基于 tvm 的神经网络算子性能优化

武家伟 (1600012924)　　丁聪 (1600011785)

2019 年 6 月

摘要

tvm 是一个针对 CPU 和 GPU 的深度学习编译栈。它通过将算子的描述（compute）和实现（schedule）分离的方式，实现了性能的大幅提升。本次课程报告利用 tvm 定义的原语和提供的 autotvm 接口，实现了批量矩阵乘和卷积两个算子的自动优化方法。

关键字：tvm, 神经网络, 卷积, GEMM

1 问题描述

- 批量矩阵乘（GEMM）算子

矩阵乘法是深度学习实现中的常用函数，也是高性能计算领域的经典问题。算子的数学描述如下：

```
def batch_gemm(N, H, K, W):
    A = tvm.placeholder((N, H, K))
    B = tvm.placeholder((N, K, W))
    k = tvm.reduce_axis((0, K))
    C = tvm.compute((N, H, W), lambda b, i, j: tvm.sum(A[b, i, k] *
   B[b, k, j]))
    return [C.op], [A, B, C]
```

对于矩阵乘法的优化，由于算子已经给定，我们无法从算法上（如 Strassen 算法）降低时间复杂度。考虑到三重循环式的 naive 实现会造成许多 cache miss，而 cache miss 会造成运行时间的大大增加。因此我们主要考虑的是提升缓存命中率以提高算子的性能。

- 卷积（CONV）算子

卷积运算是卷积神经网络的基本运算之一，而卷积运算中也涉及大量的矩阵乘法。由于 tvm 指定了编译后端为 llvm，而且 schedule 的过程也存在大量限制，因而我们在卷积的优化中也主要考虑了提高缓存命中的内存访问优化。除此之外，我们还针对 bias 和 padding 进行了各自的处理，进一步增大了搜索空间。

2 解决方案

2.1 原语的选择

在分析过问题之后，我们认为最重要的任务就在于访存优化。由于单核 CPU 测试环境的限定，我们首先排除了 **parallel** 和 **bind**。在实验过程中，**unroll**、**compute_inline** 和 **vectorize** 并没有对性能造成很大影响，甚至会稍有降低。我们认为这部分优化已经被编译后端 llvm 完成，因而也不使用这两个原语。而 **compute_at** 对实验结果也未造成显著影响。因而我们主要考虑的原语有：

- split
- reorder

2.2 搜索空间的定义

我们利用了 tvm 自带的 autotvm 模块以实现自动化搜索。首先我们定义了 **getSplit** 函数，以获得参数的待定序列。

```
def getSplit(maxNum):
    splitList = []
    splitList.append(1)
    para = 2
    while (True):
        if para <= maxNum / 2 and para <= 32:
            splitList.append(para)
            para *= 2
        else:
            break
    if 16 in splitList:
        splitList.remove(16)
    return splitList
```

这个函数给出了一些除 16 外的 2 的幂备选值。

然后我们分别对矩阵乘算子和卷积算子定义了一些搜索结点（knob）。在 knob 上进行 split 原语操作，此后再利用 reorder 指定顺序。而搜索结点的备选值由 **getSplit** 函数给出。

```
x = gemm_op.op.axis[1]
y = gemm_op.op.axis[2]
k = gemm_op.op.reduce_axis[0]

cfg = autotvm.get_config()

cfg.define_knob("split_y", getSplit(int(x.dom.extent)))
cfg.define_knob("split_x", getSplit(int(y.dom.extent)))
cfg.define_knob("split_k", getSplit(int(k.dom.extent)))

xo, xi = gemm_op.split(x, cfg["split_x"].val)
```

```
yo, yi = gemm_op.split(y, cfg["split_y"].val)
ko, ki = gemm_op.split(k, cfg["split_k"].val)
gemm_op.reorder(xo, ko, yo, xi, ki, yi)
```

在卷积算子中，我们分别对二维卷积、padding 和 bias 做了类似的搜索空间的定义。除此之外，我们对各个 knob 进行了人为的 reorder 排序。因为把 reorder 放在搜索空间中代价过高，很容易造成超时。我们人为排序的规则是：将最需要变动的结点置于最底层。最靠近里层的结点变动最频繁，这样也可以提升一定的性能。

2.3 自动搜索与调优

经过多次实验之后，我们选择了 autotvm 的遗传算法（GATuner）进行了自动化参数搜索。除此之外，我们对于不同的算子设定了不同的迭代次数上限。矩阵乘算子为 100, 卷积算子为 100, 可以满足 20 分钟的时间上限。autotvm 将根据选定的 Tuner 和给定的搜索空间对参数进行启发式搜索，并记录在本地文件中，据搜索结果选定最佳的参数完成 schedule 过程。

3 本地实验结果

3.1 矩阵乘算子

- case 1: 1.4 分（ 99.90ms v.s. Pytorch:24.95ms ）
- case 2: 2.1 分（ 22.80ms v.s. Pytorch:6.98ms ）
- case 3: 4.2 分（ 26.92ms v.s. Pytorch:18.55ms ）

小计：7.7 分

3.2 卷积算子

- case 1: 1.4 分（ 152.60ms v.s. Pytorch:31.14ms ）
- case 2: 2.1 分（ 19.47ms v.s. Pytorch:5.94ms ）
- case 3: 2.1 分（ 12.57ms v.s. Pytorch:4.45ms ）

小计：5.6 分

得分总计：13.3 分

4 结论与反思

在经过多次实验之后，我们发现：

- 相较于 naive 的实现，tvm 对两个算子已经有明显的优化效果。

这证明我们降低 cache miss 率的方向基本正确，也取得了显著的效果。

- 一些原语效果不明显，甚至有反作用。

原因分析如下：一是算子经过 tvm 的 schedule 过程之后，指定了 llvm 作为后端进行编译。许多操作（如 unroll）已经由 llvm 执行优化了，因而在 schedule 过程中的性能改变不明显，甚至会造成性能损失。

- 运行时间依然逊色于 pytorch，而且在部分实例上效果相差明显。

在尝试多个原语、扩大搜索空间之后依然无法取得令人满意的效果。在课程中期报告中，其他同学也普遍反映了类似的现象。相较之下，pytorch 可以达到汇编指令级别的优化。tvm 优化的部分局限于循环和内存访问。CPU 上汇编级别的优化只能依赖于 llvm。而 llvm 的优化不如 pytorch 的手工优化，从而造成了运行时间的差距。

5 参考文献

1. TVM 官网, https://tvm.ai/
2. autoTVM 论文, Learning to Optimize Tensor Programs, https:// papers.nips.cc/ paper/ 7599-learning-to-optimize-tensor-programs.pdf
3. TVM 文档, https://docs.tvm.ai