Lab2

3240104875

王耀

2025/7/5

任务一

AVX 部分，看到优化提示意识到，转置 32bit 后一组是四个 uint8_t。但是这么读进去就算是转化成 uint64_t 也不过 256 个 bit。我去求助 GPT，但是它给我回了一堆有点胡言乱语的东西。

根据 lab2 的优化提示，如果要对那样的 for 循环进行向量化，最直接的想法是一次读 4 个运算 4 个，但是这显然太浪费读取时间，而且也没有这么小的连续地址类型，能实现的函数又太大材小用，不合适。

那我多读一点，进行分块矩阵运算？可是 AVX 似乎只是向量化工具，而非分块矩阵工具，或许可以成为一个思路。

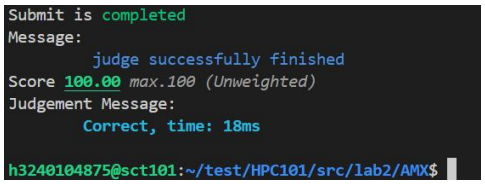
为什么要用 B_reshape 呢？这不是限制吗？如果用 B，可以一次读入更多的数据，_mm512_dpbusd_epi32 可以直接对 64 个 8bit 数进行分组内积，只要求和就能得到大量的内积片段，似乎比用 B_reshape 的效率高多了。

难道是我用 B_reshape 的方法不对？还是先着手于 B_reshape 的应用，但是考虑如何降低读取次数还能计算乘法想了很久。现决定暴力优化，硬读硬算，然后成功得到了 1.6s 的好成绩，优化系数是小于 0.1 的，有点幽默。

我尝试减少 A 读取的次数，这次实现了 0.69 的速度，算是加速了，但是不多。

我抛弃了 B_reshape，优化到 0.09 了，1.57619 倍。核心是_mm512_dpbusd_epi32 和 _mm512_reduce_add_epi32，这是很好的，但是我感觉还能再优化，明显的 A 和 B 的读取次数还没有压到最小，寄存器还能利用。

AMX 部分，我几乎就按照样例写了一遍，就是用错了 dpbusd 和 dpbsud，导致了一个错误。AMX 加速了 166.676 倍，用时 0.0208975。这个数据很震撼我。可见我的 AVX 还要再摸索一下提升性能。快乐 AMX。



接下来我继续优化 AVX。目前的思路是减少 A 或 B 的读取次数。我写的新版本里，以 4 个 A 中的向量作为一组，对应计算所有 B 向量。但是学出现了我意料之外的结果：加速仅仅 1.16 倍。我分析了一下，新版本虽然减少了 B 的读取，但是显著增加了保存结果的 C 块清零的次数和计算 C 块和的函数的调用次数，整整翻了个倍。_m512i_reduce_add_epi32 和 _m512i_setzero_si512 的执行耗时显然是要大于读取的。

更正后略微提升，优化倍数 1.76503。下一步，我尝试扩大读取 A 的数据规模，并加入预取功能。使用了_mm_prefetch，优化系数 2.10。有所提升。我在调整了预取的提前量之后，优化了一点。但是我尝试扩大 A 一次读取的规模，却发现效果不是很明显。可能是因为第一轮没有预取的慢速和之后 B 减少读取的提速相抵消了。

提交了一下，得到了幽默 62 分。我感觉 reduce_add 浪费了很长时间，有没有办法能用 storeu 来做呢？可以免去这部分的加法时间。

我尝试转换思路，按照优化提示中的那种方法来做，我打算试试广播一行中的每一个元素，dpbusd 然后进行累加最后 storeu。将矩阵乘法拆解，会发现 C 的第一行就是 A 的第一行的每个元素（其实是每四个）与 B_reshape 中每行做乘法然后累加。新思路得到了 72 分，看看能否继续优化。经过取更多的 B 后，时间来到 39ms，加速比到了 3.995564，得分 91.43

（测试 40ms），这是好的。

思考题

首先，a 是输入的图像数据，b 中储存了需要求的插值点坐标。

21 行和其他的 None 都是为了拓展维度，虽然不改变数据，但是会多加一层 C 语言中的[1]，增维来适配后续运算。

24 行 a[n_idx,x_idx,y_idx]的 shape 是 N*H2*W2*C，这些 idx 在这里实际上就是取所有图片（n_idx）的所有插值点的坐标向下取整（x_idx,y_idx）的所有通道值（最后一个参数未传入）构成的矩阵。

24 行中参与运算的三个向量是 a[...](N*H2*W2*C), x_mul(N*H2*W2*1), y_mul(N*H2*W2*1), 根据 numpy 运算的默认广播，会把 x_mul 的值复制粘贴 C-1 成一个(N*H2*W2*C)的向量，然后再进行运算。

对于 reshape.cpp，整体是将矩阵分块（16*64），逐个改变。第一版函数，完全不知道是怎么想出来的，太神奇了。首先是从分块的第一行和第九行读取前一半，然后第一行在前，第九行在后，也就是 t0 为 a00 a01 ... a0 31 a80 a81 ... a8 31，其他的类似。实现过程是读取 256 位(_mm256_loadu_epi64)，延拓成 512 位(_mm512_castsi256_si512)，高位插入第九行的 256 位(_mm512_inserti64x4)。然后利用_mm512_mask_permutexvar_epi64 进行 64 位的置换，结果储存在 r 中。这里仍然以 r0 为例。0xcc 是 11001100，但是在内存中读取认为的第一位却是 0，所以前 16 个数取 t0，也就是第一行的的 0-15，下 16 个取 t4，第五行的的 0-15，九行的 0-15，十三行的 0-15。

同理，r2 分别是 3，7，11，15 的 0-15 位，故而下一步 t0 按照 10101010 来，t0 是 1，3，5，7，9，11，13，15 的 0-7，t1 是 2，4，6，8，10，12，14，16 的 0-7，最终 r0 就是 1，2，3，...，16 的 0-3 位，存入内存即可。

这个思路真的很奇妙，很难想象是怎么想出来的。

对于另一个函数，这个函数开始时考虑了行数不为 16 的倍数的情况，但是我觉得好像有误改 B_reshape 之外数据的风险，可能会将 B_reshape 外的某些数据置 0。整体上来讲这个函数和第一个的思路是相似的。_mm512_unpacklo_epi32（两个参数向量取 32，弃 32，轮流进行）这个函数使得 t0 为 a00 a01 a02 a03 a10 a11 a12 a13 a08 a09 a0,10 a0,11，类似的交错。然后再使用_mm512_unpacklo_epi64 处理 t0 和 t2，使得 r0 成为 0,0-3 1,0-3 2,0-3 3,0-3 ,0,16-19 如此类似的交错。r4 则对应 4-7 行的同样类似的交错。

之后使用_mm512_shuffle_i32x4 函数（0x88 下就是取两个向量的 0-127 256-383 位然后拼接），使得 t0 成为 0,0-3 1,0-3 2,0-3 3,0-3 0123 32-35，4567 0-3 32-35。然后在使用一次，因为 t8 是 8-11 0-3 32-35，12-15 0-3 32-35，再结合，即可得 r0 为转置后的第一行。