深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第5篇 Tensor IR（续）

关于作者以及免责声明见序章开头。

题图源自网络，侵删。

本篇文章继续讨论Tensor IR在Graph Compiler中的相关实现代码。这篇文章将会讨论一些代码上的细节，以帮助读者之后更好地阅读GraphCompiler的相关代码。本文的相关示例代码在 https://github.com/Menooker/graphcompiler-tutorial/tree/master/Ch5-TensorIR github.com/Menooker/graphcompiler-tutorial/tree/master/Ch5-TensorIR

上篇中我们说到了Tensor IR节点互相之间的引用是通过`expr`和`stmt`这两个GraphCompiler定义的C++指针对象完成的。`expr`和`stmt`是STL指针`std::shared_ptr`的包装，来实现对被引用对象的声明周期管理，并且在原有shared ptr的基础上，拓展了一些功能。我们先来看一下这两种指针对象的实现。

expr和stmt 指针对象

我们先从Expr的指针包装`expr`开始。在Expr各个子类的定义中，已经反复出现了`expr`这样的成员类型，例如：

```
/**
 * The cast node, which converts an expression from one type
 * to another
 * @param type the destination type of the casting
 * @param in_expr the expression to convert
 * */
class cast_node : public expr_base, public visitable_t<cast_node, expr_base> {
public:
    static constexpr sc_expr_type type_code_ = sc_expr_type::cast;
    expr in_;
    void to_string(ostream &os) const override;
    bool equals(expr_c other, ir_comparer &ctx) const override;
    expr remake() const override;
};
```

类型转换节点`cast_node`中，就存放了`expr in_;`，表示类型转换的输入表达式。我定位到`expr`的定义，在`sc_expr.hpp`中：

```
using expr = node_ptr<expr_base, expr_base>;
```

看到`expr`只是`node_ptr<expr_base, expr_base>`这个类的别名。而模板类`node_ptr`则是`stmt`和`expr`智能指针对象共同的实现来源。在精简暂时无关的代码后，定义如下：

```
template <typename T, typename Base>
class node_ptr : public node_ptr_impl_t<T, Base> {
public:
    // ...
};
```

有读者可能会觉得奇怪，为什么不直接使用`node_ptr_impl_t<T, Base>`，而是要先让`node_ptr`继承`node_ptr_impl_t`，然后使用`node_ptr`。这主要是为了特化`node_ptr<expr_base, expr_base>`，并且能复用`node_ptr_impl_t`的代码。这个我们在稍后就能看到。

`node_ptr_impl_t`最核心的代码如下所示：

```
template <typename T, typename Base>
class node_ptr_impl_t {
public:
    // the implementation is based on shared_ptr<Base>
    using impl_ptr = std::shared_ptr<Base>;
    using type = T;
```

```
    impl_ptr impl;
    //...
}
```

对于node_ptr_impl_t<T, Base>，本质上就是std::shared_ptr<Base>。第一个模板参数T是当前这个指针指向的子类类型，可以是expr_base,add_node，sub_node等等。第二个模板参数Base是这个子类T的最远基类，即expr_base或者stmt_base。和shared-ptr一样我们定义了一系列拷贝、移动构造函数和operator=:

```
template <typename T, typename Base>
class node_ptr_impl_t {
public:
    /// ...
    template <typename _Arg>
    using _assignable =
            typename std::enable_if<std::is_convertible<_Arg *, T *>::value,
                    Base>::type;
    static_assert(
            is_base_of_t<Base, T>::value, "T should be a subclass of Base");
    // constructible from a sub-class node_ptr<T2, Base>, where T2 < T
    template <typename T2>
    node_ptr_impl_t(const node_ptr<T2, _assignable<T2>> &other) noexcept
        : impl(other.impl) {}

    // Move-constructible from a sub-class node_ptr<T2, Base>, where T2 < T
    template <typename T2>
    node_ptr_impl_t(node_ptr<T2, _assignable<T2>> &&other) noexcept
        : impl(std::move(other.impl)) {}

    // Constructs an empty node_ptr_impl_t
    node_ptr_impl_t() noexcept = default;

    template <typename T2>
    node_ptr_impl_t &operator=(
            const node_ptr<T2, _assignable<T2>> &other) noexcept {
        impl = other.impl;
        return *this;
    }

    // move-assignable from node_ptr
    template <typename T2>
    node_ptr_impl_t &operator=(node_ptr<T2, _assignable<T2>> &&other) noexcept {
        impl = std::move(other.impl);
        return *this;
    }
}
```

有了这些构造函数和等号赋值函数，我们可以实现子类指针自动转换到父类指针的功能。实现方式是这样的：我们通过STL提供的enable_if定义了_assignable<_Arg>，只有当_Arg*可以自动转换到当前指针类型T*的时候，_assignable<_Arg>才有定义，这样可以实现诸如node_ptr_impl_t<add_node, expr_base>到node_ptr_impl_t<expr_base, expr_base>的自动转换。而且C++编译器能够自动拒绝node_ptr_impl_t<expr_base, expr_base>到node_ptr_impl_t<add_node, expr_base>的自动转换，因为expr_base不是add_node的子类。

node_ptr_impl_t还有更多内容，加强了子类指针转换的功能：

```
template <typename T, typename Base>
class node_ptr_impl_t {
public:
    /// ...
    template <typename T2>
    bool isa() const noexcept {
        ...
    }

    template <typename T2>
    bool instanceof () const noexcept {
        ...
    }

    template <typename T2>
    node_ptr<typename T2::type, Base> static_as() const noexcept {
        ...
```

```
    }

    template <typename T2>
    node_ptr<typename T2::type, Base> as() const noexcept {
        ...
    }

    template <typename T2>
    node_ptr<typename T2::type, Base> checked_as() const {
        ...
    }

    template <typename T2>
    node_ptr<typename T2::type, Base> dyn_as() const noexcept {
        ...
    }
}
```

为了阅读体验，我们略过了这些函数的内容。isa和instanceof都是用于检查一个父类指针指向的对象是否为某个子类（isa是"is a"的意思）。例如ptr是expr类型的父类指针，那么代码：ptr.isa<add>()将会返回这个指针是否指向一个add_node。isa和instanceof的不同在于，isa是通过指针指向对象的node_type_成员来判断子类类型的，而instanceof是通过C++ dynamic cast来检查的。isa速度比instanceof快，但是isa只能检查是否为继承关系的“叶子”节点，即“add_node”，“cast_node”等，无法检查继承关系的中间节点，例如“binary_node”。（读者可以思考一下为什么会有这样的限制）。instanceof则没有这样的限制。

后面的static_as可类比为"static_cast"。它可以将一个父类指针直接转换为子类指针，但是不会做类型检查。as和dyn_as也用于将父类指针转换为子类指针。它们会分别通过isa和instanceof检查指针的实际类型，如果类型不符合期待的子类，将返回空指针。checked_as则是将类型检查放到assert断言中，在release模式下，与static_as没有区别。在debug模式中，如果不是预期的类型，则会报错。

我们来看node_ptr_impl_t最后一部分代码：

```
template <typename T, typename Base>
class node_ptr_impl_t {
public:
    /// ...

    // operator *
    T &operator*() const noexcept { return *get(); }
    // operator ->
    T *operator->() const noexcept { return get(); }
    // gets the contained pointer
    T *get() const noexcept { return static_cast<T *>(impl.get()); }

    /**
     * Checks if the node_ptr contains any pointer
     * @return false if the node_ptr is empty
     * */
    bool defined() const noexcept { return impl.operator bool(); }

    /**
     * Checks if the node_ptr contains the same pointer of another
     * @param v the other node_ptr to compare with
     * @return true if the node_ptrs are the same
     * */
    bool ptr_same(const node_ptr_impl_t &v) const noexcept {
        return v.impl == impl;
    }
}
```

与shared_ptr类似，node_ptr_impl_t重载了*与->运算符。defined函数将会返回这个指针是否是智能指针。ptr_same则可以用于比较两个指针是否相等。Graph Compiler没有将"=="运算符定义为指针比较（即operator==）。对Tensor IR指针进行“==”比较将会产生一个cmp_eq_node。

对于每个Expr的子类，我们都利用node_ptr模板，定义了对应的指针。对于XXX_node，定义了指针指针XXX和XXX_c。XXX_c是指向对应的指针是const XXX_node*。例如add是add_node的智能指针，add_c是const add_node的智能指针。

对expr_base的指针指针GraphCompiler进行了特化，添加了以下这些成员：

```
template <>
class node_ptr<expr_base, expr_base>
    : public node_ptr_impl_t<expr_base, expr_base> {
public:
    // ...

    // converter from c++ float to f32 `constant` IR
    node_ptr(float v);
    // converter from c++ int32_t to s32 `constant` IR
    node_ptr(int32_t v);
    // converter from c++ uint64_t to index `constant` IR
    node_ptr(uint64_t v);
    // ...
    // converter from c++ bool to boolean `constant` IR
    node_ptr(bool v);
    /**
     * Generates a lvalue_proxy_t with a single index. The lvalue_proxy_t
     * can be further "assigned" with expr or be used as expr
     * */
    lvalue_proxy_t operator[](expr index);

    //... more definitions
};
```

形如node_ptr(float v);的构造函数可以将一个C++的值自动转换为一个constant_node指针，这个constant_node内部的值即是这个C++值。GraphCompiler同时还对expr和expr_c重载了常见的C++运算符，例如加减乘除，取余运算等等。

```
expr operator+(expr_c a, expr_c b);
expr operator-(expr_c a, expr_c b);
expr operator*(expr_c a, expr_c b);
expr operator/(expr_c a, expr_c b);
```

这样可以方便地在C++代码中构造Tensor IR。例如C++代码中，val是一个expr，那么

```
val+2
```

会是一个expr，指向的对象是一个add_node，这个加法节点由val指向的expr和2这个常量节点组成。注意到，由于有node_ptr(int32_t v);这个构造函数的存在，所以代码中的2将会自动构造一个expr。这个构造函数将会生成一个指向常量节点的expr指针。例如在sc_expr.cpp中，可以看到这个构造函数的定义是：

```
expr::node_ptr(int32_t v) : parent(builder::make_constant(v)) {}
```

代码builder::make_constant(v)会创建一个值为v的constant_node指针。构造函数将这个指针交由父类继续完成指针对象的构造。

对于加法+，GraphCompiler重载了operator+。上述val+2的代码最终会调用这个重载运算符:

```
    inline expr operator+(const expr_c &l, const expr_c &r) {
        return builder::make_add(l, r);
    }
```

node_ptr<expr_base, expr_base>同时也有[]运算符的重载，也就可以支持对于expr进行Tensor索引的操作。这样的C++代码将会生成一个index_node这样的expr节点指针。

类似地，GraphCompiler也封装了Stmt语句部分Tensor IR的智能指针，这里不再赘述。

小结:

1) GraphCompiler提供了智能指针的封装。对于Expr表达式和Stmt语句节点 XXX_node，定义了指针指针XXX。XXX_c是指向对应的指针是const XXX_node*。

2) Tensor IR智能指针提供了isa等工具来转换子类父类指针

3）Tensor IR智能指针，例如add，是node_ptr<add_node, expr_base>的别名，它继承了node_ptr_impl_t<add_node, expr_base>类的所有方法，内部存储的是std::shared_ptr<expr_base>

4) expr和expr_c除了继承自node_ptr_impl_t的方法，还提供了C++数值到Tensor IR的自动转换，以及各种运算符重载，用以方便地组合Tensor IR表达式，组成新的表达式

C++中创建Tensor IR对象

下面我们来看在GraphCompiler中，我们是如何创建Tensor IR对象的。为了调试，我们可能需要打印Tensor IR。这一章中，我们也将讨论如何输出Tensor IR。

通过make_stmt和make_expr创建Tensor IR

`make_stmt`和`make_expr`是创建Expr和Stmt最基本的方法。它们的实现较为简单，只需要通过STL的`make_shared`创建智能指针，然后转换为对应的`node_ptr`即可。`make_stmt`和`make_expr`的模板参数`T`是需要创建的IR节点的类型。注意这里的`T`是IR节点的名字（例如`add_node`），不要传入IR指针的类型，例如`add`。函数的参数是需要调用的IR节点构造函数参数。

```
template <typename T, typename... Args>
node_ptr<T, expr_base> make_expr(Args &&... args) {
    std::shared_ptr<T> ptr = std::make_shared<T>(std::forward<Args>(args)...);
    return node_ptr<T, expr_base>(std::move(ptr));
}
```

下面的例子是通过`make_stmt`和`make_expr`创建一段IR，IR中定义了变量`a`，并且将表达式`1+2`赋值给`a`变量。

```
    auto v_stmts = make_stmt<stmts_node_t>(std::vector<stmt>());
    auto var_a = make_expr<var_node>(datatypes::s32, "a");
    v_stmts->seq_.emplace_back(make_stmt<define_node_t>(var_a, linkage::local, expr()));
    auto v_1_2 = make_expr<add_node>(make_expr<constant_node>(int64_t(1)), make_expr<constant_node>(int64_t(2)));
    v_stmts->seq_.emplace_back(make_stmt<assign_node_t>(var_a, v_1_2));

    std::cout << v_stmts << std::endl;
```

Tensor IR的节点指针重载了流输出运算符`<<`，所以我们可以直接将结果输出到`std::cout`。最终在终端得到的结果是：

```
{
  var a: s32
  a = (1 + 2)
}
```

这就是这个IR的“人类可读”字符串表示。

通过IR builder创建Tensor IR

`make_stmt`和`make_expr`在实际使用中会有一些不方便的地方。它们的函数参数是通过模板方式进行传递的：

```
node_ptr<T, expr_base> make_expr(Args &&... args) { ... }
```

用户在实际使用这两个函数创建IR的时候，仅仅通过查看`make_expr`的参数列表是无法得知某个IR节点的参数，只能去查询这个IR节点的构造函数的参数列表。在IDE中开发Tensor IR同样也有这样的问题。我们输入`make_expr<add_node>(`之后，IDE也无法提示我们需要哪些参数。为了给Tensor IR用户明确的创建IR API定义，我们封装了一层IR builder接口。头文件位置在

https://github.com/oneapi-src/oneDNN/blob/dev-graph/src/backend/graph_compiler/core/src/compiler/ir/builder.hpp

相关的函数定义在`sc::builder`命名空间下（GraphCompiler项目的大部分函数和类都在`sc`命名空间）。IR builder定义了大量形如`make_*`的函数，用于创建IR节点。例如创建add节点，可以调用：

```
/**
 * Makes a add (+) node
 * @param left left hand side
 * @param right right hand side
 * @return the created node
 * */
expr make_add(const expr_c &left, const expr_c &right);
```

调用这个函数和调用`make_expr<add_node>(l, r)`是等价的，只是使用上更加方便。

对于Stmt，IR builder也提供了对应的创建节点的API。例如：

```
/**
 * Makes an if_else, the statement is not attached to any builder
 * @param condition_ the condition, should be boolean typed
 * @param then_case_ the `then` block
 * @param else_case_ the `else` block, nullable.
 * @return the pushed node
 * */
stmt make_if_else_unattached(const expr_c &condition, const stmt_c &then_case,
        const stmt_c &else_case);
```

可以看到builder namespace下面所有的Stmt的make_*函数都带有unattached后缀。这主要是为了区分“无状态”IR builder和“有状态”的IR Builder。在这一节中，我们介绍的是无状态的IR builder。即本节介绍的IR builder API在创建IR节点后，会直接返回，不会在内部记录这个IR节点。与此相对应的是有状态的IR builder。它可以被用来更方便地通过C++代码创建Tensor IR。我们将在下一节介绍。

下面我们试着用IR builder重写上一节中的代码示例：

```
    using namespace sc::builder;
    auto v_stmts = make_stmts_unattached({}).checked_as<stmts>();
    auto var_a = make_var(datatypes::s32, "a");
    v_stmts->seq_.emplace_back(make_var_tensor_def_unattached(var_a, linkage::local, expr()));
    auto v_1_2 = make_add(make_expr<constant_node>(int64_t(1)), make_expr<constant_node>(int64_t(2)));
    v_stmts->seq_.emplace_back(make_assign_unattached(var_a, v_1_2));

    std::cout << v_stmts << std::endl;
```

像写C++代码一样创建Tensor IR

通过上面IR builder，我们已经可以从expr开始构造stmt，再构造出IR function。但是这种创建IR的方式还是不够方便和直观。特别是我们内部大量的单元测试中，我们可能需要通过C++手写输入和输入的IR，用于和某些GraphCompiler的输出结果进行比较。如果使用调用Builder API的方式，那么代码会特别冗长。GraphCompiler内部还有一种神奇的方式，可以很方便地通过C++代码创建Tensor IR。

我们首先include头文件：

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

然后在main函数中，写下这样的代码：

```
    builder::ir_builder_t bld;
    _function_(datatypes::f32, some_func, _arg_("idx", datatypes::s32), _arg_("A", datatypes::f32, {100, 200}))
    {
        _bind_(idx, A);
        _var_(b, datatypes::f32);
        b = 0;
        _for_(i, 0, 200)
        {
            b = b + A[{idx, i}];
        }
        _return_(b);
    }

    std::cout << some_func;
```

没错，这是一段C++的代码。这段代码的作用是，创建了一个IR Function，这个IR Function的指针存放在了变量some_func中。我们可以把这个函数通过std::cout打印出来，最终我们可以在终端看到这个IR function和它内部的Expr和Stmt：

```
func some_func(idx: s32, A: [f32 * 100 * 200]): f32 {
  var b: f32
  b = 0
  for i in (0, 200, 1) {
    b = (b + A[idx, i])
  }
  return b
}
```

这个IR Function的内容和创建IR的C++代码可以说十分相似了。而这样的C++代码也相比上一节使用无状态IR Builder的方式更加简洁明了。GraphCompiler在创建Tensor IR的时候和单元测试中大量运用了这样的方式。

接下来我们逐一解释上面的代码。

```
builder::ir_builder_t bld;
```

首先我们创建了一个ir_builder_t对象。这就是上一节已经提到的“有状态”的IR Builder。后续代码将会通过这个对象完成IR的组建。

我们接下来申明了一个IR function，函数名为some_func，这个IR function在构建完成后，IR function对象的指针将会存储到同名C++变量some_func中。这个C++变量的类型是IR function的智能指针类型。这个Function在IR中的返回值类型是f32，有两个参数idx和A，其中，idx的类型是int32，A的类型是维度为[100*200]的f32类型Tensor。

这仍然是C++的代码。_function_和_arg_都是定义在easy_build.hpp中的C++宏。利用这两个宏，我们就能在C++代码中建立IR function和函数参数。

```
_function_(datatypes::f32, some_func, _arg_("idx", datatypes::s32), _arg_("A", datatypes::f32, {100, 200}))
{
    ...
}
```

继续解读代码：

```
_bind_(idx, A);
```

_bind_是easy_build.hpp中的C++宏，表示将当前IR function定义的函数参数绑定到C++变量中。这样后续C++代码操作idx和A，也就在操作IR function的argument列表中的这两个Expr节点。

继续解读代码：

```
_var_(b, datatypes::f32);
b = 0;
```

定义了在IR function内部的本地变量b，它的类型是f32。然后将b赋值为0。这里_var_是easy_build.hpp中的C++宏。b = 0这行代码其实并没有真的对变量b赋值，而是创建了一个assign_node，记录下来变量b和0这个常量。这是通过C++中重载等号=得以实现的。这里的C++常量0将会被自动转换为constant_node。这是通过expr的构造函数自动完成的。

后面：

```
_for_(i, 0, 200)
{
    ...
}
```

将会在IR function中插入一个for_loop_node节点，循环变量是i，循环从0开始到200结束（不包括200），循环步长可以不用填写，默认为1。

for循环内部有：

```
b = b + A[{idx, i}];
```

我们已经了解了这里的等号“=”将会创建一个assign_node。代码A[{idx, i}]返回的是一个expr。回顾上文有个expr智能指针的描述，expr类重载了运算符[]。我们知道A本身是expr（精确地说，可以等价于expr），对于A加上运算符[]，将会创建一个对Tensor进行索引的expr节点indexing_node。代码中对Tensor A索引的位置是{idx, i}。我们知道GraphCompiler的Tensor是多维数组，我们IR中也允许进行多维索引。索引的位置就是(idx,i)。代码A[{idx, i}]返回的将是Tensor A的对应读取后的expr。我们将结果和变量b相加，存储回b中。上文我们也已经提到，对于expr类型我们重载了+等常见运算符，所以我们可以简单地用加号组合expr。

需要再次强调的是，这些在_funtion_中的C++代码并没有真正执行这些赋值和加法，而是创建了对应的IR节点。_funtion_内的代码相当于是《黑客帝国》中的虚拟世界。

最后IR function可以创建return_node：

```
_return_(b);
```

有读者可能会好奇，GraphCompiler的easy_build.hpp是通过什么魔法简化IR创建的过程的呢？还记得我们在第一行代码中创建了一个ir_builder_t对象吗？所有的创建stmt的动作（包括_for_, _return_, 等号赋值等等）都会往这个对象中添加一个stmt，最后我们可以从ir_builder_t对获取添加的stmt列表，组成一个stmts_node。那么我们又是如何处理_for_这样的带有花括号域scope（"{}"）的代码的呢？这里主要运用了C++ RAII的技巧。例如_for_的实现：

```
#define _for_(IDX, ...) \
    for (auto IDX : ::sc::builder::range_nobind(#IDX, __VA_ARGS__))
```

函数range_nobind返回一个RAII对象for_range_simulator_t，这个对象在构造函数中，会在当前的ir_builder_t对象（）中push一个"scope"。在_for_的花括号"{}"结束后，这个for_range_simulator_t对象被析构。析构函数中，在当前的ir_builder_t对象pop出一个scope，这个scope将会被转换成stmts_node，并且创建一个新的for_loop_node，将得到的stmts_node作为for_loop_node的body。最后将新的for_loop_node放入ir_builder_t对象中。

由于easy_build.hpp的原理与我们GraphCompiler的核心部分关系不大，基本上直接使用即可，所以我们这里就不展开解释背后的原理了。

下一篇文章将介绍GraphCompiler的另一套IR——Graph IR。