深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第12篇 Tensor IR Visitor

关于作者以及免责声明见序章开头。题图源自网络，侵删。

本文示例代码已经上传至
https://github.com/Menooker/graphcompiler-tutorial/blob/master/Ch12-IRVisitor/irvisitor.cpp
github.com/Menooker/graphcompiler-tutorial/blob/master/Ch12-IRVisitor/irvisitor.cpp

上一篇文章中我们讨论了Graph Compiler的IR pass。对于Graph IR来说，开发者可以通过`op_visitor_t`类来遍历整个Graph，构建Graph pass。那么对于Tensor IR来说，有没有工具来帮助递归遍历一个IR函数中的所有stmt和expr呢？这就是本篇的主题，IR visitor。

IR visitor主要有两个作用：

- 按照深度优先顺序遍历IR。这个顺序也是Tensor IR实际的执行顺序
- 允许替换IR中的任意expr和stmt节点

基本接口定义

`ir_visitor_t`是一个C++类，提供的主要接口是`dispatch`和`visit`。`visit`是一个重载函数，接受一个const修饰的stmt或者expr指针作为参数。它的头文件定义在：
src/backend/graph_compiler/core/src/compiler/ir/visitor.hpp。 对于所有的stmt，expr的子类，`visit`函都提供了相应的重载版本：

```
class ir_visitor_t : private ir_visitor_base_impl_t<false> {

    virtual expr_c dispatch(expr_c e);
    virtual stmt_c dispatch(stmt_c s);
    virtual func_c dispatch(func_c v);

    virtual expr_c visit(constant_c v);
    virtual expr_c visit(var_c v);
    virtual expr_c visit(cast_c v);

    virtual expr_c visit(binary_c v);
    virtual expr_c visit(add_c v);
    virtual expr_c visit(sub_c v);
    virtual expr_c visit(mul_c v);
    virtual expr_c visit(div_c v);
    virtual expr_c visit(mod_c v);

    // ...
}
```

在IR visitor基类中，每个版本的visit函数的默认实现将会依次对这个IR节点内部所有的IR节点的调用`visit`函数，实现递归地对于整个IR“树”的每个节点调用`visit`函数。这里有个问题，IR节点中存放的一般是基类指针，例如if node中：

```
class if_else_node_t : public stmt_base_t /* ... */{
public:
    expr condition_;
    stmt then_case_;
    stmt else_case_;
    /* ... */
}
```

我们看到cond_，then_block_，else_block_都是以基类指针形式存放的：expr，stmt。而为了实现递归调用，我们需要对于一个基类指针，调用它实际子类对应的visit重载函数（visit函数需要指明子类指针类型，无法通过基类指针调用）。IR visitor提供了dispatch方法。它的内部可以将一个基类IR node指针转换为对应的子类指针，然后调用当前visitor对象内部对应的visit方法。

所以对于if_else_node_t来说，ir_visitor_t基类的visit函数会依次调用dispatch(cond_)、dispatch(then_block_)、dispatch(else_block_)，而dispatch内部将会调用对应子类的visit函数，实现递归访问所有的子节点。

上文描述的diapatch，visit这两个函数都是虚函数。开发者可以继承IR visitor，然后在自己的visitor类中override掉这两个方法。pass的开发者只需要override关心的IR节点子类的visit方法。例如，如果只override了add_node的visit函数，那么这个visitor可以实现遍历IR中所有的add节点。

visitor还能实现IR的替换。visit和dispatch方法都需要返回一个IR节点，会将原始IR“树”中被visit的节点替换为返回值返回的节点指针。如果无需替换，则需要返回这个visit或dispatch方法参数中传入的原始节点指针。

在上一篇文章中，我们已经提到，Tensor IR pass传入的IR都是const修饰的，也无法直接修改输入的IR。visit和dispatch方法传入的参数也都是const修饰的指针。那么我们是如何实现在visitor中替换IR节点的呢？我们使用了“写时复制”的思路（copy on write）。visitor中，当我们visit一个IR节点的所有成员子节点的时候，我们会记录任意子节点是否被修改（dispatch的返回的节点指针是否和原来成员存储的指针相同）。如果有至少一个子节点被修改，那么会创建一个新的IR节点，它的子节点指向visit方法中对各个子节点dispatch后的返回值。所以一个IR节点如果被visit或者dispatch方法的返回值替换，那么它所有的祖先节点都会通过copy on write方式进行重建，但是这个节点的兄弟节点如果没有变化，那么则无需重建，新的祖先节点会指向没有修改过的旧IR节点。后文我们将通过visitor的实现代码来继续解释这一过程。

例子—实现简单的常量折叠

我们来通过visitor实现简单的常量折叠的优化。在IR中，我们可能会遇到类似2+3这样形式的IR，其中所有的数据节点都是常量，而不是var或者indexing节点。对于编译器来说，我们可以在编译IR的时候就计算出这种常量表达式的结果，而无需在生成的可执行代码中在运行时计算常量表达式。例如我们可以用单个常量5来替换IR中的2+3。

我们在这个例子中，准备实现对于s32（signed int 32）类型的加减法的常量折叠。

首先需要include visitor头文件，并且创建一个新的visitor子类继承ir_visitor_t:

```
#include <compiler/ir/visitor.hpp>

using namespace sc;
```

```
class simple_constant_folder : public ir_visitor_t
```

我们的目标只是实现加减法的折叠，所以只关心加法和减法节点，通过override visit方法的方式来覆盖add_c和sub_c的重载函数：

```
expr_c visit(add_c v) override
    {
```

visit方法中，以对add_c类型节点的处理为例，我们检查当前节点的左右参数都为constant节点，如果是，那么就可以return一个constant节点，constant中的值就是add节点左右constant node的值的和：

```
if (lhs.isa<constant>() && rhs.isa<constant>() && lhs->dtype_ == datatypes::s32)
{
    int64_t lval = lhs.static_as<constant>()->value_[0].s64;
    int64_t rval = rhs.static_as<constant>()->value_[0].s64;
    return make_expr<constant_node>(lval + rval, datatypes::s32);
}
```

我们还忽略了一件重要的事情，在visit方法中，我们要手动对于每个addnode的子节点，调用dispatch方法，这样才能递归地访问IR上所有的节点。 如果当前add节点不是常量表达式，那么我们需要处理两种情况 add节点的子节点在dispatch之后没有变化，那么说明add节点本身也没有变化，直接返回原来输入的add节点 add节点的子节点在dispatch之后至少有一个发生变化，那么需要根据dispatch方法返回的新的子节点来构造新的add节点 所以整体代码如下：

```
expr_c visit(add_c v) override
{
    auto lhs = dispatch(v->l_);
    auto rhs = dispatch(v->r_);

    if (lhs.isa<constant>() && rhs.isa<constant>() && lhs->dtype_ == datatypes::s32)
    {
        int64_t lval = lhs.static_as<constant>()->value_[0].s64;
        int64_t rval = rhs.static_as<constant>()->value_[0].s64;
        return make_expr<constant_node>(lval + rval, datatypes::s32);
    }
    bool changed = !lhs.ptr_same(v->l_) || !rhs.ptr_same(v->r_);
    if (changed)
    {
        return builder::make_add(lhs, rhs);
    }
    return v;
}
```

减法节点可以按照上面的方式处理，只是代码中的add需要替换为sub。

至此我们的这个visitor已经完成了，它可以递归地将IR中的常量加减法替换为等价的常量值。我们来试验一下，首先创建一个simple_constant_folder对象

```
simple_constant_folder folder;
```

创建想要处理的expr，其中b这个expr混合了不能常量折叠的var和可以折叠的constant:

```
expr a = expr(1) + expr(2) - expr(3);
expr b = expr(1) + expr(2) + builder::make_var(datatypes::s32, "var_b");
```

然后调用simple_constant_folder对象的dispatch方法：

```
std::cout << "expr a before folder:" << a << ", after folder:" << folder.dispatch(a) << '\n';
std::cout << "expr b before folder:" << b << ", after folder:" << folder.dispatch(b) << '\n';
```

最后折叠的结果为：

```
expr a before folder:((1 + 2) - 3), after folder:0
expr b before folder:((1 + 2) + var_b), after folder:(3 + var_b)
```

我们这个简单的常量折叠pass没法处理一些较为复杂的IR，例如1+a+2，它是无法优化为a+3的。GraphCompiler中实现了功能更强的常量折叠，实现在了`constant_folder`中：

对IR只读访问：ir_viewer_t

有些IR pass只是对Tensor IR进行分析，不会对IR进行修改。这时如果在Visitor中override的`visit`方法还需要return返回值，一是会造成代码的冗余，而是开发者有可能无意中返回了错误的返回值，造成IR的改变。所以Graph Compiler提供了`ir_viewer_t`。IR Viewer从接口上杜绝了开发者修改IR（包括直接修改IR节点本身、以及通过返回值来Copy on write修改），这样开发者可以放心将IR交给`ir_viewer_t`来进行只读的分析而不用担心IR会被修改。它的头文件在：src/backend/graph_compiler/core/src/compiler/ir/viewer.hpp。在接口上，`ir_viewer_t`与`ir_visitor_t`类似，都为每个IR节点的子类提供了重载方法：

```
// ...
virtual void view(constant_c v);
virtual void view(var_c v);
virtual void view(cast_c v);

virtual void view(binary_c v);
virtual void view(add_c v);
virtual void view(sub_c v);
virtual void view(mul_c v);
virtual void view(div_c v);
virtual void view(mod_c v);

virtual void view(cmp_c v);
// ...
```

开发者可以通过继承`ir_viewer_t`来override感兴趣的子类的`view`方法。注意到`view`方法的返回值是`void`，而不是`ir_visitor_t`中的`expr_c`或者`stmt_c`。

`ir_viewer_t`的内部实现是通过私有继承`ir_visitor_t`，并且借由`visit`方法来实现了`view`方法。

IR Visitor原理

这里我们简单讨论以下IR visitor的实现原理。IR visitor继承自`ir_visitor_base_impl_t`类，实现在src/backend/graph_compiler/core/src/compiler/ir/visitor.cpp。`ir_visitor_base_impl_t`类提供了`visit_impl`和`dispatch_impl`方法，分别对应了`ir_visitor_t`的`visit`和`dispatch`方法。

我们先来看`visit_impl`方法。上文我们已经讨论了，这个重载函数在基类中默认实现应该对输入节点的所有子节点调用调用`dispatch`（在这里应该是`dispatch_impl`），然后通过`dispatch_impl`返回的指针，判断有无子节点被改变。如果有，那么需要重新创建一个新的、同输入节点类型相同类型的IR节点，并且新节点的子节点指向`dispatch_impl`返回的新IR对象——这样就完成了Copy On Write的过程。

例如对于`for_loop`，它的`visit`实现为：

```
template <bool is_inplace>
stmt ir_visitor_base_impl_t<is_inplace>::visit_impl(for_loop v) {
    auto var = dispatch_impl(v->var_);
```

```
    auto begin = dispatch_impl(v->iter_begin_);
    auto end = dispatch_impl(v->iter_end_);
    auto step = dispatch_impl(v->step_);
    auto body = dispatch_impl(v->body_);

    changed_ = !(var.ptr_same(v->var_) && begin.ptr_same(v->iter_begin_)
            && end.ptr_same(v->iter_end_) && step.ptr_same(v->step_)
            && body.ptr_same(v->body_));
    if (is_inplace) {
        v->var_ = var;
        v->iter_begin_ = begin;
        v->iter_end_ = end;
        v->step_ = step;
        v->body_ = body;
        return std::move(v);
    } else {
        if (changed_) {
            return copy_attr(*v,
                    make_stmt<for_loop_node_t>(std::move(var), std::move(begin),
                            std::move(end), std::move(step), std::move(body),
                            v->incremental_, v->kind_));
        }
        return std::move(v);
    }
}
```

其中模板参数is_inplace用来注明这是不是inplace visitor。inplace visitor较少用到，我们可以暂时忽略，认为代码中的is_inplace为false。代码中的copy_attr将第一个参数中IR的attr表拷贝到第二个参数指向的IR节点上，返回第二参数的IR节点指针。

在visitor.cpp中，Graph Compiler对于所有的IR节点类型都实现了与上述代码类似的visit_impl。这样就在基类中实现了对于所有节点中子节点的访问。

我们再来看dispatch（和基类中的dispatch_impl）如何实现。dispatch方法的作用应该是，传入一个基类（expr、stmt）指针，根据指针指向的实际子类对象，调用对应的visit（visit_impl）方法。visitor.cpp中，有关dispatch，只有这两个函数：

```
expr ir_visitor_base_t::dispatch_impl(expr e) {
    return e->visited_by(this);
}

stmt ir_visitor_base_t::dispatch_impl(stmt s) {
    return s->visited_by(this);
}
```

这两个函数只是简单地调用expr和stmt节点的visited_by方法，并且将visitor本身传给这个方法。我们继续追踪visited_by方法的代码。如果我们简单的实现visited_by方法，可以在每个Tensor IR节点类中，实现以下的函数（以var节点为例）：

```
class var_node: public expr_base {
    virtual expr visited_by(ir_visitor_base_t* vis) override {
        return vis->visit_impl(this->node_ptr_from_this().static_as<var>());
    }
}
```

上面的简单版visited_by函数中，需要先把this指针（指向var_node）转换为GraphCompiler包装的智能指针：var，然后调用ir_visitor_base_t中的visit_impl中对应的重载函数expr visit_impl(var v)，这样由调用链：ir_visitor_base_t::dispatch_impl,expr_base::visited_by,ir_visitor_base_t::visit_impl完成了输入基类指针，调用子类的visit_impl的过程。

上面代码中this->node_ptr_from_this()会将this指针（指向var_node）转换为一个expr指针对象。这个node_ptr_from_this函数的用法类似于std::shared_ptr配套的shared_from_this()函数，将this指针转换为智能指针（其实node_ptr_from_this的内部实现也是基于shared_from_this的）。后面的static_as<var>()则是将基类指针对象expr转换为子类智能指针var。由于我们知道this指针是var_node，所以这个指针转换一定是有效的。

为了实现IR节点子类的visited_by，我们其实只要复制上面的visited_by代码，将其中的static_as<var>改成转换到对应的子类指针即可。但是为每一个类都复制黏贴相似的代码，会造成大量的代码重复，并且很重要的一点是，代码的逼格就下降了（哈哈哈哈）。我们想到C++为我们提供了自动生成大量类似代码的方式，~~宏：正是在下~~即“模板”。我们看到不同子类的visited_by代码其实只有static_as后面的类型不同，提供一个模板，为所有的子类都提供这样的visited_by代码即可。

在src/backend/graph_compiler/core/src/compiler/ir/visitable.hpp这个文件中，实现了模板化的visited_by：

```
template <typename T, typename Base>
node_ptr<Base, Base> visitable_t<T, Base>::visited_by(ir_visitor_base_t *vis) {
    using ptr_ty = node_ptr<T, Base>;
    return vis->visit_impl(static_cast<T *>(this)
                                    ->node_ptr_from_this()
                                    .template static_as<ptr_ty>());
}
```

这个visited_by方法属于visitable_t<T, Base>模板类，其中模板参数T是需要visit子类的类型，Base是T类型这个IR class的基类（expr_base或者stmt_base_t）。每个IR节点类的子类需要从visitable_t中继承visited_by方法。例如tensor_node:

```
class tensor_node : public expr_base,
                    public visitable_t<tensor_node, expr_base> {
    ...
}
```

有些读者可能会发问了，在定义tensor_node的时候，让它继承visitable_t<tensor_node, expr_base>这不会有循环定义的问题吗？这其实是C++模板编程中常见的代码模式：奇异递归模板模式(Curiously Recurring Template Pattern，CRTP)，可以参考这篇文章。有了CRTP，可以将重复的代码放在CRTP父类中，通过继承的方式完成模板代码复用。

我们对IR Visitor的介绍就到这里。下一篇我们将讨论如果将Tensor IR转换编译为可执行代码：代码生成和即时编译（JIT）。