深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第一章 - 序

英特尔最近开源了oneDNN Graph API和神经网络计算图编译模块 oneDNN Graph Compiler。oneDNN Graph Compiler作为oneDNN Graph API的一个后端实现，用于针对深度学习的计算图（Graph）生成高效的可执行代码。本系列将深入了解Graph Compiler的实现细节。后续的文章中我们有时会将oneDNN Graph Compiler简称为Graph Compiler 。

这是本系列的第一篇。在序章中首先先讨论一个问题，为什么需要深度学习编译器?

关于本文作者 本人供职于Intel上海，是Graph Compiler的开发人员，Github ID: Menooker

更多官方文档, 请访问 https://github.com/oneapi-src/oneDNN/tree/dev-graph/doc

文章目录和计划内容：

第一部分：项目基础介绍

在第一部分中，我们将探索oneDNN Graph Compiler的“基础设施”，这些部分构成了oneDNN Graph Compiler的骨架。

第2篇 整体架构

第3篇 小试牛刀（项目编译和运行）

第4篇 Tensor IR

第5篇 Tensor IR细节和构造Tensor IR

第6篇 Graph IR

第7篇 Graph IR变换和Op遍历

第8篇 Graph Tensor IR执行调度和转换到Tensor IR

第9篇 Tensor IR语义详解 (1)

第10篇 Tensor IR语义详解 (2)

第11篇 IR Pass

第12篇 Tensor IR Visitor

第13篇 编译运行Tensor IR

第二部分：专题介绍

第二部分中我们将分若干个专题，对oneDNN Graph Compiler的部分子模块进行剖析。

算子融合 (TBD)

Tensor内存复用和调度

……

听说要有题图，文章才有人看（滑稽）。题图源自网络，侵删。

传统编译器在深度学习领域的不足

深度学习作为计算机科学的一个分支，目前已经广泛应用于生活的各个领域。在相同的硬件条件下，如何加速深度学习应用（包括训练以及推理）是深度学习领域研究的重要方向。最早的Caffe，以及早期的Tensorflow、PyTorch等框架都是通过手工优化深度学习中使用到的算子（Operator）来追求更好的性能。

我们先来看如何通过传统编译器实现一个简单模型，并且手工优化之。假设有这样的简单深度学习“模型”-单层Linear层之后接上一个relu层。下面我们来试图通过传统编译器实现这个模型代码，

简易的C语言实现如下

```
void matmul(float data[8192*8192], float weight[8192*8192], float temp[8192*8192]) {
    for(int i=0;i<8192*8192;i++) {
        temp[i]=0;
    }
    for(int i=0;i<8192;i++) {
        for(int j=0;j<8192;j++) {
            for(int k=0;k<8192;k++) {
                temp[i*8192+j] += data[i*8192+k] * weight[k*8192+j];
            }
        }
    }
}

void relu(float input[8192*8192], float output[8192*8192]) {
    for(int i=0;i<8192*8192;i++) {
        output[i] = max(input[i], 0);
    }
}


void linear_and_relu(float data[8192*8192], float weight[8192*8192], float output[8192*8192]) {
    float* temp = (float*)malloc(8192*8192*sizeof(float));
    matmul(data, weight, temp);
    relu(temp, output);
    free(temp);
}
```

Linear本质上就是一个矩阵乘法（matmul），所以先把输入data（8192*8192维）和权重weight（8192*8192维）矩阵相乘，存储在临时buffer temp中，然后对这块buffer计算relu，结果存到output中。

首先看到矩阵乘法的实现（matmul）中，是一个简易的三层for循环。这样的循环其实对于缓存是不友好的。我们将矩阵data的一行`data[i*8192+k]`，与矩阵weight的每一列`weight[k*8192+j]`乘点积。但是矩阵data的一行已经占用了32KB的空间，假设CPU的L1缓存只有32KB（实际上没有这么小，我的远古7代Intel i7也有256KB，这边只是举个例子），那么在计算data与weight一行点积的时候，缓存已经无法放下这两行，这也使得data矩阵内容被不停地换出缓存，然后重新载入缓存。

为解决缓存使用的问题，可以改变matmul的计算顺序，使得data矩阵的一部分数据可以长久地驻扎在缓存中，避免重复从内存读取这部分数据，这种技术被称为Blocking（或tiling）。它将矩阵划分几块，然后在小块中进行矩阵乘法，最后将数据汇集到输出矩阵中。

```
void matmul(float data[8192*8192], float weight[8192*8192], float temp[8192*8192]) {
    for(int i=0;i<8192*0124;i++) {
        temp[i]=0;
    }
    constexpr int block_size=1024;
    for(int i_o=0;i<8192/block_size;i_o++) {
        for(int j_o=0;j<8192/block_size;j_o++) {
            for(int i_i=0;i<block_size;i_i++) {
                for(int j_i=0;j<block_size;j_i++) {
                    int i = i_o*block_size+i_i;
                    int j = j_o*block_size+j_i;
                    for(int k=0;k<8192;k++) {
                        temp[i*8192+j] += data[i*8192+k] * weight[k*8192+j];
                    }
                }
            }
        }
    }
}
```

以上代码中，将对于i的for循环拆成了`i_o`和`i_i`两个for循环，类似的将j的for循环拆成了`j_o`和`j_i`两个for循环。最内侧的for循环处理的是`block_size*block_size`大小的矩阵，保证其中的data矩阵部分可以一直在缓存中。

我们将一个for循环拆解成两个for循环的变换称为`split`。如果交换两个嵌套的for循环，称之为`reorder`。

接下来再看下一个优化。观察到计算完矩阵乘法之后，我们需要从内存中重新遍历中间结果矩阵，来计算relu。而其实relu是一种简单的运算，我们在计算完matmul每个元素的结果后，可以马上计算relu，然后直接写入最终结果的矩阵中。这样，就让relu操作“融合”进入matmul中。可以省去重复读取内存的开销。实验发现这样的优化对于性能有巨大的影响。我们称这样的操作为“算子融合”，即`fusion`.

```
void linear_and_relu(float data[8192*8192], float weight[8192*8192], float output[8192*8192]) {
    constexpr int block_size=1024;
    for(int i_o=0;i<8192/block_size;i_o++) {
        for(int j_o=0;j<8192/block_size;j_o++) {
            for(int i_i=0;i<block_size;i_i++) {
                for(int j_i=0;j<block_size;j_i++) {
                    int i = i_o*block_size+i_i;
                    int j = j_o*block_size+j_i;
                    float v = 0;
                    for(int k=0;k<8192;k++) {
                        v += data[i*8192+k] * weight[k*8192+j];
                    }
                    temp[i*8192+j] = max(v,0);
                }
            }
```

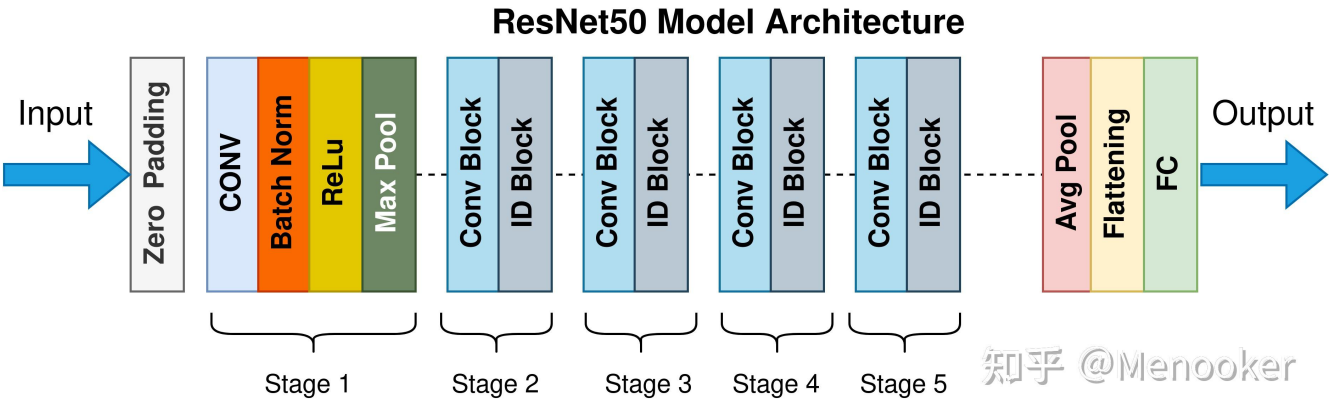
```
        }
    }
}
```

以上代码优化的例子，引出了传统编译器在深度学习领域的不足，这里只是列举一些：

1. 算子的开发者很难对代码进行调度和变换 - 比如编译器并不提供对循环代码进行Split、Fuse、Reorder等变换的操作。如果开发者要对一个matmul运算进行tiling，那么他需要手工更改源代码。
2. 对于不同的应用场景（例如不同大小的矩阵乘法），以及不同的硬件，代码调度的方式也会不同（例如tile的大小，上面例子中选用了`constexpr int block_size=1024`，是否交换两个循环的顺序等等）。传统编译器很难做到为同一个算子的不同场景生成足够高效的代码。当然开发者可以通过为不同的硬件、矩阵大小来编写不同的代码，但是显然会大大增加工作量。
3. 在将深度学习计算图转换成算子实现代码的时候，事实上对编译器来说已经丢失了算子在计算图内连接关系的信息，而这些信息其实可以帮助生成更有效的代码，例如可以交换两个算子的执行顺序，得到更好的缓存性能、可以将两个相邻算子进行算子融合，避免重复读写内存。而且对于不同的图可以自动生成优化后的代码，无需手动重复修改代码。

深度学习编译器

深度学习编译器也就应运而生。它们生而针对深度学习的应用负载而设计，针对性地去优化生成的代码。市面上常用的深度学习编译器有TVM，MLIR等（MLIR不仅可以用于深度学习编译器）。深度学习编译器通常的输入是一张计算图，图上的节点表示了算子（Op），而算子直接的边表示一个算子输出的张量（Tensor）传递给另一个算子（Op）作为输入。例如经典的ResNet.



https://commons.wikimedia.org/wiki/File:ResNet50.png

深度学习编译器通常会定义自己的计算图表示，例如TVM使用Relay，而MLIR则由开发者定义dialect。oneDNN Graph Compiler则是定义了一套Graph IR来表示。

深度学习编译器在接收到计算图后，会为这张计算图生成可执行的代码。用户最后可以通过调用生成的代码来进行深度学习模型的训练和推理。

在oneDNN Graph Compiler中，上一节提到的传统编译器遇到的问题，都得到了一定程度的解决。

- Graph Compiler中，算子的开发者可以方便地对代码进行调度和变换。算子实现的代码本身是内存中的对象，可以自由地变换。
- Graph Compiler对于不同形状的输入Tensor以及不同的硬件，可以通过基于人工规则的方式生成对应的代码配置，使得代码达到较优的性能。
- Graph Compiler实现了许多在计算图层面的优化。

从oneDNN到oneDNN Graph API到oneDNN Graph Compiler

熟悉PyTorch，TensorFlow内部实现的同学一定对cuDNN有所耳闻，这是NV官方实现的深度神经网络（DNN）的算子库，在NV GPU上TF和PT两个框架有许多的实现是基于这个库的。选择基于cuDNN是因为GPU硬件提供商（NV）是最了解硬件优化的人，官方实现的算子库通常可以达到接近理论性能的成绩。对于x86 CPU有没有官方实现的DNN算子库呢？答案就是Intel的oneDNN。话说最早这个库的名字是MKL-DNN。MKL其实是Intel的另一个高性能数学库。后来嘛，Intel的各种支持软件都改名叫oneXXXX了。oneDNN可以看作是各大深度学习框架在Intel CPU和GPU（对，我们还有GPU）上的底层实现。oneDNN提供了大量DNN会用到的算子，例如conv，matmul，还有各种激活函数relu，gelu等等。

oneDNN

oneDNN是目前Intel的主要DNN算子库。知乎上有一些文章来介绍oneDNN的基本结构。这里主要讲一下它不足的地方。oneDNN的基本逻辑是基于“原语” `primitive`的。以一个简单网络`conv, batch norm, relu, conv batch norm, bias` 为例，简要介绍一下oneDNN的工作流程。为了计算这个网络，用户需要创建两个primitive，即`conv+batch norm+relu`和`conv+batch norm+bias`，oneDNN会为这两个primitive生成JIT（即时编译）代码，然后用户将数据丢给创建后的primitive，然后将会得到结果。

在本例中，可以看到oneDNN的不足

1. 它的核心在于primitive，即以单个算子为核心的运算。oneDNN没法直接通过一个primitive处理复杂的神经网络（例如MLP中多个matmul和中间的运算）。单个primitive中能处理的算子组合通常是一个复杂操作（例如conv或者matmul），然后在输出接上简单运算（例如relu，bias等等）
2. 用户需要手动将计算图划分成不同primitive的组合。例如将`conv, batch norm, relu, conv batch norm, bias`划分为`conv+batch norm+relu`和`conv+batch norm+bias`。这需要用户非常熟悉oneDNN的API操作，与它的能力限制。
3. oneDNN要求复杂算子后连接的简单算子之间的连接关系不能太复杂，简单来说在一个primitive中，基本上只能处理单一线性的图。例如对于以下这个带简单分叉的计算图，oneDNN比较难实现通过单个primitive进行表达：

```
c=conv(a,b) out = (c*2)/log(c)
```

oneDNN Graph API

Intel提出了对oneDNN的拓展，被称为oneDNN Graph API。在2022/7它还是单独git分支的状态，合并情况可以查看：

2022年11月更新：oneDNN Graph API 已经正式合并到oneDNN的主分支中了。目前Graph Compiler的部分还未合并进主分支。

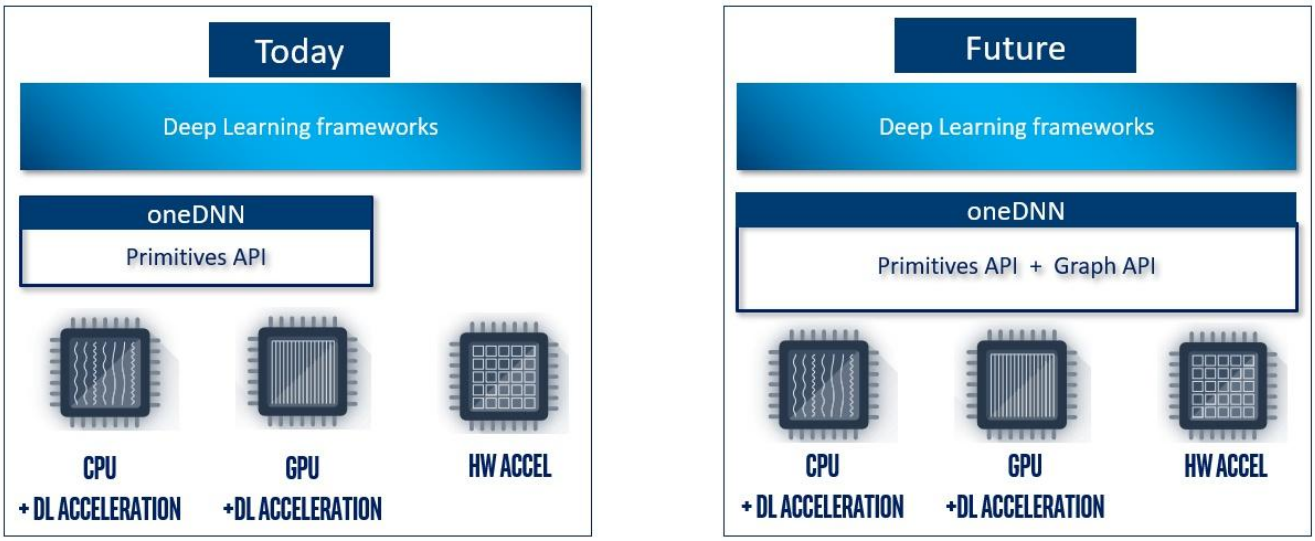
oneDNN Graph API解决了oneDNN无法整体处理计算图的问题。它还降低了将oneDNN集成到深度学习框架(Tensorflow, PyTorch等)的开发成本，提高了集成oneDNN的整体性能。具体可以参考：

oneDNN Graph API（我们简称oneDNN Graph）定义了大量DNN会用到的算子。在它的内部主要做的就是：接受用户输入的整个计算图，然后将图进行切分（partition），将被切分后的子图（partition）调度到不同的后端（backend）进行编译。在运行时，则是将数据根据计算图，导入到不同的可以执行的partition（compiled partition）中。

这里说的后端中，有一个就是基于oneDNN primitive的DNNL backend了。oneDNN Graph已知了oneDNN的算子的局限性，所以可以自动将计算图根据oneDNN的能力进行切分，而不需要人工介入。最终它也能做到自动调度运行生成的primitives。

本专栏的主角就是oneDNN Graph的另一个后端，oneDNN Graph Compiler。它可以在更大的粒度下生成计算图的代码。由于Graph Compiler的能力理论上可以生成任意复杂的图连接关系，所有它可以相比oneDNN处理更大的子图。看到更大的子图，同时也意味着更多的优化机会。

一张图说明oneDNN、oneDNN Graph API的关系（图片来源oneDNN Graph 文档）：



- Graph API allows HW backend to maximize performance
- Same integration for multiple AI HW: CPU, GPU, and accelerators

新版的oneDNN会提供两种API，primitive API和Graph API。而oneDNN Graph API的内部实现又可以分为基于primitive的和基于oneDNN Graph Compiler的后端。

我们从第二篇文章开始，逐步揭开oneDNN Graph Compiler的神秘面纱——先从项目整体架构本身开始。