深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第13篇 编译运行 Tensor IR

关于作者以及免责声明见序章开头。题图源自网络，侵删。

在之前的系列文章中，我们已经展开讨论了Graph Compiler的运行流程当中的几个重要步骤：从输入的计算图转换为Graph IR，优化Graph IR，转换到Tensor IR，和优化、变换Tensor IR。本文将探讨在经过Tensor IR pass pipeline之后的Tensor IR是如何被编译为可执行代码、然后被最终用户来调用的。

这里我们再次讨论一下编译器中JIT（即时编译，Just in time）和AOT（Ahead of time）的概念。这两个概念的区别在于“编译”的时机：JIT编译“正好”发生在用户运行想要执行的程序之前，而AOT则是预先将程序编译为可执行的代码。传统的C/C++编译器都是AOT的，我们运行gcc/clang生成的可执行程序的时候，不需要再调用编译器。JIT编译器的的一个有名的例子就是Javascript的V8环境，用户输入JS源码，编译器则会编译源码后立即执行。

Graph Compiler选用了JIT模式。主要原因在于，Pytorch Tensorflow这类框架的用户可以任意指定Tensor的shape，以及计算图的连接。我们作为底层编译器，不可能为所有的输入Tensor大小预先编译代码。并且Graph Compiler在知道计算图之后才能为这个图进行算子融合，生成优化后的代码。所以在用户输入计算图之后再进行编译是通常情况下最好的选择。

本文中，我们先来看看Graph Compiler中定义的JIT编译器和调用生成后的代码的接口，然后简单地看一下基于LLVM的JIT engine的实现，最后讨论如何使用Graph Compiler来AOT生成可执行代码。

JIT Engine和JIT module

目前的Graph Compiler有两种JIT编译器的实现：`cfake_jit`和`llvm_jit`。这两种JIT编译器通过不同的方式将Tensor IR转换为内存中的可执行代码。项目中引入了抽象的JIT engine接口类：`jit_engine_t`。`cfake_jit`和`llvm_jit`分别继承了这个类。一个JIT Engine实体类表示了JIT编译器的一种实现。JIT Engine类定义在jit.hpp,定义为：

```
class SC_API jit_engine_t {
public:
    context_ptr context_;
    jit_engine_t(context_ptr context) : context_(std::move(context)) {}

    // jit an ir_module_t into a jit_module
    virtual std::shared_ptr<jit_module> make_jit_module(
            const_ir_module_ptr module, bool generate_wrapper)
            = 0;

    /**
     * Generates a executable module and extract the entry function of the
     * ir_module_t
     * @param m module to generate. Must have entry function defined
```

```
     * @param generic if true, creates a type-erased wrapper for the
     *  function, users can further call `call_generic` on the
     *  generated executable
     * @return the executable function for the entry function
     * */
    std::shared_ptr<jit_function_t> get_entry_func(
            const ir_module_ptr &m, bool generic = true);
    virtual ~jit_engine_t() = default;

    static std::unique_ptr<jit_engine_t> make(const context_ptr &ctx);
    // negotiate with the JIT engine and get the target machine with as
    // many flags as possible the JIT can support in the user given target
    // machine
    static void set_target_machine(jit_kind kind, target_machine_t &tm);
};
```

它提供的主要接口为`make_jit_module`，这个接口接收一个Tensor IR Module（用户无需提前对这个Tensor IR跑pass进行优化），输出为编译后的JIT Module，指向可执行的而二进制代码。参数`generate_wrapper`一般设为true即可，表示为IR module里面的函数生成通用接口的包装函数。JIT编译器的内部应该在这个函数中,将输入的IR module经过Tensor IR pipeline处理。这部分内容可以参考之前的这篇文章：

`make_jit_module`然后对优化后的Tensor IR经过代码生成（codegen）转换为可执行代码，最终返回内存中管理可执行代码的C++对象：`jit_module`。

JIT module类是JIT编译器的编译结果，是编译后的可执行函数代码、以及代码依赖的全局变量的集合。JIT module中的各个元素和IR module中的一一对应，IR中的每个IR function和global tensor、variable都会分别被编译进IR module中的代码部分和数据部分。这个类定义如下：

```
class SC_INTERNAL_API jit_module {
public:
    statics_table_t globals_;
    // the unique id for a JIT module in a process scope
    size_t module_id_;
    // whether to use managed thread pool
    bool managed_thread_pool_;
    jit_module(bool managed_thread_pool);
    jit_module(statics_table_t &&globals, bool managed_thread_pool);
    virtual void *get_address_of_symbol(const std::string &name) = 0;
    virtual std::shared_ptr<jit_function_t> get_function(
            const std::string &name)
            = 0;
    /// This method only exists to help with debugging.
    virtual std::vector<std::string> get_temp_filenames() const {
        return std::vector<std::string>();
    }
    virtual ~jit_module() = default;
};
```

其中成员变量`globals_`是这个模块的静态存储空间，包括了全局变量和全局Tensor所需的内存空间。

由于不同的JIT方式管理生成的代码和在代码中提取想要的函数指针的方式都不尽相同，所以JIT Module和JIT Engine一样也是一个虚基类，不同的JIT编译器会创建不同的JIT Module子类对象。

JIT Module包含了同一个模块中被编译的多个函数，那么如何从JIT Module中选择某个函数来执行呢？我们看到这个类中提供了`get_function`接口，输入是函数的名字，输出则是函数对象的指

针。通常一个IR module会有一个“主函数”。例如通过Graph IR生成的IR module中，主函数就是这个Graph的入口。在JIT Engine中也提供了`get_entry_func`函数，将IR module编译后，直接返回“主函数”的函数对象。

JIT编译器中，通过类`jit_function_t`来表示一个编译后的可执行函数。它的定义如下：

```
class SC_API jit_function_t {
public:
    virtual ~jit_function_t() = default;

    virtual std::shared_ptr<jit_module> get_module() const = 0;
    virtual void *get_function_pointer() const = 0;

    /**
     * Calls the generic wrapper function with default stream context. The
     * module must have been compiled with `generate_wrapper=true`.
     * @param args the arguments
     */
    virtual void call_generic(
            runtime::stream_t *stream, generic_val *args) const = 0;

    /**
     * Calls the generic wrapper function and specifies a user-defined module
     * data. The module must have been compiled with `generate_wrapper=true`.
     * @param stream the runtime stream context
     * @param module_data the module data buffer. It should hold the module
     * scope vars and tensors
     * @param args the arguments
     */
    virtual void call_generic(runtime::stream_t *stream, void *module_data,
            generic_val *args) const {
        throw std::runtime_error("Not implemeneted");
    }

    virtual void *get_module_data() const { return nullptr; }
};
```

这个类中最主要的接口就是`call_generic`，它接收两个参数：stream指针和参数列表指针。这里的stream指向了onednn的stream类，它提供了onednn内部的内存分配器。Graph Compiler生成的kernel在运行时通过这个指针来分配运行时需要的内存。`call_generic`的第二个参数是`generic_val`的指针，表示在内存上连续的多个函数参数。`generic_val`是64位长度的union，可以存放一个指针、整数或者浮点数。定义如下：

```
union generic_val {
    uint16_t v_uint16_t;
    float v_float;
    int32_t v_int32_t;
    int8_t v_int8_t;
    uint8_t v_uint8_t;
    uint64_t v_uint64_t;
    void *v_ptr;
    generic_val() = default;
    generic_val(uint16_t v) : v_uint16_t(v) {}
    generic_val(float v) : v_float(v) {}
    generic_val(int32_t v) : v_int32_t(v) {}
    generic_val(int8_t v) : v_int8_t(v) {}
    generic_val(uint8_t v) : v_uint8_t(v) {}
    generic_val(uint64_t v) : v_uint64_t(v) {}
    generic_val(void *v) : v_ptr(v) {}
};
```

用户在调用JIT function的时候需要先将参数列表存入一个数组中，然后将数组指针传入call_generic的第二个参数。

至此，我们可以整理出用户通过JIT编译器来执行Tensor IR的流程：首先需要创建一个JIT Engine实例，然后调用`make_jit_module`生成JIT module，并且从JIT Module中通过`get_function`方法得到JIT function。最终，调用`call_generic`方法调用生成的函数。用户也可以直接将IR module交给JIT engine的`get_entry_func`函数，直接得到“主函数”的JIT function。

简析基于LLVM的JIT Engine

LLVM是一个著名的编译器后端库，它自带了JIT编译功能。Graph Compiler的llvm jit利用了LLVM来完成Tensor IR到可执行代码的转换。Graph Compiler的llvm jit会首先将Tensor IR转换为LLVM IR，然后调用LLVM的JIT编译器完成编译。llvm jit实现在了

`llvm_jit::make_jit_module`中，Graph Compiler首先调用Tensor IR pass——`llvm_generator_pass`来将Tensor IR转换为LLVM IR，并且提取出Tensor IR用到的静态全局变量，放入`statics_table`中。然后它将调用llvm的Execution Engine（MCJIT）生成内存中的可执行代码。最终将Execution Engine和`statics_table`放入Graph Compiler的`llvm_jit_module`中。

我们再来展开看一下`llvm_generator_pass`的实现。它的源码在

这个pass首先调用了`precodegen_passes`（我们在之前的这篇文章中已经讨论了这个Tensor IR pipeline），对输入的未经规整化和优化的Tensor IR进行处理和优化，然后将优化后的Tensor IR交给`codegen_llvm_vis_t`这个Tensor IR viewer（详见这篇文章对IR viewer的描述）来遍历整个IR module中的所有函数，逐一将每个Tensor IR节点翻译为LLVM IR节点。`codegen_llvm_vis_t`中的部分定义代码和成员变量有：

```
class codegen_llvm_vis_t : public ir_viewer_t {
public:
    context_ptr ctx_;
    LLVMContext &context_;
    IRBuilder<> builder_;
    std::unique_ptr<Module> module_;
    Function *current_func_;
    Value *current_val_;
    // the **pointer** of local var in a function
    std::unordered_map<expr_c, Value *> var_ptr_in_func_;
    std::unordered_map<std::string, Function *> name_to_func_;
    bool is_lvalue_mode_ = false;
    ...
};
```

可以看到它是`ir_viewer_t`的子类。其中，`ctx_`成员是Graph Compiler的上下文。`context_`是LLVM的上下文引用。`builder_`是LLVM的IR builder，这个pass就是通过这个LLVM IR builder来创建LLVM IR。成员`module_`是输出的LLVM IR Module。`current_func_`和`current_val_`是Tensor IR viewer当前访问的Tensor IR所翻译到的LLVM IR。

这个IR viewer重载了所有的expr和stmt的子类的`view`函数。它们的作用是，自顶而下遍历每个IR function中的每个IR节点，并且调用相应的`view`函数。对于表达式expr节点来说，通过`view`访问这个expr节点后，对应的LLVM Value将会存储在这个viewer的`current_val_`成员中。对于stmt节点，view函数调用后将会把对应的LLVM IR生成到LLVM IR Builder中。我们以select表达式为

例。select是Tensor IR中表示C语言中“问号冒号表达式”（例如aaa?123:321）的节点。codegen_llvm_vis_t中处理select的代码在简化后如下所示：

```
void view(select_c v) override {
    auto l = generate_expr(v->l_);
    auto r = generate_expr(v->r_);
    auto cond = generate_expr(v->cond_);
    current_val_ = builder_.CreateSelect(cond, l, r);
}
```

select节点由三部分组成，cond_成员表示“问号”前的布尔类型值，成员l_和r_表示冒号左右的两个待选值。这里通过codegen_llvm_vis_t::generate_expr成员函数将select节点的三个成员转换为llvm::Value*，然后通过llvm的IR builder创建LLVM中的select节点，然后通过成员变量current_val_返回。

codegen_llvm_vis_t的generate_expr的实现如下：

```
Value *generate_expr(const expr_c &e) {
    dispatch(e);
    return current_val_;
}
```

它首先调用IR viewer的dispatch方式，实现递归访问这个expr和它的子节点。dispatch函数中，IR viewer会自动调用输入expr指针对应的子类重载的view函数（例如上文的处理select的view函数）。在dispatch完成后，按照约定当前codegen_llvm_vis_t对象的成员变量current_val_中存储的将会是输入的expr对应的LLVM Value指针（例如上文中处理select后，就是通过这个成员变量记录生成的LLVM select节点的）。然后generate_expr函数只需返回这个LLVM Value指针即可。

类似地可以通过重载对应的view函数实现其他expr和stmt到LLVM IR的转换。这也是Tensor IR Viewer的惯常做法。

在有些场景下，用户可能希望Graph Compiler将编译的结果保存到文件，而不是JIT直接运行。这就相当于我们之前说的AOT编译模式。Graph Compiler中，我们只要手动调用llvm_generator_pass（而不是直接调用llvm jit），就可以从Tensor IR得到LLVM IR，用户代码中可以选择保存LLVM IR或者是将它进一步转换为Object file（LLVM提供了相应的功能）。这样就可以通过手动调用的方式实现AOT编译模式。

至此，我们在专栏的文章中已经讨论了Graph Compiler最基础的部分，包括了Tensor IR和Graph IR的语义、定义，实现Tensor IR pass的基础工具（IR Visitor、Viewer），实现Graph IR pass的基础工具（Op Visitor），Graph IR到Tensor IR的转换，以及本篇的代码生成和JIT。基于这些基础模块，Graph Compiler实现了各种pass来对IR进行调整和优化，同时Graph IR中每个Op都有内部实现来生成对应的Tensor IR。专栏接下来的主要内容将关于Pass和Op内部的实现，以及Graph Compiler如果进行算子融合（Fusion）。