week-16

JX-Ma

2023/12/30

1 本周工作

本周工作是把上周寄存器输出张量宽度按照 8 分块的代码修改了一下，使用两个内联函数分别处理输出张量宽度以 8 分块和剩下的部分，处理输出张量宽度以 8 分块的内联函数我写了另外 3 个版本，一个是使用 256 位向量寄存器存储数据，对于输入张量中不连续的数据分别使用了 gather 和先将数据放在 data 数组中，在进行 loadu 加载，另一个是使用了 2 个 128 位寄存器读取，乘加操作使用了 fmadd 指令，最后实现了使用 padding 填充输入张量来进行卷积

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

具体写的算法如下

3.1 none

这是上周写的 1x8 的代码

```
void Register_Block_Output_width_4( FTensor_1D& input, FTensor_1D& fiter ,FTensor_1D& output,size_t s)
{
    float* inptr = input.getDataPtr();                          // 此时指针指向input[0]
    float* fitr = fiter.getDataPtr();                         // 此时指针指向filter[0]
    float* outptr = output.getDataPtr();                    // 此时指针指向output[0]
    float c00,c01,c02,c03,c04,c05,c06,c07;                  // hoist Output连续的8个元素
    size_t nk = output.width / 8;
    // 输入张量 a(i,r,m*s+u,n*s+v) 卷积核 b(j,r,u,v) 输出张量 c(i,j,m,n)
    for (size_t i = 0; i < output.batch; ++i){
        size_t input_i = i * input.chw;                         // 计算输入张量第一个维度的索引i
        size_t output_i = i * output.chw;                   // 计算输出张量第一个维度的索引i
        for (size_t j = 0; j < output.channel; ++j) {
            size_t output_j = j * output.hw;                // 计算输出张量第二个维度的索引j
            size_t filter_j = j * fiter.chw;                // 计算卷积核第一个维度索引 j
            for (size_t m = 0; m < output.height; ++m){
                size_t output_m = m * output.width; // 计算输出张量中的索引 m
                size_t ms = m * s;                                      // 计算输入张量第三维度中的m*s
                // 按照8分块处理的部分
                for (size_t n = 0; n < nk; ++n){
                    // 使用8个寄存器存储输出张量按照8分块的每一个元素的值
                    c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
                    c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
                    size_t output_index =output_i + output_j + output_m + n*8;              // 计算此时输出张量的索引 因为每次处理后计算出了8个元素, 所以n*8
                    size_t ns = n * 8 * s;                                                                                  // 输入张量的窗口应该向右移动8 * s个
                    for (size_t r = 0; r < input.channel; ++r){
                        size_t input_r = r * input.hw;                                                          // 计算剩余的索引
                        size_t filter_r = r * fiter.hw;
                        for (size_t u = 0; u < fiter.height; ++u){
                            size_t input_ms = (ms + u) * input.width;
                            size_t filter_u = u  * fiter.width;
                            for (size_t v = 0; v < fiter.width; ++v) {
                                // 对于输出张量来说, 我们需要求连续8个元素的值, 对应的输入张量应该是连续的8个窗口的值, 卷积核不变
                                // v和u, 以及r三层for循环分别遍历输入张量窗口和卷积核的宽, 高, 通道维度
                                size_t input_index =input_i + input_r + input_ms +ns + v;
                                size_t filter_index = filter_j + filter_r + filter_u + v;
                                c00 += inptr[input_index] * fitr[filter_index];                 // 输入张量第一个窗口的第一个元素 * 卷积核的第一个元素,
                                c01 += inptr[input_index + s] * fitr[filter_index];         // 输入张量第二个窗口的第一个元素 * 卷积核的第一个元素
                                c02 += inptr[input_index + 2 * s] * fitr[filter_index];
                                c03 += inptr[input_index + 3 * s] * fitr[filter_index];
                                c04 += inptr[input_index + 4 * s] * fitr[filter_index];
                                c05 += inptr[input_index + 5 * s] * fitr[filter_index];
                                c06 += inptr[input_index + 6 * s] * fitr[filter_index];
                                c07 += inptr[input_index + 7 * s] * fitr[filter_index]; //  输入张量第八个窗口的第一个元素 * 卷积核的第一个元素
                            }
                        }
                    }
                    // 将卷积计算的值赋值给输入张量
                    outptr[output_index] = c00;outptr[output_index + 1] = c01;
                    outptr[output_index + 2] = c02;outptr[output_index + 3] = c03;
                    outptr[output_index + 4] = c04;outptr[output_index + 5] = c05;
                    outptr[output_index + 6] = c06;outptr[output_index + 7] = c07;
                }
                // 对于按8分块后剩余的部分处理
                // 剩余部分 输出张量的宽为55 剩余部分宽的索引应该为48,49,50,51,52,53,54, 对应的输入张量的窗口为第48个窗口
                 for (size_t n = 0; n < output.width%8; ++n){
                    size_t output_index =output_i + output_j + output_m + nk*8 + n;         // 此时输出张量的索引为(i,j,m,48)
                    size_t ns = (nk*8+n) * s;                                                                               // 计算输入张量窗口第一个元素的位置
                    c00 = 0.0;
                    for (size_t r = 0; r < input.channel; ++r){
                        size_t input_r = r * input.hw;
                        size_t filter_r = r * fiter.hw;
                        for (size_t u = 0; u < fiter.height; ++u){
                            size_t input_ms = (ms + u) * input.width;
                            size_t filter_u = u  * fiter.width;
                            for (size_t v = 0; v < fiter.width; ++v) {
                                size_t input_index =input_i + input_r + input_ms +ns+ v;
                                size_t filter_index = filter_j + filter_r + filter_u + v;
                                c00 += inptr[input_index] * fitr[filter_index];
                            }
                        }
                    }
                    outptr[output_index] = c00;
                }
            }
        }
```

```
        }
    }
```

3.2 inline

使用内联函数将块处理和单个处理分开

```
    void Register_Block_Output_width1x8( FTensor_1D& input, FTensor_1D& fiter ,FTensor_1D& output,size_t s)
    {
        size_t nk = output.width / 8;
        for (size_t i = 0; i < output.batch; ++i){
            size_t input_i = i * input.chw;
            size_t output_i = i * output.chw;
            for (size_t j = 0; j < output.channel; ++j) {
                size_t output_j = j * output.hw;
                size_t filter_j = j * fiter.chw;
                for (size_t m = 0; m < output.height; ++m){
                    size_t output_m = m * output.width;
                    size_t ms = m * s * input.width;
                     /**
                     * 此时我们需要对输出张量每一行进行分块卷积, 输出张量一行55个元素, 可以分为6个1x8的块, 剩下的7个元素按照普通单个直接卷积
                     * 参数中的索引代表卷积位置刚开始的索引, 例如我们这个方法进行的输出张量的元素为0-47也就是output[i,j,m,0]-->output[i,j,m,47]
                     * 我们带入的索引值output_index就是&output[i,j,m,0]
                     * 对应的输入张量的索引值为input[i,0,m*s,0], 卷积核的此时的索引值为filter[j,0,0,0]
                    */
                    CONV1x8(input,fiter,output,ms + input_i,filter_j,output_i+output_j+output_m,s,nk);
                    CONV1x8_remain(input,fiter,output,ms + input_i + nk * 8 * s,filter_j,output_i+output_j+output_m + nk * 8,s);


                }
            }
        }
    }
```

接下来是块处理函数

```
    // 下面是各部分的函数具体实现
    // 分块处理
    /**
 * CONV1x8 对输出张量的宽度按照8分块并进行卷积
 * param (input) 输入张量
 * param (filter) 卷积核
 * param (output) 输出张量
 * param (input_index) 带入输入张量起始位置的索引 此时已经计算了i和m*s input(i,r,m*s+u,n*s+v)
 * param (filter_index) 带入卷积核起始位置的索引 此时仅仅计算了j filter(j,r,u,v)
 * param (output_index) 带入输出张量起始位置的索引 此时计算了 i,j,m output(i,j,m,n)
 * param (s) 卷积步长
 * param (nk) 对输出张量宽进行分块的数量
 *
*/
inline void CONV1x8(FTensor_1D& input, FTensor_1D& fiter ,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s,size_t nk)
{
        float* inptr = input.getDataPtr();
    float* fitr = fiter.getDataPtr();
    float* outptr = output.getDataPtr();
        float c00,c01,c02,c03,c04,c05,c06,c07;
        for (size_t n = 0; n < nk; ++n){
                c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
                c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
                size_t output_n = n * 8;
                size_t ns = n * 8 * s;
        for (size_t r = 0; r < input.channel; ++r){
                        size_t input_r = r * input.hw;
                        size_t filter_r = r * fiter.hw;
                for (size_t u = 0; u < fiter.height; ++u){
                                        size_t input_ms =  u * input.width;
                                        size_t filter_u = u  * fiter.width;
                        for (size_t v = 0; v < fiter.width; ++v) {
                                                size_t filter_index0 = filter_index + filter_r + filter_u + v;
                                                size_t input_index0 = input_index + input_r + input_ms +ns + v;
                                                c00 += inptr[input_index0] * fitr[filter_index0];
                                                c01 += inptr[input_index0 + s] * fitr[filter_index0];
```

```
                                        c02 += inptr[input_index0 + 2 * s] * fitr[filter_index0];
                                        c03 += inptr[input_index0 + 3 * s] * fitr[filter_index0];
                                        c04 += inptr[input_index0 + 4 * s] * fitr[filter_index0];
                                        c05 += inptr[input_index0 + 5 * s] * fitr[filter_index0];
                                        c06 += inptr[input_index0 + 6 * s] * fitr[filter_index0];
                                        c07 += inptr[input_index0 + 7 * s] * fitr[filter_index0];
                                    }
                }
            }
                        size_t output_index0 = output_index + output_n;
                        outptr[output_index0] = c00;outptr[output_index0 + 1] = c01;
                        outptr[output_index0 + 2] = c02;outptr[output_index0 + 3] = c03;
                        outptr[output_index0 + 4] = c04;outptr[output_index0 + 5] = c05;
                        outptr[output_index0 + 6] = c06;outptr[output_index0 + 7] = c07;
        }
}
```

剩余部分处理

```
inline void CONV1x8_remain(FTensor_1D& input, FTensor_1D& fiter,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s )
{
    float* inptr = input.getDataPtr();
    float* fitr = fiter.getDataPtr();
    float* outptr = output.getDataPtr();
    float c00;
    for (size_t n = 0; n < output.width%8; ++n){
        size_t ns = n * s;
        c00 = 0.0;
        for (size_t r = 0; r < input.channel; ++r){
            size_t input_r = r * input.hw;
            size_t filter_r = r * fiter.hw;
            for (size_t u = 0; u < fiter.height; ++u){
                size_t input_ms =  u* input.width;
                size_t filter_u = u  * fiter.width;
                    for (size_t v = 0; v < fiter.width; ++v) {
                        size_t input_index0 = input_index + input_r + input_ms +ns+ v;
                        size_t filter_index0 =filter_index +  filter_r + filter_u + v;
                        c00 += inptr[input_index0] * fitr[filter_index0];
                    }
                }
            }
            outptr[output_index + n] = c00;
         }
}
```

分开的主要难点就是索引的计算，索引带入的参数不能出错

3.3 inline-hoist

这个主要优化的是分块处理和剩余部分函数，对索引进行进一步的 hoist

```
inline void CONV1x8_1(FTensor_1D& input, FTensor_1D& fiter,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s, size_t nk)
{

    float* inptr,*fitr,*outptr;
    outptr = output.getDataPtr() + output_index;
    float c00,c01,c02,c03,c04,c05,c06,c07;

     for (size_t n = 0; n < nk; ++n){
        c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
        c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
        // 变换指针的位置，第一次循环的元素为output[i,j,m,0] —> output[i,j,m,7] 第二次开始时的位置为output[i,j,m,8]
        // 参考直接卷积 input(i,j,m*s+u,n*s+v) 我们指针此时指向的是input(i,0,m*s,0) 这里x8因为一次循环计算output8个元素，所对应的input的窗口快也要移动这么多的位
        size_t ns = n * 8 * s;
        for (size_t r = 0; r < input.channel; ++r){
            // filter和input第二个维度 input(i,r,x,x), filter(j,r,x,x)
            size_t input_r = r * input.hw;
            size_t filter_r = r * fiter.hw;
            for (size_t u = 0; u < fiter.height; ++u){
                // 计算filter和input第三个维度 input(i,r,m*s+u,n*s+v) 因为带入的索引已经计算了m * s * input.width，因此只需要计算u * input.width
                size_t input_ms =  u * input.width;
                size_t filter_u = u  * fiter.width;
```

```
                // 此时inptr指向 input[i,r,m*s+u,ns]
                inptr = input.getDataPtr()+input_index + input_r + input_ms +ns;
                // 此时fiptr指向 filter[i,r,u,0]
                fitr = fiter.getDataPtr() + filter_index + filter_r + filter_u;
                for (size_t v = 0; v < fiter.width; ++v) {
                    float b=*(fitr+v);
                    c00 += *(inptr+v) * b;
                    c01 += *(inptr + s + v) * b;
                    c02 += *(inptr + 2 * s + v) * b;
                    c03 += *(inptr + 3 * s + v) * b;
                    c04 += *(inptr + 4 * s + v) * b;
                    c05 += *(inptr + 5 * s + v) * b;
                    c06 += *(inptr + 6 * s + v) * b;
                    c07 += *(inptr + 7 * s + v) * b;
                }
            }
        }
        *outptr++ = c00;*outptr++ = c01;
        *outptr++ = c02;*outptr++ = c03;
        *outptr++ = c04;*outptr++ = c05;
        *outptr++ = c06;*outptr++ = c07;
    }
}
```

主要变化点就是输入张量和卷积核的指针，在倒数第二个循环内指向特定的值，输入张量的指针指向了窗口的第一个元素，而卷积核指针则指向了卷积核的第一个元素，这样做的好处就是不用再最后一层循环计算前几个维度相加，在最后一层循环中只需要计算宽变化的值即可。

3.4 inline-m256-gather

这一部分主要修改的还是内联函数，我用了 256 位寄存器去存储分块的元素，因为我们需要使用寄存器去存储连续 8 个元素的值

对于输出张量来说，存储的 8 个元素是连续的，而卷积核存储的 8 个元素的值是相等的，主要是输入张量是连续 8 个窗口的第一个元素，由于步长的存在，他们并不是连续的，我的处理方法是使用 __mm256__i32gather__ps, 它的参数是（基址，索引，加载数据大小）三个参数，索引我使用了 ___m256i，这个函数 __mm256__set__epi32 他带入的 8 个参数假设为 A,B,C,D,E,F,G,H，返回值为 H,G,F,E,D,C,B,A，因此我们需要存入的索引值是逆序的。

```
inline void CONV1x8_2(FTensor_1D& input, FTensor_1D& fiter,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s, size_t nk)
{

    float* inptr,*fitr,*outptr;
    outptr = output.getDataPtr() + output_index;
    // 定义单精度256位寄存器
    // c0_c7 存储8个连续的元素，用来存储输出张量宽1x8的快
    // b0v存储8个一样的单个元素
    // a0_7 存储需要用到的输入张量的元素
        __m256 c0_7,b0v,a0_7; //float c00,c01,c02,c03,c04,c05,c06,c07;


     for (size_t n = 0; n < nk; ++n){
        c0_7 = _mm256_setzero_ps(); // 设置元素值为0 函数后面ps代表单精度
        // c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
        // c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
        // 变换指针的位置，第一次循环的元素为output[i,j,m,0] —> output[i,j,m,7] 第二次开始时的位置为output[i,j,m,8]
        // 参考直接卷积 input(i,j,m*s+u,n*s+v) 我们指针此时指向的是input(i,0,m*s,0) 这里x8因为一次循环计算output8个元素，所对应的input的窗口快也要移动这么多的位
        size_t ns = n * 8 * s;
        for (size_t r = 0; r < input.channel; ++r){
            // filter和input第二个维度 input(i,r,x,x), filter(j,r,x,x)
            size_t input_r = r * input.hw;
            size_t filter_r = r * fiter.hw;
            for (size_t u = 0; u < fiter.height; ++u){
                // 计算filter和input第三个维度 input(i,r,m*s+u,n*s+v) 因为带入的索引已经计算了m * s * input.width, 因此只需要计算u * input.width
                size_t input_ms =  u * input.width;
                size_t filter_u = u  * fiter.width;
```

```
                // 此时inptr指向 input[i,r,m*s+u,ns]
                inptr = input.getDataPtr()+input_index + input_r + input_ms +ns;
                // 此时fiptr指向 filter[i,r,u,0]
                fitr = fiter.getDataPtr() + filter_index + filter_r + filter_u;
                for (size_t v = 0; v < fiter.width; ++v) {
                    __m256i indexVec = _mm256_set_epi32(7*s,6*s,5*s,4*s,3*s,2*s,s,0); // 设置索引值 用于求下面的输入张量位置
                    a0_7 = _mm256_i32gather_ps(inptr+v,indexVec,sizeof(float));//BASE, INDEX, SCALE 参数名称， BASE 基址，INDEX 索引， SCALE 取数大小
                    b0v = _mm256_set1_ps(*(fitr+v));
                    c0_7 = _mm256_fmadd_ps(a0_7,b0v,c0_7);
                    // c00 += *(inptr+v) * *(fitr+v);
                    // c01 += *(inptr + s + v) * *(fitr+v);
                    // c02 += *(inptr + 2 * s + v) * *(fitr+v);
                    // c03 += *(inptr + 3 * s + v) * *(fitr+v);
                    // c04 += *(inptr + 4 * s + v) * *(fitr+v);
                    // c05 += *(inptr + 5 * s + v) * *(fitr+v);
                    // c06 += *(inptr + 6 * s + v) * *(fitr+v);
                    // c07 += *(inptr + 7 * s + v) * *(fitr+v);
                }
            }
        }
        *outptr++ = c0_7[0];*outptr++ = c0_7[1];
        *outptr++ = c0_7[2];*outptr++ = c0_7[3];
        *outptr++ = c0_7[4];*outptr++ = c0_7[5];
        *outptr++ = c0_7[6];*outptr++ = c0_7[7];
    }
}
```

3.5 inline-m256-loadu

这一部分修改的也是内联函数，主要是对于输入张量的元素加载采用的是 loadu，_mm256_loadu_ps, 这个函数参数为一个数组的地址，它可以加载从该地址开始的连续 8 个元素，加载 8 个元素的原因是因为，寄存器大小为 256 为，ps 代表单精度元素，也就是 float，float4 字节 =32 位，因此可以加载 8 个元素。因为输入张量存入寄存器的元素不是连续的，因此我们需要先定义一个数组去加载每个窗口的第一个元素，在使用 loadu 去读取。

```
inline void CONV1x8_3(FTensor_1D& input, FTensor_1D& fiter,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s, size_t nk)
{

    float* inptr,*fitr,*outptr;
    outptr = output.getDataPtr() + output_index;
    // 定义单精度256位寄存器
    // c0_c7 存储8个连续的元素， 用来存储输出张量宽1x8的快
    // b0v存储8个一样的单个元素
    // a0_7 存储需要用到的输入张量的元素
        __m256 c0_7,b0v,a0_7; //float c00,c01,c02,c03,c04,c05,c06,c07;


     for (size_t n = 0; n < nk; ++n){
        c0_7 = _mm256_setzero_ps(); // 设置元素值为0 函数后面ps代表单精度
        // c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
        // c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
        // 变换指针的位置， 第一次循环的元素为output[i,j,m,0] —> output[i,j,m,7] 第二次开始时的位置为output[i,j,m,8]
        // 参考直接卷积 input(i,j,m*s+u,n*s+v) 我们指针此时指向的是input(i,0,m*s,0) 这里x8因为一次循环计算output8个元素， 所对应的input的窗口快也要移动这么多的位
        size_t ns = n * 8 * s;
        for (size_t r = 0; r < input.channel; ++r){
            // filter和input第二个维度 input(i,r,x,x), filter(j,r,x,x)
            size_t input_r = r * input.hw;
            size_t filter_r = r * fiter.hw;
            for (size_t u = 0; u < fiter.height; ++u){
                // 计算filter和input第三个维度 input(i,r,m*s+u,n*s+v) 因为带入的索引已经计算了m * s * input.width, 因此只需要计算u * input.width
                size_t input_ms =  u * input.width;
                size_t filter_u =  u  * fiter.width;
                // 此时inptr指向 input[i,r,m*s+u,ns]
                inptr = input.getDataPtr()+input_index + input_r + input_ms +ns;
                // 此时fiptr指向 filter[i,r,u,0]
                fitr = fiter.getDataPtr() + filter_index + filter_r + filter_u;
                for (size_t v = 0; v < fiter.width; ++v) {
                    float data[8] = {*(inptr + v),*(inptr + s + v),*(inptr + 2 * s + v),*(inptr + 3 * s + v),*(inptr + 4 * s + v),*(inptr + 5 * s + v),*(inptr +
                    a0_7 = _mm256_loadu_ps(data);
```

```
                b0v = _mm256_set1_ps(*(fitr+v));
                c0_7 = _mm256_fmadd_ps(a0_7,b0v,c0_7);
                // c00 += *(inptr+v) * *(fitr+v);
                // c01 += *(inptr + s + v) * *(fitr+v);
                // c02 += *(inptr + 2 * s + v) * *(fitr+v);
                // c03 += *(inptr + 3 * s + v) * *(fitr+v);
                // c04 += *(inptr + 4 * s + v) * *(fitr+v);
                // c05 += *(inptr + 5 * s + v) * *(fitr+v);
                // c06 += *(inptr + 6 * s + v) * *(fitr+v);
                // c07 += *(inptr + 7 * s + v) * *(fitr+v);
            }
        }
    }
    *outptr++ = c0_7[0];*outptr++ = c0_7[1];
    *outptr++ = c0_7[2];*outptr++ = c0_7[3];
    *outptr++ = c0_7[4];*outptr++ = c0_7[5];
    *outptr++ = c0_7[6];*outptr++ = c0_7[7];
  }
}
```

3.6 inline-2xm128-loadu

这一部分修改的也是内联函数，修改的部分是使用 2 个 128 位寄存器去存储 8 个 float 元素，因为 128 位寄存器方法中没有 gather 函数，因此我使用了 loadu 去加载输入张量的元素，同样的我也需要先用数组加载输入张量中不连续的元素。

```
inline void CONV1x8_4(FTensor_1D& input, FTensor_1D& fiter,FTensor_1D& output,size_t input_index,size_t filter_index,size_t output_index,size_t s, size_t nk)
{

    float* inptr,*fitr,*outptr;
    outptr = output.getDataPtr() + output_index;
    // 定义单精度256位寄存器
    // c0_c7 存储8个连续的元素，用来存储输出张量宽1x8的快
    // b0v存储8个一样的单个元素
    // a0_7 存储需要用到的输入张量的元素
        __m128 c0_3,c4_7,b0v,a0_3,a4_7; //float c00,c01,c02,c03,c04,c05,c06,c07;


     for (size_t n = 0; n < nk; ++n){
        c0_3 = _mm_setzero_ps();
        c4_7 = _mm_setzero_ps();// 设置元素值为0 函数后面ps代表单精度
        // c00 = 0.0;c01=0.0; c02=0.0;c03=0.0;
        // c04 = 0.0;c05=0.0; c06=0.0;c07=0.0;
        // 变换指针的位置，第一次循环的元素为output[i,j,m,0] —> output[i,j,m,7] 第二次开始时的位置为output[i,j,m,8]
        // 参考直接卷积 input(i,j,m*s+u,n*s+v) 我们指针此时指向的是input(i,0,m*s,0) 这里x8因为一次循环计算output8个元素，所对应的input的窗口快也要移动这么多的位
        size_t ns = n * 8 * s;
        for (size_t r = 0; r < input.channel; ++r){
            // filter和input第二个维度 input(i,r,x,x), filter(j,r,x,x)
            size_t input_r = r * input.hw;
            size_t filter_r = r * fiter.hw;
            for (size_t u = 0; u < fiter.height; ++u){
                // 计算filter和input第三个维度 input(i,r,m*s+u,n*s+v) 因为带入的索引已经计算了m * s * input.width, 因此只需要计算u * input.width
                size_t input_ms =  u * input.width;
                size_t filter_u = u  * fiter.width;
                // 此时inptr指向 input[i,r,m*s+u,ns]
                inptr = input.getDataPtr()+input_index + input_r + input_ms +ns;
                // 此时fiptr指向 filter[i,r,u,0]
                fitr = fiter.getDataPtr() + filter_index + filter_r + filter_u;
                for (size_t v = 0; v < fiter.width; ++v) {
                    float data[8] = {*(inptr + v),*(inptr + s + v),*(inptr + 2 * s + v),*(inptr + 3 * s + v),*(inptr + 4 * s + v),*(inptr + 5 * s + v),*(inptr +
                    a0_3 = _mm_loadu_ps(data);
                    a4_7 = _mm_loadu_ps(&data[4]);
                    b0v = _mm_set1_ps(*(fitr+v));
                    c0_3 = _mm_fmadd_ps(a0_3,b0v,c0_3);
                    c4_7 = _mm_fmadd_ps(a4_7,b0v,c4_7);
                    // c00 += *(inptr+v) * *(fitr+v);
                    // c01 += *(inptr + s + v) * *(fitr+v);
                    // c02 += *(inptr + 2 * s + v) * *(fitr+v);
                    // c03 += *(inptr + 3 * s + v) * *(fitr+v);
                    // c04 += *(inptr + 4 * s + v) * *(fitr+v);
                    // c05 += *(inptr + 5 * s + v) * *(fitr+v);
```

```
                    // c06 += *(inptr + 6 * s + v) * *(fitr+v);
                    // c07 += *(inptr + 7 * s + v) * *(fitr+v);
                }
            }
        }
        *outptr++ = c0_3[0];*outptr++ = c0_3[1];
        *outptr++ = c0_3[2];*outptr++ = c0_3[3];
        *outptr++ = c4_7[0];*outptr++ = c4_7[1];
        *outptr++ = c4_7[2];*outptr++ = c4_7[3];
    }
}
```

3.7 inline-padding

这一部分主要就是对于分块有剩余的处理，我使用 padding，也就是使用 0 去填充输入张量宽度，本来有想过填充输出张量的元素，但是如果选择填充输出张量元素的话，输入张量的窗口移动会出现错误，在输出张量宽度最后一个元素卷积的时候，窗口移动会有问题，所以我选择填充输入张量的宽度，因为只是寄存器只是按照了宽度分块，因此我只填充了输入张量的宽度。我的做法就是将输入张量的宽度填充 +4，填充的元素是 0，最后得到的输出张量在进行处理。对于分块处理我采用的函数是 inline 中的函数。

```
void Register_Block_Output_width1x8_padding( FTensor_1D& input, FTensor_1D& fiter ,FTensor_1D& output,size_t s)
{
    // 对input张量进行填充0
    FTensor_1D A1(10,3,227,231);
    FTensor_1D C1(10,96,55,56);
    float *A,*a;
    for(size_t i = 0; i< A1.batch;++i){
        size_t i1 = i * input.chw;
        size_t i2 = i * A1.chw;
        for(size_t j = 0; j< A1.channel;++j){
            size_t j1 = j * input.hw;
            size_t j2 = j * A1.hw;
            for(size_t m = 0; m< A1.height;++m){
                size_t m1 = m * input.width;
                size_t m2 = m * A1.width;
                a = input.getDataPtr() + m1 + i1 + j1;
                A = A1.getDataPtr() + m2 + i2 + j2;
                for(size_t n = 0; n< input.width;++n){
                    *A++ = *a++;
                }
                for(size_t n = 0; n < A1.width - input.width;++n){
                    *A++ = 0;
                }
            }
        }
    }
    size_t nk = C1.width / 8;
    for (size_t i = 0; i < C1.batch; ++i){
        size_t input_i = i * A1.chw;
        size_t C1_i = i * C1.chw;
        for (size_t j = 0; j < C1.channel; ++j) {
            size_t C1_j = j * C1.hw;
            size_t filter_j = j * fiter.chw;
            for (size_t m = 0; m < C1.height; ++m){
                size_t C1_m = m * C1.width;
                size_t ms = m * s * A1.width;
                 /**
                  * 此时我们需要对输出张量每一行进行分块卷积，输出张量一行55个元素，可以分为6个1x8的块，剩下的7个元素按照普通单个直接卷积
                  * 参数中的索引代表卷积位置刚开始的索引，例如我们这个方法进行的输出张量的元素为0-47也就是output[i,j,m,0]-->output[i,j,m,47]
                  * 我们带入的索引值output_index就是&output[i,j,m,0]
                  * 对应的输入张量的索引值为input[i,0,m*s,0]，卷积核的此时的索引值为filter[j,0,0,0]
                  */
                CONV1x8(A1,fiter,C1,ms + input_i,filter_j,C1_i+C1_j+C1_m,s,nk);
            }
        }
    }
```

```
    for(size_t i = 0; i< C1.batch;++i){
        size_t i1 = i * output.chw;
        size_t i2 = i * C1.chw;
        for(size_t j = 0; j< C1.channel;++j){
            size_t j1 = j * output.hw;
            size_t j2 = j * C1.hw;
            for(size_t m = 0; m< C1.height;++m){
                size_t m1 = m * output.width;
                size_t m2 = m * C1.width;
                a = output.getDataPtr() + m1 + i1 + j1;
                A = C1.getDataPtr() + m2 + i2 + j2;
                for(size_t n = 0; n< output.width;++n){
                    *a++ = *A++;
                }
            }
        }
    }
}
```

4 实验结果和分析

横坐标的标签是上述的每个算法

gflops 比较

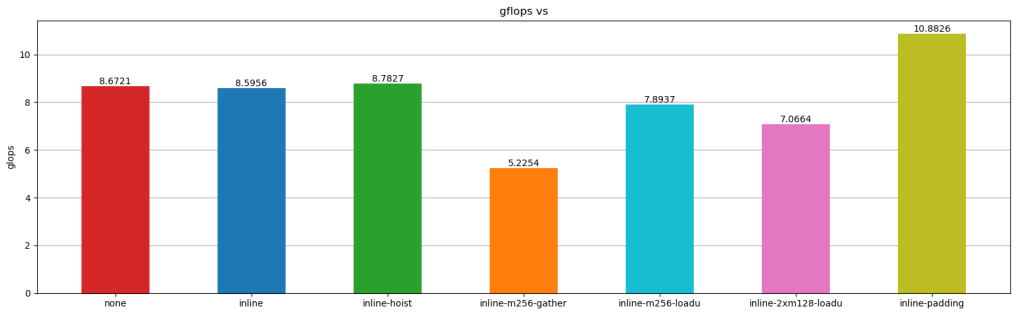

图 1: gflops

从 none-inline，所做的是就是使用内联函数将分块处理和剩余的分开，gflops 慢了一点
inline- inline-hoist，就是进一步做了索引 hoist，gflops 快了一点
inline-m256-gather,inline-m256-loadu 可以看到使用 Gather 速度会慢很多
inline-m256-loadu - inline-2xm128-loadu 使用 2 个 m128 寄存器存 8 个元素 gflops 会慢一点
inline-padding,inline 可以看到使用 padding 填充比先处理分块部分在处理剩余分布快很多
因此可以看到对于离散元素的存储，目前发现还是使用 loadu 加载比较快，但是这样的话使用 AVX2 寄存器并没有加快速度，看到 0 内存卷积论文中提出更改输入张量的结构，使输入张量存入寄存器的元素连续，或者还有不知道的更好的处理方法。