深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler Graph IR

关于作者以及免责声明见序章开头。

题图源自网络，侵删。

本篇将讨论GraphCompiler中的Graph IR的具体定义。在本系列的第二篇中，已经介绍了Graph IR和Tensor IR，以及它们的关系。详见：

先来回顾以下Graph IR。Graph IR是用来描述计算图的。里面保存了Op之间的连接关系的。Graph IR由Op（算子）和Logical Tensor组成。一个Op即表示对于输入Tensor进行的计算，例如add，matmul等等。它可以有多个Tensor作为输入，也可以输出多个Tensor。每个Tensor只会有一个Owner，表示生成这个Tensor的Op。Op之间则是通过Tensor相互连接-一个Op的输出Tensor可以作为另一个Op的输入Tensor。Graph IR组成的计算图应该是一张有向无环图（DAG），这个图的入口节点必须是input_op或者constant_op。这两种特殊的Op不表示任何计算，而是表示整个图的输入参数和常量。Graph IR的输出节点（即DAG的Sink节点）应为output_op。它的所有输入tensor都会变成计算图的输出。

接下来先讨论一下Graph、Op、Tensor在GraphCompiler中是如何存储和表示的。

Graph IR在GraphCompiler中C++上的定义

与Graph IR相关的pass和基础数据类型定义主要在目录 src/backend/graph_compiler/core/src/compiler/ir/graph下。Graph IR的基本定义在头文件graph.hpp中。我们通过自下而上的顺序来逐一讨论Graph IR中的Tensor、Op、Graph等概念。

Tensor

Tensor是Graph IR中最底层的概念，用来表示一个算子的输出。整个Graph的输入Tensor通过Input Op的“输出”表示。常量Tensor则是Constant Op的输出。整个Graph的输出Tensor则会是Output Op的输入。Graph Compiler通过Input、Output Op来标记整张图的输入和输出Tensor。

与Tensor IR的Tensor不同，Graph中的Tensor除了需要记录形状、数据类型等信息外，还需要记录Tensor本身和Graph中Op的连接关系，包括这个Tensor的“所有者”（是哪个Op的输出）、哪些Op会使用这个Tensor。GraphCompiler使用`graph_tensor`这个对象来表示Graph中的Tensor。为读者便于理解，简化后定义如下：

```
// the logical tensor for in the graph to represent a result value in the graph.
// It contains the tensor details (shape, dtype, etc.) and the connectivity of
// the value in the graph
struct graph_tensor {
    logical_tensor_t details_;
    sc_op *producer_owner_ {nullptr};
    // the op nodes that use this tensor as input
```

```
    std::vector<std::pair<int, sc_op_weak_ptr_t>> uses_;

    graph_tensor(sc_op *owner, const sc_data_format_t &format,
            const sc_dims &plain_shape, const sc_data_type_t &type);

    ~graph_tensor() = default;

    // other members omitted

    static graph_tensor_ptr make(const sc_dims &shape,
            const sc_data_format_t &fmt = sc_data_format_t(),
            sc_data_type_t dtype = sc_data_type_t(sc_data_etype::F32, 1)) {
        return std::make_shared<graph_tensor>(nullptr, fmt, shape, dtype);
    }
};
```

graph_tensor中存储了logical_tensor_t类型的成员：details_。这个成员中记录了Tensor的大小、数据类型还有format（下文马上将讨论有关format的内容）。sc_op即GraphCompiler中的Op对应的C++类。producer_owner_成员记录了这个graph_tensor的“所有者”Op。uses_成员则是记录了所有使用当前graph_tensor的作为输入的Op。注意uses_是pair<int, sc_op_weak_ptr_t>的vector。每个pair记录了使用这个graph_tensor的op，以及当前graph_tensor是这个op的第几号输入。sc_op_weak_ptr_t是std::weak_ptr<sc_op>，在uses_使用了weak_ptr用于破除循环引用的问题。

开发者可以通过graph_tensor::make来创建一个graph_tensor，并且使用std::shared_ptr<graph_tensor>来管理它。graph_tensor类中有多个成员函数用于修改Graph中Op的连接关系，为了不干扰读者对于graph_tensor基本概念的理解，我们暂时先略过。后文将会介绍这些成员函数。

Tensor的布局(Layout)和格式(Format)

在上一节中，我们以及引入了tensor的format这个概念。在GraphCompiler中，我们一般认为布局(Layout)和格式(Format)这两者是等价的概念。属性深度学习框架底层实现的读者可能有所了解，深度学习算子可能会使用和普通排布不同的特殊内存排布方式来管理Tensor内存，可能会对算子起到加速作用。对于每个Graph Tensor而言，它的需要记录的形状（shape）有两种：1.逻辑上的形状（logical dims，或称plain dims）2.实际内存中的形状（real dims，或称blocking dims）。这里的dims是dimensions的简写。Tensor的logical dims较为简单，即一块Tensor在用户心目中的形状，这个形状也定义了算子具体执行的运算。例如矩阵乘法matmul可以在数学定义上可以接受的是两个[M，K]和[K，N]形状的矩阵。所以GraphCompiler中matmul的输入tensor的logical dims也就必须是形如
[M，K]和[K，N]。

为了加速算子的速度，业界通用的做法会对Tensor进行重新排布。例如CV中常见的Tensor形式是NCHW这样的4D Tensor，即最高维N是batch维，C维度是channel维，H和W是Height和Width维。例如某个CV模型的输入可以是logical dims为[16,64,224,224]的tensor，其中16,64,224,224分别是NCHW这几个维度的长度。在Intel CPU中，常见的卷积（Conv）的实现中会希望C（channel）这个维度在内存中每16个元素是连续存放的。这样的好处是可以充分利用AVX512的SIMD指令，每条指令一次处理16个不同channel上的数据。那么我们就要对NCHW这样的原始内存排布进行重新布局，变成NCHW16c这样的布局。这里和NCHW的不同在于，最低维度变成了16c，表示C（channel）维被拆成了两个维度c和c，而最低维度c的长度为16。所以NCHW16c的布

局，对于逻辑上[16,64,224,224]大小的tensor，实际内存中会变成[16,4,224,224,16]这样的多维Tensor。注意最高第二维的4和最后一维16，对应了c和16c。

对于矩阵乘法而言，GraphCompiler也可能需要对内存进行重新排布，获得更好的性能。对于逻辑上的MK x KN的两个Tensor乘法，可能实际的内存布局是MK16m16k和NK16k32n。

相同的logical dims可能可以对应不同的内存布局(layout或称format)，那么它们的Tensor的实际内存排布也就会不同。我们可以通过Reorder这个Op来将一种Format的tensor转换为另一种format。当然这样的转换是需要进行内存中的拷贝的，需要一定的开销。在不同的深度学习底层实现中，这种转换tensor format的操作有不同的称呼，TVM称之为reorder（GraphCompiler同样如此），oneDNN称之为prepack。它们说的都是同一件事。

上文中NCHW16c、MK16m16k等布局在GraphCompiler中通过sc_data_format的类型表示。需要注意的是NCHW这样的原始内存布局也是一种format，GraphCompiler通常称为plain format，即每个原始logical dim都没有被拆分为多个维度。如果有维度被拆分为多个维度，GraphCompiler中称之为blocking format。

所以除了logical dims，Graph中的Tensor还需要记录real dims，即内存中这块Tensor实际的形状。上一节说到graph_tensor类中的logical_tensor_t成员记录了这个Tensor的具体信息，那么logical_tensor_t类的的定义也就呼之欲出了（在tensor_detail.hpp）：

```
struct logical_tensor_t {
public:
    sc_data_type_t dtype_;

    logical_tensor_t() = default;

    bool operator==(const logical_tensor_t &other) const;

    logical_tensor_t(const sc_data_format_t &format, const sc_dims &plain_dims,
            const sc_data_type_t &type);

    // gets the dims, taking blocking into consideration, using cache
    const sc_dims &get_blocking_dims() const;
    // gets the logical dims in plain format
    const sc_dims &get_plain_dims() const { return plain_dims_; }
    // sets the logical dims in plain format
    void set_plain_dims(const sc_dims &plain_dims);
    // sets the logical dims in blocking format
    void set_blocking_dims(const sc_dims &blocking_dims);
    // gets the data format
    const sc_data_format_t &get_format() const { return format_; }
    // sets the data format and invalidate the cached blocking_dims
    void set_format(const sc_data_format_t &newv);
    // gets the size of the tensor in bytes
    size_t size() const;

private:
    sc_data_format_t format_;
    // The real dims, which may be blocking.
    sc_dims dims_;
    // the logical dims in plain format
    sc_dims plain_dims_;
    // sync real dims based on plain dims and format
    void internal_update();
};
```

`logical_tensor_t`记录了数据类型、format、blocking dims和plain dims信息。通过getter setter来获取、设置这些值。其中`sc_dims`是`std::vector<int64_t>`的别名，用于存放Tensor的维度。

这里多提一句，和GraphIR的Tensor不同，Tensor IR中的Tensor只需要记录real dims，而logical dims一般不会被记录进Tensor IR中的Tensor中。

小结一下Graph IR中的tensor：
1) `graph_tensor`总有一个“owner” op，即这个Op的输出是这个Tensor
2) `graph_tensor`中还存储了以这个Tensor作为输入的Op
3) `graph_tensor`中`logical_tensor_t details_`成员记录了这个Tensor本身的具体信息：包括数据类型，format，具体维度等

Op

Op是"Operator"（算子）的简写。在GraphCompiler中，Op表示的是对一组输入Tensor的计算，然后Op将会生成一组输出Tensor。以编译器的视角，Op可以认为是一次函数调用（例如调用了“add”这个函数），输入参数是一组Tensor，返回值会“创建”一组新的Tensor。以计算图的视角，Op是图上的一个节点，输入边和输出边都是与Tensor节点相连。

`sc_op`是所有Op的基类。它主要定义了一些虚函数接口，以及声明了所有Op都需要的成员变量：输入、输出Tensor的列表。要实现一个具体的Op，需要通过继承`sc_op`类来做到。

它的定义在头文件graph.hpp中。简化后大致如下：

```
struct sc_op_info_t {
    std::vector<graph_tensor_ptr> outputs_;
    std::vector<graph_tensor_ptr> inputs_;
};

class sc_op : public virtual op_base_trait_t,
                              public std::enable_shared_from_this<sc_op> {
public:
    sc_op_info_t info_;
    any_map_t attrs_;
    // the logical op ID in the op in the manager, default is 0.
    int logical_op_id_ = 0;
    std::string op_name_;

    /**
     * Compares the contents (op_name/attrs/other fields in the op). The default
     * implementation only compares op_name and attrs. This function does not
     * check the op-tensor connections, which will be checked by the
     * graph_comparer.
     * @note we ingore the attrs with keys starting with "temp."
     * @return true if the contents (not including the op connections) are the
     * same
     * */
    virtual bool compare_contents(const sc_op *other) const;

    /**
     * Hash the contents. The default implementation only hashs op_name and
     * attrs. The function can be used to make hash map. When hash conflict
     * happened, we can compare them with `compare_contents`.
     * @note we ingore the attrs with keys starting with "temp."
     * @return hash value with size_t datatype.
     * */
    virtual size_t hash_contents() const;
```

```
    // constructor
    sc_op(const std::string &op_name,
            const std::vector<graph_tensor_ptr> &producer_lt,
            const std::vector<graph_tensor_ptr> &consumer_lt,
            const any_map_t &attrs);
    virtual ir_module_ptr get_func(context_ptr ctx) = 0;

    virtual void query_format(context_ptr ctx,
            std::vector<std::vector<sc_data_format_t>> &in_formats,
            std::vector<std::vector<sc_data_format_t>> &out_formats)
            = 0;
    virtual float get_gflop();
};
```

我们现在来逐一介绍这个类中的各个成员。`sc_op_info_t info_;`这个成员记录了这个Op所有的输入和输出Tensor。需要注意的是，Op的输入输出Tensor的顺序是这个Op（子类）所定义的。例如矩阵乘法`matmul_core`这个Op的第一、二个输入就是数学上矩阵乘法的第一二个输入（注意到矩阵乘法不满足交换律，所以Op输入Tensor的顺序是很重要的）。

每个Op都用于一个称为“属性表”的成员，即`any_map_t attrs_;`。“属性表”（attribute，简称attr）是一个字符串(`std::string`)到任意类型的map。GraphCompiler中通过一个称为`any_map_t`的类来实现了字符串到任意类型的map。它和Python中的`dict`类似，可以将任意字符串“键”映射到任意类型的“值”，而且同一个`any_map_t`中可以存储不同类型的值。例如对于`any_map_t`类型的对象`attrs_`来说，下面的代码都是合法的：

```
attrs_["key1"]=1; // int
attrs_["key1"]="some value"; // std::string
```

我们知道，C++语言无法在程序运行时为一个类动态地添加成员。但是有了“属性表”（attr），GraphCompiler就可以为Op动态地添加属性。例如Graph IR上的pass也可以通过attr记录中间结果。

下面一个成员`int logical_op_id_;`是这个Op的ID。一个Op有且只有一个Graph来管理它。在一个Graph对象中，每个Op的`logical_op_id_`是唯一的，而且这个ID保证会在`0`到`(Op数量-1)`之间。有了这个在同一个Graph中唯一的ID，我们可以方便地把一个Op映射到一个数字上，或者是将Op ID对一个数组进行索引。

成员变量`std::string op_name_;`是这个Op的名字。注意这个Op名字是Op“具体执行的计算”的名字，例如“add”，“sub”，“conv_fwd”等等。Op名字并不是在Graph中唯一的，多个Op对象可以有同一个`op_name_`。

成员函数`compare_contents`是个虚函数，用来比较这个Op与其他Op的内容，比较的内容包括包括attr、成员变量等。但是这个函数无需比较两个Op的连接关系是否相同。成员函数`hash_contents`类似，用来计算这个Op本身内容的哈希值。

成员函数`get_func`是纯虚函数，这个函数就是将Op转换成Tensor IR的接口。这个函数需要返回的是一个IR module。有关Tensor IR，我们在前文中已经有所介绍：

`sc_op`的子类需要实现这个函数来实现对应的Op的计算。返回的内容应该是实现这个Op所定义的计算的具体Tensor IR。在Graph IR lower到Tensor IR的时候，GraphCompiler将会调用这个函数来获取Op的Tensor IR上的实现。具体GraphCompiler中的Op如何实现`get_func`，以及如何将整个Graph lower到Tensor IR，后续的文章将会介绍。

下一个成员函数是query_format。它也是纯虚函数，需要子类提供相应实现。接口定义如下：

```
void query_format(context_ptr ctx,
            std::vector<std::vector<sc_data_format_t>> &in_formats,
            std::vector<std::vector<sc_data_format_t>> &out_formats);
```

query_format用于查询一个Op所期望的输入和输出Tensor的format。这个函数的输入的是一个context指针，GraphCompiler通过context来存储编译器本身的配置，例如目标CPU支持的指令集等等。同一段代码用不同的context可能会产生不同的编译结果。我们使用get_default_context()来获取当前机器的context。query_format通过in_formats和out_formats这两个引用来返回结果。每一个Op会根据自己的需要，选择自己想要的输入Tensor的format以及输出Tensor的format。返回的format以std::vector<std::vector<sc_data_format_t>>的形式存储。内层std::vector<sc_data_format_t>中每个元素对应了这个Op期望的每个输入/输入Tensor的format。外层的std::vector<...>用于返回多组候选的format组合。例如有一个matmul可以接受的输入输出format组合为in=[MK16m16k, KN16n16k] out=[MK16m16k]。它也可以接受Plain format的组合： in=[MK, KN] out=[MK]。那么这个op的query_format的结果应该是in_formats=[[MK16m16k, KN16n16k], [MK, KN]] out_formats=[[MK16m16k], [MK]]。

最后是get_gflop()这个函数。它可以返回这个Op的计算量，以FLOPs计（floating point operations）。

Graph

Graph对象是Graph IR中最顶层的概念，用于表示一张计算图。Graph中主要管理了这个图中所有的Op，存放在一个std::vector中。另外，与Op类似，Graph还允许动态添加属性到attr中。用户也可以在创建Graph的时候管理这个属性表。GraphCompiler中，Graph对象的类型为sc_graph_t，去除暂时与本篇无关的代码后，内容如下：

```
class sc_graph_t {
public:

    std::vector<sc_op_ptr> ops_;
    any_map_t attrs_;

    // adds an existing node to the graph
    void add(const sc_op_ptr &node);

    std::shared_ptr<sc_op> make(const std::string &op_name,
            const std::vector<graph_tensor_ptr> &inputs,
            const std::vector<graph_tensor_ptr> &outputs,
            const any_map_t &attrs);

    template <typename T, typename... Args>
    std::shared_ptr<T> make(Args &&... args) {
        // ...
        auto ret = std::make_shared<T>(std::forward<Args>(args)...);
        add(ret);
        return ret;
    }

    /**
     * Hash the contents. The default implementation only hashs ops and
     * attrs. The function can be used to make hash map. When hash conflict
     * happened, we can compare them with `compare_graph`.
     * @note we ingore the attrs with keys starting with "temp."
     * @return hash value with size_t datatype.
     * */
```

```
    size_t hash_contents() const;

    // Get the total gflop from all tunable ops contained in the graph.
    float get_gflop() const;
    // output op
    std::shared_ptr<sc_op> make_output(
            const std::vector<graph_tensor_ptr> &inputs,
            const any_map_t &attrs = any_map_t());
    // input op
    std::shared_ptr<sc_op> make_input(
            const std::vector<graph_tensor_ptr> &inputs,
            const any_map_t &attrs = any_map_t());
};
```

这个类中数据成员主要就是

```
    std::vector<sc_op_ptr> ops_;
    any_map_t attrs_;
```

其中ops_存放了所有的Op指针（通过sc_op_ptr指针，即std::shared_ptr<sc_op>）。可以通过Op中的logical_op_id_来从它所属Graph的ops_数组中索引到这个Op。attrs_即这个Graph的属性表。

```
void add(const sc_op_ptr &node);
```

add方法将一个Op加入到当前graph的ops_中，并且会自动设置Op的logical_op_id_。

```
    std::shared_ptr<sc_op> make(const std::string &op_name,
            const std::vector<graph_tensor_ptr> &inputs,
            const std::vector<graph_tensor_ptr> &outputs,
            const any_map_t &attrs);
```

make方法可以在当前Graph中创建一个新的Op。参数中op_name是这个Op所执行操作的名字，例如“add”，“sub”，“conv_fwd”等等。这个op_name与Op中的op_name_属性对应。GraphCompiler维护了一张从op_name到创建某个Op子类对象的函数的映射表。make方法可以通过op_name查询到创建对应Op的函数，来创建这个Op。后面的参数是新创建Op的输入和输出Tensor，以及这个Op的属性表（attr）。

```
    template <typename T, typename... Args>
    std::shared_ptr<T> make(Args &&... args) {
        // ...
        auto ret = std::make_shared<T>(std::forward<Args>(args)...);
        add(ret);
        return ret;
    }
```

这是另一个版本的make方法。如果需要创建的Op在GraphCompiler编译时已经可以确定，那么可以不用动态地通过op_name查找映射表，而是直接通过C++模板地方式创建Op对象，然后调用add方法加入到这个Graph中。

对于input_op和output_op这两个特殊的Op来说，需要通过make_input和make_output方法来创建Op对象。

与Op类似，Graph也提供了哈希函数hash_contents，以及get_gflop用于计算整个计算图的计算量。

小结

在这篇文章中，我们讨论了Graph IR中Op、Tensor、Graph等概念在C++代码中的实现，包括它们的主要成员变量和函数。在下一篇文章中，我们将会继续讨论如何通过C++对Graph IR进行修改和变换，以及如何按照各种自定义的顺序遍历Graph中的各个Op。这些是我们实现Graph IR pass，和从Graph IR lower到Tensor IR的基础。