深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第4篇 Tensor IR

关于作者以及免责声明见序章开头。

本篇将讨论GraphCompiler中的Tensor IR的具体定义和如何搭建一个可以被JIT Engine执行的Tensor IR。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：

简单来说，Tensor IR是用来详细描述算子内具体运算的IR（中间表达）。在深度学习的计算中，一个计算图最初是通过Graph IR表达的，然后会被lower成Tensor IR。Tensor IR有着类似C语言的表达能力，可以通过IR形容的计算来表达if for等控制流，并且表达出输出tensor每个具体位置的计算方式。Tensor IR在Graph Compiler内部是通过C++对象进行存储的。一个Tensor IR函数可以认为是一颗C++ IR对象的“树”。GraphCompiler的tensor IR pass可以通过对IR对象的变换来优化IR代码。

Tensor到底是什么

首先先解释一下在数学上、Graph IR、Tensor IR中Tensor代表了什么。

在数学上，Tensor（张量）可以简单理解为多维向量。我们常说的向量可以认为是1维张量。深度学习中，各个算子得到的结果，以及训练时产生的梯度都是张量。有兴趣可以看一下百度百科的相关词条：

Graph IR中，每个算子Op的输入输出都是Tensor，在代码中，我们使用`graph_tensor`对象来表示Graph IR中的Tensor。而在Tensor IR中的Tensor则代表了`graph_tensor`在内存中的实际表示。Tensor IR中的`tensor_node`对应了C语言中的多维数组。在GraphCompiler中，使用行优先（row major）方式存储Tensor——即对于二位矩阵AxB来说，每B个矩阵元素的一行，将存放在连续的内存空间中。下一行的起始地址和上一行的结束地址相邻。一般来说，对于N维Tensor `[D1 x D2 x D3 x ... x Dn]`, 元素下标为`[i1, i2, i3, ..., in]`的地址为`i1 * (D2*D3*...*Dn) + i2 * (D3*D4*...*Dn) + ... + i_x * (D_{x+1}*...*D_{n}) + ... + in`。 注意`tensor_node`依然只是逻辑上的IR节点，而没有为存储Tensor分配实际的内存。实际内存分配需要在IR被编译成可执行代码之后才会被分配或传入。

Tensor IR的语义详解

详见本系列的这两篇文章：

第9篇 Tensor IR语义详解 (1)

第10篇 Tensor IR语义详解 (2)

本篇中主要以分析C++代码中的Tensor IR为主。

Tensor IR的定义和分类

Graph Compiler的Tensor IR定义在backend/graph_compiler/core/src/compiler/ir目录中，最核心的头文件是sc_expr.hpp, sc_stmt.hpp和 sc_function.hpp。Tensor IR主要有expr、stmt、IR function和IR module这几个概念。

expr即Expression，表达式，可以参照C语言的表达式定义。Tensor IR中，表达式是一类代表某个计算结果的IR，例如float类型常量`1.0`就是expr。又比如一个加法表达式`a + 1.0`表达了将变量`a`与常量标量`1.0`相加的结果。

stmt即Statement，语句。可以类比于C语言中的if、for等控制流、赋值语句、以及分号“;”所分隔的每一行代码。Tensor IR中，stmt可能会引用到其他的expr或者stmt节点。statement和expression主要的区别在于，expr是带有“返回值”的，一个expr节点可以有指向其他的expr成员指针，但是不存在指向stmt的指针。stmt则是没有“返回值”的代码，例如for循环或者if语句。

IR function即用expr和stmt搭建出来的Tensor IR函数，当中可以包含一系列用expr和stmt表达的运算。一个IR function中允许通过`call_node`这个expr来调用其他的IR function。和C语言的函数类似，一个IR function可以指定一个返回类型，同时可以有参数列表，表示输入参数。IR function有一个成员指针`body_`来指向这个函数的内容（stmt）。

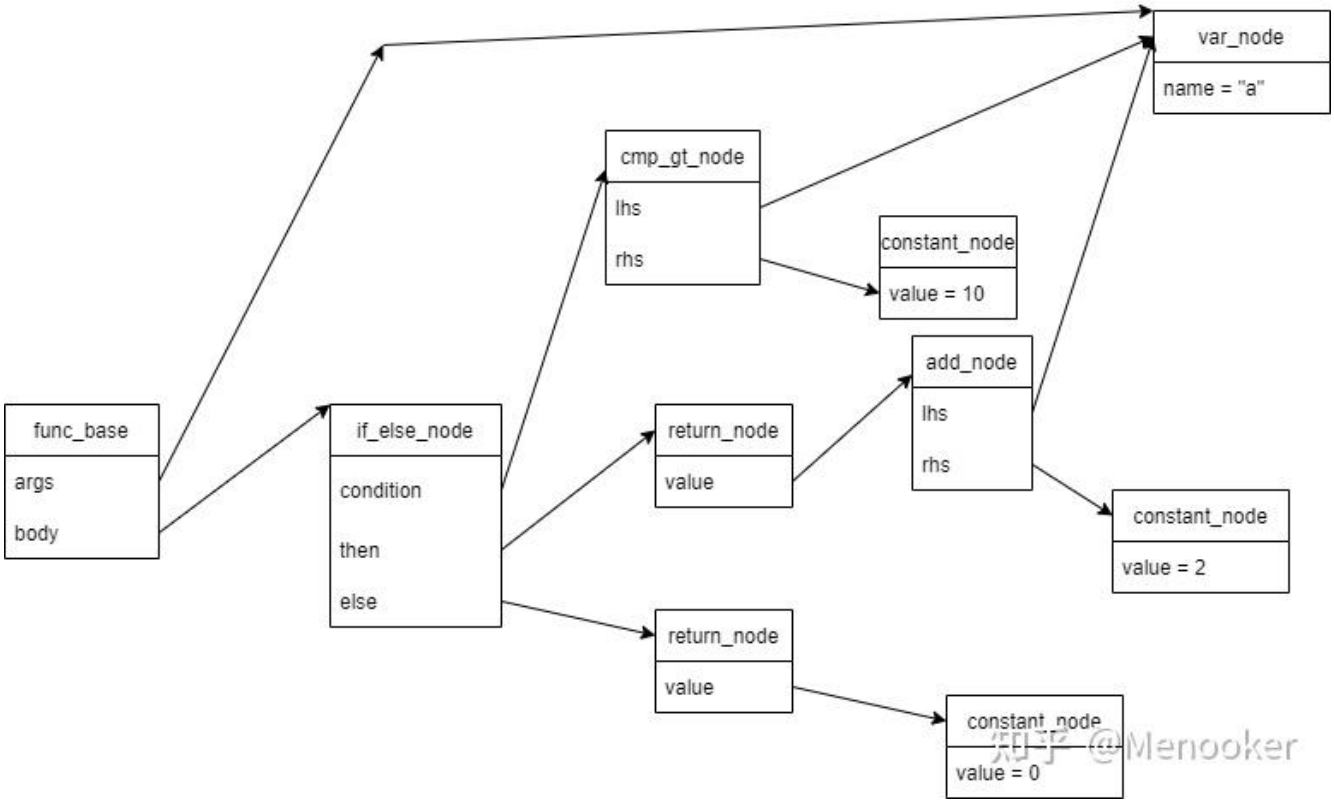
IR module是由多个IR function组成的集合。同一个IR module内的函数可以相互之间调用（调用IR module之外的函数相比之下则会较为复杂）。此外，IR module中还可以定义模块内的全局变量，供IR module内的函数使用。

在Graph Compiler中，一个Graph通常会被编译为一个IR module。其中，每个Op会成为一个IR function。那么IR module中的全局变量有什么用呢？其中一个应用场景就是：在推理（inference）的时候，通常权重（weight）是不变的，而在模型中可能需要对权重进行预处理，例如reorder、量化/反量化。由于每次推理计算中，都需要得到权重预处理后的结果，但是权重不改变的时候，预处理的结果其实可以被缓存下来，避免重复计算。那么我们就需要在IR module中定义一块全局Tensor，用于存放权重预处理的结果。

我们来看一个相对“真实”的Tensor IR的例子，一个IR function的字符串表达为：

```
func some_func(a: s32): s32 {
  if(a > 10) {
      return (a + 2)
  } else {
      return 0
  }
}
```

这个IR function定义了一个叫做"some_func"的函数，输入一个参数，变量类型是`s32`，即"signed 32bit" 整数，返回类型也是`s32`。这个IR function在Graph Compiler的C++对象，以及对象链接关系表示如下图：



Expr代码简析

Expr节点定义在sc_expr.hpp。所有expr节点的基类是`expr_base`，定义如下（删减了一些暂时对本文不重要的代码）：

```
/**
 * The base class of expression IR nodes
 * */
class expr_base : public node_base,
                  public ... /*some other base classes*/ {
public:
    // the data type of the expression
    sc_data_type_t dtype_ = datatypes::undef;
    // the expression type id of the IR node
    sc_expr_type node_type_ = sc_expr_type::undef;

    virtual ~expr_base();
    /**
     * Dumps the IR node as string to the ostream
     * @param os the output stream_t
     * */
    virtual void to_string(ostream &os) const = 0;
    /**
     * Checks if `this` is same as another IR node. May change the internal
     * states of `ctx`
     * @param other the other IR node to compare
     * @param ctx the context of the comparison: how "same" is defined,
     *  the internal states, etc.
     * @return true if the nodes are the same
     * */
    virtual bool equals(expr_c other, ir_comparer &ctx) const = 0;
};
```

expr_base是一个虚类，主要是定义了两个重要的接口to_string和equals，用于将这个expr以人类可读的字符串形式输出，以及比较两个expr是否完全相同。expr_base也定义了两个成员。dtype_是这个expr的数据类型，例如f32，bf16等等。node_type_则是这个expr的节点类型，例如add就表示这个节点是加法节点。如果有一个加法节点，返回的数据类型是f32，那么dtype_将会是f32，node_type_是add。

expr的数据类型由sc_data_type_t表示，定义在

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

Graph Compiler中的类型由基本类型（pointer, s32, u32, u8, s8, f32, bf16等等）以及基本类型的向量类型（例如f32x16）组成。向量类型可以使得我们显式地进行向量化操作——通过一条指令操作多个数据，又被称为SIMD。这在深度学习编译器领域是必不可少的操作。我们可以通过向量类型声明一个向量变量，而在底层这个变量可能会被映射到CPU、GPU中的向量寄存器。同时如果对于内存的操作的数据类型是向量类型，那么将会通过向量指令来进行访存。

简化后的sc_data_type_t定义如下：

```
struct sc_data_type_t {
    sc_data_etype type_code_;
    uint32_t lanes_;
}
```

type_code_即基础类型的enum枚举值。lanes_即向量类型的宽度，即有多少个基础类型元素组成的向量。sc_data_etype枚举了我们支持的每一种基础数据类型，定义如下：

```
enum class sc_data_etype : uint32_t {
    /// Undefined data type.
    UNDEF = 0,
    /// 16-bit/half-precision floating point.
    F16 = 1,
    /// non-standard 16-bit floating point with 7-bit mantissa.
    BF16 = 2,
    /// 16-bit unsigned integer.
    U16 = 3,
    /// 32-bit/single-precision floating point.
    F32 = 4,
    /// 32-bit signed integer.
    S32 = 5,
    /// 32-bit unsigned integer.
    U32 = 6,
    /// 8-bit signed integer.
    S8 = 7,
    /// 8-bit unsigned integer.
    U8 = 8,
    /// data type used for indexing.
    INDEX = 9,
    /// sc::generic_val type, a union type for all supported scalar types
    GENERIC = 10,
    /// boolean
    BOOLEAN = 11,
    /// void type
    VOID_T = 12,
    /// the max enum value + 1
    MAX_VALUE = 13,
    /// general pointer type, also used as a pointer bit mask
    POINTER = 0x100,
};
```

我们回到expr的讨论。每一种不同的expr运算都有对应的expr_base的子类，而每个子类都会对应一个sc_expr_type的枚举值。通过观察sc_expr_type的定义，我们可以看到Tensor IR的expr支持的所有运算：

```
enum class sc_expr_type {
    undef = 0,
    constant,
    var,
    cast,
    add,
    sub,
    mul,
    div,
    mod,
    cmp_eq,
    cmp_ne,
    cmp_lt,
    cmp_le,
    cmp_gt,
    cmp_ge,
    logic_and,
    logic_or,
    logic_not,
    select,
    indexing,
    call,
    tensor,
    tensorptr,
    intrin_call,
    ssa_phi,
    func_addr,
    low_level_intrin,
};
```

其中undef不表示任何一种计算，只是一个占位符。除了undef，这里的每个枚举值都可以找到对应的expr_base的子类。例如add，我们可以找到add_node的定义：

```
/**
 * The node for addition (+)
 * */
class add_node : public binary_node, public visitable_t<add_node, expr_base> {
public:
    static constexpr sc_expr_type type_code_ = sc_expr_type::add;
    add_node(const expr &l, const expr &r)
        : binary_node(sc_expr_type::add, l, r) {};
    void to_string(ostream &os) const override;
    expr remake() const override;
};
```

它是binary_node的子类。binary_node是有两个操作数的节点的基类，它依然是expr_base的子类：

```
class binary_node : public expr_base {
public:
    binary_node(sc_expr_type expr_type, const expr &l, const expr &r)
        : expr_base(l->dtype_ == r->dtype_ ? l->dtype_ : datatypes::undef,
                expr_type)
        , l_(l)
        , r_(r) {}
    // the left hand side expr
    expr l_;
    // the right hand side expr
    expr r_;
    bool equals(expr_c other, ir_comparer &ctx) const override;
};
```

binary_node定义了两个成员变量l_和r_，表示了左手边和右手边的两个操作数。代码中expr l_;声明了一个指向expr_base的智能指针。其中expr是GraphCompiler对std::shared_ptr<expr_base*>的封装。其他代码中出现的expr_c是对std::shared_ptr<const expr_base*>的封装。我们将在下一篇文章中讨论对expr和expr_c智能指针的封装。

从add_node的例子中我们可以看到，Tensor IR中的节点通过拓展expr_base类型的子类来定义各种运算。而每个运算的操作数通过子类的成员变量保存指向操作数expr_base的指针。这样通过遍历一颗expr的“树”，可以得到一个表达式的所有组成部分。

Stmt代码简析

Stmt节点定义在sc_stmt.hpp。Stmt和Expr类似，所有的stmt节点都是stmt_base_t的子类。只是stmt_base_t内部无需记录数据类型，因为与expr不同stmt是没有“返回值”的。简化后的基类定义如下：

```
class stmt_base_t : public node_base,
                    public ... /*other base classes*/ {
public:
    // the statement type id of the IR node
    sc_stmt_type node_type_ = sc_stmt_type::undef;

    stmt_base_t(sc_stmt_type type);
    virtual ~stmt_base_t();
    /**
     * Dump the IR node as string to the ostream
     * @param os the output stream
     * */
    virtual void to_string(ostream &os, int indent) const = 0;

    /**
     * Check if `this` is same as another IR node. May change the internal
     * states of `ctx`
     * @param other the other IR node to compare
     * @param ctx the context of the comparison: how "same" is defined,
     *  the internal states, etc.
     * @return true if the nodes are the same
     * */
    virtual bool equals(stmt_c other, ir_comparer &ctx) const = 0;
};
```

与Expr类似，Stmt中，GraphCompiler也定义了指向stmt_base_t的智能指针stmt，和指向const stmt_base_t的智能指针stmt_c。

sc_stmt_type枚举了所有的Stmt类型，每种枚举值对应一种stmt_base_t的子类：

```
/**
 * The IDs for each statememt node
 * */
enum class sc_stmt_type {
    undef = 0,
    assign,
    stmts,
    if_else,
    evaluate,
    for_loop,
    returns,
    define,
};
```

assign_node对应于C语言的赋值语句。assign_node由两个成员变量var_和value_组成，分别代表赋值等号的左值和右值。其中var_必须是var_node或者indexing_node，即赋值的左值必须是变量(var)或者是对Tensor索引后的结果（indexing）。

stmts_node_t可以认为是Stmt的容器，用来存放Stmt的序列，而它本身也是stmt_base_t的子类:

```
class stmts_node_t : public stmt_base_t,
                     public visitable_t<stmts_node_t, stmt_base_t> {
public:
    static constexpr sc_stmt_type type_code_ = sc_stmt_type::stmts;
    std::vector<stmt> seq_;
    void to_string(ostream &os, int indent) const override;
    bool equals(stmt_c other, ir_comparer &ctx) const override;

    stmts_node_t(std::vector<stmt> &&seq_)
        : stmt_base_t(sc_stmt_type::stmts), seq_(std::move(seq_)) {}
};
```

可以看到stmts_node_t存放了一个指向子节点指针的数组：std::vector<stmt>。在C语言中，与stmts对应的是花括号{...}，花括号内应该有数条Statement语句。

stmts的用途是将多个stmt按顺序组合到一起，作为一个stmt节点被其他stmt或IR function引用。

if_else和for_loop这两个IR节点对应了C语言的if和for，这里不再展开。evaluate内部包含了一个expr，作用是将一个expr转换成stmt。我们可以看到，如果将一个Tensor IR function比喻为一个“人”，那么Stmt是“骨架”，通过stmts和控制流搭起整个IR的运算流程，而Expr是“血肉”，基于Stmt的骨架表示实际的运算。如果一个Expr想要在最终生成的可执行代码中出现，那么它一定需要直接或者间接“依附”在某个Stmt或IR function的参数列表中。在GraphCompiler中，函数调用是以Expr中的call_node形式来表示的。然而如果这个函数的返回值没有出现在赋值语句中，而是想单纯调用函数，丢弃返回值，那么需要将这个call_node，包装成一个Stmt。这就需要evaluate这个Stmt节点。类比C语言，evaluate节点就像是C语言表达式后面的分号";"，表示这个表达式变成了一个语句。

这篇文章介绍了Tensor IR的大致分类，并且初步介绍了Expr和Stmt节点在GraphCompiler中C++类的定义。下一篇将继续讨论GraphCompiler中对于Stmt、Expr指针的封装以及如果手工搭建Tensor IR。

（吐槽：知乎的md文件转换突然把我文档的`XXXX` 都识别成了会换行的代码块了，这什么情况）