中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　专业：计算机科学与技术

实验序号：5.1　　实验名称：深度学习算法与硬件加速器

一．实验目的

1. 软件部分：实现深度学习的核心算法——2D 卷积+池化；
2. 硬件部分：为 RISCV 定制处理器添加对乘法指令的支持；
3. 软件部分：编写加速器基本控制软件（I/O 地址空间访问），测试处理器对硬件加速器访问。（实验项目已经提供了集成核心算法的硬件加速器）

二．实验背景知识

深度学习

深度学习（DL，Deep Learning）是机器学习（ML，Machine Learning）领域中一个新的研究方向。它的最终目标是让机器能够像人一样具有分析学习能力，能够识别文字、图像和声音等数据

目前，作为深度学习的代表算法之一，卷积神经网络在计算机视觉、分类等领域上，都取得了当前最好的效果。后来，基于深度神经网络和搜索树的智能机器人“AlphaGo”在围棋上击败了人类，这是 CNN 给人们的一个大大的惊喜。一年后的 Master 则更是完虐了所有人类围棋高手，达到神一般的境界，人类棋手毫无胜机。

可以说，卷积神经网络是深度学习算法应用最成功的领域之一。

卷积神经网络（Convolutional Neural Networks，CNN）**是一种专门用于处理图像、语音等数据的神经网络**。CNN 基于卷积操作来提取输入数据中的特征，这些特征可以被用于分类、检测、分割等任务。

卷积、池化算法

卷积、池化是深度学习中常用的一种特征提取方法，主要用于**图像处理**任务。

卷积操作是通过**滑动一个滤波器**（也称为**卷积核**）在图像上进行局部相乘累加的过程，用于提取图像的特征。**卷积操作可以捕捉到图像的局部细节信息**，并且具有平移不变性。

池化操作是对卷积后的特征图进行**降采样**的过程，通过取特定区域内的最大值（最大池化）或者平均值（平均池化）来减少特征图的尺寸。池化操作可以减少特征图的维度，提高计算效率，并且具有一定的平移不变性和部分尺度不变性。

卷积池化算法的主要步骤如下：

1. 定义卷积核的大小和数量。
2. 对输入图像进行卷积操作，得到卷积特征图。
3. 对卷积特征图进行池化操作，得到池化特征图。
4. 可选地，可以多次重复步骤 2 和步骤 3，以提取更高级的特征。
5. 将最终的池化特征图输入到分类器或者其他任务模型中进行训练或者预测。

对于网络结构而言，上面的层看下面的层经过 pooling 后传上来的特征图，就好像在太空上俯瞰地球，看到的只有山脊和雪峰。这即是对特征进行宏观上的进一步抽象。

那么为什么需要进行抽象呢？

因为：经过池化后，得到的是概要统计特征。它们不仅具有低得多的维度（相比使用所有提取得到的特征），同时还会改善结果(不容易过拟合）。

max_pooling：夜晚的地球俯瞰图（如下面的左图），灯光耀眼的穿透性让人们只注意到最 max 的部分，产生亮光区域被放大的视觉错觉。故而 max_pooling 对较抽象一点的特征（如纹理）提取更好。

average_pooling：白天的地球俯瞰图（如下面的右图），幅员辽阔的地球表面，仿佛被经过了二次插值的缩小，所有看到的都是像素点取平均的结果。故而 average_pooling 对较形象的特征（如背景信息）保留更好。

（以上的资料来自于 chatGPT 以及 CSDN 的我喝酸奶不舔盖）

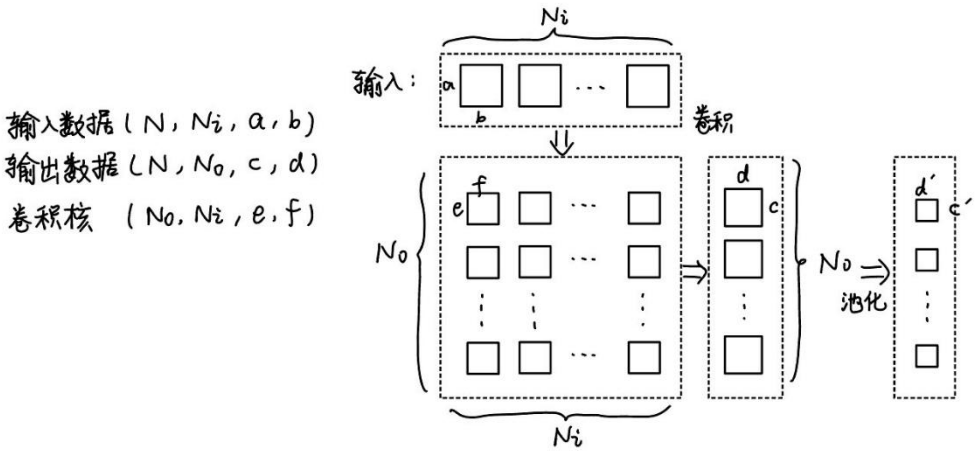
三．实验过程

第一阶段：实现 2D 卷积+池化算法

conv.c 中的一些宏定义的涵义如下所示：

```
#define FRAC_BIT 10                 // 数据中小数部分的有效位数

#define RD_ADDR 135106448           // 输入数据起始地址
#define RD_SIZE_D0 1                // 输入数据组数
#define RD_SIZE_D1 1                // 每组输入数据的通道数
#define RD_SIZE_D2 28               // 每个通道的行数
#define RD_SIZE_D3 28               // 每个通道的列数

#define WEIGHT_ADDR 134217728       // 卷积核起始地址
#define WEIGHT_SIZE_D0 20           // 卷积核组数
#define WEIGHT_SIZE_D1 1            // 每组卷积核的个数
#define WEIGHT_SIZE_D2 5            // 卷积核的行数
#define WEIGHT_SIZE_D3 5            // 卷积核的列数

#define WR_ADDR 135108240           // 输出数据起始地址
#define WR_SIZE_D0 1                // 输出数据组数
#define WR_SIZE_D1 20               // 每组输出数据的通道数
#define WR_SIZE_D2 12               // 每个通道的行数
#define WR_SIZE_D3 12               // 每个通道的列数

#define KERN_ATTR_CONV_PAD 0        // 卷积边界宽度
#define KERN_ATTR_CONV_STRIDE 1     // 卷积步长
#define KERN_ATTR_POOL_PAD 0        // 池化边界宽度
#define KERN_ATTR_POOL_KERN_SIZE 2  // 池化块大小
#define KERN_ATTR_POOL_STRIDE 2     // 池化步长
```

对于 2D 卷积，输入矩阵以及输出矩阵都有 4 个参数，用(d0,d1,d2,d3)表示，d0 代表数据的组数，d1 代表每组数据的通道数，d2，d3 代表每个通道的矩阵的行数和列数。每一组输入数据对应着一组输出数据，所以如果输入数据有 N 组，则输出数据也应该是 N 组；对于某一组输入和输出数据，输入数据通过与卷积核的卷积产生输出数据，输出数据再经过池化产生最终结果，其简化流程如下图所示。



从图中可以看出，一个输出通道的矩阵等于所有输入通道的矩阵与对应位置的卷积核的卷积结果之和。所以卷积核矩阵的行数就等于输出的通道数，卷积核矩阵的列数就等于输入的通道数。同时一个输出矩阵的大小可以由一个输入矩阵的大小和一个卷积核的大小决定（以及边界宽度和步长）。

由此，就可以得到卷积函数 convolution()的框架结构如下：

```
void convolution()
{
    // 输入数据、卷积核以及输出数据的基地址
    short *in = (short *)addr.rd_addr;
    short *weight = (short *)addr.weight_addr;
    short *out = (short *)addr.wr_addr;

    // 首先计算卷积输出矩阵的大小
    unsigned pad = KERN_ATTR_CONV_PAD;
    unsigned pad_len = pad << 1;

    unsigned extended_in_w = rd_size.d3 + pad_len;
    unsigned extended_in_h = rd_size.d2 + pad_len;

    unsigned conv_out_w = extended_in_w - weight_size.d3;
    unsigned conv_out_h = extended_in_h - weight_size.d2;

    unsigned stride = KERN_ATTR_CONV_STRIDE;

    conv_out_w = div(conv_out_w, stride);
    conv_out_h = div(conv_out_h, stride);

    conv_out_w++;
    conv_out_h++;

    conv_size.d0 = wr_size.d0;
    conv_size.d1 = wr_size.d1;
    conv_size.d2 = conv_out_h;
    conv_size.d3 = conv_out_w;

    //wr_size=(1,20,12,12)
    //rd_size=(1,1,28,28)
    //weight_size=(20,1,5,5)
```

```
    short input_size = (short)mul((short)rd_size.d2,(short)rd_size.d3);// 一个输入矩阵的大小
    short core_size = (short)(mul((short)weight_size.d2,(short)weight_size.d3)+1);// 一个卷积核的大小

    unsigned na; // 当前输入数据组数
    unsigned no, ni;// 当前输入、输出通道数
    unsigned x, y;// 当前输出数据在conv矩阵中的行数和列数

    short* input_base_1;// 当前使用的输入矩阵所在组的起始地址(input_base_1 = in + na * rd_size.d1 * input_size)
    short* input_base_2;// 当前使用的输入矩阵的起始地址(input_base_2 = input_base_1 + ni * input_size)
    short* core_base_1;// 当前使用的卷积核所在行的起始地址(core_base_1 = out + no * weight_size.d1 * core_size)
    short* core_base_2;// 当前使用的卷积核的起始地址(core_base_2 = core_base_1 + ni * core_size)

    unsigned kx, ky;// 当前乘法所用的权重值在卷积核矩阵中的行和列
    int temp_data;// 暂存乘法与加法的中间结果
    unsigned head_line, head_column;// 在扩展了边界后的输入矩阵中与weight相乘的一块的起始行与列
    unsigned real_line, real_column;// 在实际输入矩阵中与weight相乘的一块的起始行与列
    short* input_target_cell, * weight_target_cell, * output_target_cell;
    // 输入指针指向当前乘法所用的数据位置；权重指针指向当前乘法所用的权重值的位置；输出指针指向当前输出数据的位置

    output_target_cell = out;// 输出指针每次加1，顺序地计算各个卷积输出矩阵的每一个单元

    for(na = 0, input_base_1 = in; na < rd_size.d0; na++, input_base_1 += mul((short)rd_size.d1,input_size)){
        for(no = 0, core_base_1 = weight; no < wr_size.d1; no++, core_base_1 += mul((short)weight_size.d1,core_size)){
            for(x = 0, head_line = 0; x < conv_out_h; x++){
                if(head_line < pad)
                    real_line = 0;
                else if(head_line >= extended_in_h - pad)
                    real_line = rd_size.d2 - 1;
                else
                    real_line = head_line - pad;
                for(y = 0, head_column = 0; y < conv_out_w; y++){
                    if(head_column < pad)
                        real_column = 0;
                    else if(head_column >= extended_in_w - pad)
                        real_column = rd_size.d3 - 1;
                    else
                        real_column = head_column - pad;

                    core_base_2 = core_base_1;
                    input_base_2 = input_base_1;
                    for(ni = 0; ni < rd_size.d1; ni++, input_base_2 += input_size, core_base_2 += core_size){
                        // bias值的处理
                        weight_target_cell = core_base_2;
                        if(ni == 0)
                            *output_target_cell = *weight_target_cell;// 先加上bias值
                        // 卷积的处理
                        weight_target_cell ++;// 移动到卷积核的权重值区域
                        for(kx = 0, temp_data = 0; kx < weight_size.d2; kx++){
                            // 将输入指针指向输入中下一行的开始位置
                            input_target_cell = input_base_2 + mul((short)(real_line + kx),(short)rd_size.d3) + real_column;

                            for(ky = 0; ky < weight_size.d3; ky++, weight_target_cell++){
                                if(!(head_line + kx < pad||
                                     head_column + ky < pad||
                                     head_line + kx >= extended_in_h - pad||
                                     head_column + ky >= extended_in_w - pad))
                                {
                                    temp_data += mul(*input_target_cell,*weight_target_cell);
                                    input_target_cell++;
                                }
                            }
                        }
                        temp_data = temp_data >> FRAC_BIT;
                        *output_target_cell += (short)temp_data; // 将temp_data加到输出矩阵对应的位置上
                    }
                    head_column += stride; // 起始列数加上步长
                    output_target_cell++;
                }
                head_column = 0; // 起始列数归零
                head_line += stride; // 起始行数加上步长
            }
        }
    }
}
```

卷积函数最麻烦的地方在于它的循环控制。在写的时候需要考虑清楚需要几层循环，各层循环之间的嵌套关系应该是什么，各个循环变量何时初始化、何时“+1”，以及如何将指针指到目标矩阵的目标单元。这里我是依次计算各个卷积输出矩阵的各个单元，每计算完一个单元将输出指针“+1”。因此第一层循环是对输出数据组数遍历，第二层循环是对输出数据通道数遍历，第三、第四层循环是对输出矩阵的行与列遍历。

接下来，为了计算输出中每一个单元的值，需要将每一个通道的输入矩阵与对应位置的卷积核进行卷积并累加起来。所以第五层循环是对输入数据的通道数进行遍历，而第六、第七层循环是对一次卷积的行与列进行遍历。

为了减少计算地址时的乘法次数，我添加了一些中间变量(input_base_1,input_base_2,core_base_1,core_base_2 等)，用来存储每次内层循环开始时的基地址。这样理论上就可以始终通过加法来计算输入指针和卷积指针，而完全避免乘法。但我在输入指针每次换行时，还是使用乘法来重新定位到下一行的开头，主要是因为边界宽度的存在，导致换行时输入指针的变化量不确定，会比较复杂。

每一次将输入矩阵中的一块与对应的卷积核卷积后，都需要先将结果暂存到 int 型的 temp_data 中，因为两个 short 型(16 位)的数相乘，其结果很可能超过 16 位，但不会超过 32 位，所以保存在 temp_data 中不会因为乘法而丢失信息。

将每一次乘法结果都累加到 temp_data 中，直到一次卷积完成后，再将结果转化为 short 类型。因为我们使用了 16 位定点数来模拟浮点数，在 short 型的 16 位中，低 10 位代表小数部分，高 6 位代表整数部分。所以经过乘法后，在 temp_data 的 32 位中，应该是低 20 位代表小数部分，剩余的高位代表整数部分。由于我们只要取 10 位小数精度，所以可以将 temp_data 的低 10 位舍去，这就是 temp_data = temp_data >> 10 的含义。然后再将 temp_data 强制类型转换为 short 型，即直接取它的低 16 位，这样在不溢出的情况下就保持了数据类型和精度的一致性。当然，如果乘法结果比较大，在强制类型转换时还是可能发生溢出错误的。

池化过程相对来说会更清晰简单一些。池化函数需要对每一个卷积输出矩阵进行池化操作，只保留每一个池化块中的最大值。同时池化输出地址与卷积输出地址相同，所以池化的结果会覆盖卷积的结果。

池化函数 pooling()如下：

```
void pooling()
{
    short *out = (short *)addr.wr_addr;
    unsigned pad = KERN_ATTR_POOL_PAD;
    unsigned pad_len = pad << 1;

    unsigned pad_w_test = conv_size.d3 - KERN_ATTR_POOL_KERN_SIZE;
    unsigned pad_h_test = conv_size.d2 - KERN_ATTR_POOL_KERN_SIZE;

    unsigned pool_out_w = pad_w_test + pad_len;
    unsigned pool_out_h = pad_h_test + pad_len;

    unsigned stride = KERN_ATTR_POOL_STRIDE;

    unsigned pad_w_test_remain = pad_w_test - mul(div(pad_w_test, stride), stride);
    unsigned pad_h_test_remain = pad_h_test - mul(div(pad_h_test, stride), stride);

    pool_out_w = div(pool_out_w, stride);
    pool_out_h = div(pool_out_h, stride);
    pool_out_w++;
    pool_out_h++;

    if ((!pad) && (pad_w_test_remain || pad_h_test_remain))
    {
        pool_out_w++;
        pool_out_h++;
    }
```

```
    short conv_core_size = (short)mul((short)conv_size.d2,(short)conv_size.d3); // 一个卷积矩阵的大小

    unsigned na, ni;                // 当前输入conv矩阵的数据组数和通道数(na为0~conv_size.d0;ni为0~conv_size.d1)
    unsigned x, y;                  // 当前输出矩阵位置的行和列(x为0~pool_out_h;y为0~pool_out_w)

    short* conv_base;               // 当前卷积矩阵的起始地址(conv_base=out+na*conv_size.d1*conv_core_size+ni*conv_core_size)

    unsigned kx, ky;                // 当前正在池化的一块的行、列的循环变量(kx,ky的取值为0~KERN_ATTR_POOL_KERN_SIZE)
    short temp_max;                 // 临时变量暂存池化快的最大值

    unsigned head_line, head_column;// 在边界扩展后的conv矩阵中正在池化的一块的起始行与列
    // (head_line的取值为0~conv_size.d2+pad_len;head_column的取值为0~conv_size.d3+pad_len)
    unsigned real_line, real_column;// 在实际conv矩阵中正在池化的一块的起始行与列
    // (real_line的取值为0~conv_size.d2;real_column的取值为0~conv_size.d3)

    short* conv_traget_cell, * output_target_cell;// 分别指向当前conv矩阵的目标位置和池化输出矩阵的目标位置


    output_target_cell = out;// 输出指针每次加1，顺序地计算各个池化输出矩阵的每一个单元

    for(na = 0, conv_base = out; na < conv_size.d0; na++){
        for(ni = 0; ni < conv_size.d1; ni++, conv_base += conv_core_size){
            for(x = 0, head_line = 0; x < pool_out_h; x++, head_line += stride){
                if(head_line < pad) // 池化快相对于边界padding的3种情况
                    real_line = 0;
                else if(head_line >= conv_size.d2 + pad)
                    real_line = conv_size.d2 - 1;
                else
                    real_line = head_line - pad;
                for(y = 0, head_column = 0; y < pool_out_w; y++, head_column += stride){
                    if(head_column < pad) // 池化快相对于边界padding的3种情况
                        real_column = 0;
                    else if(head_column >= conv_size.d3 + pad)
                        real_column = conv_size.d3 - 1;
                    else
                        real_column = head_column - pad;

                    temp_max = SHORT_MIN; // 池化快的极大值的初值设为最小
                    for(kx = 0; kx < KERN_ATTR_POOL_KERN_SIZE; kx++){
                        conv_traget_cell = conv_base + mul((short)(real_line + kx),(short)conv_size.d3) + real_column;
                        for(ky = 0; ky < KERN_ATTR_POOL_KERN_SIZE; ky++){
                            if(!(head_line+kx < pad||
                                 head_column+ky < pad||
                                 head_line+kx >= conv_size.d2+pad||
                                 head_column+ky >= conv_size.d3+pad))
                            {
                                //printf("(%d,%d,%x)\n",head_line,head_column,*conv_traget_cell);
                                temp_max = MAX(*conv_traget_cell,temp_max);
                                conv_traget_cell++;
                            }
                            else {
                                temp_max = MAX(0,temp_max);
                            }
                        }
                    }
                    //printf("max=%x\n",temp_max);
                    *output_target_cell = temp_max;
                    output_target_cell++;
                }
            }
        }
    }
}
```

池化函数中各层循环的嵌套与卷积函数非常类似，而且池化中也需要考虑边界（在卷积输出矩阵上添加一圈“0”），这里就不再重复了。

每一次池化中，都用一个变量 temp_max 来记录池化块中的极大值。所以将 temp_max 的初值设为可以取到的最小值。

第二阶段：在功能型处理器中添加乘法指令通路

这里我使用的是 RISCV_32 处理器实现的乘法指令。在 RISCV 中，乘除法指令是通过 M 扩展实现的。指令码的格式如下所示：

6.1 Multiplication Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

我们只需要实现乘法指令中第一条 MUL 指令。MUL 指令的操作是将两个 32 位数相乘，取结果的低 32 位。在理论课上，我们学过乘法可以通过 Bush 算法等方法实现，但是它们都需要多个时钟周期才能完成，通过多次加法和移位来实现。这样乘法指令的执行阶段就需要多个时钟周期，执行起来比较慢，而且乘法器需要添加时序控制逻辑。所以我们使用了一种“取巧”的方法：

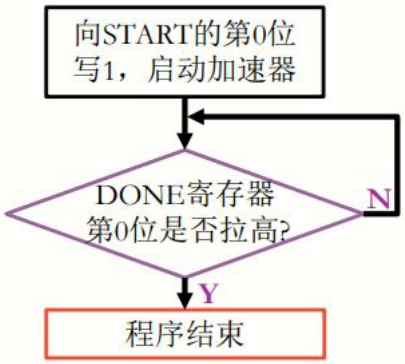
```
wire [63:0] full_multiply_result = RF_rdata1*RF_rdata2;
```

这样在硬件实现上，就不需要自己实现乘法器的内部细节，vivado 可以帮我们实现一个异步并行乘法器，在一个时钟周期内就计算出 A*B 的 64 位结果。虽然我不知道它的乘法器在硬件上是如何实现的，但这样确实又快又省事。

除了执行阶段的操作，还需要实现乘法指令的译码和写回，这些只要在已有的逻辑中添加对 MUL 指令的支持即可。

第三阶段：硬件加速器控制软件流程

对硬件加速器的控制软件流程图已经在课件中画出，如下所示：



改变 START 寄存器的第 0 位可以通过和掩码的按位操作来实现，而中间循环判断并等待的操作可以通过 while 循环来实现。控制软件 launch_hw_accel()的代码如下：

```
#ifdef USE_HW_ACCEL
void launch_hw_accel()
{
    volatile int* gpio_start = (void*)(GPIO_START_ADDR);
    volatile int* gpio_done = (void*)(GPIO_DONE_ADDR);

    //TODO: Please add your implementation here
    *gpio_start = *gpio_start | 1;// 将START寄存器的第0位置为1，启动加速器
    while(*gpio_done & 1);
    *gpio_start = *gpio_start ^ 1;// 将START寄存器的第0位置为0
}
#endif
```

四．实验总结

1. 性能分析

在 fpga_run 阶段总共有三个测试程序，它们分别使用不同的宏，将 conv.c 编译成不同的可执行程序。下表是用性能计数器统计到的三个测试程序执行时 CPU 的一些数据。

测试程序	时钟周期数（clk cycle）	指令周期数（inst cycle）	访存次数（load/store 指令数）
hw_conv	6051604	69842	11801
sw_conv	2895911106	282105208	1130241
sw_conv_mul	385767425	4734620	724134

由此可以计算出各个测试程序执行过程中的 CPI 和访存指令的比例，如下表。

测试程序	CPI	访存指令比例
hw_conv	86.65	16.90%
sw_conv	10.27	0.40%
sw_conv_mul	81.48	15.29%

可以看到，hw_conv 是使用卷积池化硬件加速器后跑的 conv.c，所以它的速度最快（实际用时是 69.85ms），而且指令周期也最少，访存次数也最少。

sw_conv_mul 是使用支持乘法指令的 CPU 跑的 conv.c，它的速度明显要比硬件加速器慢很多（实际用时是 3868.31ms），但是 CPI 和访存指令的比例与 hw_conv 差不多。

而 sw_conv 是使用不使用乘法指令跑出的 conv.c，它的速度就更慢了（实际用时是 200770.65ms）。因为不能使用乘法指令，要实现卷积过程中密集的乘法，它就需要用多次的加法和移位来组合模拟乘法操作。相比 sw_conv_mul，它多出来的开销应该就是模拟乘法所多用的时间。而且它的 CPI 要明显小于另外两个测试程序，它的访存指令比例也明显低于其他两个指令，这也说明了一条乘法指令需要用非常多的加法和移位指令来组合完成，它增加的加法和移位指令使得原本访存指令的比例下降了，而且也让 CPI 减小了（可能是因为加法和移位指令的平均 CPI 比较小）。

通过性能比较与分析，可以发现乘法器（以及乘法指令）对于卷积池化的效率提升非常关键，而且卷积池化硬件加速器更是可以显著得提升性能。不过这里的加速器是内置的，不需要我们来设计实现（估计比较复杂吧），所以我们还不知道它的实现原理和方法。

2. 调试过程总结

这次实验主要的工作量在于深度学习中卷积核池化函数的实现，属于软件部分。但是问题在于，在 fpga_run 阶段的测试只能告诉我们出错的位置，它是将我们的卷积池化输出与金标准的输出进行对比。但是光知道最终的输出哪里错了，还是不能定位到具体的错误原因：可能是卷积错，也可能是池化错，可能是输入指针错，也可能是卷积核指针错——我们需要软件执行的中间过程和中间结果！所以在调试的过程中，我大量使用了 printf 函数，将中间结果打印出来，才最终找到了出错的原因。

有一个印象深刻错误是我被课件误导了。课件中所说的输入数据“在每行后额外增加 8 字节空白，用于将每行起始地址按 64 字节对齐”，以及卷积核“在 bias 值后会插入 6 字节空白，bias 值共占据 8 字节空间……25 个权重值后再增加 6 个字节空白”，这些是对硬件加速器可见，但是在软件算法实现时不需要考虑！一开始我在写卷积核池化函数时还专门考虑了这些空白字节，费很多时间，后来打印出来和金标准对比才发现问题，非常麻烦，所以看课件还是要认真仔细啊哇哇哇！（希望以后的课件中可以专门强调一下哈哈）

虽然是软件调试，不需要看波形，但是其实要定位错误还是挺麻烦的。因为金标准只提供了输入数据、卷积核和最终输出，没有提供卷积输出，所以就只能看自己卷积输出矩阵中有没有特别奇怪的输出。而且打印也不能次数过多，否则云平台上可能会卡半天显示不出来。

后来在验收的时候，助教叶从容老师还给我介绍了一下他做的“一生一芯”项目的毕设：ByteFloat 机制。利用这个机制，可以将卷积神经网络中密集的浮点乘法运算简化，加快运算速度。ByteFloat 机制就是说在卷积神经网络中，我们不使用 IEEE754 标准的浮点数，而是自己设计一种低精度、快运算的数据格式，名为 ByteFloat。而且降低运算的精度并不会明显降低网络的预测精度。我当时听得不是太懂。我觉得可能因为浮点运算相对于定点运算，需要先对阶，还要有专门的浮点运算器，开销比较大，所以使用 ByteFloat 机制应该可以提高卷积神经网络的性能。

虽然这次实验我们实现了一个 2D 卷积和池化函数，但是其实对于卷积神经网络如何应用、如何去工作，以及它为何能够在深度学习领域很成功，这些更宏观的东西，我还不太了解，就留给后面继续去学习吧！