week-6

JX-Ma

2023/10/26

1 实验环境

- cpu:AMD Ryzen 7 6800H with Radeon Graphics
- 操作系统: win11
- 编译器:

2 算法的正确性

2.1 四维卷积验证

- 步长 s = 1, padding = 0
- 输入张量维度:1 * 1 * 3 * 3
- 卷积核维度: 2 * 1 * 2 * 2
- 输出张量维度: 1 * 2 * 2 * 2

测试结果如图 **1:** 第一个矩阵为输入的张量, 第二第三个为卷积核的矩阵，最后两个为输出张量。

2.2 一维卷积验证

- **步长 s = 1, padding = 0**
- **输入张量维度:1 * 1 * 3 * 3**
- **卷积核维度: 2 * 1 * 2 * 2**
- **输出张量维度: 1 * 2 * 2 * 2**

测试结果如图 **2:** 第一个矩阵为输入的张量, 第二第三个为卷积核的矩阵，最后两个为输出张量。

2.3 四维张量和一维张量内存元素存放地址对比

- 输入张量维度:5 * 5 * 5 * 5

具体效果有图 3 所示这里输入张量的类型为 int,4 个字节. 可以看到使用一维张量存储数据:
A[0][0][0][0] 与 A[0][0][1][0] 相差 20 个字节,5 个元素
A[0][1][0][0] 与 A[0][0][0][0] 相差 100 个字节,25 个元素
A[1][0][0][0] 与 A[0][0][0][0] 相差 500 字节,125 个元素
得到使用一维张量存储四维数组, 元素之间是连续存放的。
四维张量存储数据:
A[0][0][1][0] 与 A[0][0][0][0] 相差 240,48 个元素
得到使用四维张量存储四维数组, 元素之间并不是连续存放.

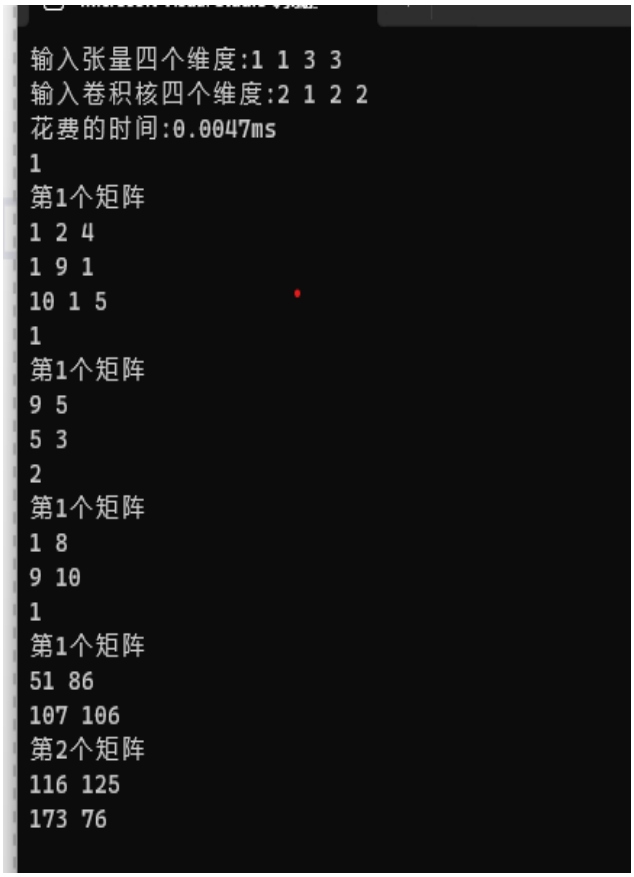

图 1: 四维卷积结果

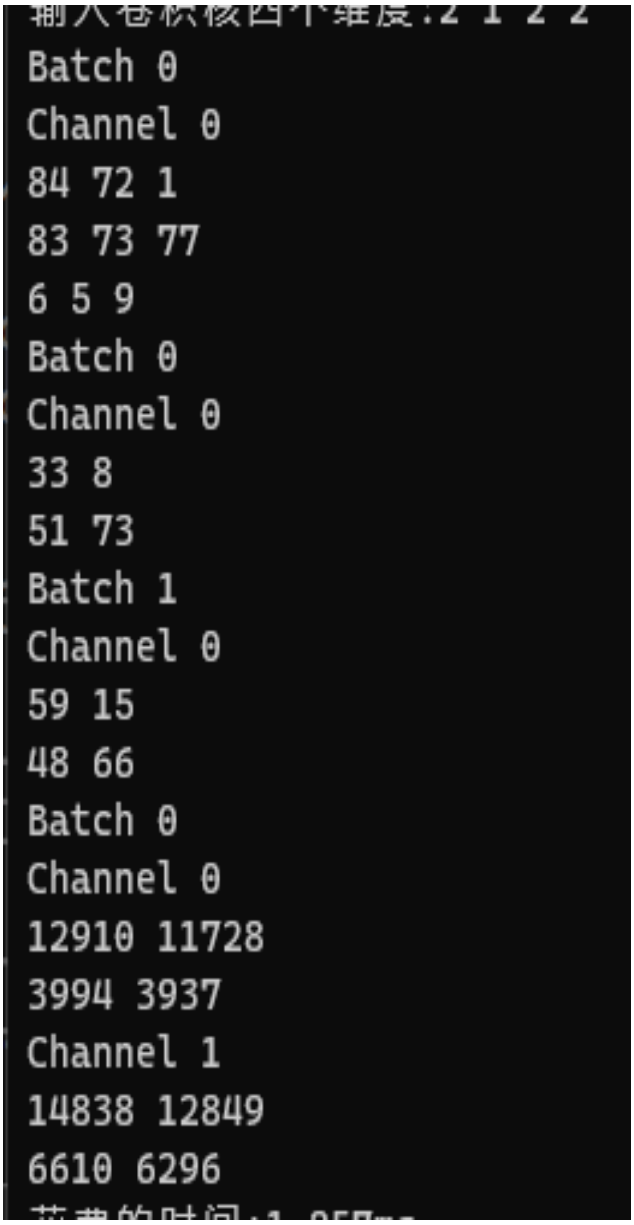

图 2: 一维卷积结果

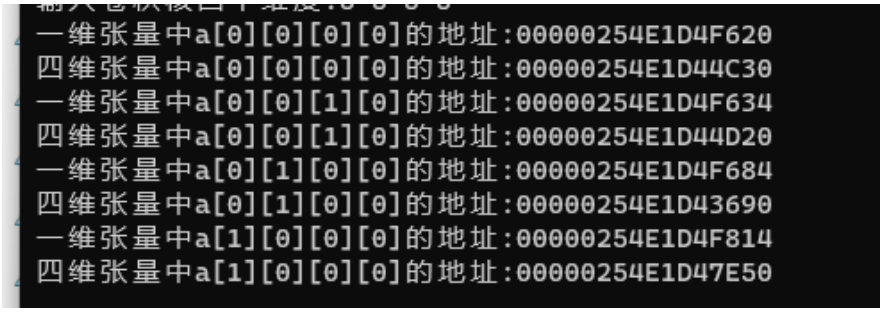

图 3: 四维和一维张量的地址比较

3 实验数据准备

数据准备

- **步长 s = 0; padding = 0;**
- **输入张量: 10 3 100 100**
- **卷积核: 5 3 3 3**

总共测 8 组数据, 前四组数据保持前三个维度不变输入张量第四维度 +10, 后四组数据保持前两个维度不变, 后四个维度 +50;
时间单位:ms

表 1: 测试结果表

数据	1	2	3	4	5	6	7	8
运算次数	5419584	11441344	17463104	23484864	53936064	96867264	152278464	220169664
4 维运行时 (O0)	412.725	799.642	1199.63	1664.93	3757.94	6615.48	11205.3	16388.5
4 维 Gfloat(O0)	0.013131	0.014308	0.014557	0.014106	0.014353	0.014643	0.013590	0.013434
1 维运行时 (O0)	192.099	384.152	589.463	767.606	1781.13	3253.77	4877.65	7272.47
1 维 Gfloat(O0)	0.028212	0.029783	0.029625	0.030595	0.030282	0.029771	0.031220	0.030274
4 维运行时 (O2)	66.3884	136.255	199.829	271.344	617.711	1101.8	1736.38	2604.45
4 维 Gfloat(O2)	0.081635	0.083970	0.087390	0.086550	0.087316	0.087917	0.087699	0.084536
1 维运行时 (O2)	9.7088	18.8681	27.987	40.0143	89.1661	169.097	253.8	381.207
1 维 Gfloat(O2)	0.5582	0.6064	0.6240	0.5869	0.6049	0.5729	0.6000	0.5776
4 维运行时 (O3)	11.1724	24.6142	34.9645	48.0855	104.947	182.595	285.515	400.06
4 维 Gfloat(O3)	0.4851	0.4648	0.4995	0.4884	0.5139	0.5305	0.5333	0.5503
1 维运行时 (O3)	10.4307	20.1631	31.4405	42.3565	96.1266	177.713	266.476	391.216
1 维 Gfloat(O3)	0.5196	0.5674	0.5554	0.5545	0.5611	0.5451	0.5715	0.5628
4 维运行时 (Ofast)	11.3358	21.8099	33.3479	46.2708	109.593	185.783	288.044	418.313
4 维 Gfloat(Ofast)	0.4781	0.5246	0.5237	0.5076	0.4921	0.5214	0.5287	0.5263
1 维运行时 (Ofast)	10.2253	21.0229	30.4687	41.124	102.986	170.475	281.86	409.296
1 维 Gfloat(Ofast)	0.5300	0.5442	0.5731	0.5711	0.5237	0.5682	0.5403	0.5379

4 实验结果

4.1 Runtime 比较

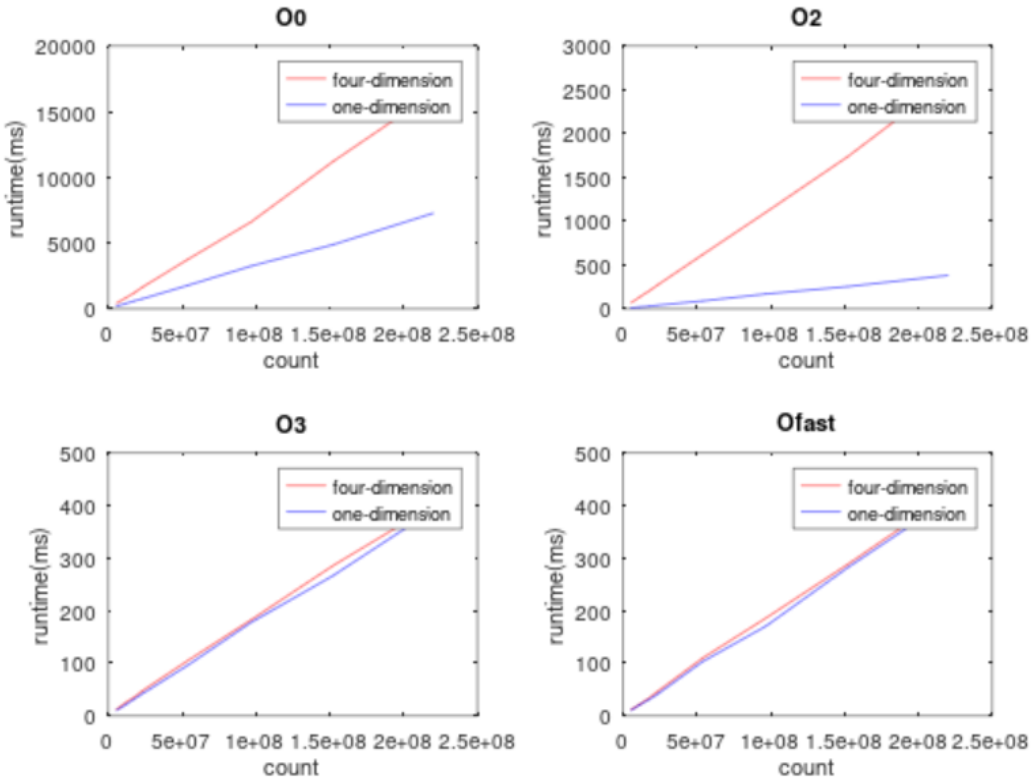

图 4: Runtime

4.2 Gfloat 比较图

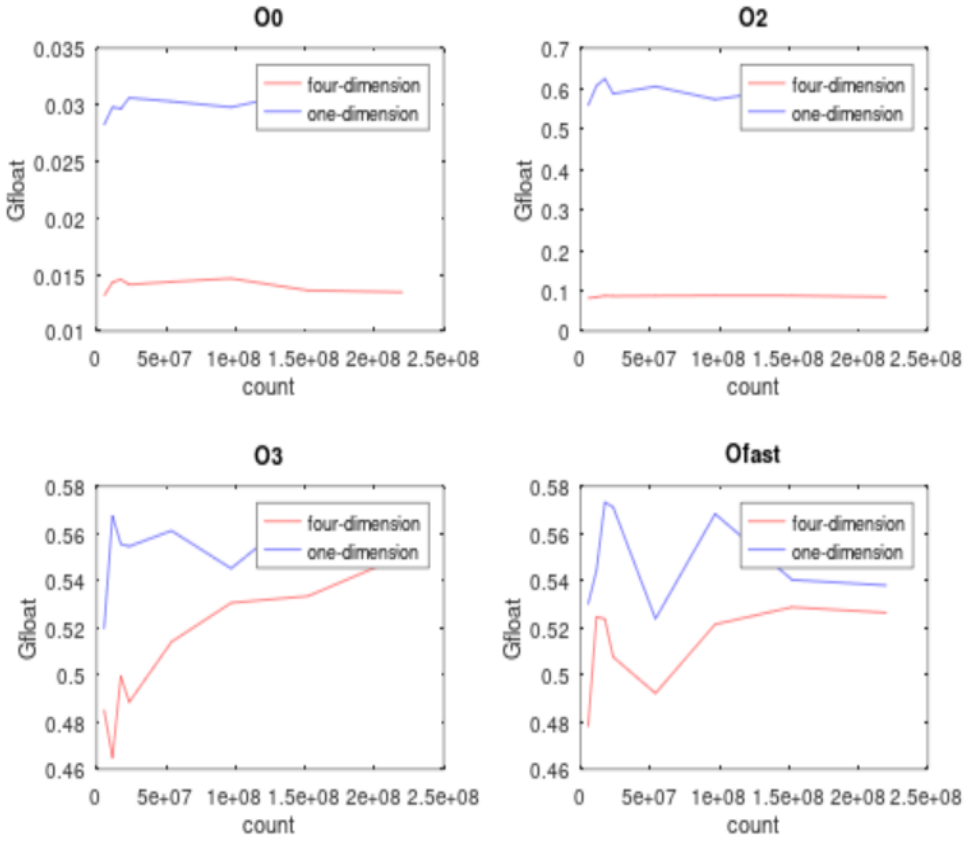

图 5: Gfloat

4.3 一维张量在不同优化后的 Runtime 和 Gfloat 比较

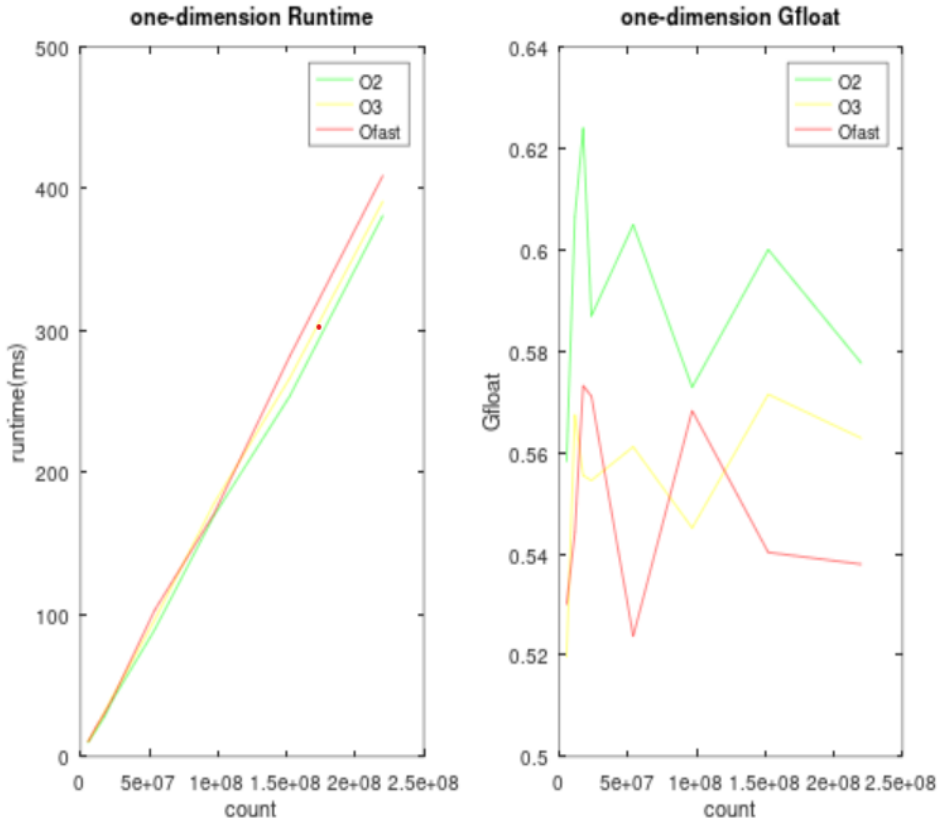

图 6: Runtime and Gfloat

5 实验总结

从实验结果中可以看出, 无论有没有优化,1 维张量的运行时比 4 维张量的快, 并且 Gfloat 也相应的多, 从图六可以看出 O3 和 Ofast 优化后的性能都不如 O2 优化后的性能, 在从前面图中看出在 O2 时 4 维和 1 维卷积性能相差很大, 而 O3 优化和 Ofast 优化两者性能相差很小.

优化等级:

O2 会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间。

O3 在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。

-Ofast 会开启所有-O3 的编译开关，且会对不符合标准的程序进行优化。

从网上查到 O3 和 Ofast 可能比 O2 的原因:

1.O3 和 Ofast 级别进行了更多的优化，包括了一些更加复杂的优化技术。这些优化技术可能会增加编译器的工作量，导致编译时间增加。

2.O3 和 Ofast 级别的优化会更加注重代码的性能，而不是编译时间。因此，编译器可能会花更多的

时间来寻找更优的优化方案，以获得更好的性能。这也导致了编译时间的增加。