week-15

JX-Ma

2023/12/30

1 本周工作

本周主要做了将前几周的所学习的优化应用到第一个 benchmark 上，将不同的优化后的算法画在 roofline 上。

2 实验环境

- 系统: Ubantu 22.01
- gcc version : 9.5.0
- 优化选项: -O3
- cpu:AMD Ryzen 7 6800H 3.20GHz
- inputTensor: 10,3,227,227
- filterTensor: 96,3,11,11
- outputTensor: 10,96,55,55
- stride : 4

3 实验部分

3.1 optnone-optnone1

将 Tensor1D 换成了上周张新鹏提出来的 Tensor1dv2
因为 Tensor1D 继承 Tensor, 调用里面的方法会增加一次寻址

3.2 optnone1-hoist-index

对索引做了 hoist 优化，减少了索引的计算次数

gflops 比较

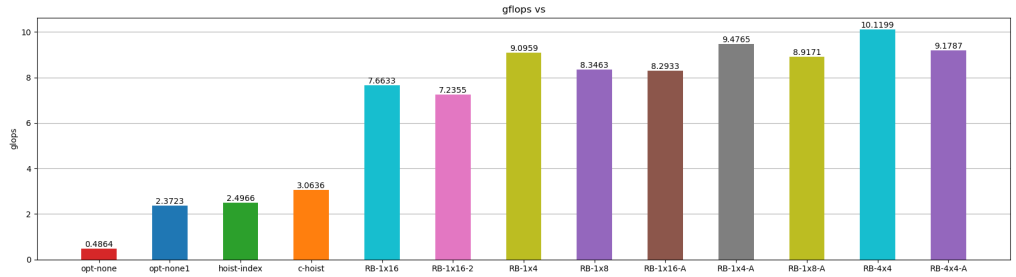

图 1: gflops

3.3 hoist-index - c-hoist

对 c 的每一个元素元素做了 hoist 优化，减少了直接从内存中读取 c 的次数 (c 是输出张量的每个元素)

3.4 c-hoist-RB

对 C 的元素进行分块，1x4 代表对输出张量的宽进行 4 分块，输出张量的宽对于分块大小不能整除，余下的部分按照正常的 hoist 计算。
4x4 代表对输出张量宽和高按照 4x4 分块。后面加了 A 的代表使用了 simd 指令集。
RB-1x16-2 这一步我是先将宽按照 16 分块剩下的部分如果可以按照 8 分块则继续分块，余下的部分能按照 4 分块则继续，最后正常的算一个的值。

4 roofline 分析不同的分块性能

roofline 中点所对应的颜色对应 bar 图中柱状图对应的颜色
从红色的点到蓝色的点是 (index-hoist - c-hoist), 红色点所在的位置为 bandwidth-bound, 这个时候可以通过适当的增加算术强度来提高性能。
对于分块 1x16 1x4 1x8，可以看到在 1x4 的时候表现很好，增加分块的大小即增加算术强度反而还使得性能降低，对于相同的算术强度，1x16 和 4x4 来说，对输出张量的高分块能够获得更高的性能。原因可能是输出张量其他维度相同时，读取输出张量不同高的代价比较高。

5 实验分析

1. 首先对于算术强度的计算，我是计算一个 for 循环内所有浮点次数与读取数据字节数的比值，也就是算术强度的下限，对于分块的算术强度计算，只计算分块内的算数强度，也就是取余部分的不计。
2. 对于分块算法来说，首先考虑一下分块的好处，分块可以增大算术强度，如果数据密集的话，cache 命中率高，就可以在更短的时间内进行更多的运算，但是分块需要考虑到步长的问题，如果步长存在，我们需要读取的输入张量是相邻为 s* 数据类型大小的数据，这个时候的 cache 命中率也可能会降低从而造成负优化的结果。

gflops 比较

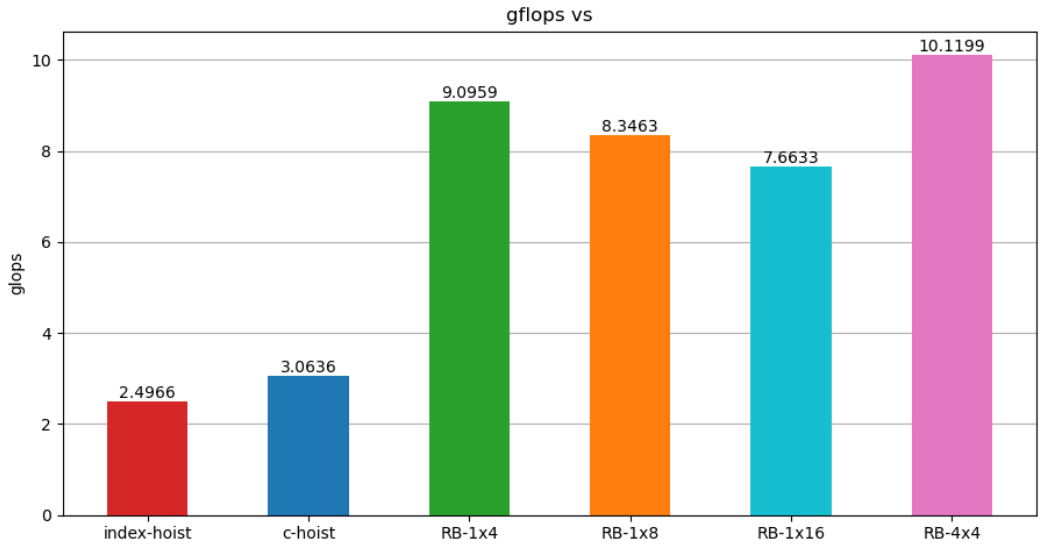

图 2: gflops

3. 对于分块后使用指令集优化效果不大的问题，首先由于步长的存在，再一次分块计算内我需要先读取分块大小个相邻为 s 的元素放到一个数组中，然后再去加载它，这就需要额外的消耗，我需要读取分块大小个地址的元素，如果步长为 1，我只需要读取首地址一次就行。

4. 对于分块 1x16 和 4x4，我认为主要是输出张量的 height 的权重比较高，[0,0,0,0] 和 [0,0,0,1] 相邻，而 [0,0,1,0] 和 [0,0,0,0] 相差 C.width 个元素。

5. 对于分块 1x16，我提出了另一种卷积方法，就是对 c 的宽整除后剩下的部分对 8 整除，然后对 4 整除，最后在直接卷积，这样会多增加 2 个判断条件，而剩下的循环最多只会循环一个，因为 cpu 会提前去执行下一个指令，当遇到判断的时候，他会选择它猜测的指令执行，如果猜测错误的话，则会重新撤回之前执行的指令。

gflops 比较

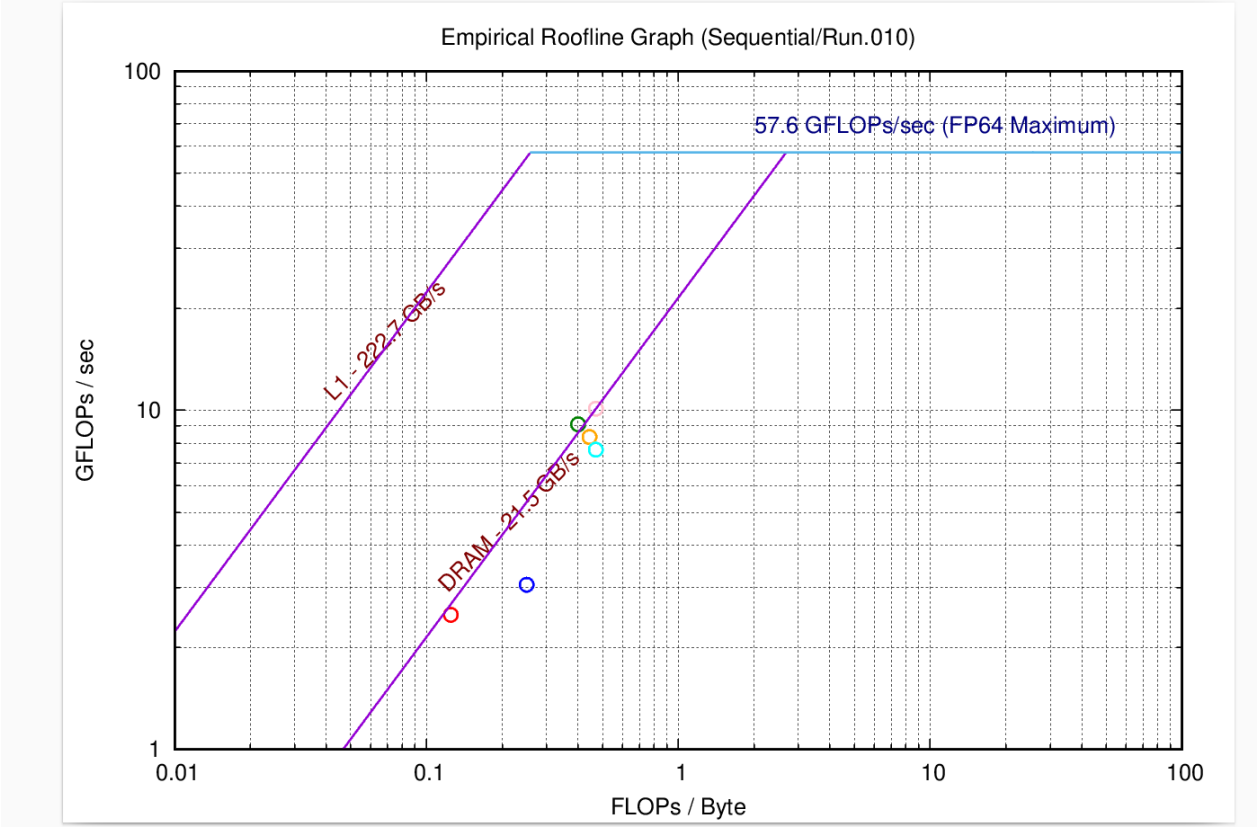

图 3: gflops