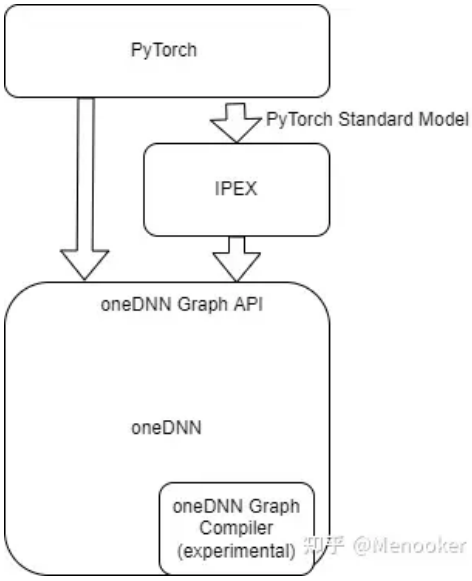
深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第2篇 整体架构

关于作者以及免责声明见序章开头。

题图来自网络，侵删。

我们在序章中引出了oneDNN Graph Compiler，同时也说明了oneDNN，oneDNN Graph Compiler之间的关系。oneDNN Graph API将成为整个oneDNN处理DNN模型的最前端。它通过桥接层，可以将PyTorch，TensorFlow等框架的模型转换为oneDNN Graph API的标准图格式。oneDNN在这里定义了Graph API。在最新的Intel® Extension for PyTorch（Intel为PyTorch提供的CPU、Intel GPU加速插件）中，已经集成了oneDNN Graph API(在此)。官方的PyTorch也集成了oneDNN的功能（在这里），可以直接调用oneDNN Graph API。oneDNN Graph API则会将图经过一些图优化之后，将图进行切分，然后对于不同的图partition，选择oneDNN primitive或者oneDNN Graph Compiler进行编译，最后调度运行编译后的代码。



这个系列文章主要讨论的是oneDNN Graph Compiler，所以展示的代码与例子可能会绕过oneDNN Graph API，直接调用oneDNN Graph Compiler的内部接口。

两层IR：Graph IR与Tensor IR

这里IR即Intermediate Representation，中间表达形式。IR是编译器中常用的概念，用于抽象出用户程序的计算，以及各个中间结果之间的关系。用户想要编译的程序在编译器内部就是通过IR来表示的。我们可以对IR进行变换来优化代码。最终编译器将IR翻译为机器汇编码完成最后的编译。

GraphCompiler选择了两层IR表示。我们内部并没有对这两层IR有正式的命名。上层IR表示的是计算图，本文称之为Graph IR。底层IR表示的是Op内部的具体计算，本文称之为Tensor IR。

Graph IR

Graph IR是用来描述计算图的Op之间的连接关系的。Graph IR是由Op（算子）和Logical Tensor组成。一个Op即表示对于输入Tensor进行的计算，例如add，matmul等等。它可以有多个Tensor作为输入，也可以输出多个Tensor。每个Tensor只会有一个Owner，表示生成这个Tensor的Op。Op之间则是通过Tensor相互连接-一个Op的输出Tensor可以作为另一个Op的输入Tensor。Graph IR组成的计算图应该是一张有向无环图（DAG），这个图的入口节点必须是`input_op`或者`constant_op`。这两种特殊的Op不表示任何计算，而是表示整个图的输入参数和常量。Graph IR的输出节点（即DAG的Sink节点）应为`output_op`。它的所有输入tensor都会变成计算图的输出。

在测试时，以及我们内部使用GraphCompiler的时候，我们有时也会跳过oneDNN Graph，直接通过Graph IR来搭建一张计算图。

Graph IR在编译器内存中是通过C++的对象来组织的。但是GraphCompiler也提供了一些手段来打印Graph IR。例如计算`out=relu(matmul(a,b)) + a`的Graph IR是长这样的：

```
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v3: f32[1024, 1024]] = matmul_core(v0, v1)
  [v4: f32[1024, 1024]] = relu(v3)
  [v2: f32[1024, 1024]] = add(v4, v0)
}
```

形如`v0: f32[1024, 1024]`的代码表示的是一个Logical Tensor，名字是`v0`，数据类型是`float32`,维度是`1024*1024`。第一行`graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {`表示的是整个计算图的输入tensor是v0和v1，输出v2。

通过打印Graph IR，我们可以直观地看到Graph的内容和连接关系。下一篇文章马上会介绍如何生成和打印Graph IR。

Graph IR描述了图中Op之间的连接关系，可以让编译器方便地进行Op级别的优化，而暂时不用考虑Op内部的实现细节。那么编译器是如何实现生成Op内部实现的呢？这就需要Tensor IR了。

Tensor IR

Tensor IR是用来描述具体运算的IR。可以将Graph Compiler的Tensor IR类比为IR中的C语言。Tensor IR可以描述在tensor内部的每个元素是如何计算的。它还可以有控制流节点，例如`for`, `if`等。每条Tensor IR基本上可以与C语言的语句一一对应。下面是一个Tensor IR function的例子，实现了一个Add op：

```
func add__1(__outs_0: [f32 * 1024UL * 1024UL], __ins_0: [f32 * 1024UL * 1024UL], __ins_1: [f32 * 1024UL * 1024UL]): bool {
  for _fuseiter0 in (0, 1024UL, 1) parallel {
    for _fuseiter1 in (0, 1024, 1) {
        __outs_0[_fuseiter0, _fuseiter1] = (__ins_0[_fuseiter0, _fuseiter1] + __ins_1[_fuseiter0, _fuseiter1])
      }
    }
  }
  return true
}
```

上面的IR大致相当于以下的C函数

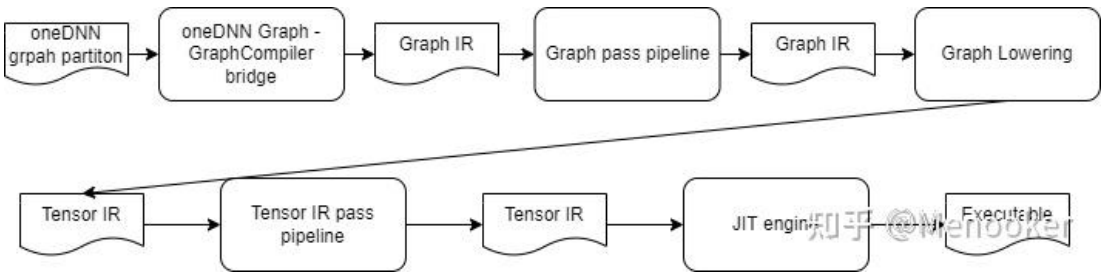
```
bool add__1(float __outs_0[1024UL * 1024UL], float __ins_0[1024UL * 1024UL], float __ins_1[1024UL * 1024UL]) {
  #pragma omp parallel for
  for (int _fuseiter0 = 0; _fuseiter0 < 1024UL; _fuseiter0 += 1) {
    for (int _fuseiter1 = 0; _fuseiter1 < 1024UL; _fuseiter1 += 1) {
        __outs_0[_fuseiter0*1024 + _fuseiter1] = (__ins_0[_fuseiter0*1024 + _fuseiter1] + __ins_1[_fuseiter0*1024 + _fuseiter1]);
      }
    }
  }
  return true
}
```

编写Tensor IR基本和写C++程序类似，只是生成的代码是IR，我们可以继续通过变换在编译器内部进行后续优化。通过（编译器）代码来操控代码（IR），这可以认为是传说中的元编程了（Meta programming）。

GraphCompiler整体架构

GraphCompiler是oneDNN Graph的一部分，GraphCompiler与oneDNN Graph桥接部分代码在src/backend/graph_compiler。GraphCompiler本身的代码位于src/backend/graph_compiler/core

接下来我们通过描述GraphCompiler的工作流程，借以一窥GraphCompiler的主要模块。



首先GraphCompiler接受到的是基于oneDNN Graph API的计算子图（partition），然后GraphCompiler与oneDNN Graph桥接代码会把这个partition翻译为GraphCompiler内部的图表示形式，即上一节说的Graph IR。

在得到Graph IR之后，GraphCompiler会有一系列的pass用来处理和优化Graph IR。这里的Pass也是编译器中常见的概念，即对IR进行分析和变换的子程序。通常一个Pass会对IR产生一种优化，而我们把一系列的pass组合起来，依次调用，就能对IR进行

各种不同的优化了。

接下来的一个模块是Graph Lowering，这是Graph IR上的最后一个pass，将Graph IR转换为更为底层的Tensor IR。对于每一个Op，我们都为其实现了IR的模板，确定具体输入输出Tensor形状和其他配置之后，我们可以为Op生成对应的底层Tensor IR。

然后在Tensor IR层面，我们也实现了一套优化pass pipeline，经过pipeline之后的Tensor IR已经被优化完毕，并且适合被用来生成机器码了。

在整个编译流程的最后，是即时编译器模块（JIT Engine，Just-in-time）。即时编译，即在程序运行的时候动态生成机器汇编码来运行用户需要的程序。与JIT相对的概念是AOT，即Ahead-of-time，即在程序开始运行之前就编译程序到汇编指令。常用的C、C++编译器例如clang、gcc都属于AOT编译器。而Java最常见的JVM实现HotSpot属于JIT编译器。选用JIT编译的好处在于，在深度学习的任务中，我们通常是在DNN框架运行，用户搭建完计算图之后才能获知具体计算图的内容和tensor shape。在编译Graph Compiler本身的时候，我们无从得知用户会用我们的项目运行什么模型、什么shape。所以Graph Compiler必须在用户输入Graph IR之后通过JIT的方式生成可执行代码。

GraphCompiler目前开源了两种JIT Engine：`llvm_jit`和`c_jit`。在`llvm_jit`中，我们将Tensor IR翻译为LLVM IR，然后调用LLVM的MCJIT完成即时编译。而`c_jit`中，我们把Tensor IR翻译为C++源文件，然后调用g++编译为动态链接库，然后通过`dlopen`加载生成的代码。`c_jit`由于需要写入文件，以及调用外部编译器，所以编译速度和安全性都不如LLVM。这个JIT模式主要用于调试生成的代码。默认情况下使用的是`llvm_jit`。在运行时，可以通过环境变量`DNNL_GRAPH_SC_CPU_JIT=llvm`或`c`来设置JIT时使用的模式。

在GraphCompiler生成的代码（我们通常称之为Kernel）运行的时候，需要提供一组支持函数，用于创建线程、分配文件、进行调试等等。这些函数是GraphCompiler提供的Runtime的一部分。

以上就是对GraphCompiler的整体架构的介绍了。下一篇文章将会介绍如何调用和编译GraphCompiler，并且通过它生成和运行一个简单的kernel。