# week14实验记录

zxp

December 23, 2023

## 1 environment

cpu:Inter i5-12400f (2.5 GHz)

System:Ubuntu 22.04.1

Compiler:gcc 12.3

## 2 code

代码和在上周的基础上将conv3的barch乘了16(之前是2)。直接卷积新增了两个不同的循环顺序的版本。新增了一个数据结构tensor_1d，和之前的tensor1d和tensor4d不同，tensor_1d储存数据用的不是vector而是double数组，tensor_1d并不继承tensor类，也不和tensor1d和tensor4d公用直接卷积函数(调用这个函数传入的是tensor类，所以tensor1d和tensor4d用的直接卷积函数是一样的)，然后用tensor_1d实现了直接卷积，有3个版本，一个是使用重载的()，一个是直接用double指针寻找数据，一个对数组下标进行了优化。

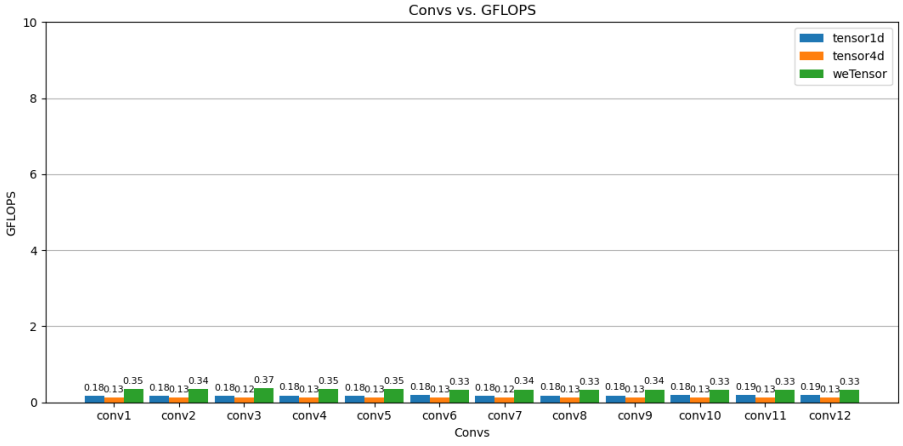


Figure 1: O0

# 3 Experiment

改变了conv3的barch，重新测试了conv3并绘制折线图

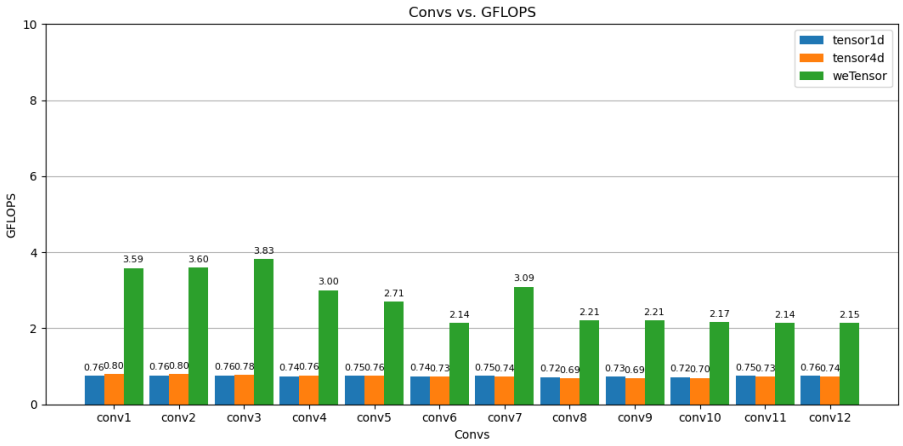


Figure 2: O2

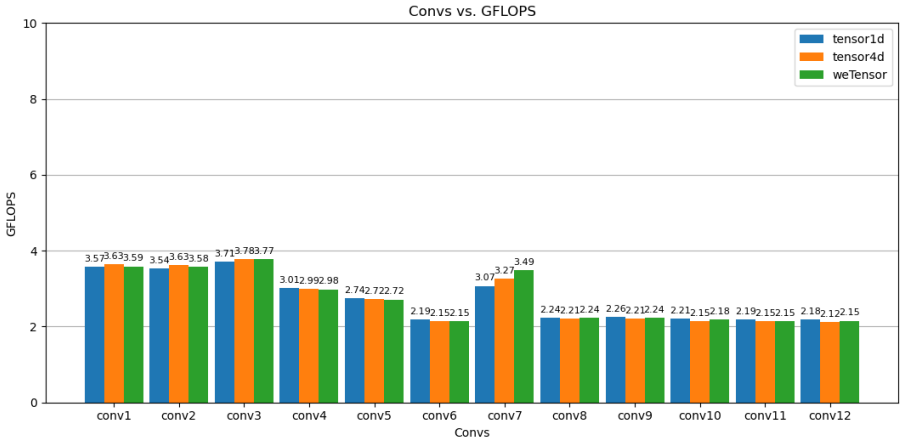


Figure 3: O3

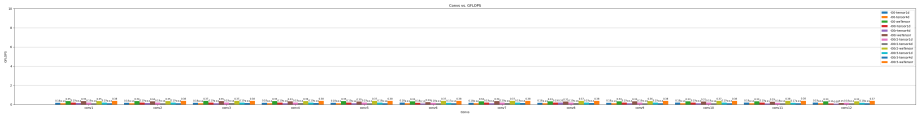

Figure 4: 把O0,O2,O3放一起比较

## 3.1 Analysis

改变conv3的barch是因为conv3太小了，设置为2每次使用conv3直接卷积只需要0.12秒左右，但是把barch改大会导致其他conv需要运行很久时间，于是只改变了conv3的barch，将conv3的barch乘了16，使每次使用conv3直接卷积超过1秒，让conv3直接卷积的时候更加稳定，不会因为误差而使得计算出的gflops变化太大。

# 4 Experiment2

前几周得出的结果中可以看到conv7在不同优化下表现十分不稳定。从conv7的输入张量尺寸特别大(长宽244×244)而卷积核特别小(长宽3×3)，按之前的直接卷积，最内层历遍卷积核，数据没有连续性。上周按论文《High performance zero-memory overhead direct convolutions》提出的优化直接卷积的方法改变直接卷积循环的顺序。但这篇论文使用的数据结构和我们的不一样，论文中的循环顺序是基于他的数据结构和他要用的优化设立的，于是我又试验了两种不同的循环顺序。下面是这些循环的代码，依次是，一开始用的循环顺序，论文中提到的循环顺序，两种不同的循环(下图中的第三种和第四种)。

```
//初始的七层循环顺序
 for (int64_t i = 0; i < C.num_batch(); ++i){
        for (int64_t j = 0; j < C.num_channel(); ++j){
            for (int64_t m = 0; m < C.num_height(); ++
                m){
                for (int64_t n = 0; n < C.num_width();
                    ++n){
                    for (int64_t r = 0; r < B.
                        num_channel(); ++r){
                        for (int64_t u = 0; u < B.
                            num_height(); ++u){
                            for (int64_t v = 0; v < B.
                                num_width(); ++v) {
                                C(i, j, m, n) += A(i,
                                    r, m * s + u, n * s
                                     + v) * B(j, r, u,
                                    v);
                            }
                        }
                    }
                }
            }
        }
    }
```

```
//论文<High performance zero-memory overhead direct convolutions 提出的循环顺序>
 for (int64_t i = 0; i < C.num_batch(); ++i){
        for (int64_t j = 0; j < C.num_height(); ++j){
            for (int64_t m = 0; m < B.num_height(); ++m){
                for (int64_t n = 0; n < B.num_width(); ++n){
                    for (int64_t r = 0; r < A.num_channel(); ++r){
                        for (int64_t u = 0; u < C.num_width(); ++u){
                            for (int64_t v = 0; v < C.num_channel(); ++v) {
                                C(i, v, j, u) += A(i, r, j * s + m, u * s + n) * B(v, r, m, n);
                            }
                        }
                    }
                }
            }
        }
    }
```

```
//更改顺序后的直接卷积
    for (int64_t r = 0; r < B.num_channel(); ++r){
        for (int64_t u = 0; u < B.num_height(); ++u){
            for (int64_t v = 0; v < B.num_width(); ++v) {
                for (int64_t i = 0; i < C.num_batch(); ++i){
                    for (int64_t j = 0; j < C.num_channel(); ++j){
                        for (int64_t m = 0; m < C.num_height(); ++m){
                            for (int64_t n = 0; n < C.num_width(); ++n){
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                            }
```

```
                        }
                    }
                }
            }
        }
    }
```

```
//更改顺序后的直接卷积
    for (int64_t i = 0; i < C.num_batch(); ++i){
        for (int64_t r = 0; r < B.num_channel(); ++r){
            for (int64_t j = 0; j < C.num_channel();
                ++j){
                for (int64_t m = 0; m < C.num_height()
                    ; ++m){
                    for (int64_t u = 0; u < B.
                        num_height(); ++u){
                        for (int64_t v = 0; v < B.
                            num_width(); ++v) {
                            for (int64_t n = 0; n < C.
                                num_width(); ++n){
                                C(i, j, m, n) += A(i,
                                    r, m * s + u, n * s
                                     + v) * B(j, r, u,
                                    v);
                            }
                        }
                    }
                }
            }
        }
    }
```

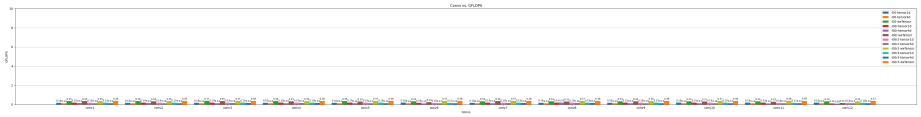

Figure 5: O0

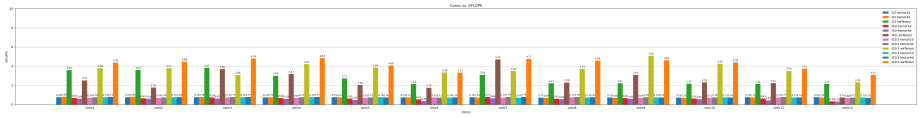

Figure 6: O2

先看将所有的gflops，放一张图，其中图中的r是论文种的顺序，r2是第三种循环顺序，r3是第四种循环顺序

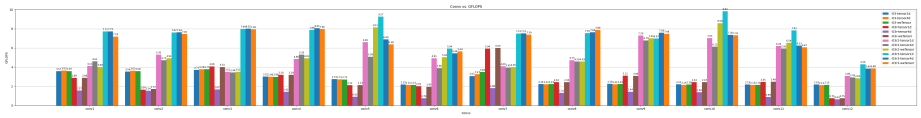

Figure 7: O3

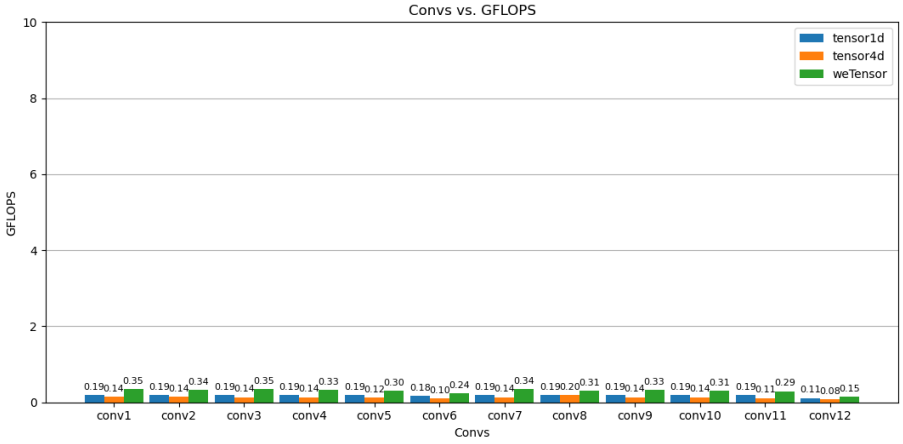


Figure 8: O0

下面是将循环顺序改成和论文《High performance zero-memory overhead direct convolutions》中一样后测试得到的gflops

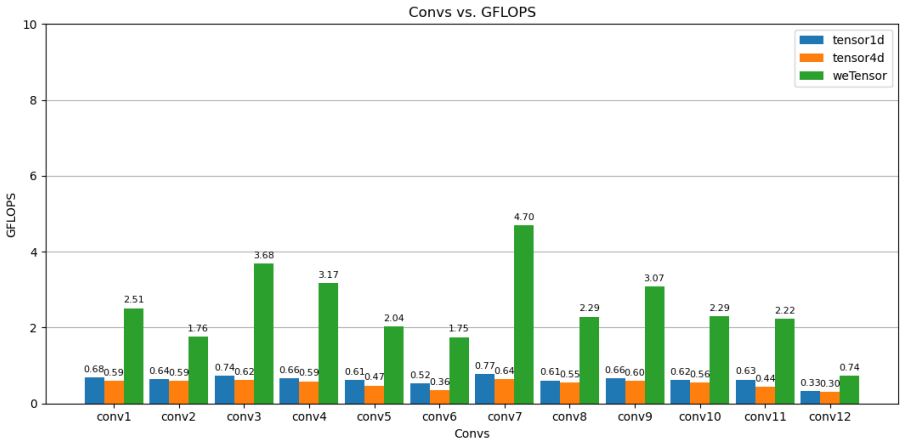
Convs vs. GFLOPS
tensor1d
tensor4d
weTensor
GFLOPS
Convs
conv1
conv2
conv3
conv4
conv5
conv6
conv7
conv8
conv9
conv10
conv11
conv12
0.68
0.59
2.51
0.64
0.59
1.76
0.74
0.62
3.68
0.66
0.59
3.17
0.61
0.47
2.04
0.52
0.36
1.75
0.77
0.64
4.70
0.61
0.55
2.29
0.66
0.60
3.07
0.62
0.56
2.29
0.63
0.44
2.22
0.33
0.30
0.74

Figure 9: O2

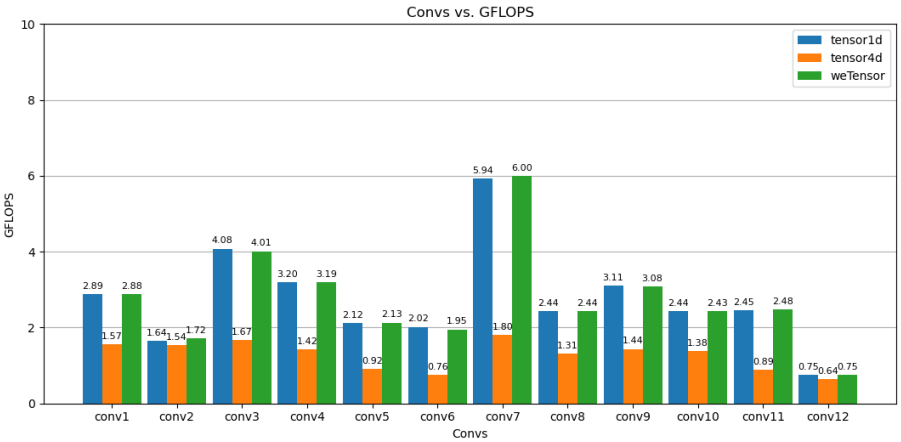
Convs vs. GFLOPS
tensor1d
tensor4d
weTensor
GFLOPS
Convs
conv1
conv2
conv3
conv4
conv5
conv6
conv7
conv8
conv9
conv10
conv11
conv12
2.89
1.57
2.88
1.64
1.54
1.72
4.08
1.67
4.01
3.20
1.42
3.19
2.12
0.92
2.13
2.02
0.76
1.95
5.94
1.80
6.00
2.44
1.31
2.44
3.11
1.44
3.08
2.44
1.38
2.43
2.45
0.89
2.48
0.75
0.64
0.75

Figure 10: O3

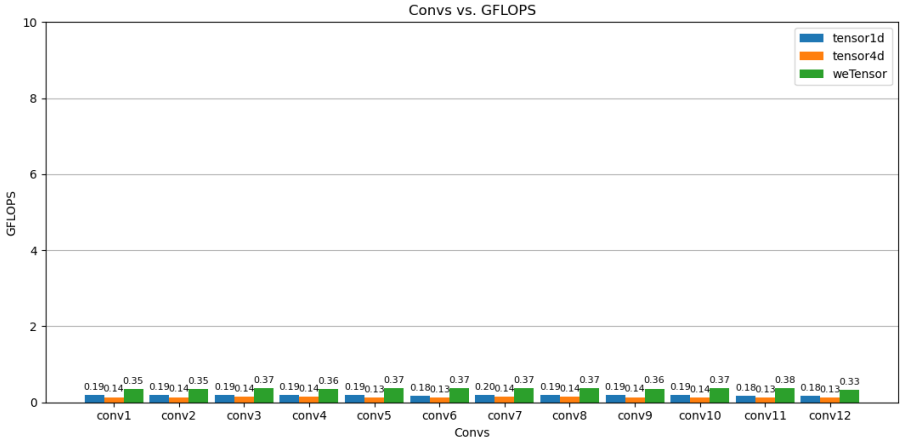


Figure 11: O0

下面是将循环顺序改成第三种测试得到的gflops

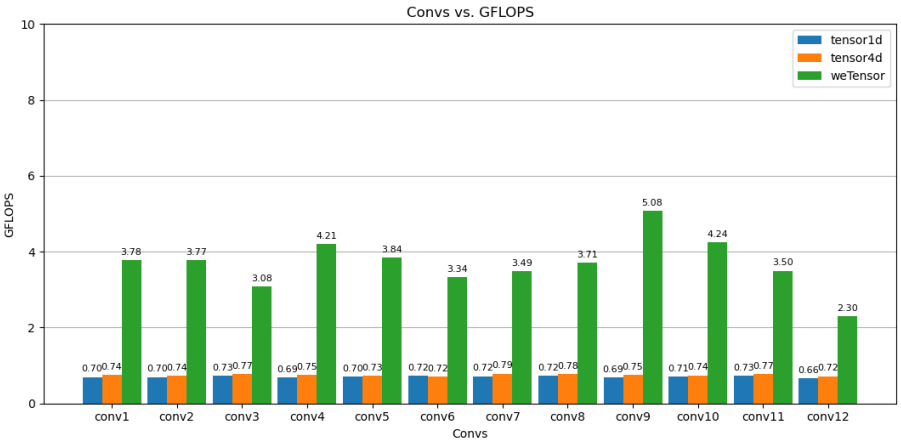
Convs vs. GFLOPS
tensor1d
tensor4d
weTensor
GFLOPS
Convs
conv1
conv2
conv3
conv4
conv5
conv6
conv7
conv8
conv9
conv10
conv11
conv12
0.70 0.74 3.78
0.70 0.74 3.77
0.73 0.77 3.08
0.69 0.75 4.21
0.70 0.73 3.84
0.72 0.72 3.34
0.72 0.79 3.49
0.72 0.78 3.71
0.69 0.75 5.08
0.71 0.74 4.24
0.73 0.77 3.50
0.66 0.72 2.30

Figure 12: O2

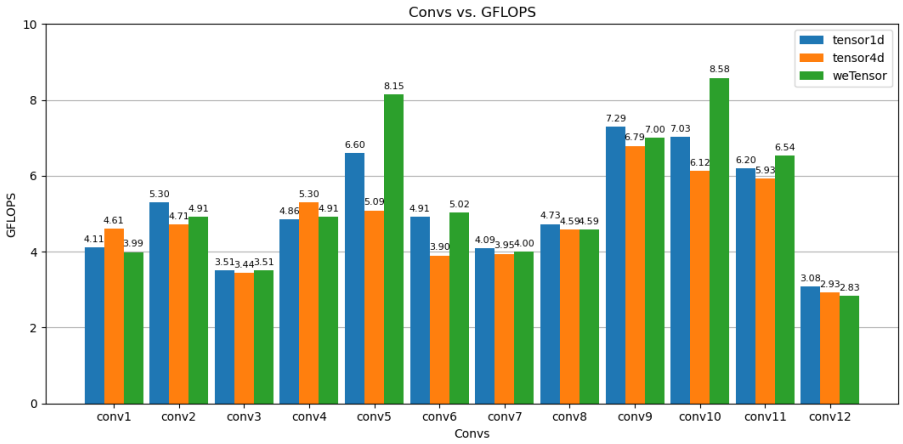
Convs vs. GFLOPS
tensor1d
tensor4d
weTensor
GFLOPS
Convs
conv1
conv2
conv3
conv4
conv5
conv6
conv7
conv8
conv9
conv10
conv11
conv12
4.11
4.61
3.99
5.30
4.71
4.91
3.51
3.44
3.51
4.86
5.30
4.91
6.60
5.09
8.15
4.91
3.90
5.02
4.09
3.95
4.00
4.73
4.59
4.59
7.29
6.79
7.00
7.03
6.12
8.58
6.20
5.93
6.54
3.08
2.93
2.83

Figure 13: O3

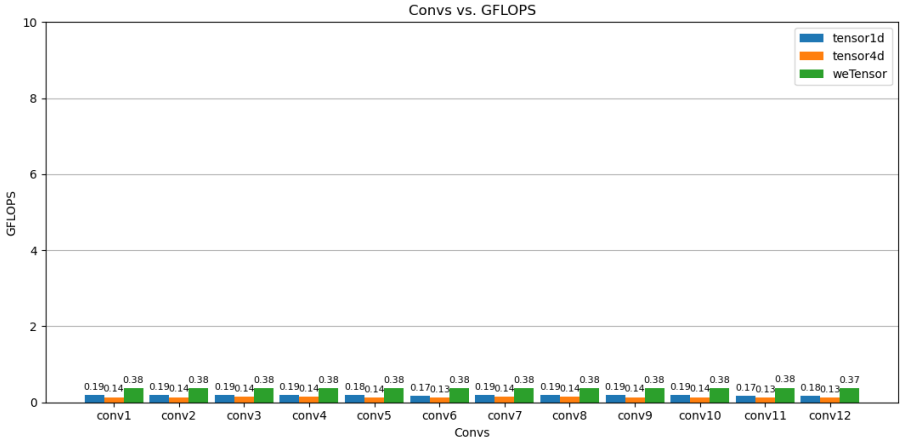


Figure 14: O0

下面是将循环顺序改成第四种一样后测试得到的gflops

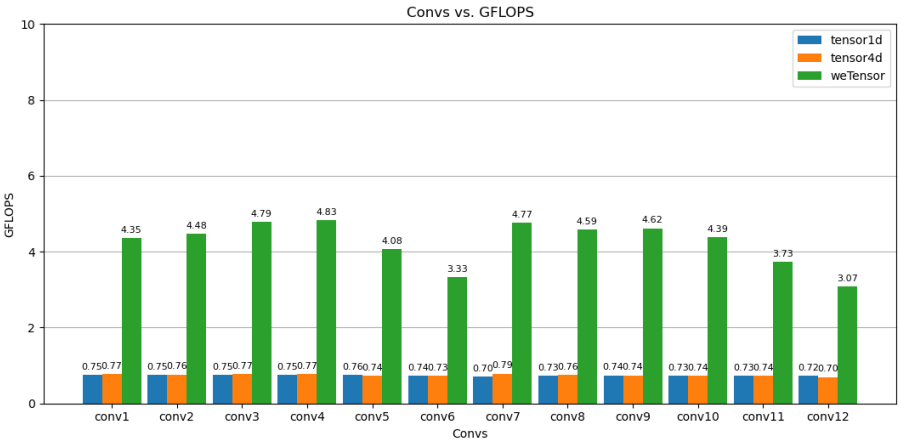
Convs vs. GFLOPS
tensor1d
tensor4d
weTensor
GFLOPS
Convs
conv1
conv2
conv3
conv4
conv5
conv6
conv7
conv8
conv9
conv10
conv11
conv12
0.75 0.77 4.35
0.75 0.76 4.48
0.75 0.77 4.79
0.75 0.77 4.83
0.76 0.74 4.08
0.74 0.73 3.33
0.70 0.79 4.77
0.73 0.76 4.59
0.74 0.74 4.62
0.73 0.74 4.39
0.73 0.74 3.73
0.72 0.70 3.07

Figure 15: O2

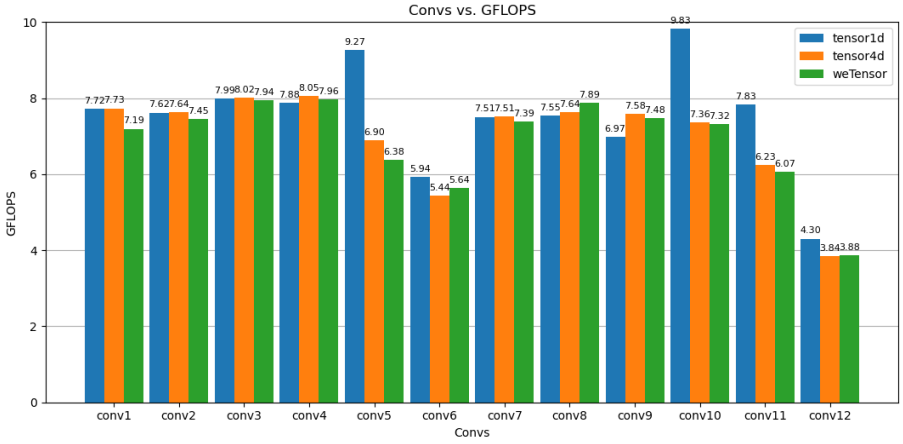


Figure 16: O3

Figure 17: 论文中的数据结构

## 4.1 Analysis

更改了循环顺序对gflops的影响是特别明显的，虽然O0和O2中tensor1d和4d变化没wetensor这么剧烈，但O3中可以看到十分夸张的变化。按论文中提出的顺序直接卷积，虽然wetenor和tensor1d在conv7的性能上升了，但在部分conv（比如conv1和conv2）的性能反而下降了，tensor4d则是直接拉跨了。这是因为论文中数据结构和我们用的不一样，他是基于他需要用到的优化提出了循环并且按这个循环设计了数据结构。

论文中的输出和输入张量的数据结构是一样的，卷积核的数据结构和输出输入张量的数据结构不同。论文中的输出的张量连续性最好的是批次，然后是宽最后是列（论文中的数据结构不包括批次）。最快的维度是块化的输出通道，其次是块化的输入通道，卷积核的宽度和高度，输入通道，然后是输出通道。tensor1d和tensor4d

```
Tensor4D(int64_t N, int64_t C, int64_t H, int64_t W) {
        this->batch = N;
        this->channel = C;
        this->height = H;
        this->width = W;
        // 给四维数组分配空间
        this->tensors.resize(N);
        for (int64_t i = 0; i < this->batch; ++i) {
            this->tensors[i].resize(C);
            for (int64_t j = 0; j < this->channel; ++j
                ) {
```

```
11                  this->tensors[i][j].resize(H);
12                  for (int64_t k = 0; k < this->height;
                        ++k) {
13                      this->tensors[i][j][k].resize(W);
14                  }
15              }
16          }
17
18      }
```

tensor1d数据连续性最好的是宽，然后才是高，通道和批次。而tensor4d几乎只在宽上有连续性。直接用论文中提出的顺序导致tensor4d没有连续性，在conv7中输出张量远大于卷积核，论文中输出张量的高比最开始的更内层，所以连续性比初始的循环顺序好。所以我按把数据多的即历遍输出张量的循环放内层，把连续性好的宽的历遍放最里面，按这个顺序得到结果比gflops要好。而且提升并不是局限在conv7，不像上面的循环有些conv减低了，而是在全部conv也是有提升。而论文提出的顺序并不是粗暴的把大的放内层，而是用一层循环使数据饱和后在这层循环外几层循环尽量快一点的把该层循环需要用到的数据搬运进内存。于是我又测试了把连续性好的输出张量的宽的历遍放最里面，然后历遍把卷积核的宽高放靠里些（不够彻底，要是把历遍卷积核的通道放在历遍张量的高里面说不定会更好一些）。按这个顺序得到的结果又比把数据多的循环放内层要好，而且是全面的好，还好很多，在O3的情况下tensor1d/4d和wetensor在很多conv的gflops都超过了7 在conv5的情况下，tensor1d甚至达到了9，比最开始用到循环顺序性能好了3倍多，这只是改变循环带来的影响。

# 5 Experiment3

从前几周的结论和改变循环顺序可以看出，四维数组的比一维数组糟糕，四维数组数据连续性很差，索引优化也没法在四维数组上用，而且tensor1d和tensor4d继承了tensor类，直接卷积函数传入的参数是tensor类的指针，这样不如给tensor1d独立的直接卷积函数快。于是我写了tensor_1d类，这个类不继承tensor类，并且写了DTensor_1D类来继承tensor_1d，DTensor_1D设定储存数据的部分是double数组（就像wetensor那样写，因为数据是double形，实际上使用的是继承了wetensor的dtensor）。并且给DTensor_1D使用的直接卷积函数传入的就是DTensor_1D。然后测试了三种方式使用DTensor_1D直接卷积（循环的顺序是前几周的顺序，并为使用这周的顺序），第一种方式使用tensor_1d中重载的()来获取数据

```
output(i,j,m,n) += input(i, r, m * s + u, n * s + v) *
    fiter(j, r, u, v);
```

第二种如同使用wetensor那样使用DTensor_1D中的double指针来获取数据

```
double* inptr = input.getDataPtr();
double* fitr = fiter.getDataPtr();
double* outptr = output.getDataPtr();
outptr[i*output.channel*output.width*output.height + j
   *output.width*output.height + m*output.width + n]
   +=
                                    inptr[i*input.channel*
                                       input.width*input.
                                       height + r*input.
                                       width*input.height
                                       + (m*s+u)*input.
                                       width + n*s+v]*
                                    fitr[j*fiter.channel*
                                       fiter.width*fiter.
                                       height + r*fiter.
                                       width*fiter.height
                                       + u*fiter.width + v
                                       ];
```

第三种，在第二种的基础上对索引进行优化

```
void directConvolution( DTensor_1D& input,DTensor_1D&
   fiter,DTensor_1D& output,int64_t s){
    double* inptr = input.getDataPtr();
    double* fitr = fiter.getDataPtr();
    double* outptr = output.getDataPtr();
    size_t O_cwh = output.channel*output.width*output.
       height;
    size_t O_wh = output.width*output.height;
```

```
7       size_t O_w = output.width;
8       size_t I_cwh = input.channel*input.width*input.
            height;
9       size_t I_wh = input.width*input.height;
10      size_t I_w = input.width;
11      size_t f_cwh = fiter.channel*fiter.width*fiter.
            height;
12      size_t f_wh = fiter.width*fiter.height;
13      size_t f_w = fiter.width;
14      for (int64_t i = 0; i < (int64_t)output.get_batch
            (); ++i){
15          size_t i_ti = i*I_cwh;
16          size_t O_ti = i*O_cwh;
17          for (int64_t j = 0; j < (int64_t)output.
                get_channel(); ++j) {
18              size_t F_tj = j*f_cwh;
19              size_t O_tj = O_ti + j*O_wh;
20              for (int64_t m = 0; m < (int64_t)output.
                    get_height(); ++m){
21                  size_t O_tm = m*O_w+ O_tj;
22                  size_t I_tm = m*s;
23                  for (int64_t n = 0; n < (int64_t)
                        output.get_width(); ++n){
24                      size_t O_t = O_tm + n;
25                      size_t I_tn = i_ti + n*s;
26                      for (int64_t r = 0; r < (int64_t)
                            input.get_channel(); ++r){
27                          size_t I_tr =  r*I_wh + I_tn;
28                          size_t F_tr = F_tj + r*f_wh;
29                          for (int64_t u = 0; u < (
                                int64_t)fiter.get_height();
                                 ++u){
30                              size_t I_t = I_tr + (I_tm+
                                    u)*I_w ;
31                              size_t F_t = F_tr + u*f_w;
32                              for (int64_t v = 0; v < (
                                    int64_t)fiter.get_width
                                    (); ++v) {
33                                  // C(i, j, m, n) += (*
                                        this)(i, r, m * s +
                                         u, n * s + v) * B(
                                        j, r, u, v);
34                                  outptr[O_t] += inptr[
                                        I_t + v]*fitr[F_t +
                                         v];
35                              }
```

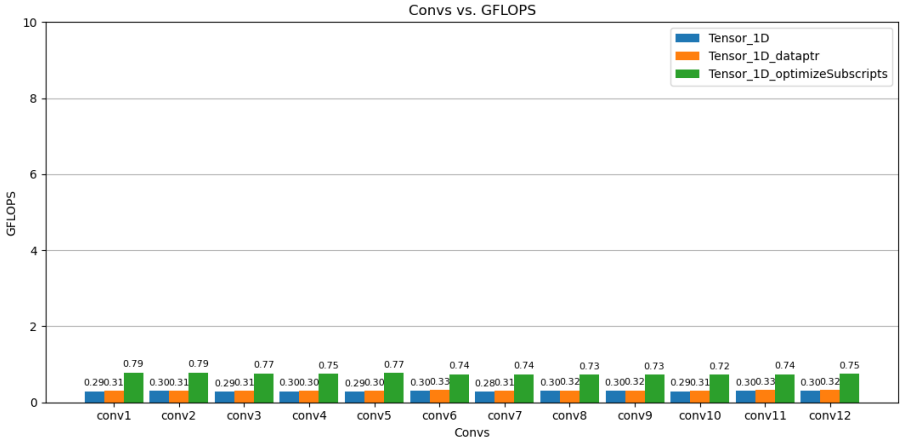


Figure 18: O0

```
36                                  }
37                              }
38                          }
39                      }
40                  }
41              }
42
43  }
```

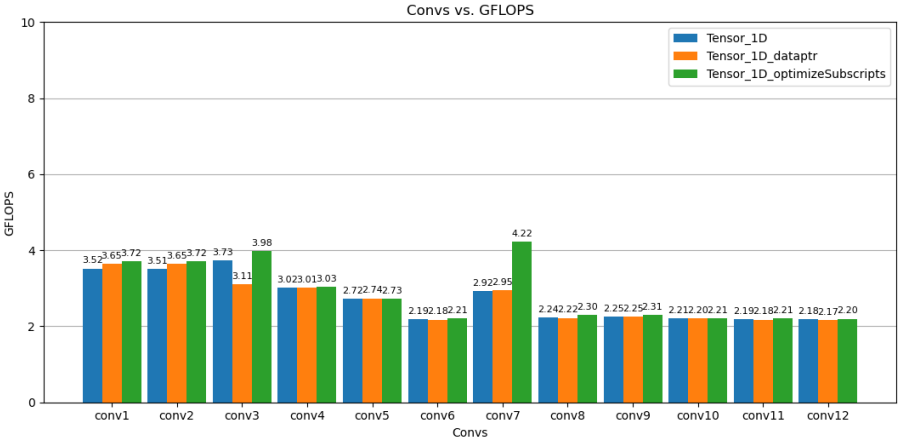


Figure 19: O2

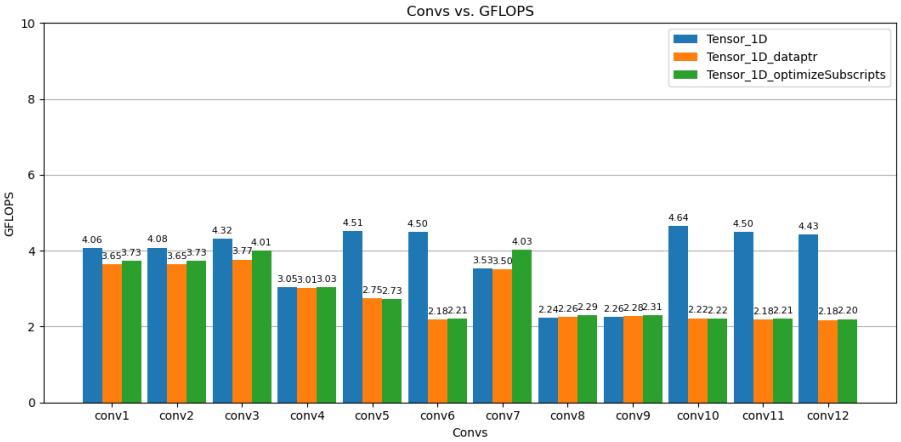


Figure 20: O3

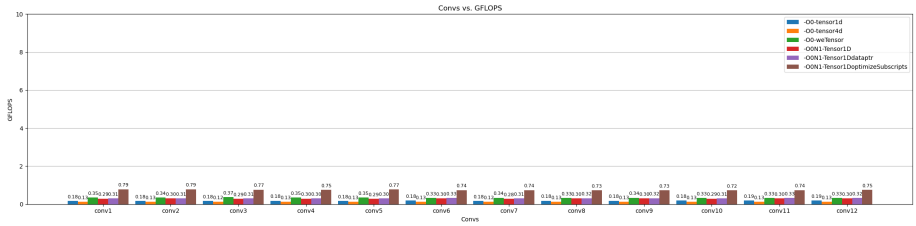


Figure 21: O0情况下对比

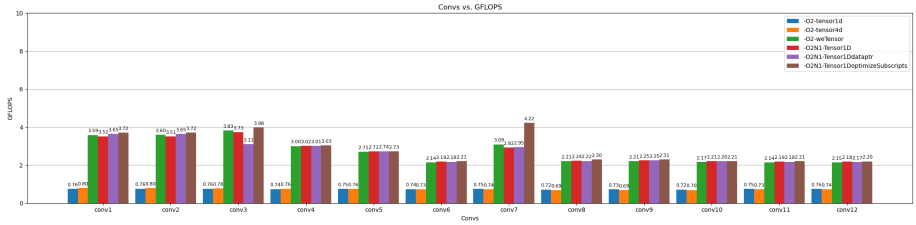


Figure 22: O2情况下对比

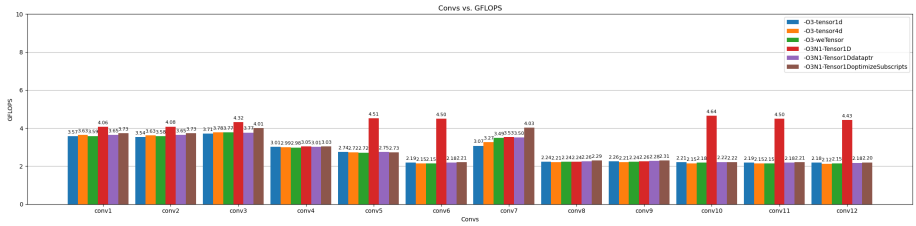


Figure 23: O3情况下对比

## 5.1 Analysis

tensor_1d十分接近wetensor了。特别是在O2的情况下，远超tensor1d和tensor4d。之前找出的优化选项-fipa-cp-clone效果是给了tensor1d和tenso4d不同的直接卷积函数，我前几周也试过直接卷积函数传入的参数不是父类tensor类而是tensor1d和tensor4d的话在O2会快特别多。现在tensor_1d不继承tensor类，有独立的直接卷积函数，所有会有这个表现。我仔细去看wetensor代码，现在的用法不知道是否正确，完全没用上libtorch的东西，直接卷积用到wetensor里的double数组，类似tensor_1d的用法。索引优化在O0的情况下特别明显，比没索引优化快了5倍多，但O2就不是这么明显，O3甚至部分conv(conv10,conv11,conv12)不索引优化比索引优化快。