目录---

--------- 一 矩阵乘法与sGEMM

--------- 二 对kernel矩阵重排(在ncnn中以aarch64为例)

--------- 三 为什么这样重排

--------- 四 与how-to-optimize-gemm教程的区别

参考：

*<https://zhuanlan.zhihu.com/p/81201840>*

*<https://zhuanlan.zhihu.com/p/65436463>*

**一 矩阵乘法与sGEMM**

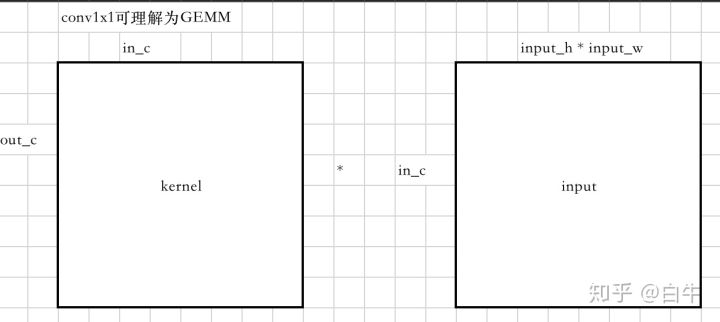
1. 设有C[m, n] = A[m, k]\*B[k, n]矩阵乘法, 则一次矩阵乘法的计算量为m\*n\*k次乘累加
2. 矩阵乘法本身的计算量在sGEMM中是不变的, 对于sGEMM的优化主要是对cache缓存的优化
3. 对于卷积计算, 以conv1x1s1为例, 可以将conv1x1s1转化为一个sGEMM, 其中：

*k = input\_channel*

*m = output\_channel*

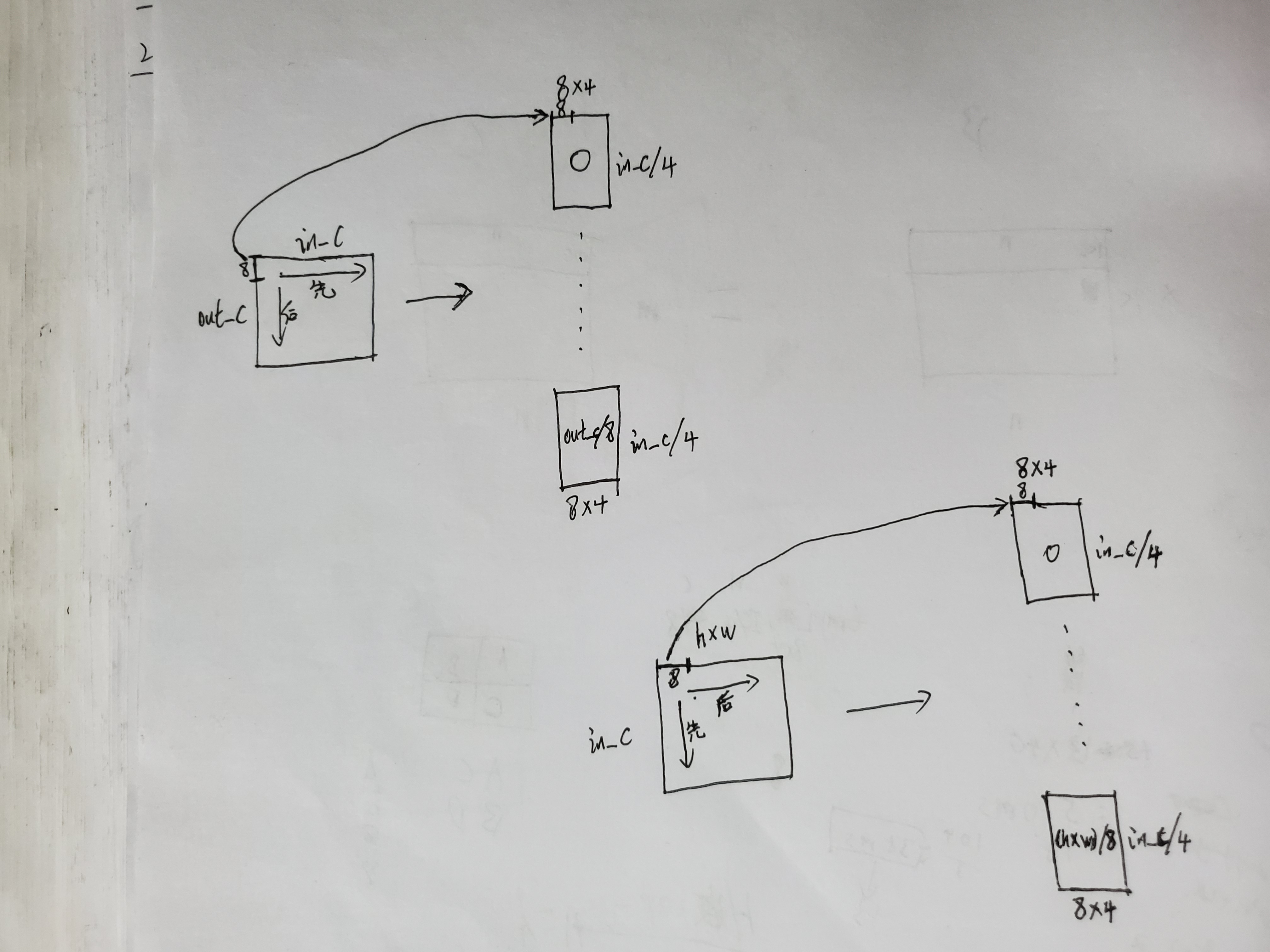
*n = input\_h\*input\_w*

即kernel可以看作m\*k的A矩阵, 输入heatmap可以看作k\*n的B矩阵, 输出为m\*n的C矩阵



**二 对kernel矩阵重排(在ncnn中以aarch64为例)**

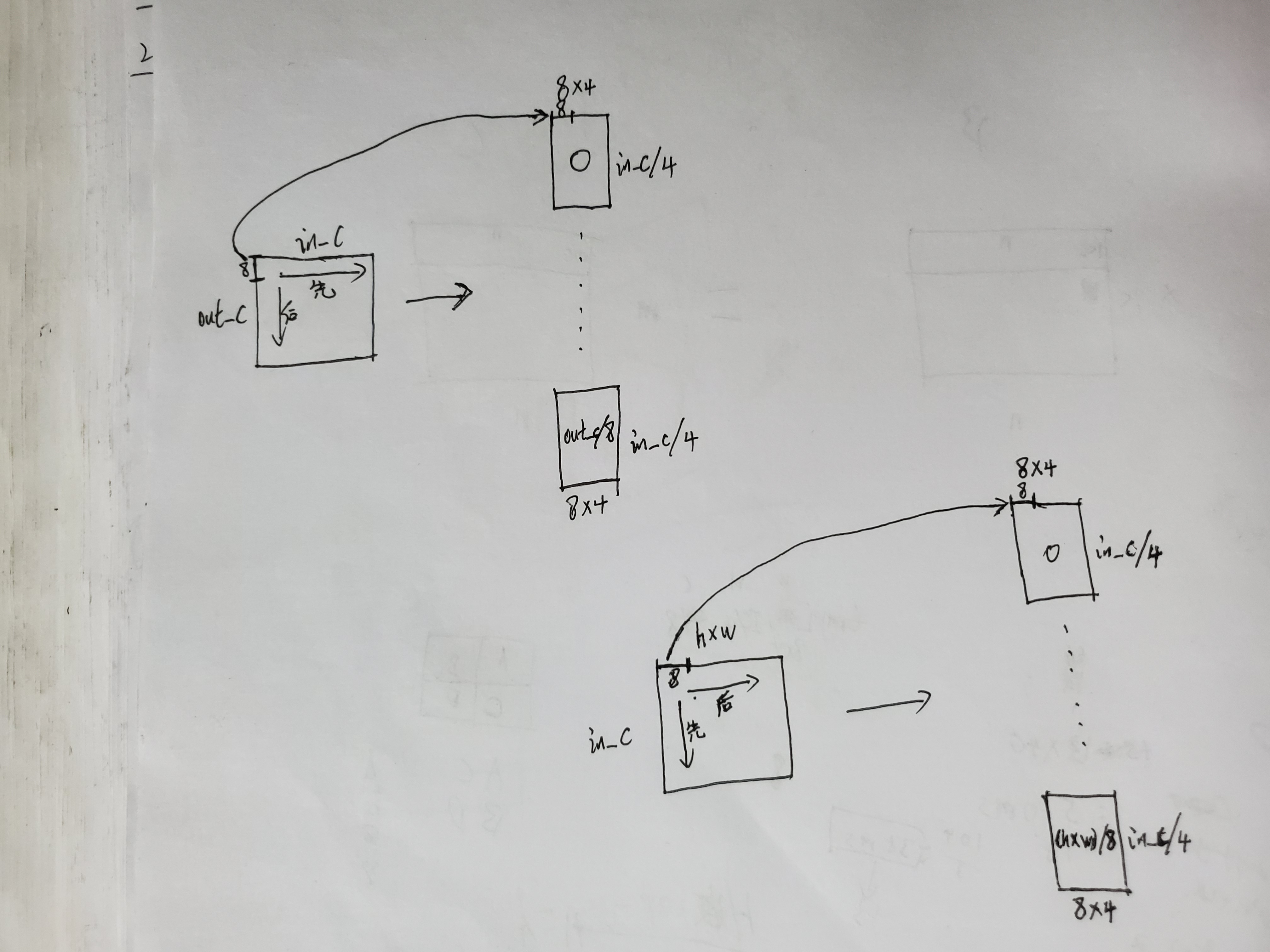
1. 对A(kernel)矩阵重排, 对应ncnn-convolution\_1x1.h中的conv1x1s1\_sgemm\_transform\_kernel\_neon函数

****

对应代码解释如下：

|  |
| --- |
| kernel\_tm.create(4\*8, inch/4 + inch%4, outch/8 + (outch%8)/4 + outch%4);  // 遍历行, 步长为8, 遍历完成后生成outch/8个channel  for (; p+7<outch; p+=8)  {  // 指向每一行的起始位置  const float\* kernel0 = kernel + (p+0)\*inch;  const float\* kernel1 = kernel + (p+1)\*inch;  const float\* kernel2 = kernel + (p+2)\*inch;  const float\* kernel3 = kernel + (p+3)\*inch;  const float\* kernel4 = kernel + (p+4)\*inch;  const float\* kernel5 = kernel + (p+5)\*inch;  const float\* kernel6 = kernel + (p+6)\*inch;  const float\* kernel7 = kernel + (p+7)\*inch;  // ktmp每次指向新的channle  float\* ktmp = kernel\_tm.channel(p/8);  // 遍历列, 步长为1, 遍历完成后生成inch/4行  // for循环完成后，生成一个数据重排后的channel  for (int q=0; q<inch; q++)  {  // 将一列的8个元素排成一行  // kernel0...7 0  ktmp[0] = kernel0[0];  ktmp[1] = kernel1[0];  ktmp[2] = kernel2[0];  ktmp[3] = kernel3[0];  ktmp[4] = kernel4[0];  ktmp[5] = kernel5[0];  ktmp[6] = kernel6[0];  ktmp[7] = kernel7[0];  ktmp += 8;  kernel0 += 1;  kernel1 += 1;  kernel2 += 1;  kernel3 += 1;  kernel4 += 1;  kernel5 += 1;  kernel6 += 1;  kernel7 += 1;  }  } |

1. 对B(输入heatmap)矩阵重排, 对应ncnn-convolution\_1x1.h中的conv1x1s1\_sgemm\_neon函数的矩阵重排部分



对应代码解释如下：

|  |
| --- |
| const int size = w \* h;  Mat tmp(8\*4, inch/4+inch%4, size/8 + (size%8)/4 + size%4, 4u, opt.workspace\_allocator);  // 遍历列, 步长为8, 遍历完成后生成(h\*w)/8个channel  for (int ii=0; ii<nn\_size; ii++)  {  int i = ii \* 8;  const float\* img0 = bottom\_blob.channel(0);  // img0每次指向新的列(步长为8)  img0 += i;  float\* tmpptr = tmp.channel(i/8);  // 遍历行, 步长为1, 遍历完成后生成inch/4行  // for循环完成后，生成一个数据重排后的channel  for (int q=0; q<inch; q++)  {  // 将一行的8个元素继续排成一行  tmpptr[0] = img0[0];  tmpptr[1] = img0[1];  tmpptr[2] = img0[2];  tmpptr[3] = img0[3];  tmpptr[4] = img0[4];  tmpptr[5] = img0[5];  tmpptr[6] = img0[6];  tmpptr[7] = img0[7];  tmpptr += 8;  // 每次循环img0指向新的一行  img0 += bottom\_blob.cstep;  }  } |

**三 为什么这样重排**

参看ncnn-convolution\_1x1.h中的conv1x1s1\_sgemm\_neon函数中的矩阵乘法部分，简写为如下三个循环：

|  |
| --- |
| for i in 0...out\_c: step=8  for j in 0...size: step=8  for k in 0...in\_c: step=4  out[8x8]=A[8x4]\*B[4x8] |

1. 最内层循环

for循环计算完成后，得到out矩阵的一个[8x8]小块矩阵结果

8x4/4x8对应重排后的kernel以及输入heatmap矩阵的一行元素

对表达式out[8x8]=A[8x4]\*B[4x8]做展开，就会得到ncnn函数中汇编计算部分

计算时：

1. 一次取kernel矩阵一列中挨着的8个元素乘以输入heatmap矩阵第2行中挨着的8个元素(注意不是常规的乘法)
2. 取kernel矩阵第2列中挨着的8个元素乘以输入heatmap矩阵第2行中挨着的8个元素，累加到A的结果中
3. 重复直到第4列和第4行，乘累加完成后得到out[8x8]
4. 以上是一次最内层for循环的计算，第2次最内层for循环会重复以上A B C三个步骤，只不过操作的数据不一样
5. 中间for循环

循环一次得到out矩阵的一个[8x8]小块矩阵结果，下一次得到第2个[8x8]小块矩阵结果

计算完成后，得到out矩阵[8xsize]块矩阵的结果

1. 最外层for循环

循环一次得到out矩阵[8xsize]块矩阵的结果，下一次得到第2个[8xsize]块矩阵的结果

计算完成后，所有计算都完成了

总结，out矩阵计算的方式决定了对kernel矩阵和输入heatmap矩阵的重排方式

**四 与how-to-optimize-gemm教程的区别**

1. 在教程中有对输入矩阵A B的划分，即划分出[mc, kc]和[kc, nc]的子矩阵，划分大小取决于cache的大小，然后对子矩阵做重排，然后计算，计算完成后再做下一次划分
2. ncnn在初始化时先对整个A矩阵做重排，然后在运行时对整个B矩阵做重排，不划分子矩阵。ncnn中的3个for循环以及计算类似教程中的kernel\_4x4单个函数，而没有教程中切分子矩阵的3个for循环