week-7

JX-Ma

2024/4/13

1 本周工作

1. 上周实验太慢了因为测试方法的问题，本周比较了上周的测试方法和这周的测试方法的内存分析，在比较前使用 valgrind 工具分别测试了两种方法是否存在内存泄漏，结果是都没出现内存泄漏的问题。2. 比较了在 28 个线程下，正常跑，使用一个 numa 节点和静态分配进程的 gflops 和内存分析。3. 将输出张量的 channel 和 batch 合并，在使用 openmp，测试了这种方法的 gflops 和内存分析。

2 实验部分

2.1 实验环境

- 系统: CentOS7
- gcc version : 13.2.0
- 优化选项: -O3 -fopenmp -avx2 -fmadd
- cpu: Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz

2.2 实验一

从内存分析, 文件 memory.xlsx 中可以看到，上周使用测试方法的 Minor_page_faults 要远远大于这周测试方法。这里需要注意，因为我们的实验是在一个程序内跑多次取最优，所以表中参数 gflops 是最优的 gflops，然后后面参数都是跑了多次求和。

2.3 实验二

测试了在 28 个线程下使用一个 numa 节点测试的 gflops，和静态分配线程的 gflops，静态线程方法是，我对输出张量的 channel 维度进行取余操作，能被最大线程数整除的就设置线程数大小为最大线程数，把取余后的大小作为剩下的分配的线程数量。从内存分析表格 memory_t28.xlsx 中看到，Minor_page_faults，Voluntary_contex_swh，InVoluntary_contex_swh，在静态分配线程上这三个参数的值都要远大于前两种，这里测试的内存是跑一次的，就是在程序里面跑一次，跑这个程序多次，将最好的性能内存分析数据记录到表格中，这里可能可能因为只跑了 20 次所以和测试的 gflops 相差挺大的.

图 1: diffrence-1

不同点 1：上周函数调用流程多一步

2.4 服务器 numa 的环境以及内存分配策略

```
numactl —hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
             16 17 18 19 20 21 22 23 24 25 26 27
node 0 size: 130723 MB
node 0 free: 117995 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39
          40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
node 1 size: 131072 MB
node 1 free: 117399 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

numa内存分配策略

policy: **default**

```
preferred node: current
```

图 2: diffrence-2
不同点 2：for 循环内不一样

```
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
             17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
             33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
              49 50 51 52 53 54 55
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

服务器上含有 2 个 numa 节点，每个节点包含 28 个 cpus，之前 cpu 信息看到，每个 cpus 汇总包含一个核心，每个核心中有一个 thread。numa 的内存大小为 130G。node distance 代表节点间的距离，节点的距离代表不同节点之间的访问成本

2.5 实验三

根据张新鹏的建议，把输出张量的 batch 和 channel 结合成一层循环，然后测试结果，结果表明和成一层的效果更好。
从内存分析表 memory_t56 中可以看出直接卷积的 cache_miss 要远小于 im2win 的，然后这里有一点我不明白的是那个 cpu 使用率为什么百分比越低算法的性能越好。

3 总结

这周测试直接卷积和 im2win 的内存分析找到了直接卷积快的原因，这里猜测还是因为 im2win 的窗口之间的距离太大导致 cache_miss 过多, 应该适当的降低按照通道 flow 的大小。

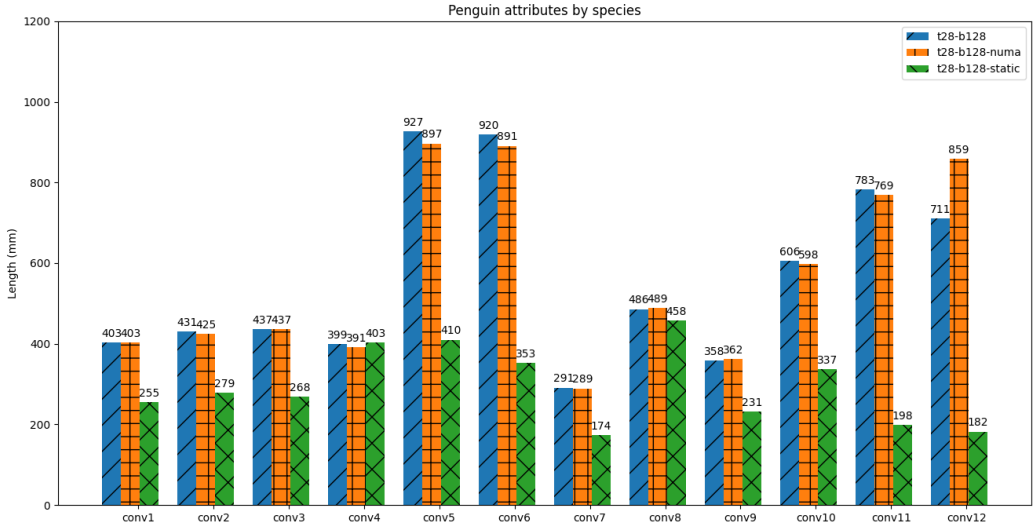

图 3: t28-gflops

从图中看出，正常情况分配和把 28 个线程分配到 numa 上性能差不多，静态分配的性能就差一点。

old

```
for (size_t i = 0; i < output_batch; ++i){
    size_t i_o_cwh = i*o_cwh;
    size_t i_i_cwh = i*i_whhc;
    #ifdef omp_flag
    #pragma omp parallel for schedule(omp_flag)
    #endif
    for (size_t j = 0; j < output_channel; ++j) {
        size_t j_o_wh = i_o_cwh + j*o_wh;
        size_t j_f = j*f_cwh;
        for (size_t m = 0; m < output_height; ++m){
            size_t m_o_w = j_o_wh + m*output_width;
            size_t m_i_whc = i_i_cwh + m*i_whc;
            float *outptrt = outptr + m_o_w;
```

new

```
#ifdef omp_flag
#pragma omp parallel for schedule(omp_flag)
#endif
for(size_t ij = 0 ;ij < output_bc;++ij){
    i = ij/output_channel;
    j = ij%output_channel;
    size_t i_o_cwh = i*o_cwh;
    size_t i_i_cwh = i*i_whhc;
    size_t j_o_wh = i_o_cwh + j*o_wh;
    size_t j_f = j*f_cwh;
        for (size_t m = 0; m < output_height; ++m){
            size_t m_o_w = j_o_wh + m*output_width;
            size_t m_i_whc = i_i_cwh + m*i_whc;
            float *outptrt = outptr + m_o_w;
            for (size_t n = 0; n < output_width/11; ++n){
```

图 4: b+c

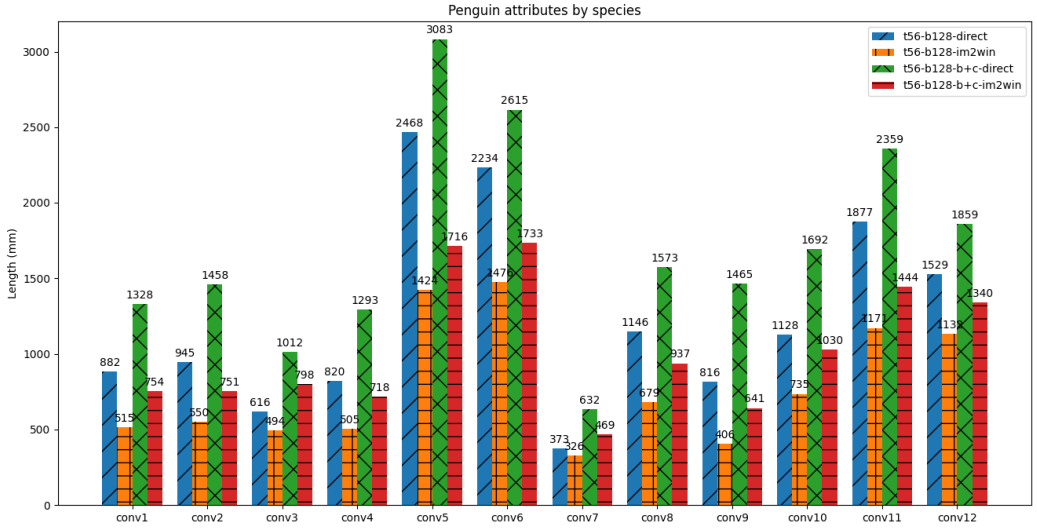

图 5: gflops

从实验结果上来看直接卷积要比 im2win 好很多，可能是因为 im2win 展开后的步长太大了。例如通道 512，卷积步长为 1，卷积核高为 3，那么对于 im2win 来说，输出张量窗口之间相差 1536 个元素。而直接卷积只有 512 个元素。

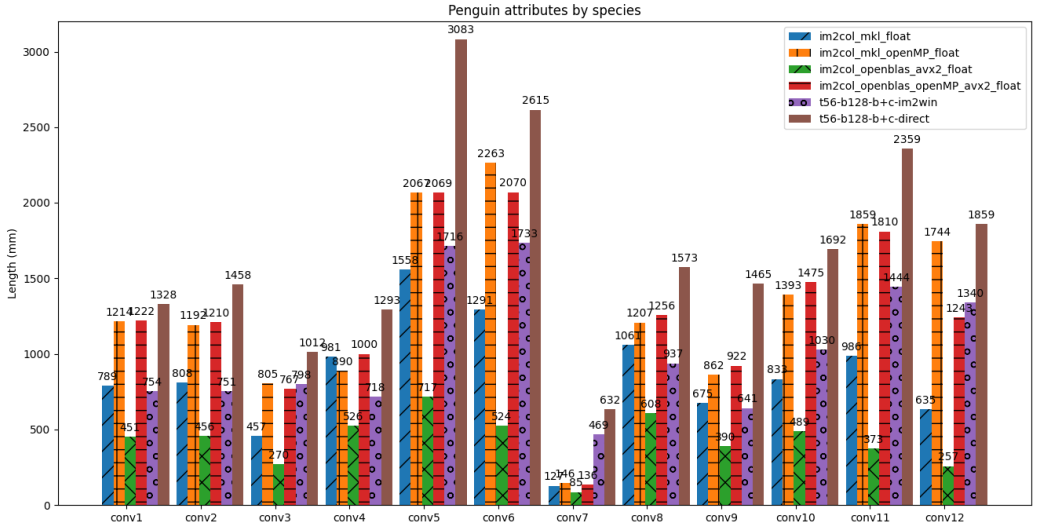

图 6: gflops