23下半年总结

zxp

February 23, 2024

1 Experiment1

开头几周出了各种问题，首先是测试了一维数组和四维数组和wetensor在编译器不同优化等级（O0、O2、O3、Ofast和Ofast-march=native）下没有任何优化的直接卷积的性能差异。
出现的问题是实验的方式出了问题，开始写的代码在跑一次实验的时候测试12个conv，并且每个conv只测试一次。这样的方式是错误的，因为是要测在不同conv的表现，测试每个conv应该是不同的实验，放在一起会相互影响。正确的方式是每次实验只测试一个conv，然后测试多遍取平均值，测试多遍可以在程序中用循环测试多遍，也可以用脚本让程序运行多遍，使用的是前者。
然后发现O2情况下一维数组和四维数组表现很差，测试了O3比O2多开的优化选项，得出O3和O2主要差异是-fipa-cp-clone这个优化选项，因为一开始一维数组和四纬数组都有个tensor父类，然后使用的时候是先调用父类导致在O2的情况下表现特别差，这个优化克隆使tensor1d和tensor4d有自己不同的直接卷积函数而不是用tensor的直接卷积函数。后面察觉到wetenor底层也是一维数组（测试了不同版本的libtorch但因为直接是用不上libtorch的东西，导致差异不大，这样测没有意义，要测不同版本的libtorch的差异得调用libtorch的函数conv2d才行），这几周测试的直接卷积完全没用到libtorch的东西，差异是来自tensor1d有个tensor父类并且不是直接使用一维数组而是用vector数组。因为四维数组没法进行后续的优化，四维数组的表现在O0情况下也不如一维数组，于是放弃了四维数组。后面直接使用tensor1d做数据结构，不继承tensor父类，不使用封装过一遍的vector数组，而是直接使用1纬数组。（这里我还犯了一个内存泄漏的问题，申请了空间没在结束的时候释放，vector数组和wetenor会自动释放，一维数组不会）
然后测试了Ofast和O3之间的性能差异是由什么优化选项带来的，但结果是-funsafe-math-optimizations。这个优化选项不能使用，因为他虽然使浮点运算变快，但会导致浮点运算不符合标准，结果的精度会下降。所以Ofast优化不能使用。
-march=native说明书上说会让机器使用最适合的优化，但gcc11.4在我机器上有问题，开了之后反而更慢很多，换成gcc12.3倒是不会慢很多，但基本没差异，所以并不使用-march=native。
还有个问题是初始化和验证正确性，之前用的随机赋值是1.0到10的均匀分布，验证正确性的时候发现差的很多，能差到10的-2次方。一个是验证正确性

的时候使用了相对误差，并且没考虑浮点数特殊的地方，后面按博客改成了绝对误差。初始化也太随意了，会和真实情况差很多，图像像素的取值一般是0到255，一些预处理会将整数变为浮点数，比如归一化会使取值变为0到1的浮点数。libtorch中随机赋值有三种，第一种是0.0到1.0的均匀分布，第二种是均值为0方差为1的正态分布，第三种是自己设置范围的均匀分布。换成了和libtorch的第一种相同的方式。

2 Experiment2

不同conv在直接卷积的性能表现不同，但即使最快的conv也远远达不到能用的速度，直接卷积需要进行优化才能使用。
首先是index hoisting，O0优化下特别明显，O2优化下不算明显但使用了index hoisting的也是最快的。
然后是循环顺序，在没有其他优化的情况下调换循环顺序带来的影响是很大的，但不同循环顺序对不同的conv影响不同，试了了5种循环顺序，不同的conv有不同的最佳循环顺序，没有普适的最优，要看具体的conv。大部分情况下把output的width放在最内层效果都会不错，其他循环的循环顺序影响没这么大，原因是大部分conv的input的width和output的width比较大，在内存中也是width连续性最好。结论是没有普适的最优循环顺序，要看具体的conv的数据和张量在内存中的数据布局，把连续性好并且比较大的放在最内层会比较好。虽然变换循环顺序后性能好了很多，但循环顺序上限很低，不是最重要的，应该先做其他优化靠其他优化来决定循环顺序。
然后是不同conv的特征，conv1到12的长宽都是N，并且步长都小于N（大部分步长都为1）。并且input的长宽大于output的长宽。
直接卷积按NCHW布局，数据的连续性会受到步长的影响，比如conv3的步长相较与其他的conv就比较大，一些其它conv表现好的循环顺序（把output的宽和高放在最两层，这两个数字是最大的）它的表现就不好，因为步长比较大导致input的连续性怎么都不好（其他步长不为1的conv（conv1-4）在这两种情况都均表现不好，只是没con3这么极端）。这种步长不为1的情况也很不好使用SIMD。
还有conv7，优化在conv7上面是十分不稳定的，很多优化在其他conv上好用但在conv7上面不好用，因为它的核太小了，他的核和input的大小差距是所有conv中最大的。核只有64×3×3×3这么大，特别是通道和高和宽，只有3这么大。但因为input的宽和高很大，所以更换循环顺序后的表现还不错。然后就conv11和12这种，通道巨大，但input和output的宽高特别小，更换循环顺序后conv12是最慢的。

3 Experiment3

然后是im2win。im2win卷积运算需要先进行im2win变换得到im2win张量（测试了im2win变换所需的时间占总卷积运算的时间，占比非常小）。然后用im2win进行卷积运算。
测试了两种不同的数据布局，第一种数据布局的im2win张量在channel上是并排的，但input张量和卷积核的channel是一一对应的，每个channel都是一个核的hight*width大小的窗口在im2win张量对应的一个上channel滑动，第二种数据布局所有channel也放到这个窗口，数据连续性的部分会更大，只是改变了数据布局占的内存大小不会增加。
完全没有优化的im2win和直接卷积表现接近。

4 Experiment4

测试了不是由初始的input张量转换成im2win张量而是直接用随机数赋值的性能，并且粗略的测试了下内存情况。直接用随机数赋值性能比由初始的input张量转换成im2win张量要好一点点，少了初始的input张量占用内存，直接用随机数赋值的内存占用肯定是比由初始的input张量转换成im2win张量要小的。但是输出张量是下一层的输入张量，im2win卷积得到的输出张量和直接卷积得到的输出张量布局上是一样的，这不是对某次卷积运算进行优化，而是对全部卷积层进行优化，需要运算后直接得到im2win张量而不是常规NCHW数据布局的output，还需要后续的实验。

5 Experiment5

对im2win进行优化，做的优化主要是使用SIMD和FMA和index-hosting和hosting。使用simd的地方是窗口部分，在256位寄存器能放8个float数据或者4个double数据，然后使用FMA。hosting优化单纯的hosting了一个输出张量的元素。
进行优化后效果很明显，第二种的数据布局表现要好，特别是在conv6-12，第一种的数据布局表现不好是一个窗口大小只有3x3，太小了，而这个数据布局把核所有的channel也放在一个窗口，这样窗口要大channel倍，所以效果要好。第二种数据布局对SIMD更加友好。

6 Experiment6

然后是对比double和float，以及int和short，对比不同的数据类型对性能产生的影响。
这个不同阶段做了几次实验，测试过没有优化的直接卷积的使用不同的数据类型，使用float并没有比使用double快一倍，反而十分接近，说明是运算速度的问题。使用int比使用short快的更明显，但也没一倍。int要比double快。
然后测试了使用SIMD指令集优化后的im2win卷积运算，使用这个优化后float明显比double快，虽然不是在每个conv都快一倍，conv1-4上float比double快的很明显，在O2优化下conv1的情况float比double快了近两倍。因为在256位寄存器能放8个float数据但只能存4个double数据。使用float更快，使用double精度更高。

7 Experiment7

然后是对im2win进行优化。
首先是并行，很自然地对历遍output的batch循环使用openMP，效果非常好，性能成倍数提升。后续实验需要实验对其他循环并行，因为每次计算得到一个output的元素，每次互不干扰。
然后是unroll，unroll方式不对导致效果不好。
然后是填充，填充效果非常好，特别是conv7，O2和O3情况下性能增加近了一倍，conv7的过滤器、只有$3 \times 3 \times 3$，总共27个元素，不填充的话，前24个元素可以用3次256位寄存器算完，而剩下的需要独立算3次。填充后可以使用4次次256位寄存器直接算完。接下来的优化应该在填充的基础上做。不过现在填充的方式还有点问题，对对im2win的每行最后一个窗口都做了填充，填充至8的倍数，实数据在内存中是以一维数组的方式储存的，下一行放在上一行后面，即使不填充也不会越界，只需要填充过滤器和防止数组越界填充im2win张量的最后一个窗口，填充的时候填充至8的倍数就可以，花费的内存很小很小。