附件6：计算结构性能优化报告

1.单指令多数据优化原理及实现

因为如果直接优化通用卷积需要考虑的条件过多，优化后的效果也很差，并且常见的轻型模型多以3*3s1d1卷积为主，所以项目就先针对3*3s1d1卷积进行了优化。1*1、5*5、7*7等大小的卷积核实现原理也可由3*3卷积类推得到。

注1：s1d1指步长为1，无空洞卷积。

注2：本文档有关源码均使用小端字节序。

1.1 ncnn原始int8卷积原理及实现

在使用单指令多数据优化卷积之前，要先了解ncnn框架对卷积计算的实现。

```
for (int p = 0; p < num_output; p++)
    {
        signed char* outptr = top_blob.channel(p);

        for (int i = 0; i < outh; i++)
        {
            for (int j = 0; j < outw; j++)
            {
                int sum = 0;

                const signed char* kptr = (const signed char*)weight_data
 + maxk * channels * p;

                // channels
                for (int q = 0; q < channels; q++)
                {
                    const Mat m = bottom_blob_bordered.channel(q);
                    const signed char* sptr = m.row<signed char>(i * strid
e_h) + j * stride_w;

                    for (int k = 0; k < maxk; k++)
                    {
                        int val = sptr[space_ofs[k]];
                        int wt = kptr[k];
                        sum += val * wt;
                    }

                    kptr += maxk;
                }
```

代码1

实现代码如上所示，代码首先是按照输出的通道数、输出的行数、输出的列数的循环层次来计算出每一个输出的值。

具体的计算代码从第14行开始，sptr是指向图像的指针，kptr是指向卷积核的指针，框架先是分通道进行卷积，再将计算结果加在一起。卷积核是一个一维数组，可以直接使用k取值。但图像的值不像卷积核的值是连续的，而且还要考虑到空洞卷积的情况，因此ncnn框架提前计算了每一次卷积像素的序列到像素实际位置的映射。

因为int8是属于量化推理的一种方法，在卷积结束之后还要进行一些处理操作。

```
1  float scale_in;
2                  if (weight_data_int8_scales[p] == 0)
3                      scale_in = 0;
4                  else
5                      scale_in = 1.f / (bottom_blob_int8_scales[0] * weight_
   data_int8_scales[p]);
6
7                  float sumfp32 = sum * scale_in;
8
9                  if (bias_term)
10                     sumfp32 += bias_data[p];
11
12                 sumfp32 = activation_ss(sumfp32, activation_type, activati
   on_params);
13
14                 if (use_int8_requantize)
15                 {
16                     // requantize
17                     float scale_out = top_blob_int8_scales[0];
18                     signed char sums8 = float2int8(sumfp32 * scale_out);
19                     outptr[0] = sums8;
20                     outptr += 1;
21                 }
22                 else
23                 {
24                     // dequantize
25                     ((float*)outptr)[0] = sumfp32;
26                     outptr += 4;
27                 }
```

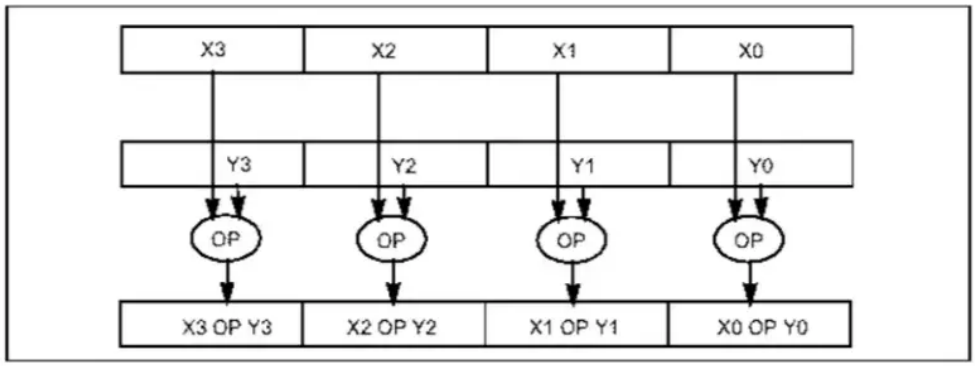
代码2

从第1行至第12行代码可以看到，ncnn框架仅在卷积计算时使用了int8类型，而在计算激活函数时将计算好的int8还原为了float类型。

第13行到第36行是有关再量化的处理，如果要求输出依旧是int8类型，就将计算结果再转换成int8类型再写入到输出top_blob中。如果不要求量化就直接写入到输出中，但因为outptr的类型是1字节的signed int，float类型是4字节，所以这里的outptr每次加4.

1.2 单指令多数据优化int8卷积原理解析

SIMD（单指令多数据）技术允许一个微指令同时对多个数据项进行操作。这对于处理大量数值数组(多媒体应用程序的典型特征)的应用程序尤其有效。例如，signed char的大小为1字节，int的大小为四字节，我们就可以使用TI提供的内联函数一次性读取四个signed char到int容器中，再使用四向SIMD乘法将四个字节与另外四个字节用类似于向量点乘的方式对应相乘，如下图所示。



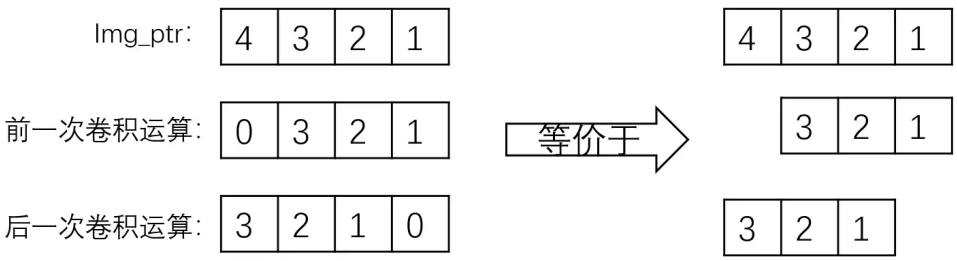
本项目的int8优化的代码是根据TI官方给出的卷积优化代码修改而成的，TI的官方优化代码仅实现了单通道、一行输出的卷积，而且并没有计算激活函数和重量化等操作，所以在实际实现时进行了修改。代码主要分为数据处理和数据计算两个部分。

首先是关于卷积核的数据处理，因为常见的卷积核是奇数，而按字节读取只能按照4字节、8字节等读入，所以要先对卷积核进行处理。

```
signed int mask1_0, mask2_0, mask3_0;
signed int mask1_1, mask2_1, mask3_1;
mask1_0 = _extu(_mem4_const(&mask_ptr[0]), 8, 8);
mask2_0 = _extu(_mem4_const(&mask_ptr[3]), 8, 8);
mask3_0 = _extu(_mem4_const(&mask_ptr[6]), 8, 8);
mask1_1 = mask1_0 << 8;
mask2_1 = mask2_0 << 8;
mask3_1 = mask3_0 << 8;
```

代码3

假设第一行卷积的值分别为(1,2,3)，_mem4_const的作用是从内存中不对齐的加载4字节的内容，此时加载进来的值为（以小端字节为例）——（x,3,2,1）。_extu的作用是将指针指向的空间左移8位再右移8位后的值返回，此时高位就清0，mask1_0的值为（0,3,2,1），再左移8位后，mask1_1的值为（3,2,1,0），这正好就是连续两次卷积操作所对应的卷积核，如下图所示。




```
for (int p = 0; p < num_output; p++)
            {
                signed char* outptr = top_blob.channel(p);
                for (int i = 0; i < outh; i++)
                {
                  const signed char *mask_ptr = (const signed char *)weigh
t_data + maxk * channels * p;
                  signed int mask1_0, mask2_0, mask3_0;
                  signed int mask1_1, mask2_1, mask3_1;
```

```
                mask1_0 = _extu(_mem4_const(&mask_ptr[0]), 8, 8);
                mask2_0 = _extu(_mem4_const(&mask_ptr[3]), 8, 8);
                mask3_0 = _extu(_mem4_const(&mask_ptr[6]), 8, 8);
                mask1_1 = mask1_0 << 8;
                mask2_1 = mask2_0 << 8;
                mask3_1 = mask3_0 << 8;

```

代码4

这是卷积数据核处理的完整代码，前面两个循环嵌套依旧是输出个数、输出的高。

接着就要对图片进行数据处理，由卷积数据的处理过程可得，每次我们一次性读入4个字节（4个int8像素）就可以进行两次卷积运算，这就意味着我们一次循环可以计算2个输出，但这一做法限制了输入和输出都必须是2的倍数，且输入图像的长度必须大于4。C6678允许一次性读入8个字节，为了最大化计算效率，我们选择以一次性计算4个输出为例子，这样限制了输出必须是4的倍数。如果要增加泛用性，也可以牺牲一定的效率，将代码简化为一次计算2个的模式，具体方式可以见下文有关float卷积的内容。

```
                short width = outw;
                short pitch = w;
                int j, count;
                const signed char *IN1, *IN2, *IN3;
                double r1_76543210, r2_76543210, r3_76543210;
                int r1_7654, r1_3210;
                int r2_7654, r2_3210;
                int r3_7654, r3_3210;
                int r1_5432, r2_5432, r3_5432;
                int sum0=0;
                int sum1=0;
                int sum2=0;
                int sum3=0;
                count = width >> 2;
                for (j = 0; j < count; j++)
                {
                    int offside=4*j;
                    sum0=0;
                    sum1=0;
                    sum2=0;
                    sum3=0;
            for (int q = 0; q < channels; q++)
                {
               const Mat m = bottom_blob_bordered.channel(q);
             const signed char *imgin_ptr = m.row<signed char>(i);
                      IN1 = imgin_ptr;
                      IN2 = IN1 + pitch;
                       IN3 = IN2 + pitch;
                         IN1 += offside;
                         IN2 += offside;
                         IN3 += offside;
                         r1_76543210 = _memd8_const(IN1);
                         r2_76543210 = _memd8_const(IN2);
```

```
                                r3_76543210 = _memd8_const(IN3);
                                r1_3210 = _lo(r1_76543210);
                                r2_3210 = _lo(r2_76543210);
                                r3_3210 = _lo(r3_76543210);
                                r1_7654 = _hi(r1_76543210);
                                r2_7654 = _hi(r2_76543210);
                                r3_7654 = _hi(r3_76543210);
                                r1_5432 = _packlh2(r1_7654, r1_3210);
                                r2_5432 = _packlh2(r2_7654, r2_3210);
                                r3_5432 = _packlh2(r3_7654, r3_3210);
```

代码5

代码5上接代码4，第三层循环不再是outw，而是outw/4，第四层循环继续为图像通道数。

我们继续以第1行的卷积为例。在第32行代码中，我们使用内联函数_memd8_const，允许从内存中不对齐的加载8个字节，加载后的结果即为从第一行开始处往后的8个像素r1_76543210。

第1次和第2次的卷积对应的内存空间为r1_3210，但第3次和第4次对应的空间为r1_5432，如下图所示。

代码先用_lo从r1_76543210取低位4个字节r1_3210，再使用_hi取高位4个字节r1_7654，最后用_packlh2将r1_7654的低为2字节和r1_3210的高位2字节拼接起来组成r1_5432。这样我们就将四次卷积计算所需的值准备好了。

```
sum0 = sum0+(_dotpu4(mask1_0, r1_3210) + _dotpu4(mask2_0, r2_3210) +
                                    _dotpu4(mask3_0, r3_3210));

 sum1 = sum1+(_dotpu4(mask1_1, r1_3210) + _dotpu4(mask2_1, r2_3210) +
                                    _dotpu4(mask3_1, r3_3210));

 sum2 = sum2+(_dotpu4(mask1_0, r1_5432) + _dotpu4(mask2_0, r2_5432) +
                                    _dotpu4(mask3_0, r3_5432));

 sum3 = sum3+(_dotpu4(mask1_1, r1_5432) + _dotpu4(mask2_1, r2_5432) +
                                    _dotpu4(mask3_1, r3_5432));
```

```
                        }
```

代码6

代码6上接代码5，此时还在第四层循环中。_dotpu4是将两个四字节空间中每一对字节按向量乘的方式对应相乘后再加起来，因此我们仅需要用三个指令就计算出了一个通道卷积后的值，再将这些值叠加到sum1、sum2、sum3中就可以得到多通道卷积后的值了。

因为ncnn设计的要求，计算后的值并不能像TI官方例程一样直接写入，还要进行一些处理后才能写入。

```
float scale_in;
                        if (weight_data_int8_scales[p] == 0)
                            scale_in = 0;
                        else
                    scale_in = 1.f / (bottom_blob_int8_scales[0] * weight_d
ata_int8_scales[p]);
                        float fsum0 = sum0 * scale_in;
                        float fsum1 = sum1 * scale_in;
                        float fsum2 = sum2 * scale_in;
                        float fsum3 = sum3 * scale_in;
                        if (bias_term)
                        {
                            fsum0 += bias_data[p];
                            fsum1 += bias_data[p];
                            fsum2 += bias_data[p];
                            fsum3 += bias_data[p];
                        }
    fsum0 = activation_ss(fsum0, activation_type, activation_params);
    fsum1 = activation_ss(fsum1, activation_type, activation_params);
    fsum2 = activation_ss(fsum2, activation_type, activation_params);
    fsum3 = activation_ss(fsum3, activation_type, activation_params);
              if (use_int8_requantize)
                {
                            // requantize
                    float scale_out = top_blob_int8_scales[0];
                    signed char ssum0 = float2int8(fsum0 * scale_out);
                    outptr[0] = ssum0;
                    outptr += 1;
                    signed char ssum1 = float2int8(fsum1 * scale_out);
                    outptr[0] = ssum1;
                    outptr += 1;
                    signed char ssum2 = float2int8(fsum2 * scale_out);
                    outptr[0] = ssum2;
                    outptr += 1;
                    signed char ssum3 = float2int8(fsum3 * scale_out);
                    outptr[0] = ssum3;
                    outptr += 1;
                }
                        else
                        {
```

```
                            // dequantize
                            ((float *)outptr)[0] = fsum0;
                            outptr += 4;
                            ((float *)outptr)[0] = fsum1;
                            outptr += 4;
                            ((float *)outptr)[0] = fsum2;
                            outptr += 4;
                            ((float *)outptr)[0] = fsum3;
                            outptr += 4;
                        }
                    }
                }
            }
```

代码7

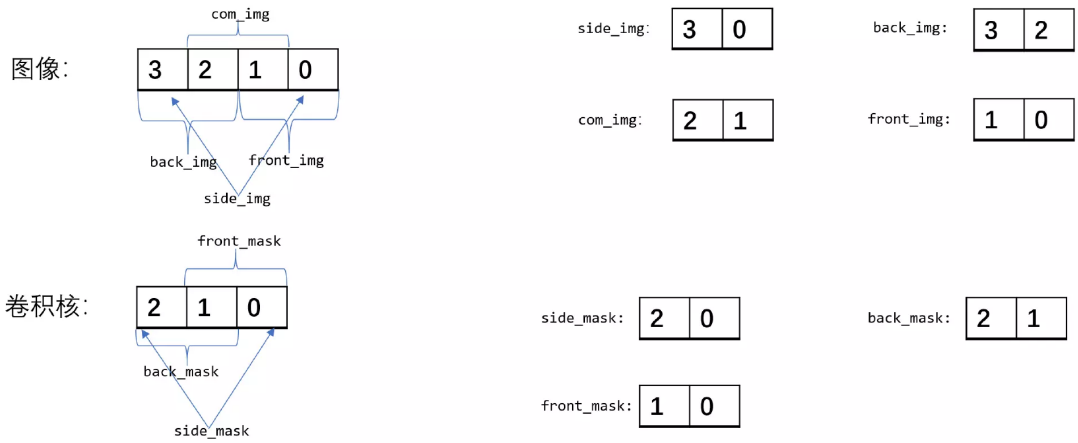
代码7上接代码6，此时在第3个循环嵌套中。后续处理的代码就相对简单很多，代码首先将int8的值还原为float的值，进行加偏移量和计算激活函数的操作。处理后的值如果需要再量化则调用ncnn提供的float2int8方法再量化后写入；如果不需要再量化则直接写入。

到这里有关单指令多数据优化int8卷积的部分就结束了，如果还有问题，可以参考项目源码中的完整源码和TI官方的源码。有关性能测试的部分放在性能测试报告中。

1.3 单指令多数据优化float卷积原理解析

float为四字节，一次最多读入两个，不能像int8一样一次性读入大量的数据再进行计算。因此需要对计算方式进行一些修改。

因为项目针对的是s1d1卷积，所以连续2次卷积之间会有两个公共的像素，2次卷积涉及到4个像素，我们将中间2个公共的像素设为com_img，两边不重叠的像素设为side_img，卷积核每一行的前两个像素设为front_mask，后两个设为back_mask，两边的像素设为side_mask如下图所示。




```
for (int p = 0; p < outch; p++)
{
    float* outptr = top_blob.channel(p);
    for (int i = 0; i < outh; i++)
    {
        const float *maskptr = (const float *)weight_data + maxk * inch * p;
```

```
        const float *IN0, *IN1, *IN2;
        __float2_t back_mask0, back_mask1, back_mask2;
        __float2_t front_mask0, front_mask1, front_mask2;
        __float2_t side_mask0, side_mask1, side_mask2;

        __float2_t back_img0, back_img1, back_img2;
        __float2_t front_img0, front_img1, front_img2;
        __float2_t side_img0, side_img1, side_img2;
        __float2_t com_img0, com_img1, com_img2;

        __float2_t sidesum,comsum0,comsum1;
        _mem8_f2(&back_mask0)=_mem8_f2_const(&maskptr[1]);
        _mem8_f2(&back_mask1)=_mem8_f2_const(&maskptr[4]);
        _mem8_f2(&back_mask2)=_mem8_f2_const(&maskptr[7]);

        _mem8_f2(&front_mask0)=_mem8_f2_const(&maskptr[0]);
        _mem8_f2(&front_mask1)=_mem8_f2_const(&maskptr[3]);
        _mem8_f2(&front_mask2)=_mem8_f2_const(&maskptr[6]);

        side_mask0=_ftof2(_hif2(back_mask0),_lof2(front_mask0));
        side_mask1=_ftof2(_hif2(back_mask1),_lof2(front_mask1));
        side_mask2=_ftof2(_hif2(back_mask2),_lof2(front_mask2));
```

代码8

float卷积优化同样是分为数据处理和数据计算两个部分，代码8是卷积核的数据处理源码。这里使用了TI提供的容器__float2_t，它的作用是存储两个float进去。然后我们类似的使用_mem8_f2和_mem8_f2_const允许不对齐的读8个字节，将卷积核的前两个像素组成front_mask，后两个像素组成side_mask；再使用_hif2、_lof2、_ftof2将back_mask的高位和front_mask的低位重新组成一个新的__float2_t，并赋给side_mask。

注：_mem8_f2和_mem8_f2_const返回值为__float2_t &和const __float2_t &；_hif2、_lof2、_ftof2的返回值均为__float2_t。

```
for (int j = 0; j < width; j++)
{
    int offside=2*j;
    float sum0=0;
    float sum1=0;
    if (bias_term){
        sum0 = bias_data[p];
        sum1 = bias_data[p];
    }

    for (int q = 0; q < inch; q++)
    {
        const Mat m = bottom_blob.channel(q);
        const float *imgin_ptr = m.row<float>(i);
        IN0 = imgin_ptr;
        IN1 = IN1 + w;
        IN2 = IN2 + w;
        IN0 += offside;
```

```
        IN1 += offside;
        IN2 += offside;

        _mem8_f2(&back_img0)=_mem8_f2_const(&IN0[2]);
        _mem8_f2(&back_img1)=_mem8_f2_const(&IN1[2]);
        _mem8_f2(&back_img2)=_mem8_f2_const(&IN2[2]);

        _mem8_f2(&front_img0)=_mem8_f2_const(&IN0[0]);
        _mem8_f2(&front_img1)=_mem8_f2_const(&IN1[0]);
        _mem8_f2(&front_img2)=_mem8_f2_const(&IN2[0]);

        com_img0=_ftof2(_lof2(back_img0),_hif2(front_img0));
        com_img1=_ftof2(_lof2(back_img1),_hif2(front_img1));
        com_img2=_ftof2(_lof2(back_img2),_hif2(front_img2));

        side_img0=_ftof2(_hif2(back_img0),_lof2(front_img0));
        side_img1=_ftof2(_hif2(back_img1),_lof2(front_img1));
        side_img2=_ftof2(_hif2(back_img2),_lof2(front_img2));
```

代码9

代码9上接代码8，是图像像素的数据处理源码。代码9和代码8相似，先读入了图像连续的四个像素的前两个像素front_img和后两个像素back_img。再使用back_img的低位和front_img的高位组成com_img；再用back_img的高位和front_img的低位组成side_img。

代码9和代码8唯一的不同就是back_img和front_img只参与数据处理，不参与数据计算。而back_mask和front_mask要参与数据计算。

这样数据处理阶段就大致完成了，接下来就是数据计算阶段。

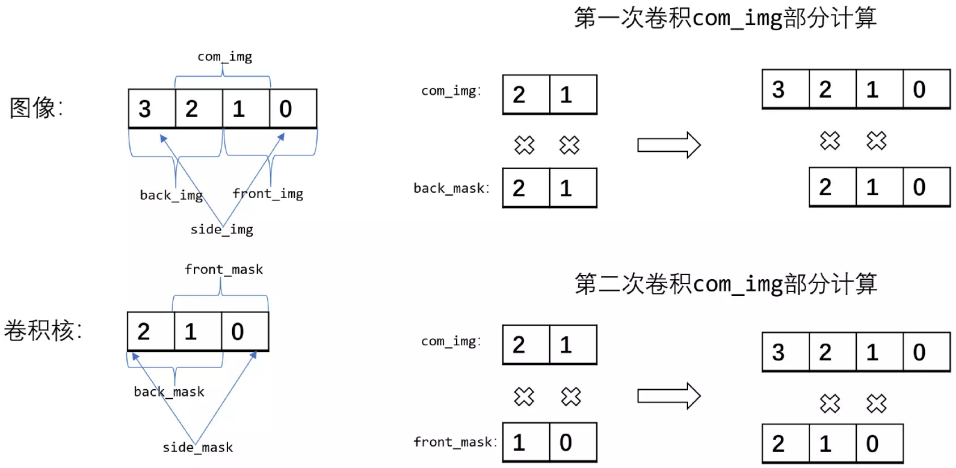
```
comsum0=_daddsp(comsum0, _dmpysp(com_img0, back_mask0));
comsum0=_daddsp(comsum0, _dmpysp(com_img1, back_mask1));
comsum0=_daddsp(comsum0, _dmpysp(com_img2, back_mask2));

comsum1=_daddsp(comsum1, _dmpysp(com_img0, front_mask0));
comsum1=_daddsp(comsum1, _dmpysp(com_img1, front_mask1));
comsum1=_daddsp(comsum1, _dmpysp(com_img2, front_mask2));

sidesum=_daddsp(sidesum, _dmpysp(side_img0, side_mask0));
sidesum=_daddsp(sidesum, _dmpysp(side_img1, side_mask1));
sidesum=_daddsp(sidesum, _dmpysp(side_img2, side_mask2));

sum0=_hif2(comsum0)+_lof2(comsum0)+_lof2(sidesum);
sum1=_hif2(comsum1)+_lof2(comsum1)+_hif2(sidesum);
```

代码10

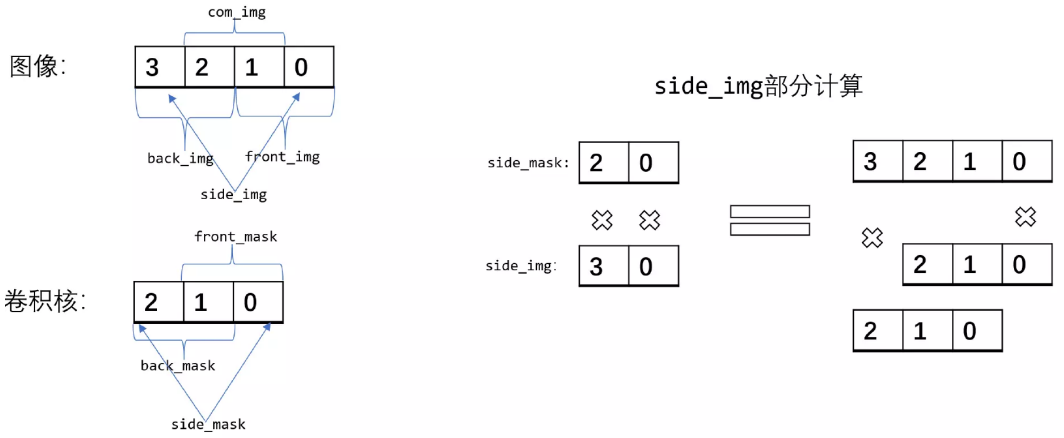
代码10上接代码9，此时代码在第四层循环中。代码10就是卷积计算部分，先使用dmpysp将com_img0的2个float数和back_mask0中的两个float数一一对应相乘，返回一个__float2_t结果，再将这个结果用_daddsp以向量加的方式将第一行的计算结果加到comsum0中，这样就计算出了第一次卷积中第一行com_img部分卷积计算的值，再将每一行的值累加起来即可。

第二次卷积中com_img部分的卷积计算和第一次相似，但是要乘以front_mask而不是back_mask。

com_img部分的计算如下图所示。



接下来就是计算side_img部分的值，计算方式和计算com_img部分的方式相似，但sidesum的高位和低位分别属于第二次卷积和第一次卷积，因此只需要一个sidesum来储存就好。这样算出的结果同样也只是一行的值，同样需要将三行的值累加到sidesum中。side_img部分的计算如下图所示。



将comsum0的高位float和低位float相加就是第一次卷积中间6个属于com_img部分的像素卷积计算后的和，取sidesum的低位float就是第一次卷积除中间6个之外的3个像素卷积计算后的和。

第二次卷积的值的计算和第一次相似，将comsum1的高位float和低位float相加就是中间6个属于com_img部分的像素卷积计算后的和，取sidesum的高位float就是第二次卷积除中间6个之外的3个像素卷积计算后的和。

最后还要对结果进行处理，因为float卷积没有进行量化，所以只需要计算激活函数即可

```
outptr[0]=activation_ss(sum0, activation_type, activation_params);
outptr[1]=activation_ss(sum1, activation_type, activation_params);
outptr+=2;
```

完整代码以包含在提交的源码中，可以参考源码进行理解。如有错漏，还请斧正。

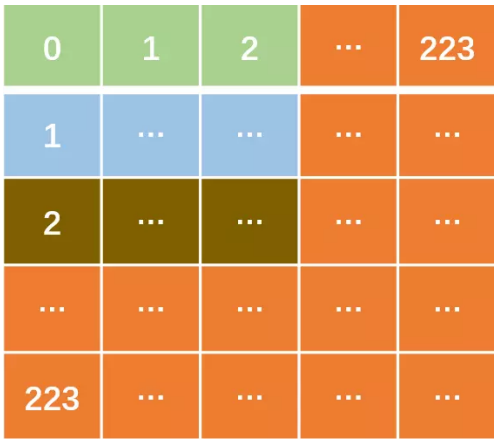
2. im2col+sgemm优化卷积运算

2.1 原始卷积性能低的原因

原始卷积代码如下

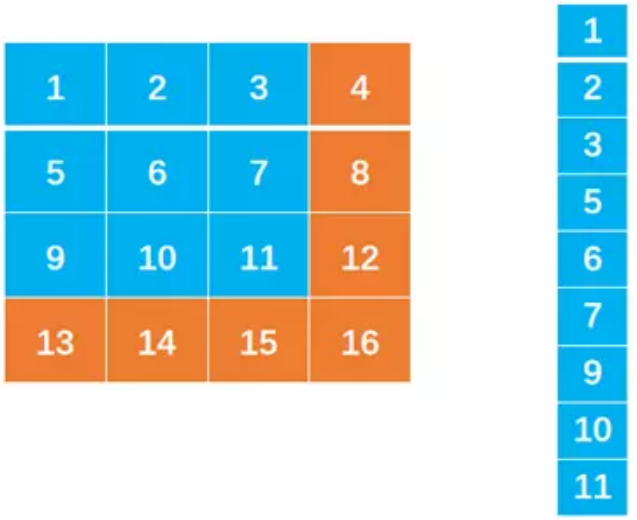
```
    for (int p = 0; p < outch; p++)
    {
      float* outptr = top_blob.channel(p);

      for (int i = 0; i < outh; i++)
      {
          for (int j = 0; j < outw; j++)
          {
              float sum = 0.f;

              if (bias_term)
                  sum = bias_data[p];

              const float* kptr = (const float*)weight_data + maxk *
                          inch * p;

              for (int q = 0; q < inch; q++)
              {
                  const Mat m = bottom_blob.channel(q);
                  const float* sptr = m.row(i * stride_h) + j *
                                       stride_w;

                  for (int k = 0; k < maxk; k++) // 29.23
                  {
                      float val = sptr[space_ofs[k]]; // 20.72
                      float wt = kptr[k];
                      sum += val * wt; // 41.45
                  }

                  kptr += maxk;
              }

              outptr[j]=activation_ss(sum, activation_type,
                                      activation_params);
          }

          outptr += outw;
      }
    }
```

对于每个输出通道的每个位置的数据，都需要访问每个输入通道的所有需进行卷积操作的元素。因此原始卷积的实现对访存优化不佳，例如一个224*224*3的输入特征图与一个3*3卷积核进行卷积时，当要读入卷积核对应的输入特征图的数据时，core访存(0,0)并读入(0,1),(0,2)同在一行的三个数据,然后再访存读入(1,0),(1,1),(1,2)，然后第三次访存(2,0),(2,1),(2,2)一共三次访存，然后再读入输入特征图的第二通道相同位置的数据。

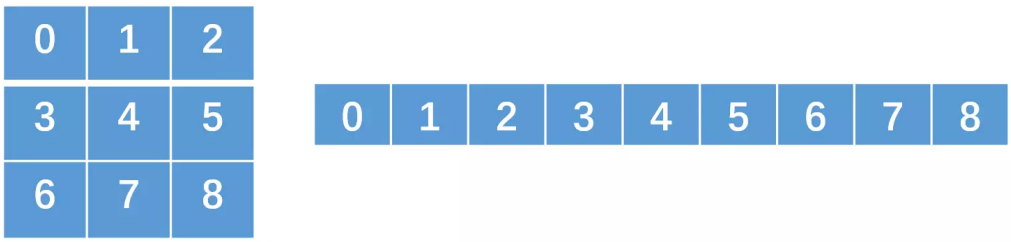


2.2 im2col转换输入特征图和卷积核的方式

如下图所示，输入特征图为4*4，在卷积核为3*3的情况下，输入特征图中第一次要参与卷积核运算的9个数按行展开为一列。卷积核大小为3*3，输入特征图为4*4的情况下，完成四次卷积，则转换后的输入特征图大小为9行4列。



同样地，3*3的卷积核也要展开成为一行



有多个输入通道的情况下，卷积核向列方向延展，有多个输出通道的情况下，卷积核向行方向延展。

2.3 传统im2col+sgemm的实现

传统的卷积优化就将卷积核和输入通道按上述方式转换后使用sgemm进行计算，再将输出特征图进行转换就完成了。但是在ti提供的dsplib中，本组发现其矩阵计算存在一定的局限性（对输入的行、列有要求），可能不能很好地满足推理框架的需要。

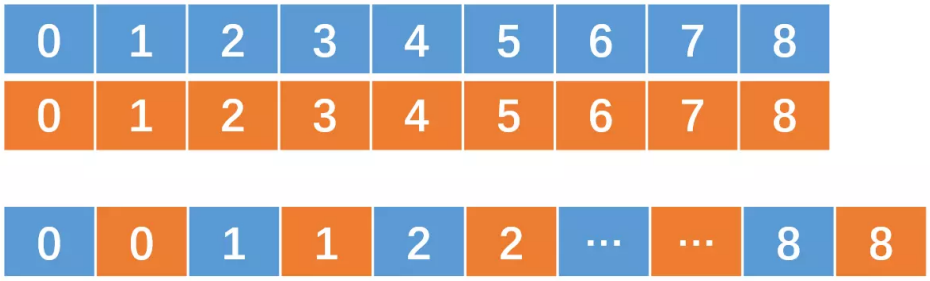
2.4 使用pack方法进行访存优化

传统的im2col+sgemm在输出矩阵的行数较大的情况下，会发生矩阵列数少，而行数多的情况，因为矩阵的一行在内存中是连续排布的，这种情况下，其访存效果会比较差。

解决这种情况可以使用pack方法来对im2col+sgemm做进一步加速。

实现的具体pack策略是对卷积核进行pack2*2，即是把两个通道的元素并在一起。

如图所示，将经过im2col后的卷积核进行pack2*2操作，把两个通道相同位置的元素按行放在一起。



对输入的特征图，则是使用pack4*4的方法，将经过im2col处理的输入特征图按4列4列的方式在内存连续分配。

最后使用带pack的sgemm进行计算，每次计算取出2个卷积元素，再取出4个输入特征图的元素。

3.算子优化结果

以下是完成计算结构优化前后的运行时间对比，优化后的运行速度显然高于优化前的速度。

神经网络模型	优化前平均运行时间 (单位/ms)	优化后平均运行时间 (单位/ms)
ShuffleNet	34037	560
ShuffleNet_v2	35755	355
blazeface	6555	150
yolo-fastestv2	22275	520
FastestDet	24820	557