中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009915022 姓名：栗祺菁 专业：计算机科学与技术

实验序号：5.3 实验名称：深度学习算法及硬件加速

一、 逻辑电路结构与仿真波形的截图及说明

阶段一：卷积神经网络核心算法

2D卷积：

in_size表示输入图像的大小（宽乘高）

filter_size表示一个权重值矩阵的大小（宽乘高，再加1 ，第一位存储的是该组权重值的bias）

out_size表示一个输出特征图的大小（宽乘高）

按照PPT上的模板写，先对输出特征图进行循环。然后里面是对输入图像的循环，从第零个输入图开始，先将bias值加到对应输出图像的（x，y）位置上，对于后面的图跳过权重值矩阵的第零位，按照一维卷积的算法进行累加。

在没有pad的情况下，baseX=x乘以stride，x对应的是输出矩阵的行，表示输出特征图的（x,y）位置对应的输入矩阵的运算核的x起始地址。同理baseY=y乘以stride，表示输入图像中运算核的y起始地址。若有pad，因为pad是在原有的图像的四周围上一圈0，则将起始地址在原有的基础上减去一个pad即可。相当于第一列对应的y为-1，第一行x=-1，最后一列y=input_fm_h，最后一行x=input_fm_w。在取数的时候判断一下，如果超出了图像的边界，则直接将乘数temp当做0。

运算核的大小和weight矩阵的大小相同，每次按位相乘之后，将两个的地址都累加1即可（input地址的累加由kx,ky的累加实现），这样直到进行完filter_size-1次运算（kx,ky分别从0遍历到weight_size.d3、weight_size.d2），input的运算核进行stride大小的移位，再重复上述运算。

input_offset由baseX、baseY以及ni的值按照行优先存储的方式，确定对于Input图像进行运算的起始地址相对于原起始地址in的偏移量。

output_offset由x，y，以及no的值按照行优先存储的方式，确定当前计算的输出特征图（x，y）的地址相对原起始地址out的偏移量。

weight_offset表示当前输出特征图和输入特征图对应的权重值矩阵的第一个权重的地址（即除去了0位的bias），可以知道，每一个输出图对应一组权重值，这一组中的一个权重值矩阵又对应一个输入矩阵。

累加的结果先用一个int型变量ans存储(因为mul函数是将两个short型变量相乘得到一个32位的int结果)，然后在最后将ans右移10位（即省去乘法结果的低10位小数位），转化为short型之后加到out相应地址。

```
for(no=0;no<mul(conv_size.d1,conv_size.d0);no++){
    for(ni=0;ni<rd_size.d1;ni++){
        for(y=0;y<conv_size.d2;y++)
            for(x=0;x<conv_size.d3;x++){

                int baseX=mul(x,stride)-pad;
                int baseY=mul(y,stride)-pad;
```

```
                input_offset=mul(ni,in_size)+mul(baseY,input_fm_w)+baseX;
                output_offset=mul(no,out_size)+mul(y,conv_size.d3)+x;
                int weight_offset=
                mul(no,mul(weight_size.d1,filter_size))+mul(ni,filter_size)+1;
                int ans=0;

                if(ni==0)
                    *(out+output_offset)=*(weight+weight_offset-1);

                for(ky=0;ky<weight_size.d2;ky++)
                    for(kx=0;kx<weight_size.d3;kx++,weight_offset++){
                        short temp
            =((baseY+ky)<0||(baseY+ky)>=input_fm_h||(baseX+kx)<0||
(baseX+kx)>=input_fm_w)?
                        0:*(in+mul(ky,input_fm_w)+kx+input_offset);
                        ans+=mul(temp,*(weight+weight_offset));
                    }
                    *(out+output_offset)+=(short)(ans>>FRAC_BIT);
            }
        }
}
```

池化：

in_size等于input_fm_h乘以input_fm_w，即输入的特征输出图像的大小。out_size等于pool_out_h乘以pool_out_w，即输出的池化之后的结果矩阵的大小。kern_size=KERN_ATTR_POOL_KERN_SIZE,即要池化的各个小模块的大小。在长、宽均为kern_size的小模块中选择最大值。

池化逻辑比卷积简单，按照stride和x,y选择要池化的小模块的初始地址，按照kern_size决定ix,iy的循环上界。遍历小模块，找到最大值，然后将最大值赋值给*(out+output_offset)，每次赋值后，output_offset就加加一次。仍然采用卷积中一样的策略：先根据x,y,ni,no决定大的offset，再由ix,iy决定具体取数的地址。

对于有pad的情况，在超出边界的时候，不用比较值和max的大小，跳过该位置就可以了。注意：不能将它赋值为0，因为存的数有是负数的情况，而0比负数大，最后比较得出的max的结果就不对了。

```
for(no=0;no<mul(wr_size.d1,wr_size.d0);no++){
        for(x=0;x<pool_out_h;x++)
            for(y=0;y<pool_out_w;y++){
                int baseX=mul(x,stride);
                int baseY=mul(y,stride);
                input_offset=mul(no,in_size)+mul(baseX,input_fm_w)+baseY;
                output_offset=mul(no,out_size)+mul(x,pool_out_w)+y;
                max=*(out+input_offset);
                for(ix=0;ix<kern_size;ix++)
                    for(iy=0;iy<kern_size;iy++){

                        if((ix+baseX)<0 || (ix+baseX)>=input_fm_w || (iy+baseY)
<0 || (iy+baseY)>=input_fm_h) ;
                        else{
                            comp=*(out+input_offset+mul(ix,input_fm_w)+iy);
                            if(comp>max){
                                max=comp;
                            }
                        }
```

```
                    }
                *(out+output_offset)=max;
            }

    }
```

阶段二：booth乘法器

先考虑cpu加上Mul指令后的变化:

我Mul指令的乘法操作是在EX阶段进行。相比之前的cpu，有几个地方需要更改：

1.状态机：乘法操作一个周期内完成不了，所以状态机由EX转到WB的条件还需要加上一种情况，就是如果是MUL指令的话，需要在done拉高的时候，再更改next_state为WB，否则就停留在EX阶段。

2.rd_write加一个MUL指令的情况。

3.RF_wen信号增加一个MUL指令的情况。

4.RF_wdata加一个mul_result。

5.实例化mul部件，run信号在EX阶段，并且当前是MUL指令的话拉高。A和B直接用之前Reg file中读出的reg_read_data_r值就可以。

逻辑比较简单，不放代码了。

然后看Booth乘法器的实现：

设计一个三段式状态机，用四个状态，分别是INIT，ADD，SHIFT，OUTPUT。INIT表示是初始状态，即还没有用到这个booth乘法器（不是MUL指令或者是MUL指令但是没有到EX阶段，即run信号没有拉高时）。ADD顾名思义，实现了加被乘数（或被乘数的补码）的操作。SHIFT，实现移位操作。ADD的下一个状态一定是SHIFT（最后一次加法也让它移位），所以实际上可以将ADD和SHIFT合并为一个状态。OUTPUT，输出结果，并将done信号拉高。

按照Booth乘法器的原理，乘数A需要有两位符号位，被乘数最后需要加一位0，即都是33位。为了方便移位运算，将累加和移位的变量都设置为66位，对于A和-A，将后33位设置为0，对于B将前33位设置为0。sum_reg存的是相加的结果，p_reg存的是相加后再移位的结果。每进行完一次加法和移位操作，cnt变量加1，加到32的时候，SHIFT状态的next_state变为OUTPUT状态，否则就停留在SHIFT阶段。

三段式状态机：

其中的第三段，根据当前处于何种状态决定进行什么操作，我直接在一个always块里面写了。这个当然可以在一个always块里面只写一个信号，但是我觉得信号比较多，像INIT阶段的初始化就有7个信号，而且这是一个连贯性很强的流程（开始乘法，加法，移位，……输出），放在一起写，感觉可读性比较强，也比较好看（希望老师不要扣我分）。

因为我最后一次加法也移了位，所以在取64位的结果product的时候，要用p_reg的【64:1】位，如果最后一次不移位的话，就取【65:2】位。

按照MUL指令的要求，取低product的低32位作为输出result的值。

下面是状态机的逻辑。

```
parameter INIT=4'b0001,
          ADD=4'b0010,
          SHIFT=4'b0100,
          OUTPUT=4'b0100;
```

```
always @(posedge clk)begin
    if(rst) current_state<=INIT;
    else current_state<=next_state;
end

always @(*)begin
    case(current_state)
        INIT:if(run) next_state=ADD;
            else next_state=INIT;
        ADD:next_state=SHIFT;
        SHIFT:if(cnt==32) next_state=OUTPUT;
            else next_state=ADD;
        OUTPUT:next_state=INIT;
        default:next_state=current_state;
    endcase
end

always @(posedge clk)begin
    if(rst)begin
        {a_reg,s_reg,p_reg,sum_reg,product,cnt,done}<=335'b0;
    end
    else begin
        if(current_state[0])begin//INIT
            a_reg<={A[31],A,{33{1'b0}}};
            s_reg<={A_neg[31],A_neg,{33{1'b0}}};
            p_reg<={{33{1'b0}},B,1'b0};
            cnt<=0;
            done<=1'b0;
        end
        else if(current_state[1])begin//ADD
            if(p_reg[1:0]==2'b01)
                sum_reg<=p_reg+a_reg;//add
            else if(p_reg[1:0]==2'b10)
                sum_reg<=p_reg+s_reg;//sub
            else if(p_reg[1:0]==2'b00)
                sum_reg<=p_reg;
            else if(p_reg[1:0]==2'b11)
                sum_reg<=p_reg;
            cnt<=cnt+1;
        end
        else if(current_state[2])begin//SHIFT
            p_reg<={sum_reg[65],sum_reg[65:1]};
        end
        else if(current_state[2])begin//OUTPUT
            product<=p_reg[64:1];
            done<=1'b1;
        end

    end
end
```

阶段三：硬件加速器控制软件流程

和prj3的第三阶段差不多。将addr设置为char*类型的指针，这样每次加1，就是地址加上一个字节的大小。DONE寄存器第0位没有拉高时，就一直处于while循环内，相当于等待。逻辑比较简单，不再赘述了。

```
    volatile  char* addr=(void *)0x40040000;
    *(addr+HW_ACC_START)=1;
    while(!(*(addr+HW_ACC_DONE) & 1)) ;
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如RTL代码中出现的逻辑bug，仿真、云平台调试过程中的难点等）

实验很简单，但是写的却不是很顺畅。

首先是在第一阶段的卷积和池化的部分，最后赋值的时候没有将结果右移FRAC_BIT位，再转化为short类型，而是直接转化为short类型。这个错误一直没发现，这个实验又不好debug，只能肉眼看，浪费了很多时间没有找到bug，最后才发现这点。两个16位的short型数据相乘之后，小数位也翻倍了，为了满足定点数存储的要求，应该取前10位小数，然后再取小数点左侧6位（5位整数位，1位符号位）。即将结果先右移10位，再转化为short型即可满足此要求。

然后就是在实现Booth乘法器的时候，尽管加上了向0C地址写0的操作，但是它的行为仿真还是一直超时，说明它都没有执行到写0的语句。仿真超时，也没法下波形来看，又是只能肉眼检查。一遍遍检查Booth的逻辑和cpu的逻辑，都没发现有错，最后，采用控制变量法，将cpu的关于mul的语句，一条条加，最终发现了错误。本来应该是RF_wen=（xxx || xxx || mul）&& current_state[8]，结果我将mul写在了括号外面，变成了（xxx || xxx ）||mul && current_state[8]，可想而知，逻辑就完全不对了，显然是一个粗心大意的错误。之前一直没发现这个错误，一是因为我一直觉得错误在于，第一次碰到乘法指令，done就没有拉高，所以才一直处于等待状态。然后我就一直找booth的错误，没太关注cpu。另外一个原因是我cpu的RF_wen语句括号里面有括号，一层又一层，看起来很不方便，不容易发现错误。

还有一个问题就是在写卷积和池化的时候，offset用的unsigned型变量，一是它本来就不对，例如baseX=stride*x-pad可能是赋值，不能将baseX变成unsigned型，但是因为测试案例中没有pad，所以也顺利通过了第一阶段。二是因为两个unsigned的数相乘会编译出Mult指令，这也是应该避免的。

以上两个问题耗费了我很多时间，但也没有通过这两个Bug学到什么有用的知识，因为这基本都是可以避免的低级错误，纯粹是浪费了时间。下次一定要想明白再写，写的时候就认真一点，不然有些小错误，之后真的很难发现。

三、 对讲义中思考题（如有）的理解和回答

四、 在课后，你花费了大约_*8*小时完成此次实验。

五、 对于此次实验的心得、感受和建议

实验比较简单，但是非常不好调试，因为一点小问题debug了很久。这个实验没有设置有pad的测试案例，没法检验有pad的情况我的卷积和池化有没有问题，不过老师大概也就是想通过这个实验让我们了解一下卷积和池化的概念的吧，案例宽松一点也行。感谢助教王嵩岳和楼持恒同学对我的帮助。