深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第8篇 执行调度和转换Graph IR到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。

本文示例代码已上传至：
https://github.com/Menooker/graphcompiler-tutorial/blob/master/Ch8-GraphLower/graphlower.cpp🔗github.com/Menooker/graphcompiler-tutorial/blob/master/Ch8-GraphLower/graphlower.cpp

在之前的几篇文章中，我们已经大致了解了GraphCompiler中Graph IR和Tensor IR。Graph IR用于表述计算图的Op级别的语义，而Tensor IR则更接近底层，用类似C语言的表达方式，表述了计算的具体代码。对于GraphCompiler来说，从用户这里接收到的是Graph IR，而它提供的是硬件可执行代码。只有Tensor IR可以被翻译为可执行代码。于是GraphCompiler需要一个桥梁来跨越Graph IR和Tensor IR的鸿沟，那就是Graph Lowering，即从Graph IR到Tensor IR的转换过程。由于Tensor IR是较为接近底层的IR，所以称这个过程为Lowering。在传统编译器领域，经常有lowering这个概念，表示高层IR转换为底层IR。相反地，也有编译器有底层IR转换为高层IR的过程，称之为lifting，这里不再展开。

在本文中，我们暂时先不直接讨论现有GraphCompiler中的lowering实现，而是假设自己是开发Graph Lowering的开发者，通过讨论以下几个问题，来一步步实现现在GraphCompiler项目中的Graph Lowering部分：

1）首先讨论Graph lowering到底需要做些什么
2）然后我们试着手动将一个简单的固定的Graph IR图转换为对应的Tensor IR
3）如何通过程序，将任意计算图自动转换为Graph IR
4）如何决定Op的执行顺序

Graph lowering到底需要做些什么

Graph lowering其实就是Graph IR中各个概念转换到Tensor IR对应概念的过程。基于Graph IR的内容创建对应的Tensor IR的对象，然后将Tensor IR对象正确连接，也就完成了转换的过程。Graph IR中有Graph、Op、Tensor、format等概念，Tensor IR中有IR module、function、tensor等概念。下面将它们对应起来。

转换Graph对象

对于GraphCompiler底层来说，一张计算图（Graph）对应的应该是一个Tensor IR module。我们知道，IR module是IR function和它依赖的全局变量的集合。这个IR module中应该有一个“主函数”，对应于整个计算图。主函数中应该通过参数传入计算图的输入和输出Tensor。

转换Op对象

Graph中每个Op节点有两个含义。第一，是定义了针对输入的一组“运算”，第二，它在图中表示了对这个“运算”的调用，包括记录了这次调用的输入和输出Tensor。所以每一个Op需要对应Tensor IR中的两个概念，一个是实现Op本身“运算”的IR function，还有就是在“主函数”中应该依次调用Graph中的每一个Op，将前后Op的结果串联起来，直到计算出最后的结果Tensor。

转换Tensor和Format

Graph IR中的Tensor对应于Tensor IR中的tensor。Tensor IR中`tensor_node`更接近于计算机底层指针的概念，表示一块内存上的多维有序空间。Graph Tensor中有format的概念，表示了tensor的实际内存排布。而在Tensor IR中，已经没有format的概念了，而是直接细化到访问Tensor的IR代码中。

手动转换固定的Graph IR

为了加深理解Graph IR和Tensor IR，我们来试着将一个固定的Graph转换为对应的Tensor IR。实际使用中，我们当然不会为不同的Graph编写不同的lowering
代码，而是会使用通用代码来转换。这里只是作为一个操作Graph IR和Tensor IR的例子。

假设我们想要转换的Graph为：

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = matmul_core(v0, v1)
  [v2: f32[1024, 1024]] = add(v3, v1)
}
```

生成这个Graph的C++代码为：

```
sc_graph_t graph;
in = graph.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t::MK(), datatypes::f32),
                       graph_tensor::make({1024, 1024}, sc_data_format_t::KN(), datatypes::f32)});
matmul = graph.make("matmul_core", in->get_outputs(), {}, {});
add = graph.make("add", {matmul->get_outputs()[0], in->get_outputs()[1]}, {}, {});
out = graph.make_output(add->get_outputs());
```

下面我们开始进行转换。首先创建一个IR function作为“主函数”，参数列表中有三个Tensor，分别为输出Tensor，和两个输入Tensor：

```
expr in0 = builder::make_tensor("in0", {1024, 1024}, datatypes::f32);
expr in1 = builder::make_tensor("in1", {1024, 1024}, datatypes::f32);
expr out0 = builder::make_tensor("out", {1024, 1024}, datatypes::f32);
stmts func_body = make_stmt<stmts_node_t>(std::vector<stmt>());
func_t func = builder::make_func("main_entry", {in0, in1, out0}, stmt(func_body), datatypes::void_t);
```

上面的代码中，我们首先创建了三个Tensor IR中的Tensor对象`in0`、`in1`和`out0`。然后为函数创建了空的函数体`func_body`，它是一个`stmts`节点。最后创建了函数`main_entry`。

然后基于刚刚创建的IR函数，我们创建一个IR module对象，将刚刚的`main_entry`函数作为IR模块的“main”函数：

```
auto ctx = get_default_context();
ir_module_ptr mod = ir_module_t::from_entry_func(ctx, func);
```

接下来，我们开始对IR module `mod`和函数体`func_body`填入函数的内容。首先我们先生成matmul的IR函数，并且加入到IR模块中：

```
ir_module_ptr matmul_mod = matmul->get_func(ctx);
func_t matmul_func = matmul_mod->get_entry_func();
mod->merge(*matmul_mod);
```

上面代码的第一行调用了`matmul`这个Op对象的`get_func`方法。在之前的文章中已经提到了，`get_func`方法将创建这个Op的具体实现的Tensor IR代码，存储在一个IR module中。上面代码的第二行，从返回的IR module中获得`matmul`这个函数的IR function指针。代码第三行将matmul IR module合并入我们主Module `mod`中。

然后就是对函数体添加内容了。首先我们在函数中定义一块临时Tensor用于存储matmul的输出：

```
expr mm_out = builder::make_tensor("matmul_out", {1024, 1024}, datatypes::f32);
func_body->seq_.emplace_back(builder::make_var_tensor_def_unattached(mm_out));
```

`func_body`指向的是一个`stmts_node_t`对象，这个stmt节点内可以存放多个stmt对象指针，起到了类似C语言花括号`{}`的作用。它的`seq_`成员变量是一个`std::vector<stmt>`，我们在这个vector中添加元素，即可对函数体添加内容。代码`builder::make_var_tensor_def_unattached`创建一个定义Tensor的`define_node_t`节点。在IR function中出现`define_node_t`节点，就表示定义和申请一块在函数内部的Tensor或者Var（变量），它的生命周期是在函数内部的，在它所在的`stmts`结束后，`define_node_t`节点定义的Tensor或者Var将不再有效。在上文代码中通过`define`节点包裹的`tensor`节点定义了在函数内部有效的一块buffer，大致相当于C语言中，在函数内部定义`float matmul_out[1024*1024]`。

然后就是生成一个调用matmul IR函数的Tensor IR节点，代码如下：

```
func_body->seq_.emplace_back(builder::make_evaluate_unattached(builder::make_call(matmul_func, {mm_out, in0, in1})));
```

上面的代码生成了一个`evaluate_node_t`节点，里面包裹了一个`call_node`。`evaluate_node_t`可以将一个expr包裹称为一个stmt，相当于C语言中的分号“;”。`call_node`即函数调用节点。我们在`call_node`中传入matmul函数IR的指针，和参数——前面定义的三个Tensor。在Tensor IR中，如果一个函数需要返回一个Tensor，那么和C/C++一样，我们可以把Tensor（对应C/C++的指针）作为“出参”传给函数，函数可以通过指针直接将结果填入Tensor中。所以上面代码中，matmul的结果`mm_out`也作为参数，填入参数中。

类似地，我们可以生成`add` Op对应的IR函数和调用函数节点，将`mm_out`作为输入参数，将整个计算图的输出`out0`作为出参：

```
ir_module_ptr add_mod = add->get_func(ctx);
func_t add_func = add_mod->get_entry_func();
mod->merge(*add_mod);
func_body->seq_.emplace_back(builder::make_evaluate_unattached(builder::make_call(add_func, {out0, mm_out, in1})));
```

最后我们打印生成的Tensor IR：

```
std::cout<< mod;
```

结果为：

```
func main_entry(in0: [f32 * 1024 * 1024], in1: [f32 * 1024 * 1024], out: [f32 * 1024 * 1024]): void {
  tensor matmul_out: [f32 * 1024 * 1024]
  evaluate{matmul_core__1(matmul_out, in0, in1)}
  evaluate{add__2(out, matmul_out, in1)}
}
func matmul_core__1(__outs_0: [f32 * 1024UL * 1024UL], __ins_0: [f32 * 1024UL * 1024UL], __ins_1: [f32 * 1024UL * 1024UL]): bool {
  for fused_0m_o__n_o_0 in (0UL, 512UL, 1UL) parallel {
    evaluate{brgemm(&__ins_0[((fused_0m_o__n_o_0 / 32UL) * 64), 0], &__ins_1[0, ((fused_0m_o__n_o_0 % 32UL) * 32)], &__outs_0[((fused_0m_o__n_o_0 / 32
  }
  return true
}
func add__2(__outs_0: [f32 * 1024UL * 1024UL], __ins_0: [f32 * 1024UL * 1024UL], __ins_1: [f32 * 1024UL * 1024UL]): bool {
  for __itr_0 in (0, 1024UL, 1) parallel {
    for _fuseiter0 in (0, 1, 1) {
      for _fuseiter1 in (0, 1024, 8) {
        {
          &__outs_0[__itr_0, 0][_fuseiter0, _fuseiter1 @ 8] = (&__ins_0[__itr_0, 0][_fuseiter0, _fuseiter1 @ 8] + &__ins_1[__itr_0, 0][_fuseiter0, _fu
        }
        {
        }
      }
    }
  }
  return true
}
```

至此，我们完成了将一个固定的Graph IR转换为Tensor IR的过程。

GraphCompiler中Lowering的实现

在上一节我们讨论了固定Graph IR的Lowering过程。GraphCompiler作为一个能接受各种不同计算图的编译器，需要实现一种能将任意合法的计算图lower到Tensor IR的实现（而不是像上一节讨论的只能lower固定Graph的代码）。这就是GraphCompiler的`lower_graph`函数，它定义在

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

头文件位于：

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

由于现有的Lowering实现了许多复杂的额外功能，为了让读者更加清晰地了解它的基本实现，本文将会从零开始，一步步重新实现一个原理相同的基础版的Graph lowering（我们称这个这个简单版的lowering为`simple_lowering`好了:)。

示例代码`simple_lowering`的完整实现已经上传到本文一开始提到的Github链接中。

我们首先定义`simple_lowering`的接口为：
```
ir_module_ptr simple_lowering(context_ptr ctx, sc_graph_t &graph) {
    ...
}
```

在这个函数中，我们需要首先定义从Graph Tensor到Tensor IR Tensor的映射，用`std::unordered_map`表示：
```
std::unordered_map<graph_tensor_ptr, expr> ltsr_rtsr;
```

然后就是定义IR模块、IR函数、以及IR函数的空实现：
```
std::vector<expr> params;
stmts func_body = make_stmt<stmts_node_t>(std::vector<stmt>());
auto func = builder::make_func(
    "main_entry",
    params, func_body, datatypes::void_t);
auto ret_mod = ir_module_t::from_entry_func(ctx, func);
```

定义一个C++工具函数用于在`ltsr_rtsr`这个map中查找GraphTensor对应的Tensor IR Tensor，如果没有找到底层Tensor，则会创建Tensor，并且根据这个Tensor是否是通过IR函数的参数传入的来决定是否需要在函数体中添加这个Tensor节点的define节点（即是否需要在函数体中定义这个Tensor，如果是从参数传入的Tensor，那么就不需要再定义它了）：
```
auto get_or_create_tensor = [&](const graph_tensor_ptr &t, bool is_arg) -> expr {
    auto itr = ltsr_rtsr.find(t);
    if (itr != ltsr_rtsr.end())
    {
        return itr->second;
    }
    if (!is_arg)
    {
        for (auto &use : t->uses_)
        {
            // finds if any of the use of the tensor is marked output
            if (use.second->isa<output_op>())
            {
                is_arg = true;
                break;
            }
        }
    }

    std::vector<expr> dims = dims_to_expr(t->details_.get_blocking_dims());
    std::string tensor_name = tensor_name = std::string("buffer_") + std::to_string(tensor_counter);
    expr tsr = builder::make_tensor(
        tensor_name, dims, t->details_.dtype_);
    tensor_counter++;
    ltsr_rtsr.insert(std::make_pair(t, tsr));

    if (!is_arg)
    {
        func_body->seq_.emplace_back(
            builder::make_var_tensor_def_unattached(tsr));
    }
    return tsr;
};
```

由于Op的执行顺序必须满足拓扑排序，所以我们需要按拓扑排序来访问Graph中的各个Op。这可以通过创建一个`op_visitor_t`来实现。在这里我们选用了`op_visitor_t::dfs_topology_sort()`这种排序。需要注意的是，在上一篇文章中我们已经讨论了，对于复杂的图来说，符合拓扑排序的访问顺序可能不止一种。而在Graph中，不同的合法的Op执行顺序虽然程序运行的结构相同，但是可能会有较大的性能差异。本文下一小节将简单地讨论GraphCompiler中是如何选择一个较好的拓扑顺序。在这一节中，我们则是简单地使用`dfs_topology_sort`。代码如下，它将会按序访问Graph中的每个Op：
```
op_visitor_t vis = op_visitor_t::dfs_topology_sort();
vis.visit_graph(graph, [&](const sc_op_ptr &node) {
    ...
});
```

在visitor中传入的lambda函数即是生成Op代码和调用Op函数的部分，上面的代码暂时略过了，我们下面一步步展开上面代码中lambda里面的的"..."部分：
```
std::vector<expr> ins;
std::vector<expr> outs;
```

首先定义两个数组，记录当前Op(即lambda函数的参数`node`)的在“主函数”中的输入和输出Tensor。
```
// special kinds of Ops that we need to take care of
enum op_kinds
{
    other = 0,
    input,
    output,
} kind = other;
if (node->isa<input_op>())
```

```
{
    kind = input;
}
else if (node->isa<output_op>())
{
    kind = output;
}
```

由于input和output Op是GraphIR中的特殊节点，仅仅表示输入输出，而不是表示计算，需要单独识别它们，如果当前访问的节点node是input或者output Op，我们先会记录下它们的类型。（这边代码做了简化，事实上不止这两种Op需要特殊处理。）

```
for (auto &ltensor : node->get_inputs())
{
    ins.emplace_back(get_or_create_tensor(ltensor, false));
}
for (auto &ltensor : node->get_outputs())
{
    outs.emplace_back(get_or_create_tensor(
        ltensor, kind == input));
}
```

上面的代码获取了当前节点node的输入输出Tensor（Graph Tensor），然后通过get_or_create_tensor转换为Tensor IR上的Tensor，然后记录在ins和outs当中，等会就可以将ins和outs作为参数列表，来调用这个Op对应的IR函数。

然后，根据Op类型的不同，我们有不同的Lower策略，代码如下：

```
        switch (kind)
        {
        case input:
        {
            for (auto &v : outs)
            {
                params.emplace_back(v);
            }
            break;
        }
        case output:
        {
            for (auto &v : ins)
            {
                params.emplace_back(v);
            }
            break;
        }
        default:
        {
            std::vector<expr> exprargs;
            exprargs.insert(exprargs.end(), outs.begin(), outs.end());
            exprargs.insert(exprargs.end(), ins.begin(), ins.end());
            auto mod = node->get_func(ctx);
            ret_mod->merge(*mod);
            auto callee = mod->get_entry_func();
            stmts_node_t *target_body = func_body.get();
            target_body->seq_.emplace_back(
                builder::make_evaluate_unattached(
                    builder::make_call(callee, exprargs)));
        }
        }
```

如果是input/output，则会将对应的输入输出Tensor按顺序push到params数组中，这个数组存放了“主函数”的参数列表。

如果是其他类型的Op，则会将输入输出Tensor，ins和outs，组合成数组exprargs中，然后通过node的get_func方法得到Op的Tensor IR function的实现，然后通过call_node和evaluate_node_t调用这个IR function，实参列表就是前面创建的exprargs。这里的代码我们已经在上一节手动lower IR的时候看到了，所以不再赘述。

这样，每个Op的Tensor IR function实现和“主函数”对它的调用都已经完成了。我们完成上文visit_graph的lambda函数，回到simple_lowering:

```
func->params_ = std::move(params);
func->decl_->params_ = func->params_;
return ret_mod;
```

我们将收集到的“主函数”形参列表params放入“主函数”对象func中。然后返回生成的IR module。至此simple_lowering函数结束，它完成了最基础的Graph Lowering的工作。GraphCompiler中当前的lower_graph函数也是从这个版本的simple_lowering添加诸多功能之后而来的。

决定Op的执行顺序

上一节讲到通过op_visitor_t遍历Graph的时候，我们已经知道，虽然是拓扑排序访问Graph，我们依然可以有多种不同的合法遍历顺序。而Op遍历顺序决定了Op的执行顺序。不同的执行顺序会产生不同的执行效率。这里我们主要考虑的是内存总用量和缓存的影响。例如以下计算图：

```
graph(v0, v1, v2) -> v6 {
    v3 = relu(v0)
    v4 = add(v3, v3)

    v5 = matmul(v1, v2)
    v6 = relu(v5)

    v7 = add(v4, v6)
}
```

这个图其实由三部分组成，我们已经用空行区分开了。看到图输出的v7 tensor是通过v4和v6加法而来。而v4和v6又是图的两个分支的结果。v4是add和relu这一个分支的结果。v6是matmul和relu分支计算得到的。那么我们在调度这个图的执行顺序的时候，我们似乎可以选择：

1) 交错运行v4和v6分支的Op，例如依次运行`v3 = relu(v0), v5 = matmul(v1, v2), v4 = add(v3, v3), v6 = relu(v5)`
2) 运行完一个分支之后再运行另一个分支，例如`v3 = relu(v0), v4 = add(v3, v3), v5 = matmul(v1, v2), v6 = relu(v5)`

一般来说，上面两种调度模式中的第二种性能较好，有两个原因：第一，前一个Op计算的结果很可能尚在缓存中，如果后一个Op的输入是前一个Op的输出，那么可以加快后一个Op的访存速度。交错运行模式下，我们选择的Op和之前的Op没有直接依赖关系，导致内存的访问都不能命中缓存。第二，在一个Op执行完成后，它所依赖的输入Tensor如果没有其他Op需要，那么这块Tensor可以被释放。如果使用交错运行Op，那么就需要同时保留多个分支的Tensor内存，这样会增大最大内存需求。

基于以上观察，我们对Op的执行顺序调度有以下几个优化目标：

1）尽量利用“热”的输入Tensor
2）同一时刻“存活”的Tensor（还会被其他没有执行的Op访问的Tensor）总量尽量小

上一篇文章讲到了`op_visitor_t`可以通过设置`selector`函数来配置拓扑遍历顺序。`selector`函数的作用是从已经可以访问的Op列表（所有依赖的Op已经执行的Op）中选择一个Op进行访问。我们通过编写一个特殊的`selector`函数来利用`op_visitor_t`按照优化的执行顺序来依次访问Graph中的所有Op。具体实现在了lowering.cpp中的`lowering_visitor_state_t`中。

在选择下一个Op进行访问（lower）的时候，它会遍历`op_visitor_t`中记录的所有可以访问的Op列表，为每一个其中的Op打分，然后选择分数最高的来访问（执行）。

打分公式如下：

$$\sum_{t \in inputTensors} \frac{NormalizedSize(t) * heatModifier(t)}{refCountModifier(t)} - \sum_{t \in outputTensors} (NormalizedSize(t) + distanceModifier(t))$$\sum_{t \in inputTensors} {\frac{NormalizedSize(t)*heatModifier(t)}{refCountModifier(t)}} - \sum_{t \in outputTensors} {(NormalizedSize(t)+ distanceModifier(t))}

上面公式的基本思想是，一个Op的“好坏”大致是它“解放”的input tensor大小（如果这个Op是最后一个使用某Tensor的Op，那么此后这个Tensor可以马上释放），减去这个Op“生成”的Tensor大小，即它的输出Tensor的总大小。如果“解放”的Tensor大小远远大于“生成”Tensor的大小，那么认为它是好的。所以上面的公式本质上是对这个Op的所有输入的加权求和减去输出的加权求和。

`NormalizedSize`是0-1.0的值表示归一化的Tensor大小，这个是Tensor大小除以Graph中最大Tensor的大小。`heatModifier`表示了这个input Tensor是多久之前被写入的。如果是上个刚刚访问的Op的输出，那么`heatModifier=2.5`，如果是两个Op前的输出，`heatModifier=1.5`，其他情况则是1.0。

`refCountModifier`表示这个Op是不是最后一个访问此输入Tensor的Op，如果是，则`refCountModifier=1`。否则`refCountModifier=useCount`，`useCount`是依赖这个这个Tensor的Op数量。设置这个系数的目的是如果Op是一个Tensor最后一个使用者，那么执行Op后可以释放这个Tensor，所以我们可以给这个Op一些分数上的“奖励”。如果Op不是这个Tensor最后的使用者，那么“释放”这个Tensor的“奖励”就会打折扣。

对于Op的输出Tensor，还需要加上`distanceModifier`，它表示这个依赖于这个Output Tensor的其他Op距离已经访问过的Op的最远距离。如果一个Output Tensor被某些其他Op依赖，而正好这些Op不太可能在近期被执行（例如有其他依赖没有完成），那么`distanceModifier`将会变大。表示这个Output Tensor将会有较长的生命周期，也就会加大内存使用量，不利于内存复用。`distanceModifier`具体公式为：

$$distanceModifier(t) = (\max_{op \in t.users} (distanceToVisitedOps(op)) - 1) * 2$$distanceModifier(t) = (\max_{op \in t.users}(distanceToVisitedOps(op)) - 1) * 2

公式中的“减一”是由于Op的距离永远大于等于1，所以如果不减去一，多个输出Tensor的Op的`distanceModifier`将会偏大。公式中“乘2”将`distanceModifier`的重要性调高。

GraphCompiler的Op执行顺序调度算法就介绍到这里。它本质上是一个启发式算法，通过为每个可以调度的Op评分来选择一个合适的Op进行访问。