中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009970001　姓名：金扬　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 格式保存在~/COD-Lab/reports 目录下。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 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、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。

为了便于访问各数据，我首先定义了三个数组指针。

```
typedef short (*IN)[WEIGHT_SIZE_D1][input_fm_h][input_fm_w];
typedef short (*WEIGHT)[WEIGHT_SIZE_D0][WEIGHT_SIZE_D1][mul(WEIGHT_SIZE_D2, WEIGHT_SIZE_D3) + 1];
typedef short (*OUT)[WEIGHT_SIZE_D0][conv_out_h][conv_out_w];
IN in_array = (IN)(in + input_offset);
WEIGHT weight_array = (WEIGHT)weight;
OUT out_array = (OUT)(out + output_offset);
```

分别对应输入图像、卷积核、以及卷积后的输出特征图数据。

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){
                    (*out_array)[no][y][x] = (*weight_array)[no][0][0];
                }
                int ih = mul(y, stride) - pad;
                int unshifted = 0;
                for(int ky=0;ky<WEIGHT_SIZE_D2;ky++,ih++){
                    int iw = mul(x, stride) - pad;
                    for(int kx=0;kx<WEIGHT_SIZE_D3;kx++,iw++){
                        if(iw>=0 && iw<input_fm_w && ih>=0 && ih<input_fm_h){
                            unshifted += mul(
                                (*in_array)[ni][ih][iw],
                                (*weight_array)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
                            );
                        }
                    }
                }
                (*out_array)[no][y][x] += unshifted >> FRAC_BIT;
            }
        }
    }
}
```

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。

2. conv.c 中 void pooling( )函数的实现

和 convolutional 函数类似地定义数组指针以便于访问特定下标的数据。

```
typedef short (*BEF_POOL)[WEIGHT_SIZE_D0][input_fm_h][input_fm_w];
typedef short (*AFT_POOL)[WEIGHT_SIZE_D0][pool_out_h][pool_out_w];
unsigned cache_offset = mul(WEIGHT_SIZE_D0, mul(input_fm_h, input_fm_w));
BEF_POOL bef_array = (BEF_POOL)(out + input_offset);
AFT_POOL aft_array = (AFT_POOL)(out + input_offset + cache_offset);
```

值得注意的是，这里的 bef_array 表示经卷积但未经池化的特征图数据；aft_array 用于临时存储经池化后的特征图数据，这部分存储空间来源于 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）。

在经过池化操作后，需要将 aft_array 中那些偏移存储的数据挪动到输出特征图数据的开始地址。

```
int i = input_offset + cache_offset;
int o = output_offset;
for(int k=0;k<mul(WEIGHT_SIZE_D0, mul(pool_out_h, pool_out_w));k++,i++,o++){
    out[o]=out[i];
}
```

池化操作的实现同卷积类似，具体如下：

```
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++){
            (*aft_array)[no][y][x] = 0x8000;
            int ih = mul(y, stride) - pad;
            for(int ky=0;ky<KERN_ATTR_POOL_KERN_SIZE;ky++,ih++){
                int iw = mul(x, stride) - pad;
                for(int kx=0;kx<KERN_ATTR_POOL_KERN_SIZE;kx++,iw++){
                    if(iw>=0 && iw<input_fm_w && ih>=0 && ih<input_fm_h){
                        if((*aft_array)[no][y][x] < (*bef_array)[no][ih][iw]){
                            (*aft_array)[no][y][x] = (*bef_array)[no][ih][iw];
                        }
                    }
                }
            }
        }
    }
}
```

这里采样前的存储值取 0x8000，也就是 short 类型能表示的最小值。在每次采样时，若样本值大于存储值，则替换存储值。从而完成取最大值的操作。

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) |= 1;
    while(!((*gpio_done) & 1));
    (*gpio_start) &= 0;
}
#endif
```

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 位是否拉高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. costum_cpu.v 中对 mul 指令的实现

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

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

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 ALUEn，ShiftEn，MULEn 三条控制信号各自互斥，定义如下：

```
assign ALUEn    = (OP & ~funct7[0] | OP_IMM) & (funct3[1] | ~funct3[0]) | JALR | LOAD | S_Type | B_Type;
assign ShiftEn  = (OP & ~funct7[0] | OP_IMM) & (~funct3[1] & funct3[0]);
assign MULEn    = OP & funct7[0];
```

5. conv.c 中 int main( )函数内性能计数器的实现

```c
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);
    printf("======Hardware Performance Counter======\n");
    printf("Cycle Count:                %u\n", res.cnt[0]);
    printf("Instruction Count:          %u\n", res.cnt[1]);
    printf("Memory Read:                %u\n", res.cnt[2]);
    printf("Memory Write:               %u\n", res.cnt[3]);
    printf("Instruction Request Delay:  %u\n", res.cnt[4]);
    printf("Instruction Response Delay: %u\n", res.cnt[5]);
    printf("MemRead Request Delay:      %u\n", res.cnt[6]);
    printf("Read Data Delay:            %u\n", res.cnt[7]);
    printf("MemWrite Request Delay:     %u\n", res.cnt[8]);
    printf("Branch Count:               %u\n", res.cnt[9]);
    printf("Jump Count:                 %u\n", res.cnt[10]);
    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。各个性能计数器的定义同之前几次实验，不再赘述。

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）

1. 对数据存放具体结构理解有误

在编写 conv.c 代码的时候，我当初参考的是第一版 PPT，其中对数据存放具体结构的描述基于硬件加速器可见的。我开始把它误解成了软件算法可见的，因此产生了一系列错误。

这个问题通过本地测试 conv.c，反复编译、调试、观察源数据格式发现。

2. 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，我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），然后将所有结果相加。

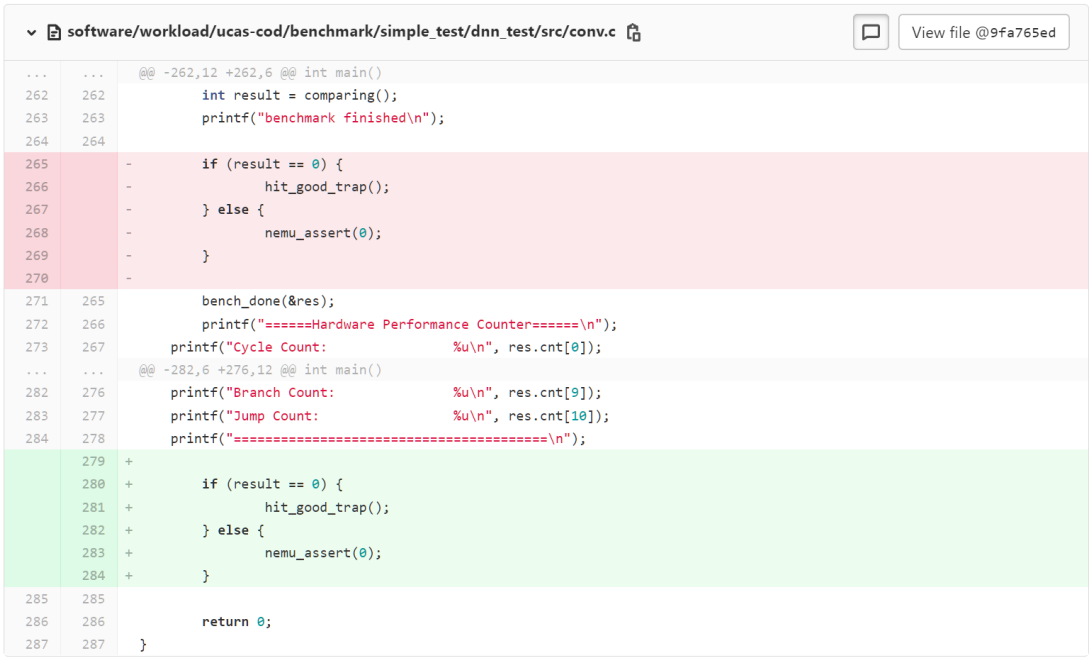
然而，这样其实是存在问题的。事实上，这样得到的运行结果与标准结果存在微小的差异:尽管大部分答案正确,但仍存在少部分数据与标准答案相差一二。

经过反复比对，我发现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结果与标准略有偏差。

解决方案是将先移位后相加改成先相加后移位。如前文的图中所示，利用 unshifted 变量存储中间结果，再将结果移位后的值加回 out 数组。

3. 性能计数器添加位置问题

最初我将性能计数器放在了 hit_good_trap 函数之后，导致计数结果无法正常显示。



解决方案是将其放在 hit_good_trap 判定之前。

至于导致这个问题的原因，我猜测是因为该程序在 hit_trap 时就已经退出了，所以并不会执行接下来的代码。

在后续的 cache 实验中，我通过阅读汇编代码发现：当 hit_good_trap 时，程序会向一个特定地址（需要通过旁路访问的地址）写入特定值，从而实现结束程序。（发现现象这个是因为我当时旁路逻辑出现了问题，这个值并没有正常写入，导致程序出现了死循环）

这个观察验证了之前的猜想，说明了 hit_good_trap 时，程序将终止。

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?

```
int ih = mul(y, stride) - pad;
int unshifted = 0;
for(int ky=0;ky<WEIGHT_SIZE_D2;ky++,ih++){
    int iw = mul(x, stride) - pad;
    for(int kx=0;kx<WEIGHT_SIZE_D3;kx++,iw++){
        if(iw≥0 && iw<input_fm_w && ih≥0 && ih<input_fm_h){
            unshifted += mul(
                (*in_array)[ni][ih][iw],
                (*weight_array)[no][ni][mul(ky, WEIGHT_SIZE_D3) + kx + 1]
            );
        }
    }
}
(*out_array)[no][y][x] += unshifted >> FRAC_BIT;
```

如图，我对 ih 和 iw 变量设置了一定的偏移，使得它们的枚举起点考虑了因边界存在而带来的偏移。

注意到边界填充的数据均为零，且不存在实际内存中。所以如红线所示，当两个访问坐标 iw 和 ih 有一个超出原特征图边界时，将直接跳过这一访问。其合理性在于，值为零的样本点不会对和造成影响。

带边界填充的池化算法实现同理。

2. 在软件算法实现中，如何避免出现溢出和精度损失?

见“二、2.精度损失导致数据输出结果出现偏差”，不再赘述。

3. 不同实现方法的性能差异

sw_conv 的性能计数器如下图所示

```
======Hardware Performance Counter======
Cycle Count:                3236515503
Instruction Count:          469140764
Memory Read:                1654485
Memory Write:               611812
Instruction Request Delay:  0
Instruction Response Delay: 893669305
MemRead Request Delay:      1654489
Read Data Delay:            68394922
MemWrite Request Delay:     611812
Branch Count:               78050709
Jump Count:                 440663
========================================
```

sw_conv_mul 的性能计数器如下图所示

```
======Hardware Performance Counter======
Cycle Count:                226488257
Instruction Count:          4454800
Memory Read:                625506
Memory Write:               33610
Instruction Request Delay:  0
Instruction Response Delay: 177173503
MemRead Request Delay:      625510
Read Data Delay:            26649477
MemWrite Request Delay:     33610
Branch Count:               1585249
Jump Count:                 23
========================================
```

hw_conv 的性能计数器如下图所示

```
======Hardware Performance Counter======
Cycle Count:                4250776
Instruction Count:          79469
Memory Read:                14877
Memory Write:               144
Instruction Request Delay:  0
Instruction Response Delay: 3312886
MemRead Request Delay:      14881
Read Data Delay:            520742
MemWrite Request Delay:     144
Branch Count:               24482
Jump Count:                 12
========================================
```

可以看到，使用 mul 指令实现乘法，比使用基于加减法的宏定义实现的乘法快了 14.3 倍；使用硬件加速器，比使用 mul 指令的普通实现快了 53.3 倍。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 15 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。

第一版的 PPT 具有一定的误导性，例如之前提到的数据存放规则。此外 PPT 上伪代码中数据访问函数的下标顺序也存在一些问题，与数据实际存放顺序不符。

非常感谢徐步骥同学在群里提供的本地编译程序模板。这个模板将测试数据通过文件输入函数读入内存，并利用*代替 mul 函数的定义，实现了本地 C 编译器编译 conv.c 的功能。该模板大大简化了调试过程的复杂度，便于打印错误信息、添加调试变量，以快速定位错误所在。

另外有个小建议，可以增加一些针对边界填充的测试样例。现在云平台上的测试样例 pad=0，因此不太不清楚自己实现的对不对。