深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 修改和遍历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中如何在已经创建好的Graph IR上进行变换、修改，以及如何按照要求的顺序遍历一个Graph内所有的Op。本文示例代码已经上传至
https://github.com/Menooker/graphcompiler-tutorial/tree/master/Ch7-GraphIR
github.com/Menooker/graphcompiler-tutorial/tree/master/Ch7-GraphIR

如何编译请参考系列文章第三篇内容。

回顾：如何创建一个Graph

在本系列的第三篇文章中已经展示了如何通过C++代码搭建一个Graph IR组成的计算图。

本文以这个计算图为基础继续探索如何对现有的计算图进行修改。我们首先来回顾一下如何通过C++代码创建Graph IR

```
    sc_graph_t g;
    auto in = g.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32),
                            graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32)});
    auto relu = g.make("relu", {in->get_outputs()[0]}, {}, {});
    auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});
    auto out = g.make_output(add->get_outputs());
    print_graph(g, std::cout, true);
```

有了上一篇对于Graph IR C++代码的解析，读者应该能更进一步理解上面的代码。我们先创建一个空的Graph，然后使用`make_input`创建这个Graph的输入节点`input_op`。这个图有两个输入Tensor，都是1024x1024的float32 Tensor。我们使用默认Plain format作为这两个输入Tensor的format。代码`sc_data_format_t()`使用默认构造函数创建了Plain format对象。

`input_op`是整个图的输入节点，它的`output tensor`就是这个图的输入Tensor。`in->get_outputs()`这个代码将会得到`input_op` “in”的两个输出Tensor（通过`const std::vector<sc_op_ptr>&`返回）。`in->get_outputs()[0]`即整个图的第0号输入，也就是`g.make_input(`这行代码中的第一个Tensor。

代码`auto relu = g.make("relu", {in->get_outputs()[0]}, {}, {});`通过调用Graph的`make`函数来创建一个"relu" Op，输入参数即整个Graph的第0号输入。`make`函数的原型在上一篇文章已经提到：

```
    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);
```

我们看到这里调用`make`函数，在参数`outputs`的地方填入了`{}`，即空的vector。GraphCompiler会调用Relu Op对象的构造函数，函数中如果`outputs`为空，将会自动根据输入Tensor的数据类型和大

小推断出输出Tensor，所以我们在调用的时候可以直接传入空vector。同样，我们对于Relu Op没有什么特殊的配置，所以attrs参数也填入空值。

代码`auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});`与创建Relu Op的代码类似，只是add需要两个输入节点，我们将relu的输入和Graph的第1号输入作为add op的输入。

最后，代码`auto out = g.make_output(add->get_outputs());`创建了一个`output_op`，这个将add op的输出标记为整个图的输出。至此，Graph IR已经构造完毕。

我们可以将内存中的Graph IR对象（即`sc_graph_t`对象）转换为人类可读的字符串形式。这是通过函数`print_graph`完成的。要使用这个函数，需要include的头文件为`compiler/ir/graph/pass/pass.hpp`。函数的原型如下：

```
void print_graph(const sc_graph_t &mgr, std::ostream &os,
        bool print_shape = false, bool print_attr = false,
        bool print_name = false);
```

第一个参数即Graph对象的引用。第二个参数`os`即将字符串化的Graph输出的输出流。如果这里填入`std::cout`，那么就会直接在终端打印Graph的字符串表示。后面三个参数`print_shape`、`print_attr`、`print_name`分别表示是否打印每个Graph Tensor的shape（blocking dims），是否打印每个Op的attr，以及是否尝试从Op的attr中找到Tensor的名字，并且将tensor的名字显示在最终的字符串中。

最终我们上面这段代码在`print_graph`之后的输出为

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = relu(v0)
  [v2: f32[1024, 1024]] = add(v3, v1)
}
```

这段输出的含义是：首先对input v0计算relu，得到v3，然后把v3和input v1做加法，得到最终输出v2。

修改Graph IR

在上一篇文章中我们简单介绍了Graph IR在C++中的定义，但是我这边没有介绍了其中修改Graph IR的部分。现在结合实例来介绍`sc_op`、`graph_tensor`中有关修改IR的成员函数。

我们先简单地修改一下上面生成Graph IR的代码，将它封装为一个函数一遍之后的代码中进行再利用。我们创建一个`make_graph`函数，返回上文生成的那个Graph，并且通过出参返回生成的Graph中`input`,`relu`,`add`,`output`这几个Op的指针。

```
sc_graph_t make_graph(sc_op_ptr& in, sc_op_ptr& relu, sc_op_ptr& add, sc_op_ptr& out) {
    sc_graph_t g;
    auto in = g.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32),
                            graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32)});
    auto relu = g.make("relu", {in->get_outputs()[0]}, {}, {});
    auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});
    auto out = g.make_output(add->get_outputs());
    return g;
}
```

sc_op::replace_input方法

sc_op类中的replace_input方法可以将当前Op的一个输入Tensor替换为另一个Tensor。原型如下：

```
class sc_op {
    // ...

    /**
     * Repalces an input logical tensor
     * @param index the index within get_inputs()
     * @param new_input the new logical tensor
     * */
    void replace_input(size_t index, const graph_tensor_ptr &new_input);

    // ...
};
```

它的作用是将这个Op的第index号输入替换为参数new_input。

我们上文中add op的第二个输入是graphinput[1](v1)，下面我们试着通过replace_input方法将它替换为graphinput[0]（v0）。

```
sc_op_ptr in, relu, add, out;
sc_graph_t graph = make_graph(in, relu, add, out);
add->replace_input(1, in->get_outputs()[0]);
print_graph(graph, std::cout, true);
```

这段代码的输出是:

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = relu(v0)
  [v2: f32[1024, 1024]] = add(v3, v0)
}
```

可以看到add的第二个输入从v1被改为了v0。

sc_op::replace_uses_with_and_remove方法

sc_op类中的replace_uses_with_and_remove方法可以将当前Op替换为另一个Op，包括将所有使用当前Op的地方替换成目标Op，以及将当前Op从Graph中移除。原型如下：

```
class sc_op {
    // ...

    // Replaces the current Op in the graph using another Op. All other Ops
    // using the output tensors of current Op will use the corresponding tensors
    // in the replacer Op instead. Finally the current node will be removed
    // Requires that the replacer has the same number of outputs of the current
    // node. Will detach from the input tensors. The replacer should manually
    // attach to the input tensors when it is needed
    void replace_uses_with_and_remove(const sc_op_ptr &replacer);

    // ...
};
```

下面我们通过这个函数来将原始Graph中的add替换为sub。

```
sc_op_ptr in, relu, add, out;
sc_graph_t graph = make_graph(in, relu, add, out);
auto sub = graph.make("sub", add->get_inputs(), {}, {});
add->replace_uses_with_and_remove(sub);
print_graph(graph, std::cout, true);
```

我们先在通过Graph的make方法在图中创建一个减法sub节点，这个节点的输入和add的输入相同。然后调用add这个op的replace_uses_with_and_remove方法即可。得到的结果是：

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = relu(v0)
  [v2: f32[1024, 1024]] = sub(v3, v1)
}
```

看到add节点被替换为了sub

graph_tensor::replace_with方法

graph_tensor类中的replace_with方法可以将所有使用当前Tensor的地方全部改为另一个Tensor。它的原型如下：

```
class graph_tensor {
    //...

    // replaces all uses of this tensors with `v`. Will call `replace_input` of
    // all uses of this
    void replace_with(const graph_tensor_ptr &v);
};
```

在前一个小结sc_op::replace_input方法的实例中中我们得到了这样的计算图：

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = relu(v0)
  [v2: f32[1024, 1024]] = add(v3, v0)
}
```

我们下面展示通过graph_tensor::replace_with方法将图中的v0全部改为v1。

```
in->get_outputs()[0]->replace_with(in->get_outputs()[1]);
print_graph(graph, std::cout, true);
```

代码的输出结果为：

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = relu(v1)
  [v2: f32[1024, 1024]] = add(v3, v1)
}
```

遍历Graph中所有的Op - Graph Visitor

在GraphCompiler中，有大量代码需要遍历图中所有的Op，包括大部分Graph IR pass，将Graph打印为字符串的print_graph函数等等。那么在在GraphCompiler中，我们是如何实现遍历Graph的呢?

首先最简单的方式是直接遍历graph中的ops_数组成员。这个数组按照Op ID顺序存放了图中所有的Op。所以直接通过for循环遍历这个数组就可以遍历到所有的Op了。但是很多时候我们对Op的遍历顺序是有要求的。在图论中，经常使用到的图遍历顺序有DFS（深度优先搜索），BFS（广度优先搜索）。此外，在GraphCompiler中最为常用的图遍历顺序是拓扑排序，即访问一个Op之前，需要优先访问这个Op依赖的其他Op。满足这样规定的Op访问顺序称之为拓扑排序（topology sort）。例如在GraphCompiler中，Op的执行顺序一定是按照拓扑排序进行的，因为Op的input tensor需要先于Op被计算出来。

我们可以为不同的遍历顺序书写不同的遍历代码，例如分别为BFS，DFS，拓扑排序设计不同的遍历Op的函数。但是这会存在两个问题：

1）实现BFS，DFS，拓扑排序的代码在很多方面是类似的，如果分别实现，会有重复的代码

2）即使同样是拓扑排序（或者DFS，或者BFS），存在有多种不同的访问顺序都满足要求（例如拓扑排序要求优先访问这个Op依赖的其他Op），那么使用者如何去定制想要的顺序呢？例如下图：

图中所有边的箭头方向指向依赖者节点，例如a->b代表b依赖于a。那么对于这个“计算图”的拓扑排序可以有多种：

1) abcde
2) abdce

即使都满足拓扑顺序，不同的场景下可能需要选择不同的顺序。例如在将Graph IR lower到Tensor IR的时候，我们可能会选择abcde的方式来遍历，因为在执行完ab这个两个Op之后，如果先执行d，可能导致b的结果被驱逐出缓存，导致c的执行效率变差。

我们再来回头看BFS，DFS，拓扑排序的实现上的异同。这三种图遍历方式本质上都能用以下的伪代码表示：

```
items = {}
while(!items.empty()) {
  i = select_and_remove(items)
  visit(i)
  update(i, items)
}
```

`items`是包含了所有可以访问的Op的集合。对于DFS来说，`items`应该是个栈；对于BFS来说它应该是一个队列；而对于拓扑排序来说，`items`则是所有依赖节点已经被访问完毕，本身可以被访问的Op的集合。`select_and_remove`对应了从`items`中取出一个元素的操作。我们前面已经知道DFS的`items`是栈，所以DFS的`select_and_remove`操作应该是对`items`的`pop`操作。相对应的，对于BFS，`select_and_remove`操作应该是对`items`的`dequeue`操作。拓扑排序较为特殊，`select_and_remove`返回`items`集合中的任意元素都是可以的，因为拓扑排序中的`items`集合是所有已经准备好被访问的Op的集合。拓扑排序中我们允许使用者提供自定义的`select_and_remove`。

`visit`是调用图遍历算法的调用者提供的回调函数，参数应该是每个单独的Op。`update`则会通过刚刚访问的Op来更新`items`。对于BFS，`update`应该将所有依赖于Op的其他Op`push`到`items`中。对

于BFS，update应该将所有依赖于Op的其他Openqueue到items中。DFS和BFS还需要记录一个Op是否已经被访问，如果已经被访问，则不需要再加入到items中。拓扑排序的update则会稍微复杂一些。update函数需要有一个内部状态，记录每个Op还有多少依赖的Op没有被访问。每当调用update的时候，将会更新Op依赖表，将依赖当前Op的其他Op的依赖数量减去1，如果一个Op的依赖数量是0，那么它就可以被加入到items中。

有以上这些观察和要求，GraphCompiler提供了Graph Visitor用于帮助开发者按照想要的顺序遍历图。

op_visitor_t

GraphCompiler中的op_visitor_t抽象了对于Graph IR中Op的访问顺序，开发者可以通过它，方便地遍历Graph中的Op。它定义在src/backend/graph_compiler/core/src/compiler/ir/graph/visitor.hpp。定义如下：

```
class op_visitor_t {
public:
    // the queue/stack for the nodes to visit
    std::list<sc_op_ptr> to_visit_;
    // the array to memorize the nodes that we have visited, indexed by the op
    // id
    std::vector<bool> visited_;

    using updater_func = std::function<void(op_visitor_t *, sc_op_ptr)>;
    using selector_func = std::function<sc_op_ptr(op_visitor_t *)>;
    // the selector to return the next node to visit in `to_visit_` list. It
    // should also remove the node from the list. It can return null if it finds
    // a node that has been visited. The visitor will try to call it again
    std::function<sc_op_ptr(op_visitor_t *)> select_next_node;
    // will be called after a node has been visited. Usually it should update
    // the `visited_`, and push/enqueue the sub-nodes to the `to_visit_`
    std::function<void(op_visitor_t *, sc_op_ptr)> update_visit_list;

    void visit(const std::function<void(sc_op_ptr)> &f);

    // set a node as visited
    void set_visited(int id);
    // returns if an id is in the visited node set
    bool has_visited(int id);

    void visit_graph(
            const sc_graph_t &mgr, const std::function<void(sc_op_ptr)> &f);

    op_visitor_t(selector_func select_next_node_func,
            updater_func update_visit_list_func);

    // updates the visitor states after a node is visited. It can be also used
    // when a new node replaces an old one. Users should call this function with
    // the new node
    void update_state_for_visited(sc_op_ptr node);
};
```

和上一节的伪代码一样，op_visitor_t的内部状态有to_visit_，即需要访问的Op的集合，以及visited_，即用来纪录哪些Op已经被访问过，通过std::vector<bool>表示，以Op的logical_op_id_作为索引（这个ID是0-N连续的ID）。op_visitor_t本身不是BFS、DFS或者拓扑排序中的任何一种。但是它搭建了这些遍历算法的基础。使用者需要提供三个函数，select_next_node用于从visited_选择并且删除一个op。update_visit_list则是需要通过已经访问的Op计算出哪些Op可以加入visited_列表。select_next_node和update_visit_list需要在

op_visitor_t的构造函数中给出。还有在实际开始遍历Graph的时候，需要给op_visitor_t一个回调函数用于访问每个单独的Op。

我们看到op_visitor_t::visit函数的实现其实与上文的伪代码高度一致：

```
void op_visitor_t::visit(const std::function<void(sc_op_ptr)> &f) {
    while (!to_visit_.empty()) {
        auto ptr = select_next_node(this);
        // if selector fails (e.g. found a node that has already been visited),
        // try again
        if (!ptr || ptr->is_removed_) { continue; }
        f(ptr);
        update_state_for_visited(std::move(ptr));
    }
}
```

op_visitor_t预定义了一些select和update函数，通过组合这些函数，足以满足日常DFS、BFS和拓扑排序的要求。这些函数作为static函数，定义在op_visitor_t类中：

```
class op_visitor_t {
    // ...

    // the updater which pushes all uses of all output logical tensors
    // to the back of the to_visit list
    static void push_back_updater(op_visitor_t *, const sc_op_ptr &sc_op_ptr);

    // the updater which pushes all nodes whose dependencies have already been
    // visited. Used in topology sort
    static updater_func create_DAG_updater(size_t total_nodes_hint);
    static updater_func create_DAG_updater_post(size_t total_nodes_hint);

    // the selector which pops a node in `to_visit_` from back
    static sc_op_ptr pop_back_selector(op_visitor_t *v);

    // the selector which pops a node in `to_visit_` from front
    static sc_op_ptr dequeue_selector(op_visitor_t *v);

    // ...
};
```

如何使用op_visitor_t呢？例如我们想要DFS遍历某个Graph对象g，并且按DFS顺序打印所有Op的名字。那么可以：

```
op_visitor_t visitor(op_visitor_t::pop_back_selector, op_visitor_t::push_back_updater);
visitor.visit_graph(g, [](sc_op_ptr op) {
    std::cout<<op->op_name_<<"\n";
});
```

第一行通过pop_back_selector和push_back_updater创建了一个DFS visitor（DFS中使用栈，所以需要pop和push）。然后调用visit_graph方法，传入Graph对象g和一个lambda函数，这个函数打印了当前op的op_name_。visit_graph方法将会按照使用者定义的遍历顺序逐个访问图中每个Op，并且调用传入的回调函数。

如果我们需要使用BFS顺序，那么只要将第一行改为：

```
op_visitor_t visitor(op_visitor_t::dequeue_selector, op_visitor_t::push_back_updater);
```

如果需要拓扑排序+先进先出顺序，我们只需：

```
op_visitor_t visitor(op_visitor_t::dequeue_selector, op_visitor_t::create_DAG_updater());
```

注意，如上文所说，拓扑排序中可以使用任意的select函数，而update函数只需选用op_visitor_t::create_DAG_updater()即可。op_visitor_t::create_DAG_updater本身是一个返回函数

(std::function)的函数，这里需要先调用它，然后使用它的返回值作为update函数参数。

像上面这样定义visitor可能还不够方便，op_visitor_t预定义了DFS、BFS和dfs_topology_sort：

```
class op_visitor_t {
    // ...

    // constructs a DFS visitor, using push_back_updater and
    // pop_back_selector
    static op_visitor_t dfs();
    // constructs a BFS visitor, using push_back_updater and
    // dequeue_selector
    static op_visitor_t bfs();
    // constructs a topology sort visitor in DFS order, using
    // create_DAG_updater and pop_back_selector
    static op_visitor_t dfs_topology_sort(size_t total_nodes_hint = 30);
};
```

例如上文创建DFS visitor的代码可以改写为：

```
op_visitor_t visitor = op_visitor_t::dfs();
```

Graph Visitor小结

GraphCompiler提供了op_visitor_t类，用于遍历Graph中的每一个Op。op_visitor_t类可以通过传入update、select和visit来定制访问顺序。如果对访问Op的顺序没有要求，只是想要遍历Graph中的所有Op，可以直接遍历Graph的ops_数组。

这篇文章我们详细探索了如何修改和遍历Graph IR。有了这些知识，我们下一篇将会讨论Graph IR如何转换（lower）到Tensor IR。