week-89

JX-Ma

2024/4/27

1 本周工作

1. 优化 im2win，改变输入张量展开的通道维度大小。
2. 尝试了不同的对齐内存的方式。
3. 测试了输出张量不同维度融合以及他们的内存分析。

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

2.2.1 优化策略

改变输入张量和卷积核通道维度展开的大小.

- v1 : 减少了卷积传入的参数。
- v2: 在 conv1 conv2 conv3 conv4 conv7 上，输入张量和卷积核张量都不展开通道维度，其他的 benchmark 上面原本是全部展开的，现在在原来全部展开的基础上/2，例如我输入张量通道为 512，现在我只展开 256.
- v3: 在 conv1 conv2 conv3 conv4 conv7 上，输入张量和卷积核张量都不展开通道维度，其他的 benchmark 上面原本是全部展开的，现在在原来全部展开的基础上/4，例如我输入张量通道为 512，现在我只展开 128.

传参变化

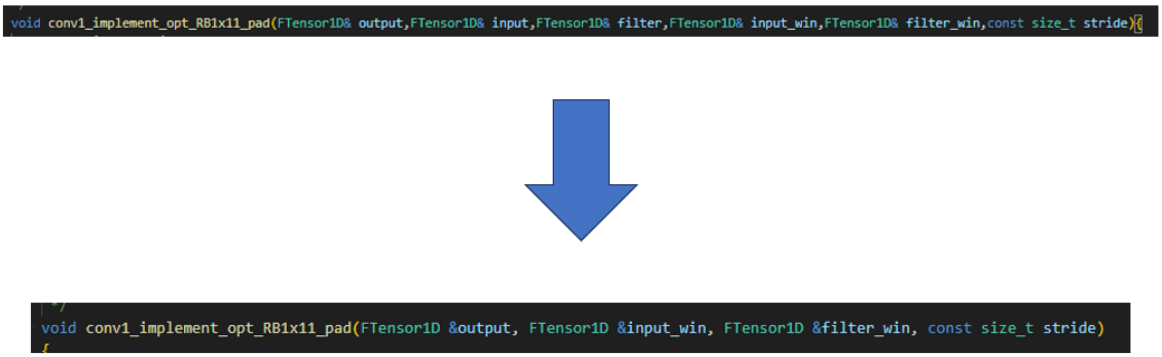

图 1: v1-dif1

改变了传参的数量，还有 stride 意义，原本 stride 代表的卷积的步长，但是经过 im2win 变化和按照 channel 展开后，步长应该改变，原本的横向步长 (在宽度上窗口移动的距离) 变为 stride * filter.height * 通道展开大小，纵向步长 (在高度上窗口移动的距离) 变为 1，这里传入的 stride 参数代表变化后的横向步长。

2.2.2 代码变化

v2v3 的变化也就是改变了展开的大小，在原本展开函数的基础上添加了 flow 变量，用于控制展开的大小。

2.2.3 实验结果

内存分析结果在 channel_flow 文件夹内, 从结果中看到在 conv1,conv2,conv4,conv7 层上不按照通道维度展开的性能比较好，conv3,conv8,conv9,conv12 性能变化不明显，而 conv5,conv6,conv10,conv11, 展开通道大小越小，性能也随之下降。然后再看看内存分析的结果，在 conv1 上的 cachemiss 不展开的要略小于展开后的，然后 conv2，conv4 上则是 InVoluntary-contex 较小，在 conv5 上面通道展开大小/2，相比全部展开的 InVoluntary_contex 要大，但是 minor-page-fault 和自愿交换次数比较小，通道/4 的则是这三个参数都要大于全部展开的。在 conv6 上感觉有点奇怪，在按通道/4 展开后，次要缺页和自愿和非自愿交换次数以及 cachemiss 都比全部展开就，就 cpu-reference 比较高，然后性能最差。conv7 上面主要影响是缺页次数，不展开缺页次数明显小于全部展开的。

函数内变化

```
float *inptr, *filptr, *outptr;
inptr = input_win.getDataPtr();
filptr = filter_win.getDataPtr();
outptr = output.getDataPtr();

//基础的input,output,filter信息
const size_t input_batch = input.batch;
const size_t input_channel = input.channel;
const size_t input_height = input.height;
const size_t input_width = input.width;

const size_t filter_batch = filter.batch;
// const size_t filter_channel = filter.channel; = input_channel
const size_t filter_height = filter.height;
const size_t filter_width = filter.width;

const size_t output_batch = output.batch;
const size_t output_channel = output.channel;
const size_t output_height = output.height;
const size_t output_width = output.width;

//卷积运算的窗口大小
const size_t window_size = filter_height*filter_width*input_channel;

const size_t window_area = filter_height*input_width*input_channel;

const size_t remainder = window_size % 8;

size_t window_size_8 = window_size;

if (remainder != 0) {
    window_size_8 = window_size + 8 - remainder; //如果余数不为0，将计算窗口增加到8的倍数
}

const size_t o_cwh = output_channel*output_width*output_height;
const size_t o_wh = output_width*output_height;
const size_t i_whhc = output_height*window_area;
const size_t i_whc = window_area;
const size_t i_dhc = stride*filter_height*input_channel;
const size_t f_cwh = window_size_8;

const size_t stride_win = stride*filter_height*input_channel;

const size_t output_bc = output_batch*output_channel;
size_t i,j;
```

```
float *inptr, *filptr, *outptr;

inptr = input_win.getDataPtr();
filptr = filter_win.getDataPtr();
outptr = output.getDataPtr();

const size_t output_batch = output.batch;
const size_t output_channel = output.channel;
const size_t output_height = output.height;
const size_t output_width = output.width;

const size_t outputChw = output_channel * output_width * output_height;
const size_t outputHw = output.width * output.height;
const size_t inputChw = input_win.height * input_win.width;

const size_t output_bc = output_batch * output_channel;
size_t i, j;
```

图 2: v1-dif2
减少代码长度

2.3 实验二

2.3.1 改变

对于内存对齐采用 3 中方式，第一种是使用了 posix_memalign 对储存数据的指针动态分配对齐内存。第二种是在定义指针的时候加上 align(64), 第三种是在初始化类的时候加上 align(64).

2.3.2 测试结果

2.3.3 结果分析

可以看到就只有 conv5,conv6,conv11,conv12 上性能有较好的提升，且只有在第一种内存对齐上有提升，内存分析文件为 align.xlsx, 在 conv5 上面的 minin-page-faults 较低，conv6 上数据不知道怎么分析，在 conv11 和 conv12 上面的自愿和非自愿缺页次数少.

2.4 实验三

实验三主要是融合不同的维度，这里因为情况很多，其中四个维度融合的有一种，三个维度融合的有 4 种，并且根据循环顺序不同每种有两种不同的顺序。2 个维度融合的有 6 种，每种有 6 种不同的循环排序。

这里有一点需要补充因为 conv4 的输出张量宽度在分块的时候不能整除，有些要单独处理，所以涉及到融合 w 的可以不把 conv4 作为比较范围内，因为他们使用的都是只融合 bc 这两个维度的同一个方法。总的来说融合 bh,bw,hw,bcw,bhw 效果都还行。

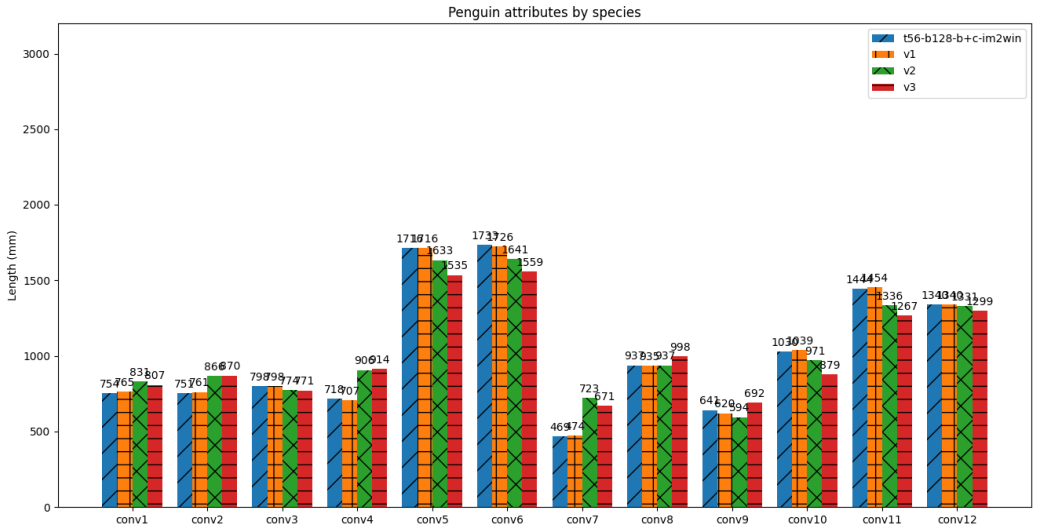

图 3: gflops

如果把那些优化总结来说，首先在 conv1,2,4,7 上面不按照通道展开，内存对齐方法在 conv5,conv6 层有不错的提升，然后再融合维度方面，可以考虑 bh-c-w 或者 bhw-c.

第一种

```
    this->width = W;
    if (0 != posix_memalign(reinterpret_cast<void**>(&this->dataPtr), 64, sizeof(T) * (N * C * H * W))) {
      cout<<"error!"<<endl;
      return;
}
}
```

第二种

```
class Tensor1D{
    protected:
        align(64) T *dataPtr; //指向存储数据的空间的指针

    public:
```

第三种

```
align(64)  FTensor1D input_d(input.batch, input.channel, input.height, input.width);
align(64)  FTensor1D filter_d(filter.batch, filter.channel, filter.height, filter.width);
align(64)  FTensor1D output_d(input.batch, filter.batch, ((input.height - filter.height) / stride + 1), ((input.width - filter.width) / stride + 1));
align(64)  FTensor1D input_win(input.batch, input.channel, input.height, input.width);
align(64)  FTensor1D filter_win(filter.batch, filter.channel, filter.height, filter.width);
 size_t s_w = stride * filter_d.height * filter_d.channel;
```

图 4: align

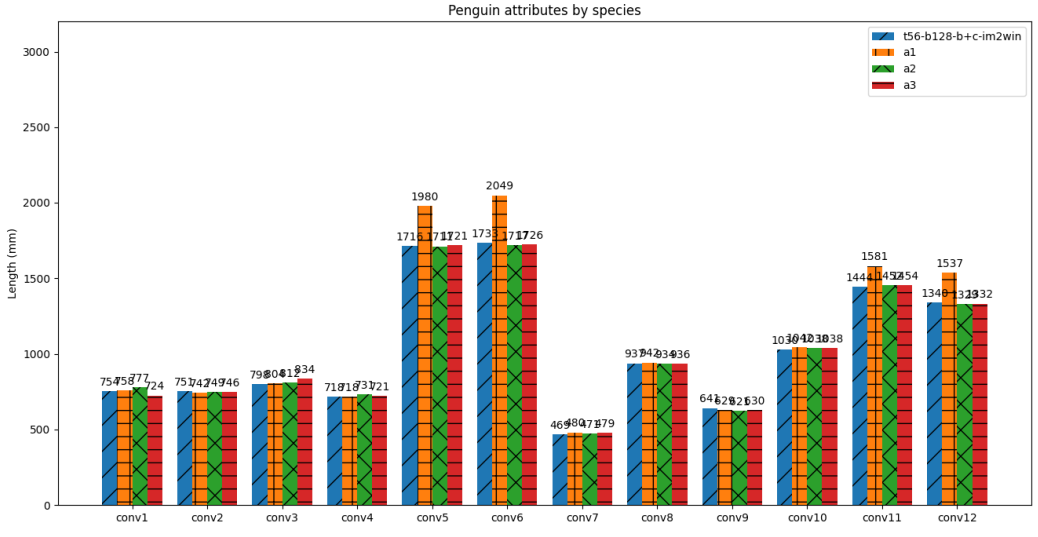

图 5: align

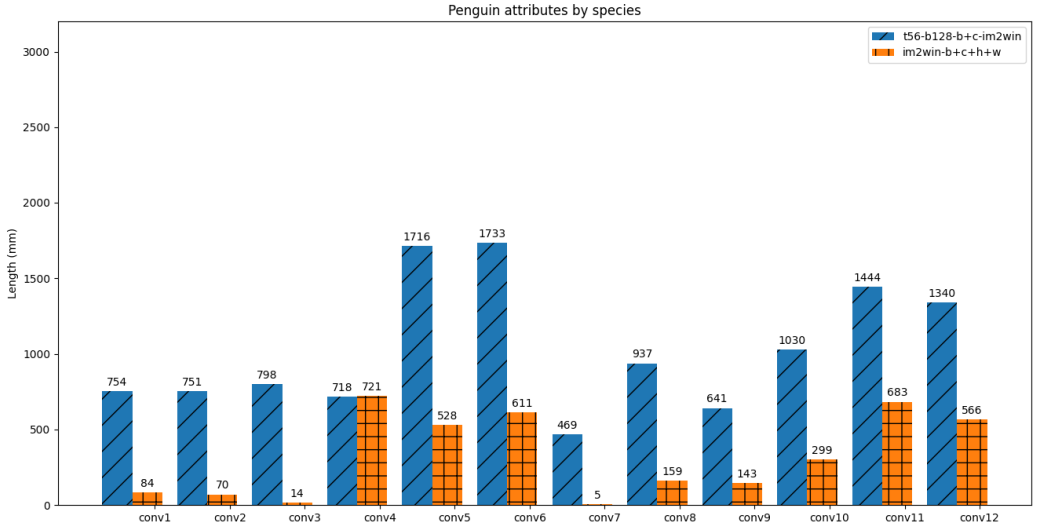

图 6: bchw

全是负优化，全部融合效果并不好

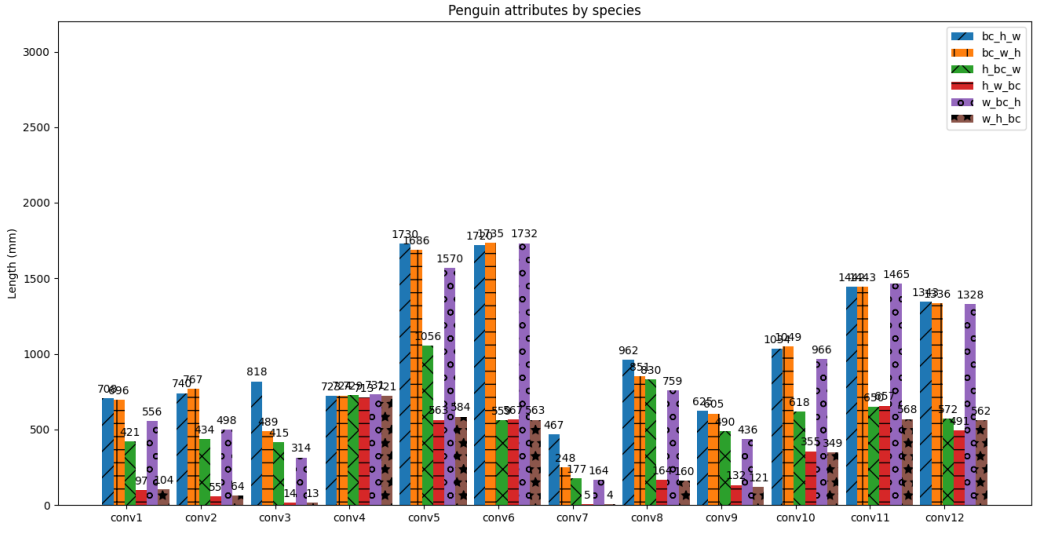

图 7: bc

只融合 bc 两个维度可以看到，最好的排序是 bchw

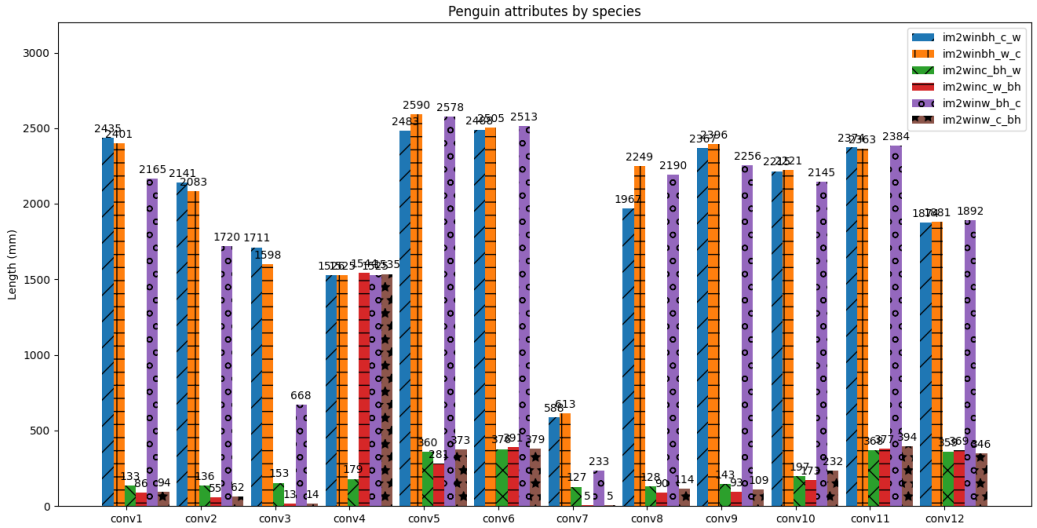

图 8: bh

可以看到融合 bh 这两个维度，在把 bh 这两个维度放到最前面的效果非常好，可以超过直接卷积。在内存分析中我把 bh 融合最好的结果和直接卷积对比，发现 cache-ref 比直接卷积少很多。

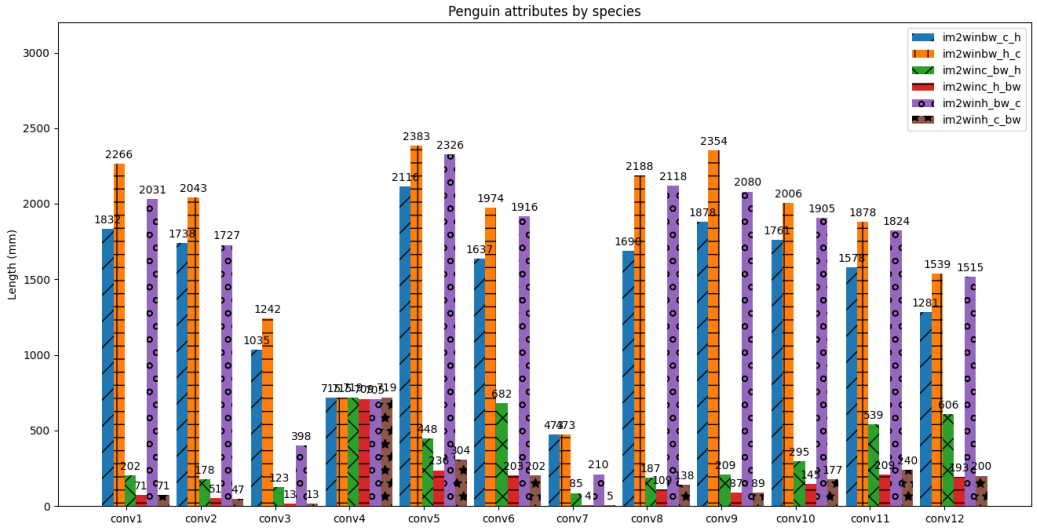

图 9: bw

融合 bw 这两个维度展示的效果也非常好，这里效果好的排序是把输出张量的通道维度放在最里面。

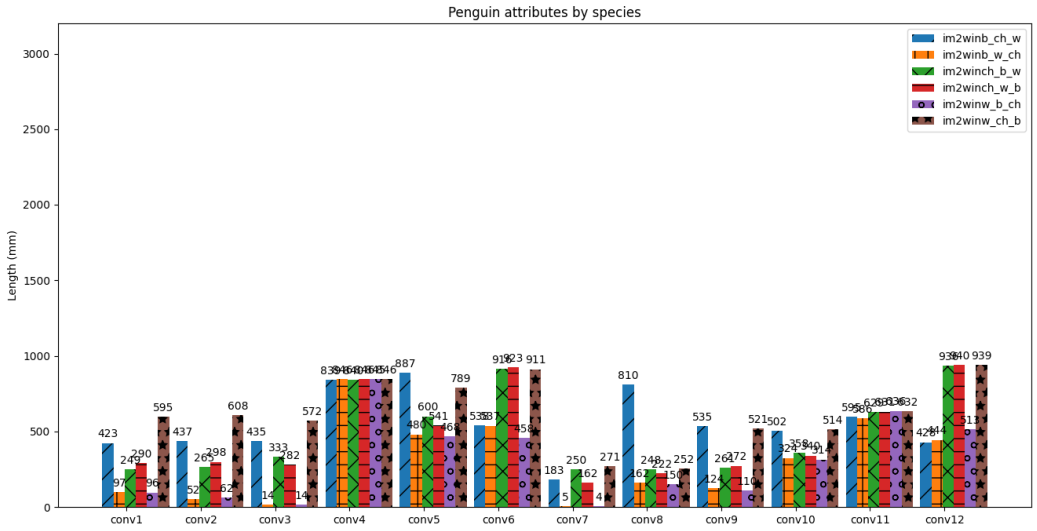

图 10: ch

融合 ch 这两个维度效果就并不是很好。

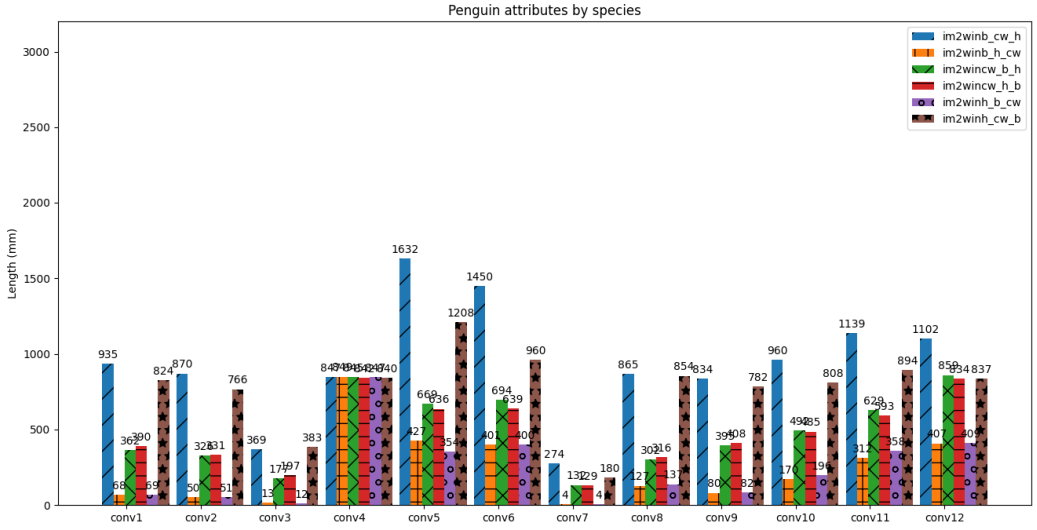

图 11: cw

融合 cw 这两个维度效在 conv1 上面的效果还行，但是相比 bh 和 bw 不太行。

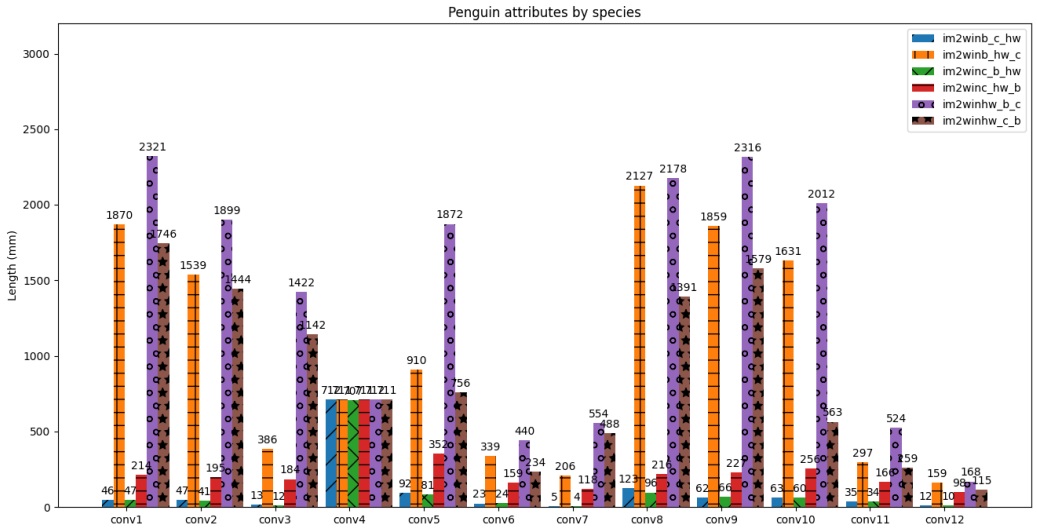

图 12: hw

融合 cw 在 conv 1 2 3 5 8 9 10 上可以取得很好的效果

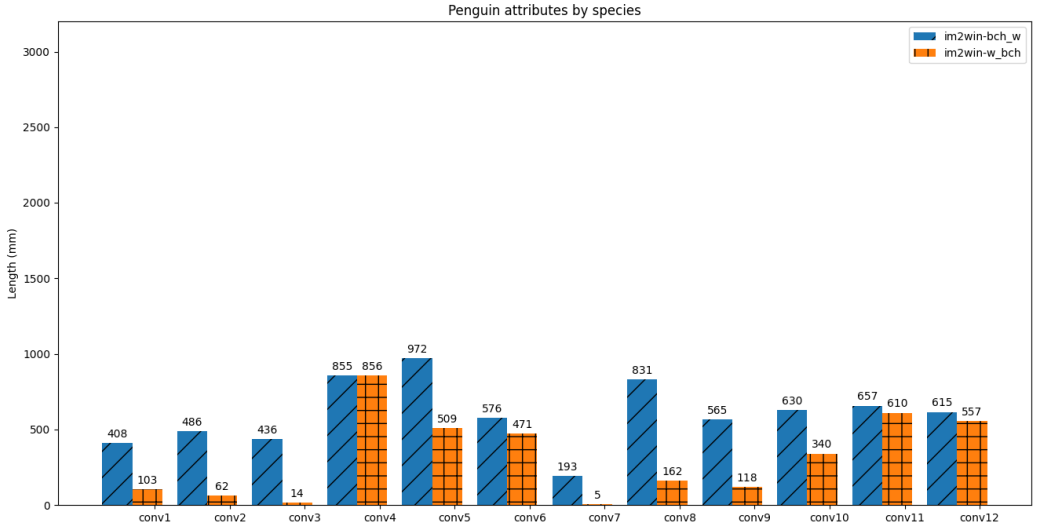

图 13: bch

效果并不是很好

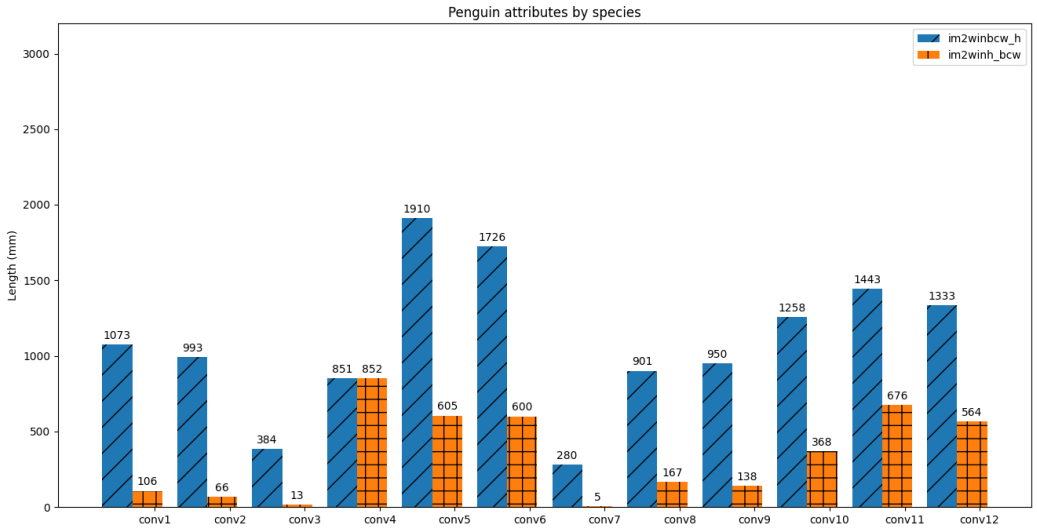

图 14: bcw
效果在某些维度上还可以

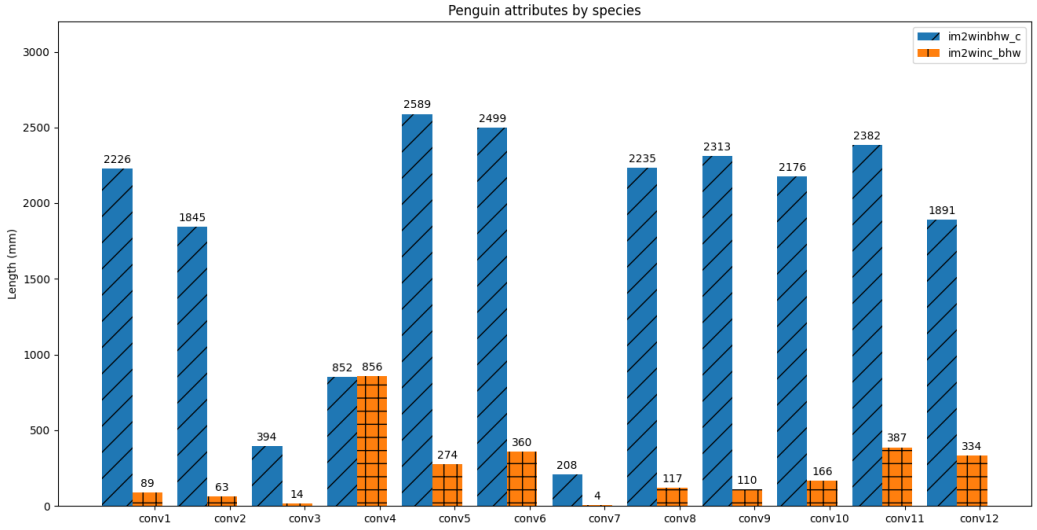

图 15: bhw
效果和 bh 融合的差不多

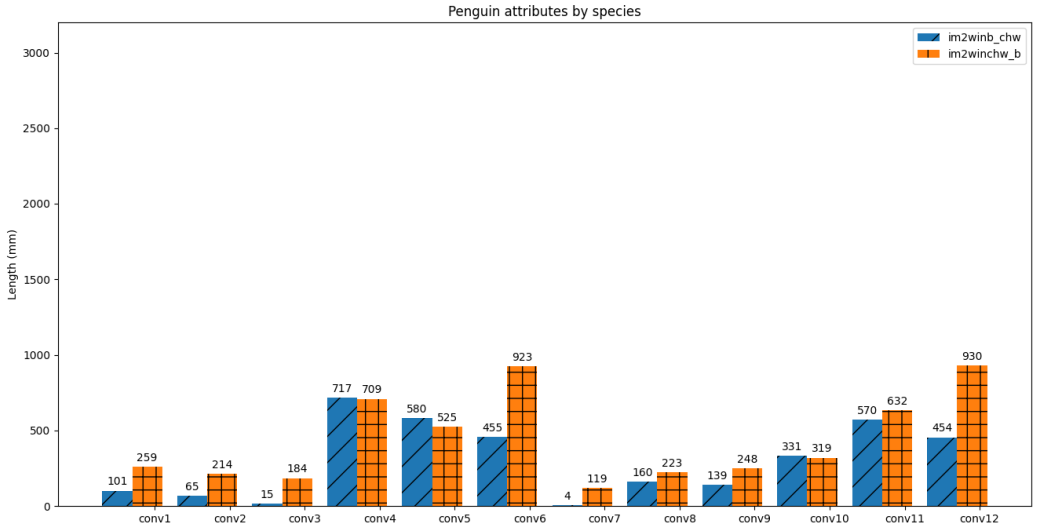

图 16: bch
效果很差

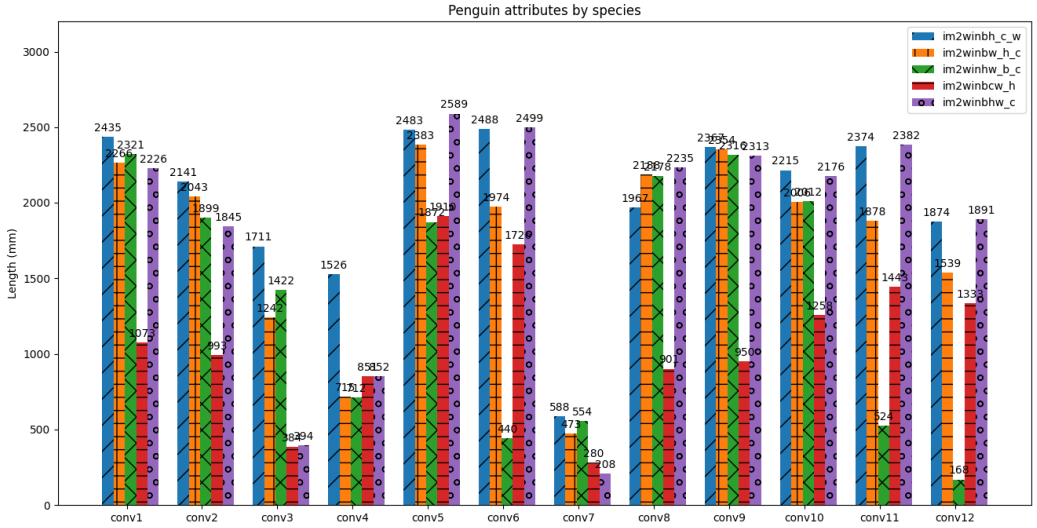

图 17: compare
效果较好的比较，可以看到在 bh_c_w, 和 bhw_c 这两种融合方式比较好。