中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 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

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

1） conv.c 中 void convolutional( )函数的实现

convolutional 函数用于完成卷积操作：

为了操作方便，我首先定义了三个数组指针：

```
int filter_size = mul(WEIGHT_SIZE_D2, WEIGHT_SIZE_D3) + 1;//卷积核大小

short (*input_vector)[WEIGHT_SIZE_D1][input_fm_h][input_fm_w];//输入图像地址
short (*output_vector)[WEIGHT_SIZE_D0][conv_out_h][conv_out_w];//输出图像地址
short (*filter)[WEIGHT_SIZE_D0][WEIGHT_SIZE_D1][filter_size];//卷积核地址

input_vector    = (void*)(in + input_offset);
output_vector   = (void*)(out + output_offset);
filter          = (void*)weight;//转化为void*才能赋值
```

分别对应输入图像、卷积后的输出特征图以及卷积核的数据的存储地址。

然后,利用多重 for 循环得到卷积后的输出特征图。

```
for(int no = 0;no < WEIGHT_SIZE_D0;no++){
        for(int ni = 0;ni < WEIGHT_SIZE_D1;ni++){
                for(int y = 0;y < conv_out_h;y++){
                        for(int x = 0;x < conv_out_w;x++){
                                if(ni == 0){
                                        (*output_vector)[no][y][x] = (*filter)[no][0][0]; //filter[i][0][0]存储第i个通道的bias值
                                }

                                int raw_data = 0;//用32位int类型存储中间计算结果
                                for(int ky = 0;ky < WEIGHT_SIZE_D2;ky++){
                                        for(int kx = 0;kx < WEIGHT_SIZE_D3;kx++){
                                                int current_x = mul(x, stride) - pad + kx;//当前输入的访问x坐标(宽度)
                                                int current_y = mul(y, stride) - pad + ky;//当前输入的访问y坐标(高度)

                                                if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){ //若输入的当前访
                                                        raw_data += mul(
                                                                (*input_vector)[ni][current_y][current_x],
                                                                (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                                                        );//用32位int类型存储中间计算结果
                                                }
                                        }
                                }
                                (*output_vector)[no][y][x] += raw_data >> FRAC_BIT;//由于用整数运算代替小数，所以最终结果需要移位
                        }
                }
        }
}
```

上述算法考虑了 padding，当进行操作的输入数据是在 padding，则直接跳过处理。(因为填充是 0，对卷积操作没有影响)

其中的中间数据用 int 类型储存，这是因为，两个 2 字节的 short 数据相乘，结果最高为 4 字节，且若仍然用 short 类型储存，则可能会溢出。

同时，这里的乘法运算通过对两数相乘再右移 10 位（小数位数）来实现。这是由于我们用 short 类型存储定点小数，所以相乘后的数据需要右移小数位数，才是真正的结果。

2） conv.c 中 void pooling( )函数的实现

pooling 函数用于完成池化操作。

和 convolutional 函数类似，为了操作方便，我定义了两个数组指针：

```
unsigned long temp_pool_store_offset = mul(WEIGHT_SIZE_D0, mul(input_fm_h, input_fm_w));//池化前数据的总大小

short (*before_pool_vector)[WEIGHT_SIZE_D0][input_fm_h][input_fm_w];//池化前的数据存储地址
short (*after_pool_vector)[WEIGHT_SIZE_D0][pool_out_h][pool_out_w];//池化后的数据临时存储地址(存储在池化前数据后面，不开新的空间)

before_pool_vector = (void*)(out + input_offset);
after_pool_vector = (void*)(out + input_offset + temp_pool_store_offset);//转化为void*才能赋值
```

值得注意的是,这里的 before_pool_vector 表示经过了卷积但未经池化的特征图数据; after_pool_vector 用于临时存储经池化后的特征图数据。

Temp_pool_store_offset 是卷积后，池化前的特征图数据的总大小，用来计算临时储存池化后特征图数据的地址。

由于 PPT 上表示不用开辟新空间，池化后特征图数据的临时存储空间来源于 out 偏移一段后的空闲空间（紧接在已被输入特征图占用的存储空间之后)。

因此，在经过池化操作后,需要将 after_pool_vector 中那些临时存储的数据搬运到输出特征图数据的开始地址，如下所示：

```
//搬运池化结果
short* temp_address = out + (input_offset + temp_pool_store_offset);//临时数据储存的地址
short* output_address = out + output_offset;//结果输出的地址

int total_num = mul(WEIGHT_SIZE_D0, mul(pool_out_h, pool_out_w));

for(int i = 0; i < total_num; i++, temp_address++, output_address++){
        *output_address = *temp_address;
}
```

其中 total_num 是需要搬运的数据总量，temp_address 是被搬运数据地址，output_address 是数据搬运到的地址，通过 for 循环完成搬运。

池化的具体操作与卷积较为类似，如下所示：

```
for(int no = 0;no < WEIGHT_SIZE_D0;no++){
        for(int y = 0;y < pool_out_h;y++){
                for(int x = 0;x < pool_out_w;x++){
                        short max = SHRT_MIN;//初始值为short类型最小值

                        for(int ky = 0;ky < KERN_ATTR_POOL_KERN_SIZE;ky++){
                                for(int kx = 0;kx < KERN_ATTR_POOL_KERN_SIZE;kx++){
                                        int current_x = mul(x, stride) - pad + kx; //当前输入的访问x坐标(宽度)
                                        int current_y = mul(y, stride) - pad + ky; //当前输入的访问y坐标(高度)

                                        if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){/
                                                if(max < (*before_pool_vector)[no][current_y][current_x]){
                                                        max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
                                                }
                                        }
                                }
                        }
                        (*after_pool_vector)[no][y][x] = max;
                }
        }
}
```

这里 max 的初始值取 SHRT_MIN，也就是 short 类型能表示的最小值。在每次采样时，若样本值大于 max 的值，则替换存储值。从而完成取最大值的操作。

上述代码也考虑了 padding，如果采样的数据在 padding 里面，则直接跳过，不做处理（这是因为最大池化算法中，padding 的数据当作负无穷，而不是 0）

最后将最大值赋值给池化后的临时存储数据，完成池化操作。

3） conv.c 中 void launch_hw_accel( )函数的实现

```
#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) |= 0x1;//START第0位写1代表加速器启动
        while(!((*gpio_done) & 0x1));//根据DONE第0位，检测是否以及完成卷积操作
        (*gpio_start) &= 0x0;//还原START
}
#endif
```

launch_hw_accel 函数用于使用硬件加速器。

gpio_start 为只写寄存器，其最低位表示加速器启动。因此，向 START 的第 0 位写 1，启动加速器。

gpio_done 为只读寄存器，其最低位表示加速器工作状态。因此，不断检测 DONE 寄存器第 0 位是否为 1 来判断加速器是否已完成卷积操作。

当加速器完成卷积操作后，向 START 的第 0 位写 0，停止加速器

4） 性能计数器的增加以及 perf_cnt.c 和 perf_cnt.h 的修改

在本次实验中，本人发现，sw_conv 的周期数会超出了能统计的最大值（即超过了 32 位 2 进制数能表示的最大值），下面是本人 sw_conv 程序中性能计数器最初的数据：

```
==============Hardware Performance Counter Result==============
       Cycles: at least 142704800
       Instruction Count: at least 475643667
       Memory Read Instruction Count: at least 1548767
       Memory Write Instruction Count: at least 30120
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 2026679942
       Memory Read Request Delay Cycles: at least 1548772
       Read Data Delay Cycles: at least 107177855
       Memory Write Request Delay Cycles: at least 30120
       Branch Instruction Count: at least 79139253
       Jump Instruction Count: at least 460823
============================================================
benchmark finished
time 345088.56ms
```

方便对比，下面给出最初的 sw_conv_mul 的程序的性能计数器结果：

```
==============Hardware Performance Counter Result==============
       Cycles: at least 370728422
       Instruction Count: at least 4467209
       Memory Read Instruction Count: at least 613883
       Memory Write Instruction Count: at least 29068
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 304282008
       Memory Read Request Delay Cycles: at least 613888
       Read Data Delay Cycles: at least 43773968
       Memory Write Request Delay Cycles: at least 29068
       Branch Instruction Count: at least 1590905
       Jump Instruction Count: at least 5783
============================================================
benchmark finished
time 3767.63ms
```

从上面两张图可以看出，sw_conv 的运行时间是 345088.56ms，几

乎是 sw_conv_mul 运行时间的 92 倍，但周期数却比 sw_conv_mul 少。除此之外，sw_conv 的 IF 阶段停留周期数已经大于了总的周期数，所以可以合理推测，sw_conv 的 cycle 数以及超过了周期计数器（32 位宽）表示的范围。

所以为了能解决溢出的问题，本人进行了一定的修改，添加了一个计数器用于存储周期数的高 32 位，同时对 perf_cnt.c 和 perf_cnt.h 进行了修改，如下所示：

```
//辅助周期计数器(帮助处理周期计数器溢出的问题)
reg  [31:0] sub_cycle_cnt;
always @ (posedge clk) begin
        if (rst) begin
                sub_cycle_cnt <= 32'd0;
        end else if (&cycle_cnt) begin
                sub_cycle_cnt <= sub_cycle_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_11 = sub_cycle_cnt;
```

首先，添加了一个计数器用于处理周期计数器溢出的问题，其中其加一的逻辑是周期计数器各位为 1，则其加一，这样，它实际上是把两个计数器拼接成了一个 64 位的计数器，用来存储周期数。

```
typedef struct Result {
        int pass;
        unsigned long long msec;

        unsigned long inst_cnt;
        unsigned long mr_cnt;
        unsigned long mw_cnt;
        unsigned long inst_req_delay_cnt;
        unsigned long inst_delay_cnt;
        unsigned long mr_req_delay_cnt;
        unsigned long rdw_delay_cnt;
        unsigned long mw_req_delay_cnt;
        unsigned long branch_inst_cnt;
        unsigned long jump_inst_cnt;
```

然后，修改 Result 的定义，将周期数数据类型改为 unsigned long long，这是由于 unsigned long long 有 64 位

```
void bench_prepare(Result *res) {
  // TODO [COD]
  //   Add preprocess code, record performance counters' initial states.
  //   You can communicate between bench_prepare() and bench_done() through
  //   static variables or add additional fields in `struct Result`
  unsigned long long temp = _sub_uptime();
  temp = temp << 32;

  res->msec = temp + _uptime();

  res->inst_cnt = _inst_cnt();
  res->mr_cnt = _mr_cnt();
  res->mw_cnt = _mw_cnt();
  res->inst_req_delay_cnt = _inst_req_delay_cycle();
  res->inst_delay_cnt = _inst_delay_cycle();
  res->mr_req_delay_cnt = _mr_req_delay_cycle();
  res->rdw_delay_cnt = _rdw_delay_cycle();
  res->mw_req_delay_cnt = _mw_req_delay_cycle();
  res->branch_inst_cnt = _branch_inst_cnt();
  res->jump_inst_cnt = _jump_inst_cnt();
```

```
void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  unsigned long long temp = _sub_uptime();
  temp = temp << 32;//周期数高位

  res->msec = temp + _uptime() - res->msec;

  res->inst_cnt = _inst_cnt() - res->inst_cnt;
  res->mr_cnt = _mr_cnt() - res->mr_cnt;
  res->mw_cnt = _mw_cnt() - res->mw_cnt;
  res->inst_req_delay_cnt = _inst_req_delay_cycle() - res->inst_req_delay_cnt;
  res->inst_delay_cnt = _inst_delay_cycle() - res->inst_delay_cnt;
  res->mr_req_delay_cnt = _mr_req_delay_cycle() - res->mr_req_delay_cnt;
  res->rdw_delay_cnt = _rdw_delay_cycle() - res->rdw_delay_cnt;
  res->mw_req_delay_cnt = _mw_req_delay_cycle() - res->mw_req_delay_cnt;
  res->branch_inst_cnt = _branch_inst_cnt() - res->branch_inst_cnt;
  res->jump_inst_cnt = _jump_inst_cnt() - res->jump_inst_cnt;
```

然后修改 bench_prepare 和 bench_done 函数，用 temp 变量读取辅助周期计数器的值，并左移 32 位，再加上周期计数器的值，得到正确的周期数。这样完成了对周期数的存储，而由于 printf 函数不支持%llu，其打印的细节将在第 5 点讲述。

5) conv.c 中 int main( )函数的实现

```
int main()
{
        Result res;
        bench_prepare(&res);

#ifdef USE_HW_ACCEL
        printf("Launching task...\n");
        launch_hw_accel();
#else
        printf("starting convolution\n");
        convolution();
        printf("starting pooling\n");
        pooling();
#endif

        int result = comparing();
```

```
bench_done(&res);

unsigned long high_cycle = res.msec >> 32;
unsigned long low_cycle  = res.msec & 0xffffffff;

printf("==============Hardware Performance Counter Result==============\n");
if(high_cycle){
        printf("\tCycles: at least %u%u\n", high_cycle, low_cycle);
}
else{
        printf("\tCycles: at least %u\n", low_cycle);
}//周期计数器可能会溢出

printf("\tInstruction Count: at least %u\n", res.inst_cnt);
printf("\tMemory Read Instruction Count: at least %u\n", res.mr_cnt);
printf("\tMemory Write Instruction Count: at least %u\n", res.mw_cnt);
printf("\tInstruction Fetch Request Delay Cycle: at least %u\n", res.inst_req_delay_cnt);
printf("\tInstruction Fetch Delay Cycles: at least %u\n", res.inst_delay_cnt);
printf("\tMemory Read Request Delay Cycles: at least %u\n", res.mr_req_delay_cnt);
printf("\tRead Data Delay Cycles: at least %u\n", res.rdw_delay_cnt);
printf("\tMemory Write Request Delay Cycles: at least %u\n", res.mw_req_delay_cnt);
printf("\tBranch Instruction Count: at least %u\n", res.branch_inst_cnt);
printf("\tJump Instruction Count: at least %u\n", res.jump_inst_cnt);
printf("================================================================\n");


printf("benchmark finished\n");

if (result == 0) {
        hit_good_trap();
} else {
        nemu_assert(0);
}

return 0;
```

如图，先在卷积与池化前定义 Result 类型变量，用于存储性能计数，并运行 bench_prepare;然后当卷积、池化、比对结束时,运行 bench_done,并输出相应结果。

由于周期计数器可能会溢出，而 printf 函数不支持 long long 数据的打印，所以我把周期数分成了高位和低位，如果高位大于 0，则先打印高位，再打印低位，否则则只打印低位，这样就能防止周期数溢出了。

各个性能计数器的定义同之前几次实验,只是多加了一个计数器用来

赋值记录周期数，具体定义如下：

名称	端口	描述
cycle_cnt	cpu_perf_cnt_0	周期计数器(周期数低 32 位)
inst_cnt	cpu_perf_cnt_1	指令计数器
mr_cnt	cpu_perf_cnt_2	读内存计数器
mw_cnt	cpu_perf_cnt_3	写内存计数器
inst_req_delay_cnt	cpu_perf_cnt_4	取指请求延误周期计数器
inst_delay_cnt	cpu_perf_cnt_5	取值延误周期计数器
mr_req_delay_cnt	cpu_perf_cnt_6	读内存请求延误周期计数器
rdw_delay_cnt	cpu_perf_cnt_7	从内存获取数据延误周期计数器
mw_req_delay_cnt	cpu_perf_cnt_8	写内存请求延误周期寄存器
branch_inst_cnt	cpu_perf_cnt_9	写内存请求延误周期寄存器
jump_inst_cnt	cpu_perf_cnt_10	跳转指令计数器
sub_cycle_cnt	cpu_per_cnt_11	辅助周期计数器(周期数高 32 位)

硬件代码与之前的实验一致，这里就不做展示了

6) costum_cpu.v 中对 mul 指令的实现

```
//MULTIPLY Instruction
assign MULTIPLE = OP_REG & funct7[0];
```

```
assign ALUEn    = (OP_REG | OP_IMM) & (~MULTIPLE) & (funct3[1] | ~funct3[0]) | JALR | LOAD | S_Type | B_Type |
                  (current_state == ID); //ID阶段Result需要对ALU的结果进行选择
assign ShiftEn  = (OP_REG | OP_IMM) & (~MULTIPLE) & (~funct3[1] & funct3[0]) &
                  ~(current_state == ID); //ID阶段Result需要对ALU的结果进行选择
assign MULEn    = MULTIPLE &
                  ~(current_state == ID);//ID阶段Result需要对ALU的结果进行选择
```

如图，引入了 MULEn 控制信号，来从一系列运算数据中选择结果数据。其中需要保证 ALUEn，ShiftEn，MULEn 三条控制信号各自互斥，乘法部件如下：

```
//MULTIPLY
assign MULResult = RF_rdata1 * RF_rdata2;
```

Result 的结果选择如下：

```
//Result
always @(posedge clk) begin
        if ((current_state == EX) || (current_state == ID)) begin
                Result <= {(32){ALUEn}}  &  ALUResult |
                          {(32){ShiftEn}} &  ShifterResult |
                          {(32){MULEn}}   &  MULResult; //Choose Result
        end
end
```

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，逻辑仿真和 FPGA 调试过程中的难点等）

本次实验的硬件部分代码实现较为容易，关键在于对于卷积，池化的理解。整体上实验不算太难，但软件代码出错后 debug 较为麻烦，也花费了一定的时间去寻找 bug。对于本次实验中遇到的 bug，主要有以下几个：

遇到的重要 bug 以及解决过程：

1. 对指针进行强制类型转换时出错：

```
input_vector     = (void*)in + input_offset;
output_vector    = (void*)out + output_offset;
filter           = (void*)weight;//转化为void*才能赋值
```

```
before_pool_vector = (void*)out + input_offset;
after_pool_vector = (void*)out + input_offset + temp_pool_store_offset;/
```

本人的最初的代码如上图所示，为了进行指针的赋值，我的最初的想法是把地址计算出来之后，然后转化为 void*，这样赋值才不会报错。但由于本人少打了括号，导致例如图中的 in、out 首先转化成 void*，然后才进行指针的加法运算，这样导致了地址的计算出错，导致最终结果出错。

解决过程：这个错误在这个实验中不容易发现，因为本次实验所提供的的 input_offset 和 output_offset 设置为 0，但是在池化操作的函数中，after_pool_vector 地址计算的偏移不再是 0，所以 fpga 运行在池化阶段报错。最后在多次调试后，本人发现了这个小错误，也耗费了很多时间，最后代码如下：

```
input_vector     = (void*)(in + input_offset);
output_vector    = (void*)(out + output_offset);
filter           = (void*)weight;//转化为void*才能赋值
```

```
before_pool_vector = (void*)(out + input_offset);
after_pool_vector = (void*)(out + input_offset + temp_pool_store_offset);
```

2. 数组指针使用错误：

```
raw_data += mul(
        *input_vector[ni][current_y][current_x],
        *filter[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
);//用32位int类型存储中间计算结果
```

这是本人最开始写的代码，其中 input_vector 后 filter 的定义如下：

```
short (*input_vector)[WEIGHT_SIZE_D1][input_fm_h][input_fm_w];//输入图像地址
short (*output_vector)[WEIGHT_SIZE_D0][conv_out_h][conv_out_w];//输出图像地址
short (*filter)[WEIGHT_SIZE_D0][WEIGHT_SIZE_D1][filter_size];//卷积核地址
```

其定义为一个多维数组的指针，但本人最开始在使用时候，没有打括号，导致读取数据出错（实际上这么做会导致数组越界）

解决过程： 在发现运行时候数组越界后，查看错误信息，发现是自己的数组指针使用出错，最后改正如下：

```
raw_data += mul(
        (*input_vector)[ni][current_y][current_x],
        (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
);//用32位int类型存储中间计算结果
```

3. 精度损失导致数据输出结果出现偏差

在最初的卷积算法实现时，本人是先对每组乘数与被乘数相乘并移位（使得结果仍然为 16-bit 定点数），然后再相加，如下图所示：

```
if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
        int raw_data += mul(
                (*input_vector)[ni][current_y][current_x],
                (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
        );//用32位int类型存储中间计算结果
        (*output_vector)[no][y][x] += raw_data >> FRAC_BIT
}
```

然而，这样其实是存在问题的。因为先移位后相加的操作会丢失

原本移位前低位的数据，带来精度上的损失。这些精度上的损失表现为所得到的结果与标准略有偏差。

解决过程：最后的解决方案其实很简单，就是保证低位的数据尽量少丢失，于是本人将先移位后相加改成先相加后移位，这样减少了移位的操作，从而减少了精度的损失。利用 raw_data 变量存储中间结果，再将结果移位后的值存储回 output_vector 数组，代码如下：

```
                if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                        raw_data += mul(
                                (*input_vector)[ni][current_y][current_x],
                                (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                        );//用32位int类型存储中间计算结果
                }
        }
}
(*output_vector)[no][y][x] += raw_data >> FRAC_BIT;//由于用整数运算代替小数，所以最终结果需要移位
```

4. 对池化算法的边界填充理解出错

对于最大池化算法中，边界填充的数据并不是 0，而是负无穷，这是为了保证边界填充的数据不会对结果产生影响，但由于这一点 PPT 并没有提及，所以本人最初的池化算法中，以为边界填充是 0，如下所示：

```
if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
        if(max < (*before_pool_vector)[no][current_y][current_x]){
                max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
        }
}
else{
        if(max < 0){
                max = 0;
        }
}
```

解决过程：后面通过对卷积和池化算法的进一步了解过程中（在

网上搜索相关的算法的讲解），发现自己的理解出错，最后对代码加以了修改，如果当前访问的数据是填充的部分，则直接不用考虑，即删除了 else 所对应的语句，如下所示：

```
if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){//
        if(max < (*before_pool_vector)[no][current_y][current_x]){
                max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
        }
}
```

5. 32 位周期计数器溢出

本人最开始没有想到 32 位周期计数器的值会溢出，直到本人看到了性能计数器打印的数值。

对于这点，其核心思想的解决方案是用两个 32 位的计数器拼接成一个 64 位的计数器。这一点的具体实现过程在第二大点的第 4.5 点已经做了详细的描述，这里就不再赘述了。

而解决的结果，将在第三大点的第 3 小点给出，这里也不再做赘述。

三、 对讲义中思考题（如有）的理解和回答

1. 卷积/池化中如果使用边界填充,算法应如何修改?

a. 卷积算法中考虑 padding:

```
for(int x = 0;x < conv_out_w;x++){
    if(ni == 0){
        (*output_vector)[no][y][x] = (*filter)[no][0][0]; //filter[i][0][0]存储第i个通道的bias值
    }

    int raw_data = 0;//用32位int类型存储中间计算结果
    for(int ky = 0;ky < WEIGHT_SIZE_D2;ky++){
        for(int kx = 0;kx < WEIGHT_SIZE_D3;kx++){
            int current_x = mul(x, stride) - pad + kx;//当前输入的访问x坐标(宽度)
            int current_y = mul(y, stride) - pad + ky;//当前输入的访问y坐标(高度)

            if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                raw_data += mul(
                    (*input_vector)[ni][current_y][current_x],
                    (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                );//用32位int类型存储中间计算结果
            }
        }
    }
    (*output_vector)[no][y][x] += raw_data >> FRAC_BIT;//由于用整数运算代替小数，所以最终结果需要移位
```

如图,我将current_x和current_y变量减去了一个偏移pad,使得它们的值考虑了因边界存在而带来的偏移。这样就可以判断访问的数据是否是填充的数据还是实际内存存储的数据,判断逻辑如下面的红框所示:

```
for(int x = 0;x < conv_out_w;x++){
    if(ni == 0){
        (*output_vector)[no][y][x] = (*filter)[no][0][0]; //filter[i][0][0]存储第i个通道的bias值
    }

    int raw_data = 0;//用32位int类型存储中间计算结果
    for(int ky = 0;ky < WEIGHT_SIZE_D2;ky++){
        for(int kx = 0;kx < WEIGHT_SIZE_D3;kx++){
            int current_x = mul(x, stride) - pad + kx;//当前输入的访问x坐标(宽度)
            int current_y = mul(y, stride) - pad + ky;//当前输入的访问y坐标(高度)

            if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                raw_data += mul(
                    (*input_vector)[ni][current_y][current_x],
                    (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                );//用32位int类型存储中间计算结果
            }
        }
    }
    (*output_vector)[no][y][x] += raw_data >> FRAC_BIT;//由于用整数运算代替小数，所以最终结果需要移位
```

注意到边界填充的数据均为0,且不存在实际内存中。而值为零的样本点不会对卷积结果造成影响。所以当两个访问坐标current_x和current_y有一个超出原特征图边界时,则可以直接

跳过这一访问，如下所示：

```
for(int x = 0;x < conv_out_w;x++){
    if(ni == 0){
        (*output_vector)[no][y][x] = (*filter)[no][0][0]; //filter[i][0][0]存储第i个通道的bias值
    }

    int raw_data = 0;//用32位int类型存储中间计算结果
    for(int ky = 0;ky < WEIGHT_SIZE_D2;ky++){
        for(int kx = 0;kx < WEIGHT_SIZE_D3;kx++){
            int current_x = mul(x, stride) - pad + kx;//当前输入的访问x坐标(宽度)
            int current_y = mul(y, stride) - pad + ky;//当前输入的访问y坐标(高度)

            if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                raw_data += mul(
                        (*input_vector)[ni][current_y][current_x],
                        (*filter)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                    );//用32位int类型存储中间计算结果
            }
        }
    }
    (*output_vector)[no][y][x] += raw_data >> FRAC_BIT;//由于用整数运算代替小数，所以最终结果需要移位
```

b. 池化算法中考虑 padding：

```
for(int ky = 0;ky < KERN_ATTR_POOL_KERN_SIZE;ky++){
        for(int kx = 0;kx < KERN_ATTR_POOL_KERN_SIZE;kx++){
            int current_x = mul(x, stride) - pad + kx; //当前输入的访问x坐标(宽度)
            int current_y = mul(y, stride) - pad + ky; //当前输入的访问y坐标(高度)

            if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                if(max < (*before_pool_vector)[no][current_y][current_x]){
                    max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
                }
            }
        }
}
(*after_pool_vector)[no][y][x] = max;
```

如图,与卷积算法类似，将 current_x 和 current_y 变量减去了一个偏移 pad,使得它们的值考虑了因边界存在而带来的偏移。这样就可以判断访问的数据是否是填充的数据还是实际内存存储的数据，判断逻辑也与卷积算法类似，如下面的红框所示：

```
for(int ky = 0;ky < KERN_ATTR_POOL_KERN_SIZE;ky++){
        for(int kx = 0;kx < KERN_ATTR_POOL_KERN_SIZE;kx++){
                int current_x = mul(x, stride) - pad + kx; //当前输入的访问x坐标(宽度)
                int current_y = mul(y, stride) - pad + ky; //当前输入的访问y坐标(高度)

                if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                        if(max < (*before_pool_vector)[no][current_y][current_x]){
                                max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
                        }
                }
        }
}
(*after_pool_vector)[no][y][x] = max;
```

注意到最大池化算法边界填充的数据为负无穷,且不存在实际内存中。所以池化操作时候，不需要考虑填充的数据。所以当两个访问坐标 current_x 和 current_y 有一个超出原特征图边界时，则也可以直接跳过这一访问，如下所示：

```
for(int ky = 0;ky < KERN_ATTR_POOL_KERN_SIZE;ky++){
        for(int kx = 0;kx < KERN_ATTR_POOL_KERN_SIZE;kx++){
                int current_x = mul(x, stride) - pad + kx; //当前输入的访问x坐标(宽度)
                int current_y = mul(y, stride) - pad + ky; //当前输入的访问y坐标(高度)

                if(current_x >= 0 && current_x < input_fm_w && current_y >= 0 && current_y < input_fm_h){
                        if(max < (*before_pool_vector)[no][current_y][current_x]){
                                max = (*before_pool_vector)[no][current_y][current_x];//最大值更新
                        }
                }
        }
}
(*after_pool_vector)[no][y][x] = max;
```

2. 在软件算法实现中，如何避免出现溢出和精度损失?

   a. 首先是，中间结果用 int 类型存储而非 short 类型，即用 32 位存储乘法计算结果，这样可以防止计算数据的溢出。

   b. 其次，由于最终结果需要移位，而移位会导致精度的损失（在写本次实验遇到的 bug 的第 3 点提到了这个问题），所以需要尽可能减

少移位次数。比如卷积算法中先对乘法结果进行相加，最后统一一次移位，而不是对每次乘法结果移位后相加。

3. 不同实现方法的性能差异：

I. sw_conv 的性能计数器如下图所示：

```
==============Hardware Performance Counter Result==============
        Cycles: at least 73773231205
        Instruction Count: at least 475643670
        Memory Read Instruction Count: at least 1548767
        Memory Write Instruction Count: at least 30121
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 1363261102
        Memory Read Request Delay Cycles: at least 1548772
        Read Data Delay Cycles: at least 106155927
        Memory Write Request Delay Cycles: at least 30121
        Branch Instruction Count: at least 79139252
        Jump Instruction Count: at least 460823
================================================================
benchmark finished
time 338445.17ms
```

II. sw_conv_mul 的性能计数器如下图所示：

```
==============Hardware Performance Counter Result==============
       Cycles: at least 360981029
       Instruction Count: at least 4467218
       Memory Read Instruction Count: at least 613885
       Memory Write Instruction Count: at least 29069
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 295762701
       Memory Read Request Delay Cycles: at least 613890
       Read Data Delay Cycles: at least 42545882
       Memory Write Request Delay Cycles: at least 29069
       Branch Instruction Count: at least 1590906
       Jump Instruction Count: at least 5783
================================================================
benchmark finished
time 3669.43ms
```

III. hw_conv 的性能计数器如下图所示：

```
==============Hardware Performance Counter Result==============
       Cycles: at least 6747804
       Instruction Count: at least 79386
       Memory Read Instruction Count: at least 14856
       Memory Write Instruction Count: at least 139
       Instruction Fetch Request Delay Cycle: at least 0
       Instruction Fetch Delay Cycles: at least 5492722
       Memory Read Request Delay Cycles: at least 14861
       Read Data Delay Cycles: at least 839168
       Memory Write Request Delay Cycles: at least 139
       Branch Instruction Count: at least 24457
       Jump Instruction Count: at least 12
================================================================
benchmark finished
time 122.84ms
```

通过周期数对比，可以很明显的看出可以看到，使用基于加减法的宏

定义实现的乘法的周期数，是使用mul指令实现乘法周期数的 204.4 倍；使用 mul 指令实现乘法的普通实现的周期数，是使用硬件加速器周期数的 53.5 倍，对比十分明显。由此可见，利用硬件替代软件实现可以较好地提升性能

四、 在课后，你花费了大约___20__小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

这次实验看似不复杂，但实际实现过程中仍然遇到了较多的问题。其中大部分是由于本人对 C 语言的语法不够熟悉和粗心大意所导致的，但通过本次实验也让我对 C 语言的一些特性和指针的应用有了更深的了解。

另外本人有一些建议，一个是可以增加一些针对边界填充的测试样例。现在所有的测试样例均是针对 pad=0，这样不太不清楚自己考虑 padding 的算法到底对不对；二是由于 sw_conv 程序的运行时间较长，而性能计数器只有 32 位，这样会导致性能计数器溢出，要解决这一点只能将两个性能计数器拼接在一块。而且这个问题没有在 PPT 中给出，希望下届能在 PPT 中有所提及。

最后感谢助教对本人的耐心帮助，感谢实验群里的同学们对本人提出

的问题的帮助与解答。