week12实验记录

zxp

December 9, 2023

1 environment

cpu:Inter i5-12400f (2.5 GHz)

System:Ubuntu 22.04.1

Compiler:gcc 12.3

2 code

因为前几周测试的方式不对，这周把测试的方式改了，并且重新把前几周的东西按新代码重新测了一遍。更改后的代码：（详细的在https://github.com/TensorConv/im2win）测试的输入张量的batch并未给出，之前设置了输入张量的batch为10，测试一次要好久，这周测试改成了测多次取平均，于是把batch设置为2 把测试tensor1d/tensor4d和wetensor的代码分开，放在三个不同的cpp文件。每个cpp文件里写了test_conv()函数用来测试直接卷积(直接卷积在同一个文件的directConvolution函数实现)，调用这个函数会用for循环对一个conv执行直接卷积50遍，并记录时间的平均值。然后写了test_conv1()-test_conv12()共12个函数给test_conv()函数传入不同的conv测试直接卷积。运行一次程序main函数会调用test_conv1()-test_conv12()中的一个。即，每次运行会测试一个conv，每次执行直接卷积50遍，记录时间的平均值。

2.1 之前代码的错误

最初版本的代码是将12个conv放入一个容器中，在main函数里用一个for循环历遍这些conv，每次用三种数据结构（tensor1d/tensor4d和wetensor）分别执行一遍直接卷积，并保存各自的运行时间。但是，马吉祥发现在O3的情况下，不用for循环历遍12个conv，而是只测试一个conv，结果只测试一个conv的结果和用for循环历遍12个conv的结果差距巨大，只测试一个conv得到的gflops只有用循环得到的结果的1/5。然后次周，我便对代码进行了更改，不使用for循环历遍12个conv，虽然还是测试代码写在main函数里面（除了实现直接卷积

的directConvolution函数），还是把12个conv放入一个容器中，但每次运行程序只测试其中一个conv，但用for函数测试一个conv多次，每次循环用三种数据结构（tensor1d/tensor4d和wetensor）分别执行直接卷积，分别记录时间。但是，得到的结果和最初版本一致，没有马吉祥出现的只测试一个conv得到的gflops只有用循环得到的结果的1/5的情况。因为for循环多少次是运行程序的时候传入的值决定的，但是即使把这个值设置为1（每次运行程序只测试一个conv并且只执行直接卷积一次），结果还是和马吉祥的结果不同。然后我把代码改了改，发现只要代码确定只测试一个conv并且只执行直接卷积一次，就会和马吉祥的结果一样。我把代码改成

```
string ttt = "1";
int tt = stoi(ttt);
for(int y = 0; y < tt; ++y) {
```

测试一个conv并且只执行直接卷积的次数是依靠string类型决定的，这样得到的结果也和马吉祥的结果不一样。然后我又改了代码，去掉了测试代码的所有循环（除了实现直接卷积的directConvolution函数）。但这次我为了让main函数看起来舒服点，把调用directConvolution的测试代码放在了test_conv()函数，test_conv1()-test_conv12()共12个函数给test_conv()函数传入不同的conv测试直接卷积，然后的出的结果又和马吉祥的不一样。然后我写了两个不同的文件，每个都是只测试一个conv并且只执行直接卷积一次，一个将调用实现直接卷积的directConvolution函数的测试代码放在main函数，另一个将调用实现直接卷积的directConvolution函数的测试代码放在自定义的test_conv()函数。只有确定只测试一个conv并且只执行直接卷积一次并且将调用实现直接卷积的directConvolution函数的测试代码放在main函数的结果和马吉祥的结果一样。

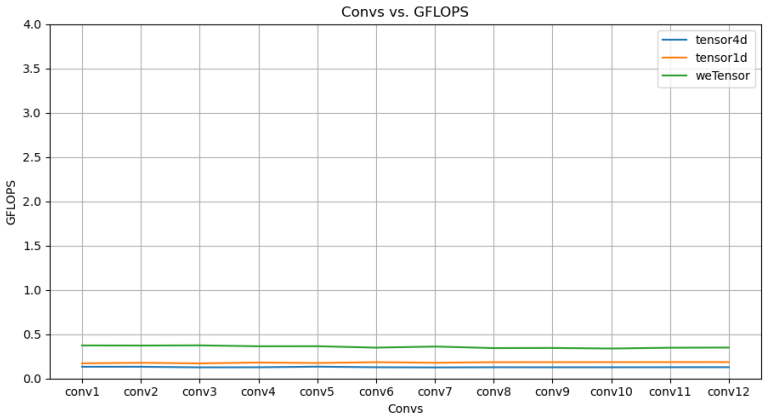


Figure 1: O0

3 Experiment

因为这周改了代码，按更改后的代码，将不同优化选项的重新测了一遍，输入张量的batch设置为2，tensor1d/tensor4d和wetensor分开测试，12个conv分开测试，测试50次取平均值。times的图效果太差不好对比，直贴出gflops的图

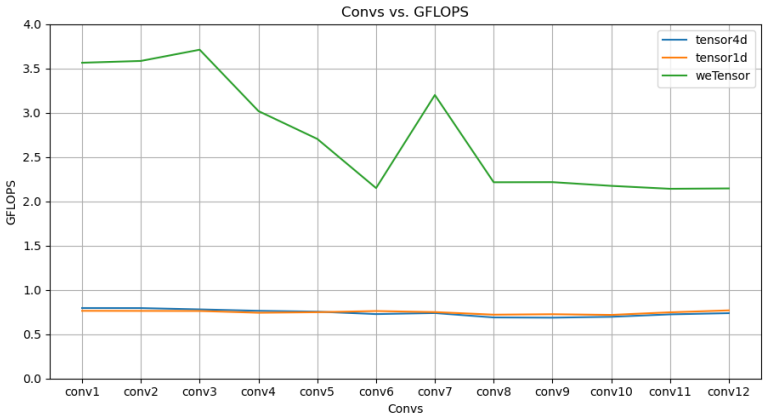


Figure 2: O2

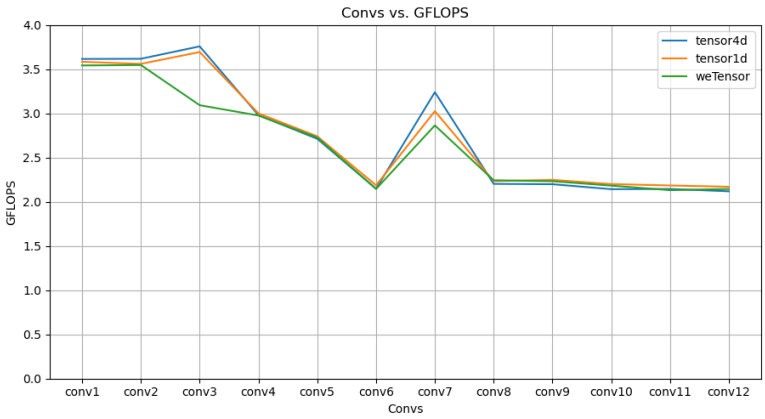


Figure 3: O3

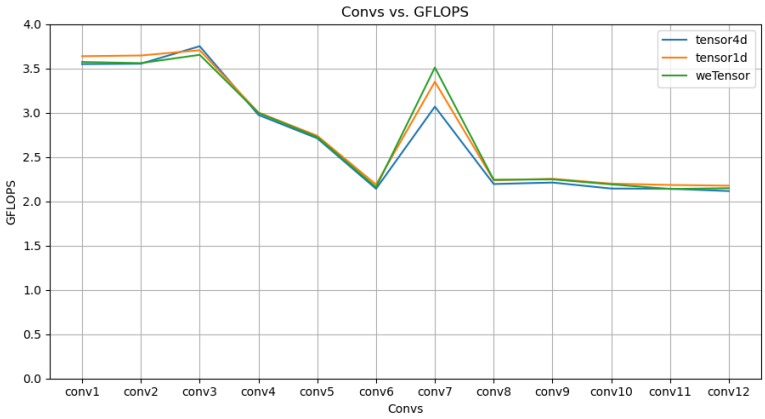


Figure 4: Ofast

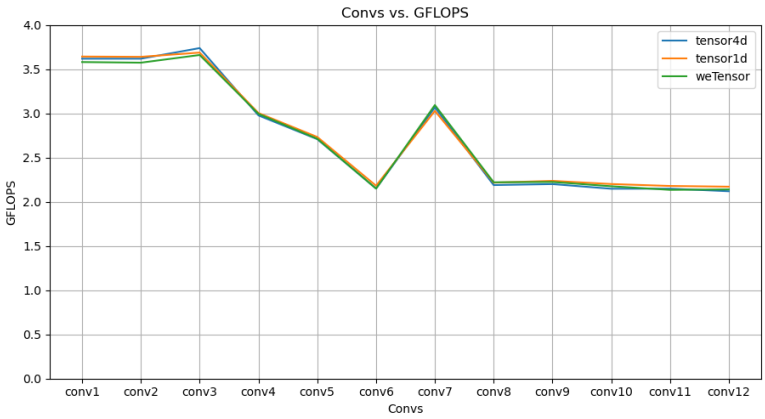


Figure 5: Ofast -march=native

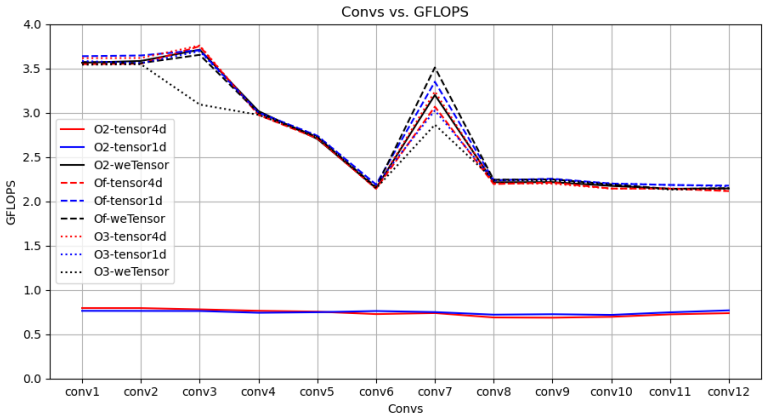


Figure 6: 把O2,O3和Ofast放一起比较

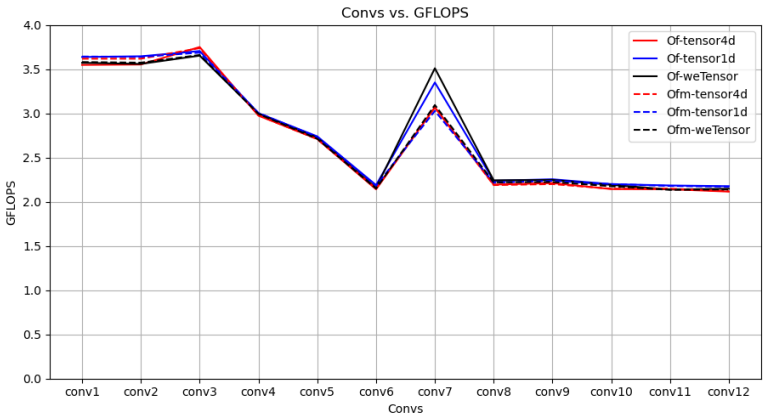


Figure 7: 把Ofast和Ofast -march=native放一起比较

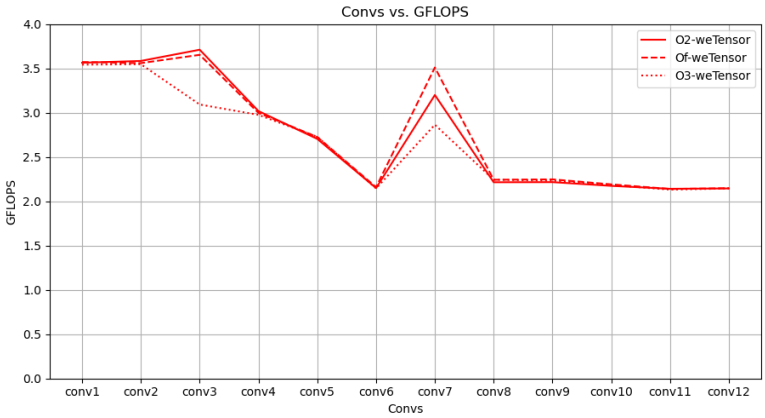


Figure 8: 只看wetensor

3.1 Analysis

使用gcc 12.3为编译器后，-march=native这个选项就不会像在gcc 11.4编译器那样大幅度降低性能，虽然在Conv7的情况下-march=native表现不是最好，但部分Conv（比如Conv1,2,11,12）下开了-march=native比不开表现好。

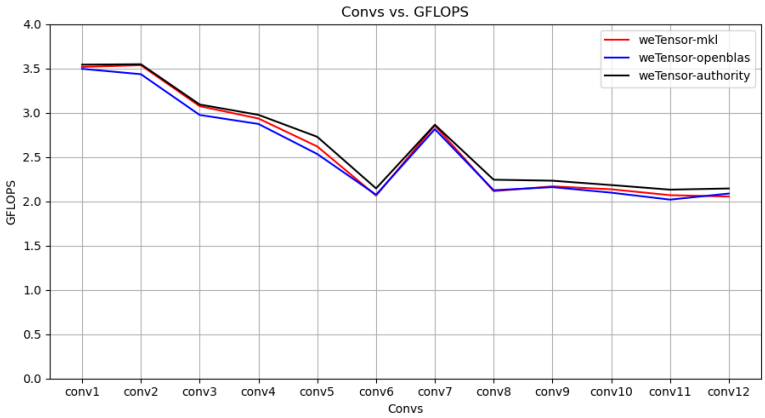


Figure 9: 对比不同版本的libtroch在O3的情况

4 Experiment2

只之前在unbuntu原生环境下编译了libtroch的cpu版本（分别使用了openblas和mkl线性代数库）和官方编译的libtroch的GPU版本（也是使用的mkl线性库）放一起进行比较（详细的libtroch信息看本文件旁边的txt文件）。
这周更改了代码，用这周写的代码对比了O3情况的性能。

4.1 Analysis

这三种的表现差不多，但官方编译的总体表现好点。我觉得这样不能说明不同版本的libtorch的差异，因为这周测试wetensor的直接卷积是我写的，就是最朴素的七层循环，没用到libtroch里的线性库的东西

5 Experiment3

查看汇编代码，前几周查看汇编代码的指令不对，导致汇编代码中的显示的函数名和源码中的不同，改用objdump -S –demangle这个命令得到的汇编代码可读性会好点，但这周还没细看汇编代码。

6 Experiment4

这周更改了代码，而且使用的和前几周用的编译器不同，编译器不同，O3开的优化选项也不同。但gcc 12.3的O3也比O2多开了-fipa-cp-clone这个优化选项，于是这周又重新测试了这个优化选项

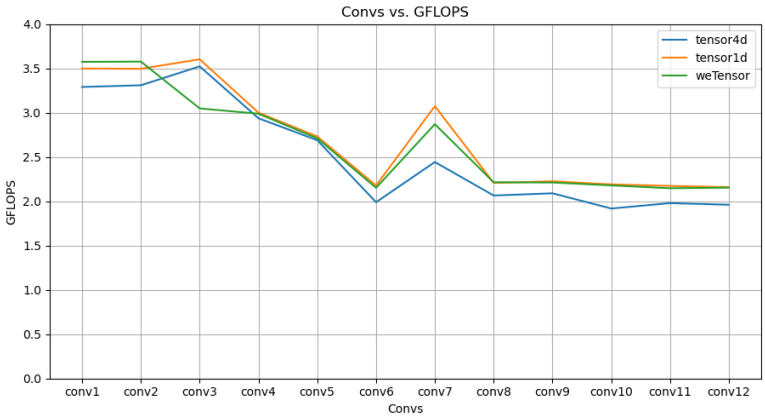


Figure 10: O2 -fipa-cp-clone

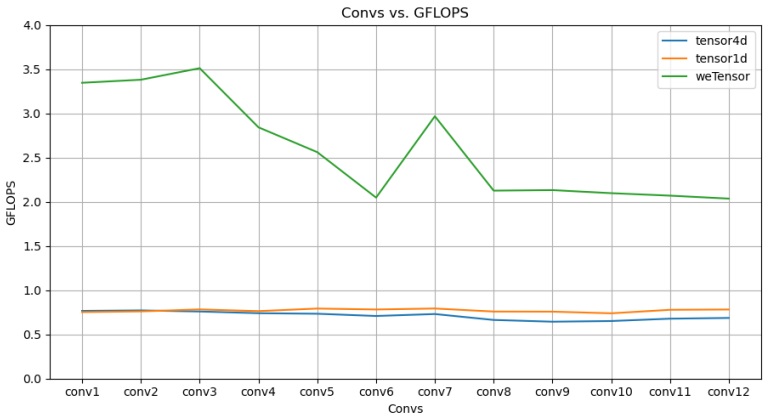


Figure 11: O2开O3多开的优化除了-fipa-cp-clone

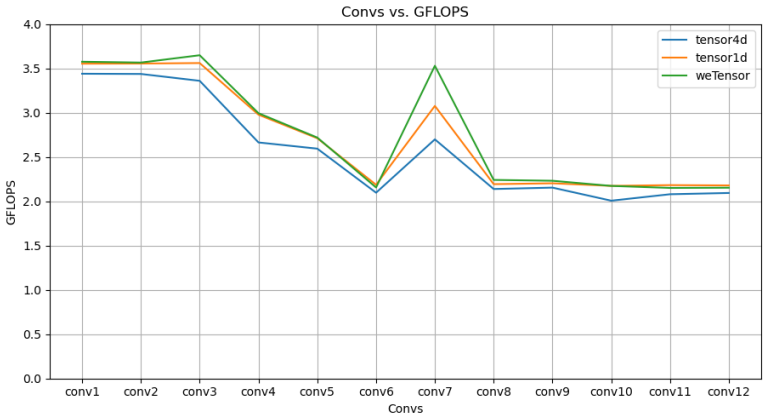


Figure 12: O3 -fno-ipa-cp-clone

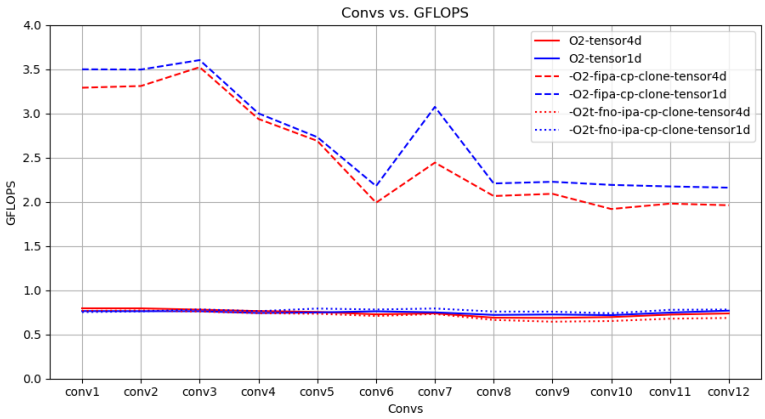


Figure 13: 对比O2 -fipa-cp-clone

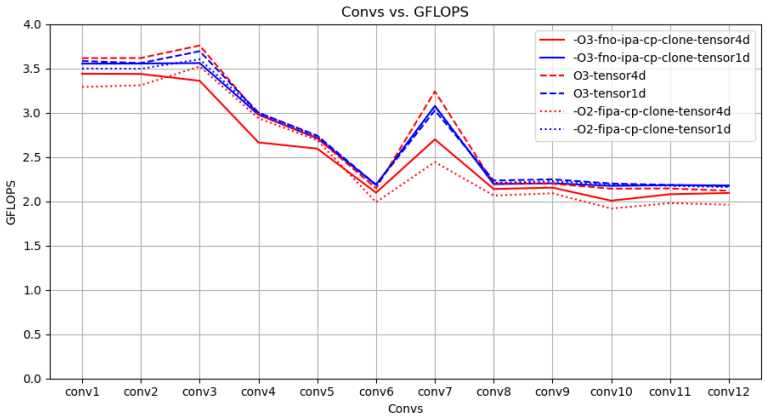


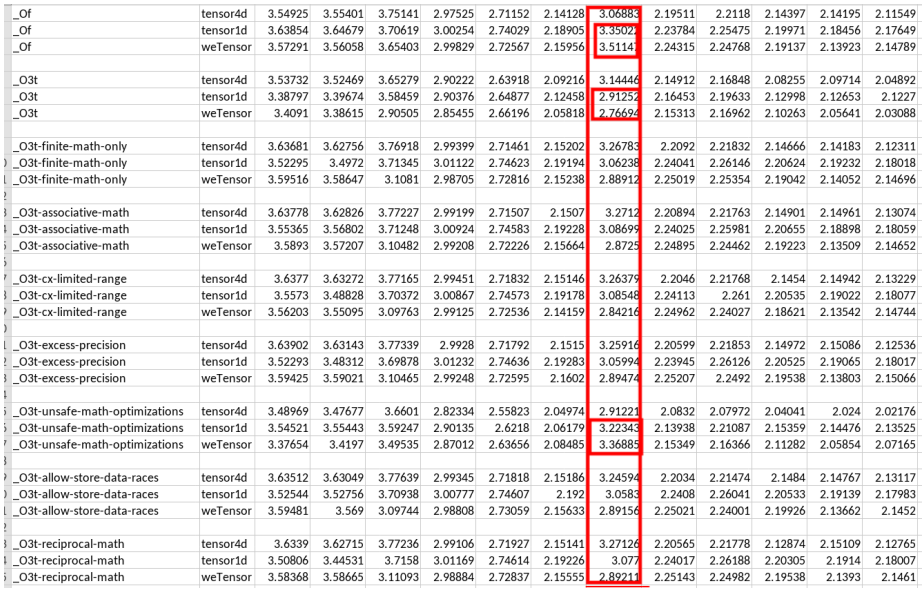
Figure 14: 对比O3用fno关闭-fipa-cp-clone

6.1 Analysis

前几周的结论是：-fipa-cp-clone这个优化是让调用函数的时候不传入常量，在调用函数传常量的时候拷贝函数实现一个不传常量的版本。我们的代码调用直接卷积函数传入的是带const，所以直接卷积函数被拷贝了，这样就和前几周写的直接卷积函数是在类里面情况接近了。但wetensor使用了libtorch里的数据结构，该数据结构不允许设置为const，所以理论wetensor是没享受到这个优化的。

7 Experiment5

这周测试了Ofast比O3多开的优化。gcc 12.3编译器。O3比Ofast多开了math-errno,semantic-interposition,signed-zeros,trapping-math。Ofast比O3多开了allow-store-data-races,associative-math,cx-limited-range,finite-math-only,reciprocal-math,unsafe-math-optimizations 其中如果开了signed-zeros或者trapping-math就没法开associative-math 我在O3的基础上把O3比Ofast多开了的优化选项用fno关掉，然后，逐个测试开启Ofast比O3多开的优化选项的效果



_Of	tensor4d	3.54925	3.55401	3.75141	2.97525	2.71152	2.14128	3.06883	2.19511	2.2118	2.14397	2.14195	2.11549
_Of	tensor1d	3.63854	3.64679	3.70619	3.00254	2.74029	2.18905	3.35021	2.23784	2.25475	2.19971	2.18456	2.17649
_Of	weTensor	3.57291	3.56058	3.65403	2.99829	2.72567	2.15956	3.5114	2.24315	2.24768	2.19137	2.13923	2.14789
_O3t	tensor4d	3.53732	3.52469	3.65279	2.90222	2.63918	2.09216	3.14446	2.14912	2.16848	2.08255	2.09714	2.04892
_O3t	tensor1d	3.38797	3.39674	3.58459	2.90376	2.64877	2.12458	2.91252	2.16453	2.19633	2.12998	2.12653	2.1227
_O3t	weTensor	3.4091	3.38615	2.90505	2.85455	2.66196	2.05818	2.76694	2.15313	2.16962	2.10263	2.05641	2.03088
_O3t-finite-math-only	tensor4d	3.63681	3.62756	3.76918	2.99399	2.71461	2.15202	3.26783	2.2092	2.21832	2.14666	2.14183	2.12311
_O3t-finite-math-only	tensor1d	3.52295	3.4972	3.71345	3.01122	2.74623	2.19194	3.06238	2.24041	2.26146	2.20624	2.19232	2.18018
_O3t-finite-math-only	weTensor	3.59516	3.58647	3.1081	2.98705	2.72816	2.15238	2.88912	2.25019	2.25354	2.19042	2.14052	2.14696
_O3t-associative-math	tensor4d	3.63778	3.62826	3.77227	2.99199	2.71507	2.1507	3.2712	2.20894	2.21763	2.14901	2.14961	2.13074
_O3t-associative-math	tensor1d	3.55365	3.56802	3.71248	3.00924	2.74583	2.19228	3.08699	2.24025	2.25981	2.20655	2.18898	2.18059
_O3t-associative-math	weTensor	3.5893	3.57207	3.10482	2.99208	2.72226	2.15664	2.8725	2.24895	2.24462	2.19223	2.13509	2.14652
_O3t-cx-limited-range	tensor4d	3.6377	3.63272	3.77165	2.99451	2.71832	2.15146	3.26379	2.2046	2.21768	2.1454	2.14942	2.13229
_O3t-cx-limited-range	tensor1d	3.5573	3.48828	3.70372	3.00867	2.74573	2.19178	3.08548	2.24113	2.261	2.20535	2.19022	2.18077
_O3t-cx-limited-range	weTensor	3.56203	3.55095	3.09763	2.99125	2.72536	2.14159	2.84216	2.24962	2.24027	2.18621	2.13542	2.14744
_O3t-excess-precision	tensor4d	3.63902	3.63143	3.77339	2.9928	2.71792	2.1515	3.25916	2.20599	2.21853	2.14972	2.15086	2.12536
_O3t-excess-precision	tensor1d	3.52293	3.48312	3.69878	3.01232	2.74636	2.19283	3.05994	2.23945	2.26126	2.20525	2.19065	2.18017
_O3t-excess-precision	weTensor	3.59425	3.59021	3.10465	2.99248	2.72595	2.1602	2.89474	2.25207	2.2492	2.19538	2.13803	2.15066
_O3t-unsafe-math-optimizations	tensor4d	3.48969	3.47677	3.6601	2.82334	2.55823	2.04974	2.91221	2.0832	2.07972	2.04041	2.024	2.02176
_O3t-unsafe-math-optimizations	tensor1d	3.54521	3.55443	3.59247	2.90135	2.6218	2.06179	3.22343	2.13938	2.21087	2.15359	2.14476	2.13525
_O3t-unsafe-math-optimizations	weTensor	3.37654	3.4197	3.49535	2.87012	2.63656	2.08485	3.36885	2.15349	2.16366	2.11282	2.05854	2.07165
_O3t-allow-store-data-races	tensor4d	3.63512	3.63049	3.77639	2.99345	2.71818	2.15186	3.24594	2.2034	2.21474	2.1484	2.14767	2.13117
_O3t-allow-store-data-races	tensor1d	3.52544	3.52756	3.70938	3.00777	2.74607	2.192	3.0583	2.2408	2.26041	2.20533	2.19139	2.17983
_O3t-allow-store-data-races	weTensor	3.59481	3.569	3.09744	2.98808	2.73059	2.15633	2.89156	2.25021	2.24001	2.19926	2.13662	2.1452
_O3t-reciprocal-math	tensor4d	3.6339	3.62715	3.77236	2.99106	2.71927	2.15141	3.27126	2.20565	2.21778	2.12874	2.15109	2.12765
_O3t-reciprocal-math	tensor1d	3.50806	3.44531	3.7158	3.01169	2.74614	2.19226	3.077	2.24017	2.26188	2.20305	2.1914	2.18007
_O3t-reciprocal-math	weTensor	3.58368	3.58665	3.11093	2.98884	2.72837	2.15555	2.89211	2.25143	2.24982	2.19538	2.1393	2.1461

Figure 15: Enter Caption

从表格可以看出开启O3 -funsafe-math-optimizations比较接近Ofast 然后测试了fast逐个关掉Ofast比O3多开的优化选项。但是fno有点问题，用fno关掉unsafe-math-optimizations这个选项gflops没明显的变化（没O3 -funsafe-math-optimizations比O3变化大）。然后我在O3的基础上把Ofast开的除了-funsafe-math-optimizations的优化选项，结果比用fno关掉unsafe-math-optimizations明显

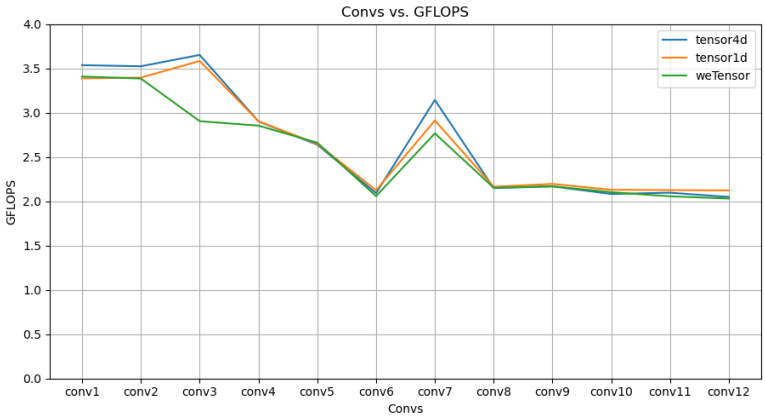


Figure 16: O3 关掉比Ofast多开的优化

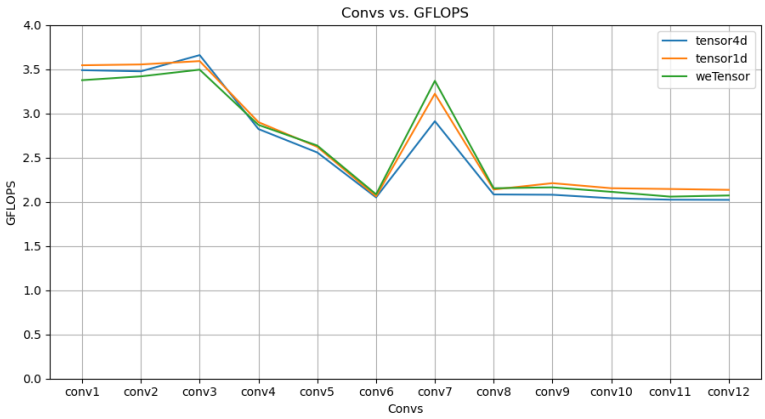


Figure 17: O3 关掉比Ofast多开的优化然后开unsafe-math-optimizations

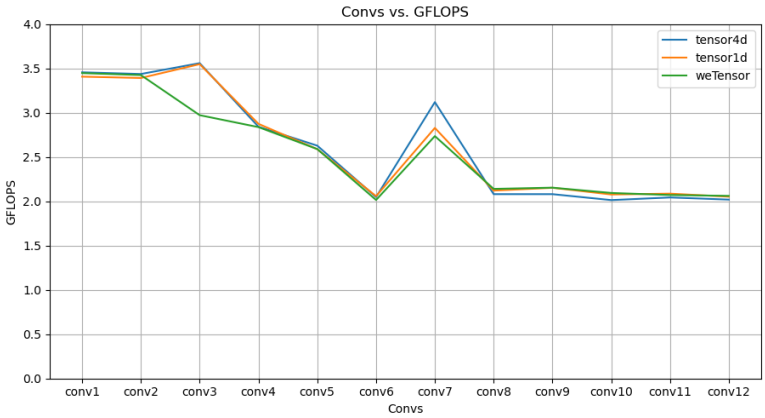


Figure 18: O3 关掉比Ofast多开的优化，然后开Ofast比O3多开的优化，除了unsafe-math-optimizations

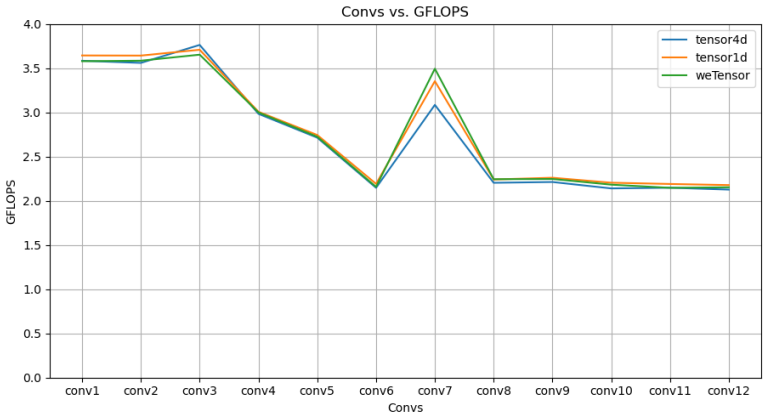


Figure 19: Ofast -fno-unsafe-math-optimizations

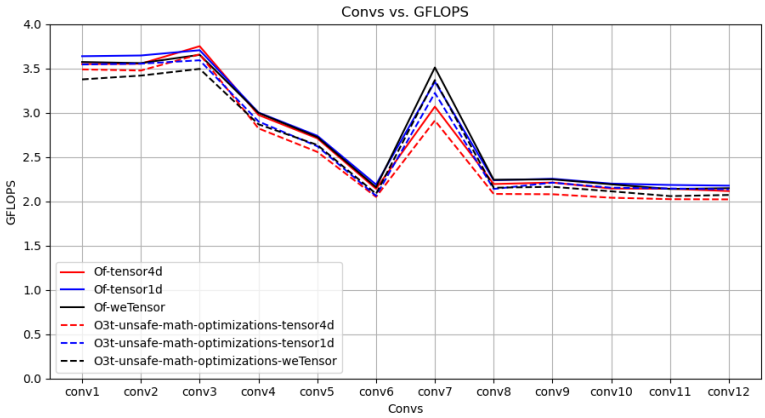


Figure 20: O3 关掉比Ofast多开的优化，然后开Ofast比O3多开的优化，除了unsafe-math-optimizations对比Ofast

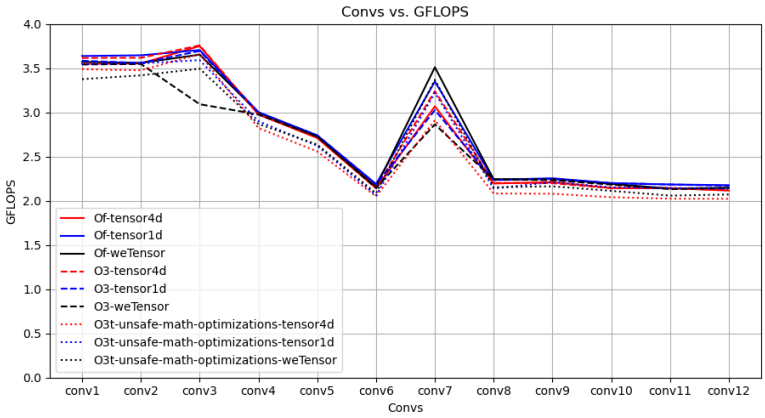


Figure 21: O3 -funsafe-math-optimizations对比O3和Ofast

7.1 Analysis

`-funsafe-math-optimizations`的官方解释是Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. When used at link time, it may include libraries or startup files that change the default FPU control word or other similar optimizations. This option is not turned on by any -O option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications. Enables -fno-signed-zeros, -fno-trapping-math, -fassociative-math and -freciprocal-math. The default is -fno-unsafe-math-optimizations. 允许对浮点运算进行优化(a)假设参数和结果是有效的，(b)可能违反IEEE或ANSI标准。当在链接时使用时，它可能包含更改默认FPU控制字或其他类似优化的库或启动文件。此选项不能通过任何-O选项打开，因为对于依赖于IEEE或ISO规则/数学函数规范的精确实现的程序，它可能导致不正确的输出。然而，对于不需要这些规范保证的程序，它可能产生更快的代码。启用-fno-signed-zero、-fno-trapping-math、-fassociated -math和-freciprocal-math。默认值是-fno-unsafe-math-optimizations。

8 Experiment6

我把机器的roofline画出来了（ubuntu原生下）

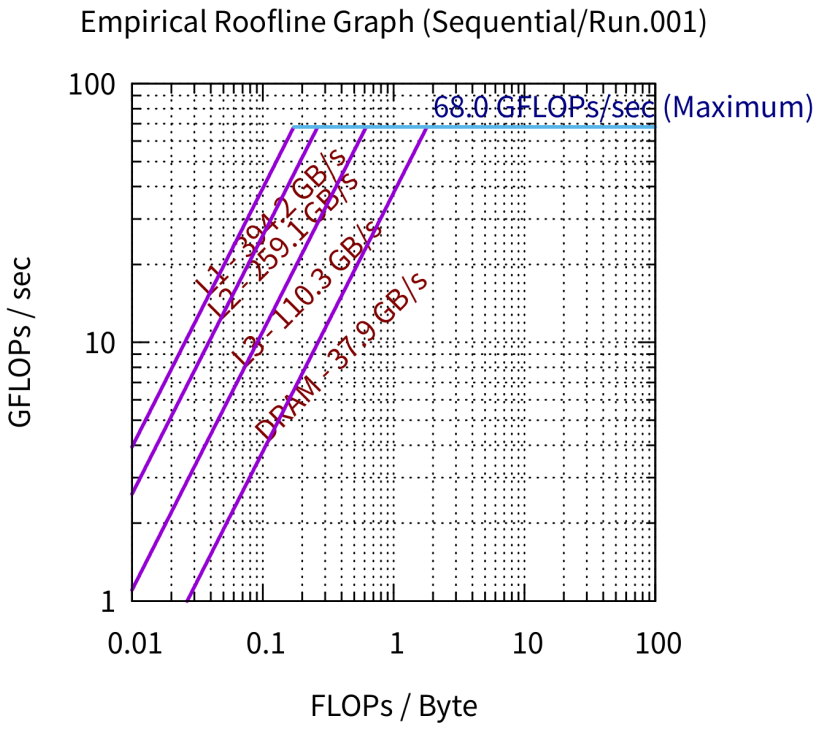

Empirical Roofline Graph (Sequential/Run.001)
100
10
1
0.01
0.1
1
10
100
GFLOPs / sec
FLOPs / Byte
68.0 GFLOPs/sec (Maximum)
L1 - 394.2 GB/s
L2 - 259.1 GB/s
L3 - 110.3 GB/s
DRAM - 37.9 GB/s

Figure 22: 单核

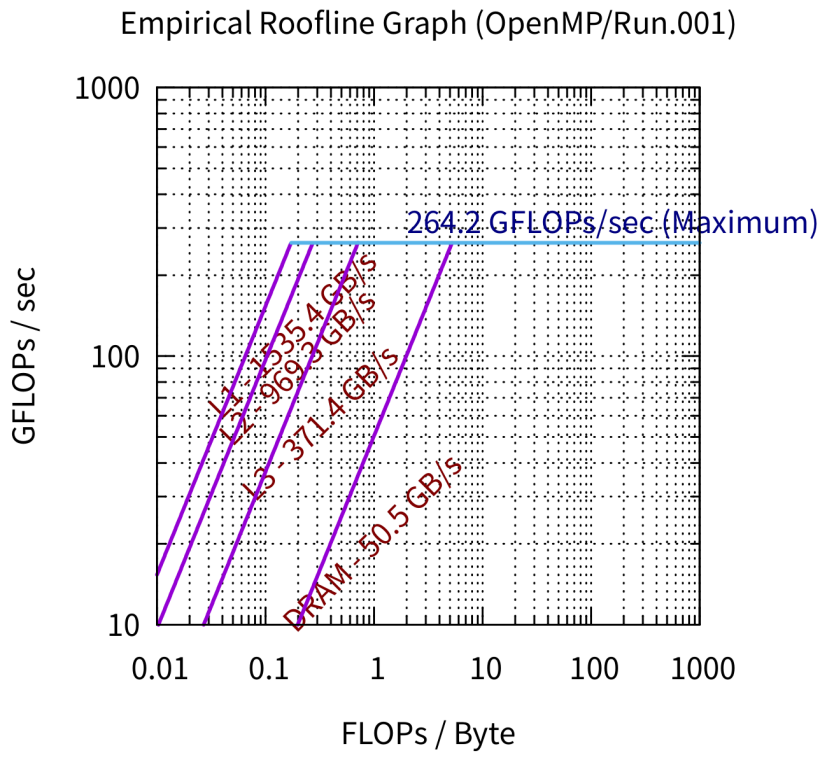


Figure 23: 多线程

但直接卷积的算数强度还不会算，于是测了几个矩阵乘法的点。

```
void multiply(const Matrix& A,const Matrix& B,Matrix&
    C){
     for(size_t i=0;i<C.num_rows();++i){
          for(size_t j=0;j<C.num_cows();++j){
               for(size_t k=0;k<A.num_cows();++k){
                    C(i,j)+=A(i,k)*B(k,j);
               }
          }
     }
}
```

N = 16 gflops = 0.141241 N = 1024 gflops = 0.118537

```
void multiply_hoisting(const Matrix& A,const Matrix& B
    ,Matrix& C){
     for(size_t i=0;i<C.num_rows();++i){
          for(size_t j=0;j<C.num_cows();++j){
               double t = C(i,j);
               for(size_t k=0;k<A.num_cows();++k){
                    t+=A(i,k)*B(k,j);
               }
               C(i,j) = t;
          }
     }
}
```

N = 1024 gflops = 0.155996

```
void multiply_tiled4x4(const Matrix& A,const Matrix& B
    ,Matrix& C){
     for(size_t i=0;i<C.num_rows();i += 4){
          for(size_t j=0;j<C.num_cows();j += 4){
               for(size_t k=0;k<A.num_cows();++k){
                    C(i,j)+=A(i,k)*B(k,j);
                    C(i,j+1)+= A(i,k)*B(k,j+1);
                    C(i,j+2)+=A(i,k)*B(k,j+2);
                    C(i,j+3)+= A(i,k)*B(k,j+3);
                    C(i+1,j)+=A(i+1,k)*B(k,j);
                    C(i+1,j+1)+=A(i+1,k)*B(k,j+1);
                    C(i+1,j+2)+=A(i+1,k)*B(k,j+2);
                    C(i+1,j+3)+=A(i+1,k)*B(k,j+3);
                    C(i+2,j)+=A(i+2,k)*B(k,j);
                    C(i+2,j+1)+= A(i+2,k)*B(k,j+1);
                    C(i+2,j+2)+=A(i+2,k)*B(k,j+2);
                    C(i+2,j+3)+= A(i+2,k)*B(k,j+3);
```

```
                    C(i+3,j)+=A(i+3,k)*B(k,j);
                    C(i+3,j+1)+= A(i+3,k)*B(k,j+1);
                    C(i+3,j+2)+=A(i+3,k)*B(k,j+2);
                    C(i+3,j+3)+= A(i+3,k)*B(k,j+3);
                }
            }
        }
}
```

N = 1024 gflops = 0.140642

不知道怎么画上去