py", line 280, in my_api_func
    return flocal(*args)
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/_ctypes/function.py", line 183, in __call__
    ctypes.byref(ret_val), ctypes.byref(ret_tcode)))
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/base.py", line 66, in check_call
    raise TVMError(py_str(_LIB.TVMGetLastError()))
TVMError: [17:12:02] /home1/repotvm/tensor_engine/src/pass/storage_flatten.cc:249: Check failed: it !=
buf_map_.end() Cannot find allocated buffer for placeholder(data, 0x27d7290)
```

The problem is solved after the parameter is modified as follows:

```
"tensor_list" : [data, res]
```


8 Appendix

8.1 Change History

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

