7. 深度学习编译器 - 算子的高效实现

图表示和图优化是在较高层级对神经网络的一种描述和优化，优化后图中的节点代表一种特定的计算逻辑/数据搬运逻辑。比如 MATMUL 表示广义的矩阵乘法运算，ADD 表示张量加法等。为了在特定硬件上运行，必须针对该硬件实现这些计算逻辑/数据搬运逻辑。这一步通常有两种实现方式：调用提前写好的高性能算子、自动生成高性能的算子。绝大部分深度学习框架最开始使用的是第一种方式，目前正在朝着两者融合以及越来越多的使用自动生成高性能算子的方向演进。

7.1 手工编写算子代码

芯片厂商在提供硬件的同时，也会提供相应的编程接口和高性能计算库，比如 cuda、cublas 、MKL、cuDNN 等。利用这些库中提供的函数，可以实现深度学习中常见的算子。芯片厂商通常会根据芯片内部架构细节对这些算子进行深度优化，因此在能使用这些算子的情况下，一般会获得比较好的性能。

然而，在一些特定情况下不能直接使用芯片厂商提供的算子库。一种情况是需要计算图中的 Operator 类型时，比如增加一种新的激活函数；另一种情况是为了优化性能而需要进行特定的算子融合时。这些情况下需要用户自己实现高性能的算子，再将其注册到框架中，使框架可以在运行时调用用户实现的算子。Tensorflow 等框架都支持这种机制。

用户可以选用不同的方法实现特定算子时，从性能角度考虑，最好的方法是使用较低层的编程接口，比如对于 GPU 而言，直接写 CUDA KERNEL。其次是使用一些已有的模板计算库，比如 EIGEN 等。还有一种方式是调用多个芯片厂商提供的计算库中的函数，组合实现需要的功能。

手工编写算子代码存在两个明显的缺陷。首先，手工编写的工程量巨大。算子的实现对不同硬件无法复用，需要单独编写。同时，不同数据类型和数据排布的算子也要单独实现。其次，性能优化空间有限。手工实现的算子通常需要支持所有可能的输入规模，对于某种特定规模的输入可能并不是最优的。除此之外，调用手工编写算子需要保证计算图中每种类型的 Operator 都有算子的实现，限制了可以支持的算子融合等图优化手段。

7.2 自动生成算子代码

为解决手工编写算子代码面临的问题，一种自然的想法就是针对每一个模型，自动生成其需要的算子代码。自动生成有两方面的好处，一是不同硬件可以复用自动生成过程中的部分逻辑，减小硬件适配的工作量。二是能够支持更极致的性能优化。例如，相对调用手工编写的算子，自动生成算子可以和算子融合等图优化更好的结合，支持更多样的算子融合。也可以利用模型中各节点输入张量的规模信息，进行特定优化。

深度学习中算子的输入和输出都是张量，计算逻辑相对统一和规则，比如大部分的激活函数，可以统一抽象为循环对输入张量中的每个元素进行某种操作。这种特征有利于对算子的计算逻辑进行抽象，将其用另外一种形式的中间表示 （Intermediate Representation）去描述，然后再根据中间表示生成针对不同硬件的代码。虽然具体做法不同，但这种中间表示要满足一些通用的要求，最主要的是能够更方便的通过中间表示的变换实现常用的算子优化手段，而这些变换又能进一步通过规则、策略或者算法去控制。

目前多个开源深度学习框架/编译器已经支持自动生成算子代码。其中，Tensorflow 中的子项目 XLA 和 TVM 是最有影响力的两个项目，也是两种不同技术路径的代表。XLA 支持由有限个 HLO （High Level Operator）组成的计算图节点，针对 CPU和 GPU，每个 HLO 有其单独的 IR 表示，内置了很多规则和策略对这些 IR 进行变换，并最终生成代码。TVM 自动生成算子代码的技术是从 Halide 演变而来，沿用了其计算 (Compute) 和调度（Schedule）分离的思路，提供了相关接口供用户描述计算逻辑和调度，从而达到对生成代码的控制。TVM 在 Halide 基础上扩展实现了很多新的 Schedule原语，同时也支持 Auto Schedule 等功能。 下面稍微扩展介绍下 TVM 的原理。

如前所述，TVM 沿用了 Halide 将计算逻辑 (Compute) 和调度（Schedule）分离的思路。 Compute 的目标是用符号化的方式描述计算输出 Tensor 中指定位置数据的逻辑，可以将其视作一个函数，参数为输出 Tensor 元素的 Index，函数内部是根据 Index 和输入 Tensor 计算输出的逻辑。 Schedule 的目标是确定计算输出 Tensor 中所有数据的顺序，比如对于二维矩阵，我们可以逐行计算；也可以将矩阵分成若干个子矩阵，循环计算每个子矩阵的元素。不同的计算顺序会影响并行性、访存局部性和重复计算的数量，从而影响最终生成的代码的性能。关于 Compute 和 Schedule 的详细讨论强烈建议参考 Halide 作者之一的博士毕业论文：Decoupling Algorithms from the Organization of Computation for High Performance Image Processing.

下面是 TVM 官网上提供的生成矩阵乘法代码的示例。参考 https://tvm.apache.org/docs/tutorials/optimize/opt_gemm.html

```
# Algorithm
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k), name="C")

# Default schedule
s = te.create_schedule(C.op)
func = tvm.build(s, [A, B, C], target=target, name="mmult")
assert func

c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), ctx)
func(a, b, c)
answer = numpy.dot(a.asnumpy(), b.asnumpy())
tvm.testing.assert_allclose(c.asnumpy(), answer, rtol=1e-5)
```

上述代码中，A、B 表示输入 Tensor, C 为输出 Tensor。通过下面的语句建立起了输入 Tensor 和输出 Tensor 之间的逻辑关系：

```
 C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k), name="C")
```

上面的代码非常直观，基本就是下面数学公式直接翻译过来的：

C[x,y]= ∑_kA[x,k]*B[k,y]

上面的语句实际上就是定义了前面所说的 Compute 函数。(x, y) 为 Index，表示输出矩阵 C 行和列。除了 Compute 函数，还需要定义 Schedule, 是通过下面语句完成的：

```
 s = te.create_schedule(C.op)
```

最后通过调用 build 函数自动生成代码，并返回相应的函数接口。这个函数接口的原型是：

```
func(input_1, input_2, output)
```

最后直接通过调用这个函数就可以完成计算, 对应的代码是 func(a, b, c) 。

下面是上面默认 Schedule 生成的 Low level IR。

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x: int32, 0, 1024) {
    for (y: int32, 0, 1024) {
      C_2[((x*1024) + y)] = 0f32
      for (k: int32, 0, 1024) {
        C_2[((x*1024) + y)] = ((float32*)C_2[((x*1024) + y)] + ((float32*)A_2[((x*1024) + k)]*(float32*)B_2[((k*1024) + y)]))
      }
    }
  }
}
```

从代码可以看出，采用默认 Schedule，会逐行计算输出矩阵 C 的每个元素。

除了使用默认 Schedule, 用户可以通过一些元语修改 Schedule, 达到优化性能的目的。例如，通过下面的代码可以创建分块计算的 Schedule:

```
bn = 32
s = te.create_schedule(C.op)
# Blocking by loop tiling
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

# Hoist reduction domain outside the blocking loop
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

采用上面的 Schedule 生成的伪代码如下：

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x.outer: int32, 0, 32) {
    for (y.outer: int32, 0, 32) {
      for (x.inner.init: int32, 0, 32) {
        for (y.inner.init: int32, 0, 32) {
          C_2[(((((x.outer*32768) + (x.inner.init*1024)) + (y.outer*32)) + y.inner.init)] = 0f32
        }
      }
      for (k.outer: int32, 0, 256) {
        for (k.inner: int32, 0, 4) {
          for (x.inner: int32, 0, 32) {
            for (y.inner: int32, 0, 32) {
              C_2[(((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)) + y.inner)] = ((float32*)C_2[(((((x.outer*32768) + (x.inner*1024)) + (y.outer*32
            }
          }
        }
      }
    }
  }
}
```

注意到经过分块后，X、Y 和 K 轴都被分割为两部分：outer 和 inner。这样原来的三重循环就会变为六重循环。这个分割功能是通过 split 和 tile 完成后。分割后的循环顺序通过 reorder 来指定。

除了 split, reorder, TVM 还提供了 vectorize, inline, cache_read, cache_write, scope，fusion 等诸多 Schedule 元语。这里不再一一介绍。

针对每个算子，由用户指定 Schedule 来获得最优的性能还是比较复杂的，特别是需要选择诸如分块大小、循环顺序等参数时。为了解决这个问题，TVM 在上述代码生成流程的基础上进一步支持了 Auto Schedule, 基于提供的 Schedule 模板去自动搜索最优的 Schedule。细节介绍可参考 TVM 论文：TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.