深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第3篇 小试牛刀

关于作者以及免责声明见序章开头。题图源自网络，侵删。

这篇文章将带领大家开始实地探索GraphCompiler——先从编译和使用项目开始！

本文基于oneDNN (dev-graph) 分支的commit e2ffeec0ee44595291cc1f6bc58e2b1a62806642 。本文示例代码已经上传至

https://github.com/Menooker/graphcompiler-tutorial

我们这一章将会创建一个可执行程序，将GraphCompiler作为一个外部库来调用，通过它编译、运行一个简单的计算图，然后验证其结果。

准备编译环境

在我本地编写本文示例代码的环境基于Ubuntu 16.04。更新的Ubuntu系统版本应该是没有问题的，在CentOS7上我们也测试过。需要提前安装的程序有（Ubuntu 包名）：llvm-8-dev或llvm-13-dev，cmake，g++，binutils，build-essential，git。如何安装LLVM可以在这里找到：

g++最低版本为4.8.5（配合LLVM-8），如果使用llvm-13，请使用支持C++17的g++版本。g++ 7和8是我们常用的版本。请注意，在高版本g++（10.0+）中，我们依赖的一个第三方库（Xbyak）无法通过编译，请使用较低版本的g++或者转用clang。

准备编译目录、下载源代码

如果想要偷懒，可以直接下载本文的示例代码。在这一小节的最后将展示直接下载示例代码的方法。GraphCompiler支持Windows和Linux平台。推荐使用Linux环境进行本文的尝试。

我们先为我们的测试代码建立一个目录。取名叫graphcompiler-tutorial。我们的第一个graphcompiler项目我们给它取名叫做HelloCompiler吧。所以在graphcompiler-tutorial目录中，再建立一个目录叫做Ch3-HelloCompiler (我们现在是第三篇文章:))。

```
mkdir graphcompiler-tutorial && cd graphcompiler-tutorial
mkdir Ch3-HelloCompiler
```

oneDNN Graph包含了GraphCompiler的代码，已经发布在Github上。在graphcompiler-tutorial目录中，我们通过git下载oneDNN（Graph）的代码，并且切换到我们需要的commit：

```
git clone https://github.com/oneapi-src/oneDNN
cd oneDNN
git checkout e2ffeec0ee44595291cc1f6bc58e2b1a62806642
git submodule init
git submodule update
# 如果想要使用最新的oneDNN Graph，可以 git checkout dev-graph
cd .. #回到 graphcompiler-tutorial 目录
```

编写我们的第一个GraphCompiler测试程序

在Ch3-HelloCompiler目录中的`hello.cpp`，创建我们第一个调用GraphCompiler的代码：

```
#include <iostream>
#include <compiler/ir/graph/graph.hpp>
#include <compiler/ir/graph/driver.hpp>
#include <compiler/ir/graph/pass/pass.hpp>
#include <compiler/ir/graph/lowering.hpp>
#include <compiler/jit/jit.hpp>

using namespace sc;

int main()
{
    auto ctx = get_default_context();
    sc_graph_t g;
    auto in = g.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32),
                            graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32)});
    auto relu = g.make("relu", {in->get_outputs()[0]}, {}, {});
    auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});
    auto out = g.make_output(add->get_outputs());
    std::cout << "Original Graph:\n";
    print_graph(g, std::cout, true);
    graph_driver(g, ctx);

    std::cout << "Graph After passes:\n";
    print_graph(g, std::cout, true);

    auto ir_modu = lower_graph(ctx, g, {in, out});
    std::cout << "Tensor IR:\n"
              << ir_modu << '\n';

    auto jit_func = jit_engine_t::make(ctx)->get_entry_func(ir_modu);
    std::vector<float> a(1024 * 1024), b(1024 * 1024), outbuffer(1024 * 1024);
```

```
    jit_func->call_default<void>(a.data(), b.data(), outbuffer.data());
}
```

如果你使用的是VSCode，在编辑窗口可能会发现提示找不到include文件，可以在graphcompiler-tutorial目录中建立一个文件夹.vscode，在这个文件夹中建立文件settings.json。内容为：

```
{
    "C_Cpp.default.includePath": [
        "oneDNN/src/backend/graph_compiler/core/src"
    ],
    "C_Cpp.default.cppStandard": "c++14"
}
```

这样我们的hello.cpp应该在VSCode编辑器中没有错误了。

来看我们的示例代码。一开始先include了一堆GraphCompiler的头文件。位置是在oneDNN/src/backend/graph_compiler/core/src。GraphCompiler目前是oneDNN Graph的一个内部模块，使用的namespace为sc：

```
#include <iostream>
#include <compiler/ir/graph/graph.hpp>
#include <compiler/ir/graph/driver.hpp>
#include <compiler/ir/graph/pass/pass.hpp>
#include <compiler/ir/graph/lowering.hpp>
#include <compiler/jit/jit.hpp>

using namespace sc;
```

GraphCompiler通过context来存储编译器本身的配置，例如目标CPU支持的指令集等等。同一段代码用不同的context可能会产生不同的编译结果。我们使用get_default_context()来获取当前机器的context。

```
auto ctx = get_default_context();
```

然后就是创建Graph IR的过程。首先建立一个空的graph：

```
sc_graph_t g;
```

然后创建图的输入，是两个1024*1024维的fp32 Tensor：

```
auto in = g.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32),
                        graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32)});
```

这里得到的in是一个input_op。它表示一个图的输入节点。input_op本身没有input tensor，但是它会输出一个或数个output tensor。这里in有两个output tensor，都是1024*1024维。in->get_outputs()[0]表示获取第一个tensor。g.make表示在graph 中创建一个调用op的节点，auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {}); 表示了创建一个add_op，输入是relu节点的0号输出和in节点的1号输出。下面建立了relu和add两个相连的op，并且将add的输出标记为graph的输出：

```
    auto relu = g.make("relu", {in->get_outputs()[0]}, {}, {});
    auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});
    auto out = g.make_output(add->get_outputs());
```

最后我们可以把图对象g打印出来，检视我们目前的成果：

```
print_graph(g, std::cout, true);
```

到这里代码输出的结果为：

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

这个图的连接关系应该十分清晰了。首先对input v0计算relu，得到v3，然后把v3和input v1做加法，得到最终输出v2。

然后继续阅读示例代码。我们调用针对graph IR的一系列优化pass，并且再次打印生成的Graph IR，查看优化结果：

```
    graph_driver(g, ctx);
    std::cout << "Graph After passes:\n";
    print_graph(g, std::cout, true);
```

从这里的输出可以看到，最终优化后的Graph IR已经发生了变化：

```
Graph After passes:
graph(v0: f32[1024, 1024], v1: f32[1024, 1024]) -> [v2: f32[1024, 1024]] {
  [v2: f32[1024, 1024]] = relu_add(v0, v1)
}
```

可以看到relu和add被合并为了一个op：relu_add。这表明Graph IR上的算子融合（Fusion）已经发生了。

接下来我们将Graph IR lower到底层的Tensor IR上：

```
    auto ir_modu = lower_graph(ctx, g, {in, out});
    std::cout << "Tensor IR:\n"
              << ir_modu << '\n';
```

这里ir_modu是一个GraphCompiler的ir_module的指针。它是Tensor IR上的IR function的集合。我们可以用std::cout直接打印ir_module。这里得到的结果如下：

```
Tensor IR:
func main_entry(buffer_0: [f32 * 1024UL * 1024UL], buffer_1: [f32 * 1024UL * 1024UL], buffer_2: [f32 * 1024UL * 1024UL]): void {
  evaluate{relu_add__2(buffer_2, buffer_0, buffer_1)}
}
func relu_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 {
    tensor _relu_buf_0: [f32 * 1024UL * 1024UL]
    for _fuseiter0 in (0, 1, 1) {
      for _fuseiter1 in (0, 1024, 8) {
        {
          &_relu_buf_0[__itr_0, 0][_fuseiter0, _fuseiter1 @ 8] = max(&__ins_0[__itr_0, 0][_fuseiter0, _fuseiter1 @ 8], (0.f))
```

```
        }
        {
          for _fuseiter3 in (0, 1, 1) {
            for _fuseiter4 in (0, 8, 8) {
              &__outs_0[__itr_0, _fuseiter1][_fuseiter3, _fuseiter4 @ 8] = (&_relu_buf_0[__itr_0, _fuseiter1][_fuseiter3, _fuseiter4 @ 8] + &__ins_1[_
            }
          }
        }
      }
    }
  }
  return true
}
```

这里展示的就是GraphCompiler内部自动生成的relu+add的fusion之后的IR代码。

最后我们通过JIT Engine将代码转换为可执行的程序，并且分配输入buffer，然后执行：

```
    auto jit_func = jit_engine_t::make(ctx)->get_entry_func(ir_modu);
    std::vector<float> a(1024 * 1024), b(1024 * 1024), outbuffer(1024 * 1024);
    jit_func->call_default<void>(a.data(), b.data(), outbuffer.data());
```

我们可以检查a、b、outbuffer中的值来确认结果。

这里展示的就是GraphCompiler编译一次计算图，并且执行代码的整个流程了。大家可以回顾上一篇文章说的GraphCompiler的整体架构，基本与这篇示例代码的流程一一对应。

编译和执行

最后是如何构建、编译我们的“HelloCompiler”。我们通过CMake去构建我们第一次使用GraphCompiler的测试代码。由于oneDNN Graph（包含了GraphCompiler）本身也是使用CMake进行构建的，我们可以很方便地将oneDNN Graph（包含GraphCompiler）作为我们HelloCompiler项目的子模块。所以在graphcompiler-tutorial目录创建文件`CMakeLists.txt`。内容为

```
project(GCHello)
set(DNNL_GRAPH_LIBRARY_TYPE STATIC)
set(DNNL_GRAPH_BUILD_COMPILER_BACKEND ON)
add_subdirectory(oneDNN)
include_directories(${PROJECT_SOURCE_DIR}/oneDNN/src/backend/graph_compiler/core/src)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -fopenmp")

add_executable(hello Ch3-HelloCompiler/hello.cpp)
target_link_libraries(hello graphcompiler dnnl)
```

代码`set(DNNL_GRAPH_LIBRARY_TYPE STATIC)`指定了使用oneDNN Graph的静态连接方式，由于GraphCompiler是oneDNN Graph的内部库，如果使用动态连接，会导致我们在外部无法调用一部分内部API。代码`set(DNNL_GRAPH_BUILD_COMPILER_BACKEND ON)`启用了GraphCompiler这个模块。代码`add_subdirectory(oneDNN)`将根目录下的`oneDNN`目录作为cmake的子模块加载进来。其他的CMake代码可以参考CMake的各种教程。

这个CMake代码会将`Ch3-HelloCompiler/hello.cpp`编译成可执行文件hello，同时将graphcompiler和dnnl链接进来。

最后我们编译和执行代码。在项目根目录graphcompiler-tutorial中建立目录build，然后在build目录中运行cmake与make

```
mkdir build && cd build
cmake .. -DDNNL_GRAPH_LIBRARY_TYPE=STATIC -DDNNL_GRAPH_BUILD_COMPILER_BACKEND=ON
make -j #或者例如make -j12，用12个进程去编译
./hello #执行示例代码
```

关于更换C++编译器 上面命令中，`cmake ..`将会选择系统默认的c++编译器。如果想要选择其他编译器，例如clang-13，可改为`CC=clang-13 CXX=clang++-13 cmake .. -DDNNL_GRAPH_LIBRARY_TYPE=STATIC -DDNNL_GRAPH_BUILD_COMPILER_BACKEND=ON`（请确保clang-13在PATH搜索路径中）。同理可以更换到某个指定的g++版本。

直接编译和执行本文的示例代码

本文的示例代码，包括CMake文件已经上传至Github。可以用如下方式下载、编译、执行

```
git clone --recursive https://github.com/Menooker/graphcompiler-tutorial
cd graphcompiler-tutorial
mkdir build && cd build
cmake .. -DDNNL_GRAPH_LIBRARY_TYPE=STATIC -DDNNL_GRAPH_BUILD_COMPILER_BACKEND=ON
make -j #或者例如make -j12，用12个进程去编译
./hello #执行示例代码
```

读者可以尝试编译其他的计算图。例如在计算中加入matmul：

```
    auto in = g.make_input({graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32),
                            graph_tensor::make({1024, 1024}, sc_data_format_t(), datatypes::f32)});
    auto mm1 = g.make("matmul_core", in->get_outputs(), {}, {});
    auto relu = g.make("relu", mm1->get_outputs(), {}, {});
    auto add = g.make("add", {relu->get_outputs()[0], in->get_outputs()[1]}, {}, {});
    auto out = g.make_output(add->get_outputs());
```

这样可以计算matmul-relu-add的计算图。很可惜，目前GraphCompiler对于matmul和conv只能运行在支持AVX512的CPU上，如果没有AVX512，应该在运行时会有报错。对于relu、add等简单计算，没有这样的限制。

在本篇中带大家初试GraphCompiler。下一篇我们将从Tensor IR开始，深入探究GraphCompiler内部。