深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第11篇 IR Pass

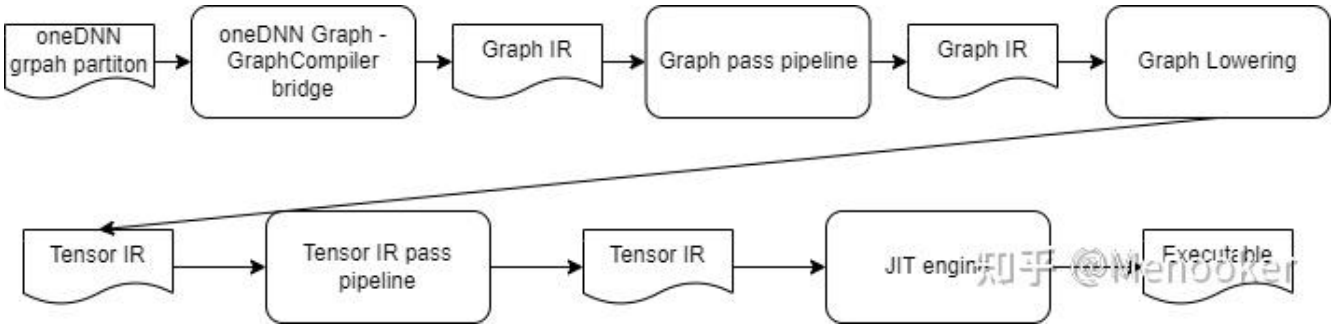
关于作者以及免责声明见序章开头。题图源自网络，侵删。

从输入Graph IR开始，GraphCompiler会将IR进行各种改写和变换。这些变换主要有两个目的：一是对IR进行lower，让IR从表达用户的编程意图变为表达对底层硬件的具体操作，二是对IR进行优化。 每一次IR的变换被称为一个pass。Pass可以理解为是一个函数y=f(x)，其中输入x和输出y都是IR。如果一个pass的输入输出都是Graph IR，我们称这个pass为Graph Pass，类似的，我们将输入输出为Tensor IR的pass为Tensor Pass。GraphCompiler中，我们上一章提到的graph lowering也是一个特殊的pass，它的输入是Graph IR，输出是Tensor IR。这个pass连接了Graph IR的世界和Tensor IR的世界。详见：

在GraphCompiler中，我们在设计pass的时候会尽量让一个pass完成一件简单的变换。如果有一个复杂的变换，我们会实现多个pass组合在一起完成。这是为了让IR的变换能够模块化，能够尽量复用pass的代码，同时调试和测试简单的pass也更为容易。GraphCompiler编译代码的主要流程就是将大量的pass前后连接到一起，组成一条“流水线”。

在这篇文章中，我们将大致浏览一下GraphCompiler会对IR做哪些变换，并且讨论pass的基础接口。

这里我们把本系列第二篇中的Graph Compiler总体流程图图重新搬出来：



可以看到，GraphCompiler的pass可以大致分为两个阶段，第一个阶段是Graph pass阶段，从输入Graph IR到将Graph IR转换到Tensor IR为止。然后是Tensor IR pass，最终通过JIT编译器这个特殊的pass翻译到可执行代码。

Graph pass

所有的graph pass都是C++函数，接口如下：

```
void (*)(sc_graph_t &, const context_ptr &);
```

pass接收一个graph对象，直接（inplace）在graph对象上进行修改。为了实现Graph pass，我们可能需要按照某些顺序（例如拓扑排序）来遍历整个计算图。这可以通过Graph visitor来实现，详见本系列文章第七篇：

目前在Graph IR上的pass有这些：

- analysis_quantized：分析计算中是否需要进行“量化”操作
- graph_inline：将复杂的组合Op（graph op）拆解成小的基本Op
- constant_optimization
- quantized_info_propagation
- quantized_graph_reschedule
- quantize_inline
- elemtwise_bcast_swap
- permute_propagation
- quantize_op_compensation
- elemwise_dimension_alignment
- layout_propagation：为每个op选择合适的input tensor layout，如果输入tensor的layout与op期望的layout不符合，则会插入reorder op进行内存重排列。还会试图进行全图优化，试图找到reorder最少的layout组合。
- tensor_view_transform
- graph_simplify
- global_reschedule
- partial_reduce_replace
- fuse_ops：在graph上将多个相邻算子进行融合
- horizontal_merge：graph compiler中默认会序列化执行op，在op内部进行并行，每个op执行时会占用全部的计算资源。如果一个op内部的并行度不高（例如在32核心的CPU上执行64x64的矩阵乘法），那么这个op独占的计算资源其实是被浪费了。所以这个pass会分析op之间的依赖关系，试图并行执行互相没有依赖关系的op（如果op内部并行度不够的话）
- constant input folding：如果graph的某个输入tensor在第一次运行graph之后就不会再改变（例如inference推理的时候，模型的weight是不变的），那么只依赖不变输入的op可以只计算一次，把结果缓存起来
- inplace_transform
- batchwise_merge：和horizontal_merge类似，试图改变op的执行方式。如果有依赖关系的多个op都有batch 维度，那么会试图让这些op在batch维度进行并行。这样可以减少线程同步开销。例如batch_matmul+relu中，会使得生成的代码中每个线程先计算完一个matmul结果，然后计算本线程matmul结果的relu，而不是跨线程计算matmul的结果。

Tensor IR pass

接口和实现

Tensor IR pass的顶层接口是module_pass_t对象。这是个虚基类，定义如下：

```
class module_pass_t {
public:
    virtual const_ir_module_ptr operator()(const_ir_module_ptr f) = 0;
    virtual ~module_pass_t() = default;
};
```

所有的Tensor IR pass都需要继承这个基类，实现`operator()`方法。可以看到Tensor IR pass是针对IR module进行变换的。每个pass输入一个IR module，输出一个变换后的新的IR module。和

Graph pass不同，Tensor IR pass不允许对输入的IR对象（IR module，IR function等）进行直接修改，如需修改，需要返回一个新的IR对象。pass接口中的IR module是被C++ const关键字修饰的。这样不允许原地修改的好处在于，每个pass都可以认为是“纯函数”，即所有对IR的修改都反映在返回的IR上，不会修改输入IR，这样符合了“函数式编程”的理念，同样也方便后续多线程编译等高级功能的开发（Graph Compiler暂时还没有这样的功能）。

有许多Tensor IR pass其实并不关心IR module里面全局变量等内容，而是只关心IR函数里面局部的信息，只对每个IR函数内部进行独立的变换，这时可以使用function_pass_t接口来编写pass。定义如下：

```
class function_pass_t {
public:
    virtual func_c operator()(func_c f) = 0;
    virtual ~function_pass_t() = default;
};
```

Function pass接收一个const修饰的IR function指针，返回变换后的IR function指针。Function pass可以通过`module_function_pass_t`这个工具类转换为Module pass。

多个Module pass可以通过sequential_module_pass_t组合到一起，变成一个Module pass。这个类中有一个Module pass的std::vector，将会按照顺序执行vector中的pass。

现有的Tensor IR pass

Tensor IR的整个pass“流水线”实现在：

里面将所有必要的Tensor IR pass组合到一起，实现了从Graph lowering得到的Tensor IR，变换到可以被JIT engine接受的、优化后的Tensor IR。

目前Tensor IR pass的序列定义在函数get_default_precodegen_passes中。其中的pass有：

- interface_generalizer_t
- tensor_shrinker_t 如果访问Tensor的pattern合适，试图将大临时Tensor缩水成小Tensor，减少内存和缓存消耗
- index_flattener_t 将多维Tensor和对Tensor的多维访问改写成一维的线性数组的访问
- auto_caster_t 自动类型转换，对于允许的隐式转换，添加cast node，详见 https://zhuanlan.zhihu.com/p/521699748 的类型系统章节
- bf16_legalizer_t 为BF16类型的计算自动插入转换到F32的cast
- validator_t 检查Tensor IR是否正确
- trace_inserter_t
- constant_folder_t 常量折叠
- func_inliner_t 实现函数内联
- loop_merger_t
- parallel_workload_dispatcher_t
- index2var_t 将连续的对Tensor相同位置的读写改为对于本地变量的读写
- bf16_eliminator_t
- target_specific_lowering_cpu_t 针对CPU后端的特殊硬件指令的lowering
- kernel_lowering_cpu_t 将BRGEMM microkernel lower到函数调用
- dead_write_eliminator_t 消除多余的内存写操作

- buffer_scheduler_t 将Tensor使用的内存buffer合并，分析Tensor的生命周期，进行内存复用
- closurizer_cpu_t 将parallel for loop改写成闭包，详见 https://zhuanlan.zhihu.com/p/522157831 的控制流和并行化章节
- dyn_boundary_check_t 添加动态检查是否数组越界的代码
- module_globals_resolver_t
- local_tensor_lowering_cpu_t
- ssa_transform_t 将ir改写为SSA 形式
- value_numbering_t 公共表达式折叠
- loop_invariant_code_motion_t 循环不变量外提
- dessa_transform_t 将SSA形式的IR转换回可以直接生成代码的IR

为了实现Tensor IR pass，Graph Compiler也提供了遍历Tensor IR中所有节点的基础工具：ir_visitor，我们将在下一篇文章中介绍。