课程项目：基于自动代码生成框架实现神经网络 PART II

May 7, 2019

1 回顾

在课程项目的第一部分中，同学们利用TVM的Tensor描述语言完成了五个卷积神经网络算子前向及反向的实现。其中包含五个算子分别为二维卷积，2x2池化，ReLU，flatten，全连接。通过与PyTorch等框架的对比，这些算子的正确性有了保证。然而，在第一部分中，所实现的算子并没有考虑性能问题。正是因为PyTorch等框架提供了高性能的算子实现，它们才被工业界和学术界广泛应用。为了实现高性能的算子，在课程项目的第二部分，我们需要对算子进行优化。通过结合两部分课程项目，同学们就有能力搭建完整的卷积神经网络并且完成快速训练与推导。

2 目的与任务

本部分是课程项目的第二部分。本部分的目的为，设计流程完成对给定算子进行自动化的性能优化。

具体任务包括：

1. 学习如何使用TVM来优化算子；
2. 实现算法完成完成对给定算子的自动优化，有效提升算子性能（即缩短算子的计算时间）。

一直以来编译器都承担了优化代码的重要任务，但是优化的形式各有不同。如数据流分析，Polyhedral模型等等。无论优化形式是什么，本质上都可以理解为对程序AST的变换，变换的限制则是要求对应的程序语义和最初相同（或者说对任何相同的输入输出结果都相同）。而编译器将这些变换实现在一个个pass中。

在TVM中提供了许多优化原语，称为Schedule原语，当我们在程序中添加这些原语后，TVM就会自动对程序的AST进行变换完成指定的优化。下面小节详细介绍TVM的Compute与Schedule这两个重要概念，并介绍如何通过TVM进行自动化优化。

2.1 TVM的Compute与Schedule

TVM作为深度学习编译器，提供了优化机制完成对算子的优化。以分块矩阵乘法为例，手工写的C++实现如下：

```
for (int i = 0; i < 64; ++i){
for (int j = 0; j < 64; ++j){
for (int k = 0; k < 64; ++k){
```

```
    // tiling
    for (int ii = 0; ii < 16; ++ii){
    for (int kk = 0; kk < 16; ++kk){
    for (int jj = 0; jj < 16; ++jj){
        C[i*16+ii, j*16+jj] += A[i*16+ii, k*16+kk] * B[k*16+kk, j*16+jj];
    }}}
}}}
```

而在TVM中，算子的数学定义与描述（Compute）与优化手段（Schedule）被特意分离开了。在分块矩阵乘法中，公式$C[i,j] = \sum_k A[i,k] \times B[k,j]$就是Compute，而用分块的形式实现矩阵乘法就是一种Schedule。为了实现各种优化手段，TVM提供了多种schedule原语。例如通过tile、split和reorder原语，分块矩阵乘法在TVM中可以被实现如下：

```
# Compute Definition
A = tvm.placeholder((1024, 1024))
B = tvm.placeholder((1024, 1024))
k = tvm.reduce_axis((0, 1024))
C = tvm.compute((1024, 1024), lambda i,j: tvm.sum(A[i,k] * B[k, j]))

# Schedule Selection
s = tvm.create_schedule(C.op)
h, w = s[C].op.axis
rk, = s[C].op.reduce_axis
ho, wo, hi, wi = s[C].tile(h, w, 16, 16)
rko, rki = s[C].split(rk, factor=16)
s[C].reorder(ho, wo, rko, hi, rki, wi)
```

对比可以发现，C++实现的方式是将Compute和Schedule融合在了一起，这种融合的方式好处在于实现起来很直接自然，缺点在于一旦优化方式变得复杂，比如添加并行优化，代码实现就会十分困难，甚至会产生失误破坏了代码原来的语义。而TVM使用Schedule的方式虽然显得不自然，有些难以理解，但是可以保证优化的正确性同时使得优化更加清晰容易，只需要一些原语就可以实现复杂优化。

但是即使TVM的Schedule使得优化更加轻松，现实中开发高性能算子仍然是一件困难的事情。困难点在于如何选择适合的Schedule方案。比如矩阵乘法分块时，分块大小是多少，如何安排循环顺序（即reorder中的排列方式）等等，都是需要小心选择的，否则性能的优化难以达到预期。而不同的schedule方案对于不同的输入数据规模（即shape）同时，由于不同计算设备的架构不同，同一个schedule方案在不同设备上会有不同的性能表现，而不同的平台的最佳schedule方案也可能会有所差异。因此，本项目要求对算子实现自动化的优化，即在算子的schedule空间中自动地搜索出较好的schedule方案。下一小节详细描述自动化schedule的要求。

2.2 自动化Schedule

为了降低本次项目的难度，本项目只选取两个算子进行自动化Schedule：“二维卷积算子”和“批量矩阵乘法”。关于这两个算子的Schedule例子在TVM的Tutorials中有详细介绍，只是要注意这

些已有的Schedule都是人手工写的。我们给定卷积算子与批量矩阵乘法的Compute，封装在给定的函数中，例如批量矩阵乘法的例子如下：

```
# batch_gemm
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]
```

函数的参数是算子的规模（shape）及一些其它参数，返回值有两个，第一个是一个list，表示算子返回的Tensor的Operation（C.op），这个list可以用来创建schedule，如s = tvm.create_schedule([C.op])，注意create_schedule函数是可以接收list作为参数的。第二个参数也是list，表示算子用到的输入输出Tensor，这些Tensor在生成最终程序时是必要的，回忆func = tvm.build(s, [A, B, C], "llvm")中第二个参数就是该list。

对于给定的Compute，我们会指定输入shape，要求自动化的优化程序根据Compute和Shape信息选择合适的优化方案，并利用Scheudle原语完成这些优化，最后将生成的Schedule（tvm.create_schedule的返回值）返回。下面给出一个模板，例子中auto_schedule就是项目要求实现的函数，其中空白的位置需要根据注释补充代码。

```
def auto_schedule(func, args):
    """Automatic scheduler

    Args:
    -----------------
    func: function object
        similar to batch_gemm function mentioned above
    args: tuple
        inputs to func
    -----------------
    Returns:
    s: tvm.schedule.Schedule
    bufs: list of tvm.tensor.Tensor
    """
    ops, bufs = func(*args)

    # do some thing with `ops`, `bufs` and `args`
    # to analyze which schedule is appropriate

    s = tvm.create_schedule(ops)

    # perform real schedule according to
    # decisions made above, using primitives
    # such as split, reorder, parallel, unroll...

    # finally, remember to return these two results
    # we need `bufs` to build function via `tvm.build`
    return s, bufs
```

3 具体要求

3.1 实验环境：

Linux, Python 3, tvm-v0.5。测评所使用的CPU架构为Intel x86_64，限制在单核下测评。

3.2 技术路线：

本部分内容是实现算子的自动化优化。自动化优化本质上可以理解为一个搜索问题，搜索空间由两部分组成：schedule原语的组合以及各个原语的参数（factor）。自动优化就是对于给定的算子，从搜索空间找到一种合适的schedule方式，使得在CPU上的运行时间最短。具体来说就是实现前面介绍过的auto_schedule函数。在实现时，可以考虑TVM中的6种schedule原语tile、split、reorder、fuse、compute_at和compute_inline，更多原语如parallel, unroll, vectorize, cache_write也是非常有用的，但是这里不再过多介绍。在搜索空间时，一般来说我们可以使用启发式搜索算法（如遗传算法，模拟退火算法）。当我们得到一种schedule后，我们既可以设计模型对其评估，也可以直接运行最后的代码在CPU上实际测出运行时间。感兴趣的同学也可以使用机器学习算法来设计这个搜索算法，如autoTVM [2].

下面我们具体介绍一下部分schedule原语 [3]：

- **tile**。这个原语可以将计算内容分块，使得我们可以一个分块接着另一个的计算完这个算子。

```
    A = tvm.placeholder((m, n), name='A')
2   B = tvm.compute((m, n), lambda i, j: A[i, j], name='B')
    s = tvm.create_schedule(B.op)
4   xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1], x_factor=10,
    y_factor=5)
    print(tvm.lower(s, [A, B], simple_mode=True))
6
    output:
8   produce B {
      for (i.outer, 0, ((m + 9)/10)) {
10      for (j.outer, 0, ((n + 4)/5)) {
          for (i.inner, 0, 10) {
12          for (j.inner, 0, 5) {
              if (likely(((i.outer*10) < (m - i.inner)))) {
14              if (likely(((j.outer*5) < (n - j.inner)))) {
                  B[(((j.outer*5) + (((i.outer*10) + i.inner)*n)) + j.inner)] =
    A[(((j.outer*5) + (((i.outer*10) + i.inner)*n)) + j.inner)]
16              } } } } } } }
```

- **split**。可以通过一个参数factor将一个维度分成两个子维度。

```
    A = tvm.placeholder((m,), name='A')
    B = tvm.compute((m,), lambda i: A[i]*2, name='B')
    s = tvm.create_schedule(B.op)
    xo, xi = s[B].split(B.op.axis[0], factor=32)
    print(tvm.lower(s, [A, B], simple_mode=True))

    output:
    produce B {
      for (i.outer, 0, ((m + 31)/32)) {
        for (i.inner, 0, 32) {
          if (likely(((i.outer*32) < (m - i.inner)))) {
            B[((i.outer*32) + i.inner)] = (A[((i.outer*32) + i.inner)]*2.000000
f)
          } } } }
```

- **reorder**。可以用来调整各维度的计算顺序。

```
    A = tvm.placeholder((m, n), name='A')
    B = tvm.compute((m, n), lambda i, j: A[i, j], name='B')
    s = tvm.create_schedule(B.op)
    # tile to four axises first: (i.outer, j.outer, i.inner, j.inner)
    xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1], x_factor=10,
y_factor=5)
    # then reorder the axises: (i.inner, j.outer, i.outer, j.inner)
    s[B].reorder(xi, yo, xo, yi)
    print(tvm.lower(s, [A, B], simple_mode=True))

    output:
    produce B {
      for (i.inner, 0, 10) {
        for (j.outer, 0, ((n + 4)/5)) {
          for (i.outer, 0, ((m + 9)/10)) {
            for (j.inner, 0, 5) {
              if (likely(((i.outer*10) < (m - i.inner)))) {
                if (likely(((j.outer*5) < (n - j.inner)))) {
                  B[(((j.outer*5) + (((i.outer*10) + i.inner)*n)) + j.inner)] =
 A[(((j.outer*5) + (((i.outer*10) + i.inner)*n)) + j.inner)]
                } } } } } } }
```

- **fuse**。可以用来合并两个相邻的计算维度。

```
    A = tvm.placeholder((m, n), name='A')
    B = tvm.compute((m, n), lambda i, j: A[i, j], name='B')
    s = tvm.create_schedule(B.op)
```

```
    # tile to four axises first: (i.outer, j.outer, i.inner, j.inner)
    xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1], x_factor=10,
    y_factor=5)
    # then fuse (i.inner, j.inner) into one axis: (i.inner.j.inner.fused)
    fused = s[B].fuse(xi, yi)
    print(tvm.lower(s, [A, B], simple_mode=True))

    output:
    produce B {
    for (i.outer, 0, ((m + 9)/10)) {
    for (j.outer, 0, ((n + 4)/5)) {
      for (i.inner.j.inner.fused, 0, 50) {
        if (likely(((i.outer*10) < (m - (i.inner.j.inner.fused/5))))) {
          if (likely(((j.outer*5) < (n - (i.inner.j.inner.fused % 5))))) {
            B[(((j.outer*5) + (i.inner.j.inner.fused % 5)) + (((i.outer*10) + (
    i.inner.j.inner.fused/5))*n))] = A[(((j.outer*5) + (i.inner.j.inner.fused %
     5)) + (((i.outer*10) + (i.inner.j.inner.fused/5))*n))]
          } } } } } }
```

- **compute_at**。当算子中包含多个tensor的计算时，默认的schedule会单独计算出各个tensor，如下：

```
    A = tvm.placeholder((m,), name='A')
    B = tvm.compute((m,), lambda i: A[i]+1, name='B')
    C = tvm.compute((m,), lambda i: B[i]*2, name='C')
    s = tvm.create_schedule(C.op)
    print(tvm.lower(s, [A, B, C], simple_mode=True))

    output:
    produce B {
      for (i, 0, m) {
        B[i] = (A[i] + 1.000000f)
      } }
    produce C {
      for (i, 0, m) {
        C[i] = (B[i]*2.000000f)
      } }
```

使用**compute_at**原语可以在计算B的同时计算C，如下:

```
    A = tvm.placeholder((m,), name='A')
    B = tvm.compute((m,), lambda i: A[i]+1, name='B')
    C = tvm.compute((m,), lambda i: B[i]*2, name='C')
    s = tvm.create_schedule(C.op)
    s[B].compute_at(s[C], C.op.axis[0])
    print(tvm.lower(s, [A, B, C], simple_mode=True))
```

```
7
    output:
9   produce C {
      for (i, 0, m) {
11      produce B {
          B[i] = (A[i] + 1.000000f)
13      }
        C[i] = (B[i]*2.000000f)
15    } }
```

- **compute_inline**。可以使用这个原语标记一个tensor，当这个tensor被计算需要，它的相关计算会被直接嵌入（inline）到需要的位置，例如:

```
1   A = tvm.placeholder((m,), name='A')
    B = tvm.compute((m,), lambda i: A[i]+1, name='B')
3   C = tvm.compute((m,), lambda i: B[i]*2, name='C')
    s = tvm.create_schedule(C.op)
5   s[B].compute_inline()
    print(tvm.lower(s, [A, B, C], simple_mode=True))
7
    output:
9   produce C {
      for (i, 0, m) {
11      C[i] = ((A[i]*2.000000f) + 2.000000f)
      } }
13
```

3.3 测评方式：

本次Project满分100分（占课程总成绩比例为20%），可组队完成。总得分由三部分构成：课上进行中期汇报展示（10分），最终代码及其优化性能（70分），简要的项目报告（20分）。

3.3.1 课堂中期汇报评分

课堂中期汇报，即在课程最后一堂课（6月5日），每组对当前project思路和完成情况进行简单展示，每组3分钟。汇报的重点为对问题进行分析，并提出解决思路和方法。希望同学们即使不能在最后一堂课时做出完整成果，也可以将已有工作和分析呈现出来。原则上不在这一部分扣分，但如果报告缺乏准备，内容过于单薄，会适当扣分。

3.3.2 性能测评方法与评分

对于卷积算子和批量矩阵乘法，我们分别准备了5个不同的shape，但是公开的shape只有3个，另外2个不公开。最终测试时在两个算子各5个shape上试验，共计10个case。本次项目只要

性能比率区间	得分比率
[0.1, 0.2)	0.1
[0.2, 0.3)	0.2
[0.3, 0.4)	0.3
[0.4, 0.5)	0.4
[0.5, 0.6)	0.5
[0.6, 0.7)	0.6
[0.7, 0.8)	0.7
[0.8, 0.9)	0.8
[0.9, $+\infty$)	1.0

Table 1: 得分表

求对CPU平台进行Schdule（即target="llvm"）. 测评所使用的CPU架构为Intel x86_64，限制在单核下测评。项目的测试框架获取见3.7节。测试框架调用auto_schedule函数，得到返回值s和bufs，之后用tvm.build产生函数，并运行测量运行时间。根据运行时间评判分数。更具体的细节可以阅读发放的测试框架代码进行更清晰的了解。要求每组独立完成，禁止抄袭，提交的代码将通过查重系统检验，发现抄袭现象将直接判处0分。

在性能测评部分，我们采用两种方式分别得到性能分和排名分，并取两个分数中的最大值作为本部分的最终得分，两种评分方式如下。

性能分：

最终测试时在两个算子各5个shape上试验，共计10个case，每个case占7分，满分70分。考虑到同学们各自的设备CPU种类不一，测试的性能也会产生差异，容易导致测量结果难以直接比较，所以我们使用相对加速比的方式确定得分——在同一平台上，通过与PyTorch对比得到的相对性能得到性能比率，再根据得分表1中的性能比率区间得到该题的得分比率。其中，性能比率指PyTorch函数执行时间除以auto_schedule输出的结果build的函数执行时间的结果（大于1表示超过PyTorch），得分比率则用来计算得分，计算分数方式为每个case的分数乘以对应的得分比率（比如对case1, auto_schedule输出结果为11ms，PyTroch为5ms，则性能比率为0.4545，查表知得分比率为0.4，所以这个case得分为0.4x7=2.8分），最终的性能分则是所有case得分之和。这里设置区间评分是考虑到，同一种schedule的相对加速比在不同平台可能会有一定范围的波动，我们期望尽可能地减少波动的影响，一定范围内的性能波动仍可以得到相对稳定的得分。

排名分：

在得到各组的性能分之后，我们对所有组的性能分进行排名，并计算出排名分。第一名的排名分为70分，之后每下降一名，排名分减少两分，即第k名的组得到的排名分为$70-2k$。

3.3.3 项目报告评分

项目报告满分20分，要求包含对问题的分析，对解决方案的阐述，着重说明解决方案的思路与亮点（如创新性、有效性等）。报告内请给出本地实验的结果，在实际测评时如果最终结果与汇报结果差异很大，我们将重新进行测试避免偶然误差。报告内不要粘贴大段代码，将分析思路、解决方案与亮点简明扼要地说明清楚即可。

3.4 时间点：

本次作业开始时间为5月8日12:00:00，截止时间为6月23日23:59:59。

3.5 组队方式：

要求每组2-3人，将分组名单在5月21日23:59:59前发至邮箱：pku_compilers@163.com。同一小队中的同学原则上最终得分相同，但如果出现分工差异悬殊的情况，可向助教组单独反映，在确定情况属实后将对组内分数评判进行调整。

3.6 提交要求：

请提交项目压缩包，压缩包要求解压后即可作为python的一个package进行import（具体参考代码标准），压缩包使用zip格式，命名格式为：‘成员1学号_成员2学号_成员3学号.zip‘，项目需要遵守代码标准要求。

需包含项目代码以及简略的技术报告，技术报告放在代码所在目录中。项目压缩包发至邮箱：pku_compilers@163.com

3.7 代码标准：

1. 代码框架的github地址位于https://github.com/pku-compiler-design-spring/Project2019Spring，请于此下载代码框架并阅读README。最后提交的部分为auto_schedule文件夹的压缩包，其余内容不要提交。auto_schedule文件夹已在__init__.py中export了auto_schedule函数接口，请不要改动这里。
2. 我们对代码的schedule搜索时间进行了限制：对于每个case，auto_schedule函数执行限时为20分钟，时限内必须返回一个schedule方案，否则计0分。由于计时器粒度较粗（有效单位为秒），请避免紧挨时限输出结果（容易被判为超时）。
3. 如果有其它包依赖，请在项目目录中添加 requirements.txt。

3.8 毕业班同学特殊通知：

毕业班学生有提早出成绩的要求，可以在以下两个选项进行选择：

1. 提前完成project（截止时间提前至6月20日23：59：59），但评分标准可以适当放宽；
2. 不做本次project，将本次project所占分数自动归入期末考试，此时期末考试的占分比为60%（原来为40%）。

要求毕业班学生将是否选择完成project的决定在5月21日23:59:59前发至邮箱：pku_compilers@163.com。

4 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中的schedule原语，https://docs.tvm.ai/tutorials/language/schedule_primitives.html?highlight=tile#schedule-primitives-in-tvm

4 Halide的auto-scheduler，http://graphics.cs.cmu.edu/projects/halidesched/mullapudi16_halidesched.pdf