

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 专业 计算机科学与技术
实验项目编号 5.3 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` 推送到实验课 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 Verilog HDL 关键代码段 {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)

• 深度学习核心算法(2D 卷积 + 池化)的软件实现。

关于 2D 卷积的理论知识, PPT 上已经讲述的很清楚了, 这里仅仅将相关知识放在下面, 不再进行进一步的解释。

利用一组卷积权重值 (weight), 对一副输入图像 (Input Image) 进行运算, 得到一张输出特征图 (Feature Map) 的过程

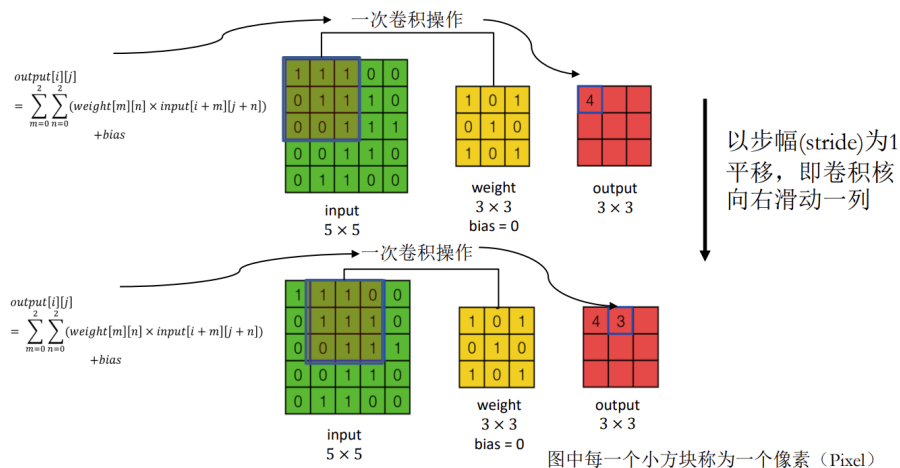


图 1: 1D 卷积算法

- 与1D卷积算法相比，输入图像和输出特征图变成多通道（如右图输入图像有3个通道，输出图像有2个通道）

- 各输出通道特征图计算都使用一个卷积核（即右图中的Filter）

- 一个卷积核包含多组权重值和一个bias值
- 每组权重值对应输入图像的一个通道

- 各输入通道与对应权重值执行1D卷积算法

- 将各1D卷积结果的对应像素位置累加，得到一张输出通道特征图

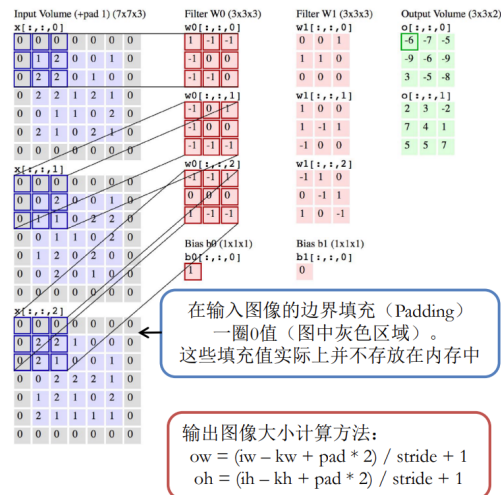


图 2: 2D 卷积算法

关于 2D 卷积算法的实现,是完成 conv.c 中的 convolution() 函数。关于实现该函数的步骤,在 PPT 中已经写出:

```

for(no = 0; no < Oc; no++) → 输出特征图数量
{
    for(ni = 0; ni < Ic; ni++) → 输入特征图数量
    {
        for(y = 0; y < Oh; y++)
        for(x = 0; x < Ow; x++) } 输出特征图尺寸
        {
            if (ni == 0)
                output_feature_map(no, x, y) = bias(no);
            for(ky = 0; ky < K; ky++)
            for(kx = 0; kx < K; kx++) } 权重值尺寸
            {
                iw = kx + x * S;
                ih = ky + y * S;
                output_feature_map(no, x, y) +=
                    input(ni, iw, ih) * weight(ni, no, kx, ky);
            }
        }
    }
}

```

图 3: 2D 卷积算法实现思路

```

1 //TODO: Please add your implementation here
2
3 /*定义输入图像、卷积核、输出特征图数组*/
4 //Input[ic][ih][iw],ic表示输入图像的通道数,ih表示输入图像的高度,iw表示输入图像的宽度
5 typedef short (*IN)[rd_size.d1][rd_size.d2][rd_size.d3];
6 //Filter[oc][ic][i+k*k],oc表示输出特征图的通道数,ic表示输入图像的通道数(行下标),k表示卷积核尺寸(列下标)
7 typedef short (*WEIGHT)[weight_size.d0][weight_size.d1][mul(weight_size.d2, weight_size.d3) + 1];
8 //Output[oc][oh][ow],oc表示输出特征的通道数,oh表示输出图像的高度(行下标),ow表示输出图像的宽度(列下标)
9 typedef short (*OUT)[conv_size.d1][conv_size.d2][conv_size.d3];
10 IN in_array = (IN)(in + input_offset);
11 WEIGHT weight_array = (WEIGHT)weight;
12 OUT out_array = (OUT)(out + output_offset);
13
14 //卷积
15 for(int no=0;no<weight_size.d0;no++){ //输出特征图数量
16     for(int ni=0;ni<rd_size.d1;ni++){ //输入特征图数量
17         for(int y=0;y<conv_size.d2;y++){ //输出特征图尺寸
18             int ybase = mul(y, stride) - pad; //乘法外移,减少乘法的运算次数
19             for(int x=0;x<conv_size.d3;x++){
20                 int xbase = mul(x, stride) - pad; //乘法外移
21                 if(ni==0){
22                     (*out_array)[no][y][x] = (*weight_array)[no][0][0];
23                 }
24                 int unshifted = 0; //
25                 for(int ky=0;ky<WEIGHT_SIZE_D2;ky++){
26                     int ih = ky + ybase;
27                     int yoffset = mul(ky, weight_size.d3); //乘法外移,且由于二维数组的存储,应当按照y*每行元素个数来表示ky
28                     for(int kx=0;kx<WEIGHT_SIZE_D3;kx++){
29                         int iw = kx + xbase;
30                         if(iw>0 && iw<input_fm_w && ih>0 && ih<input_fm_h){ //条件判断,当前坐标是否在输入范围内
31                             unshifted += mul((*in_array)[ni][ih][iw],(*weight_array)[no][ni][yoffset + kx + 1]);
32                         }
33                     }
34                 }
35                 (*out_array)[no][y][x] += unshifted >> FRAC_BIT; //用整数结果来计算最终的浮点数结果
36             }
37         }
38     }
39 }

```

图 4: convolution 函数代码实现

上图为实现 conv.c 中 convolution() 的具体代码,主要部分是实现多重 for 循环,将 2D 卷积转化为 1D 卷积,然后再进行累加,即可得到相应的卷积结果。

这里面我将部分乘法进行外提,减少了不必要的乘法次数。

然后由于实际问题是非整数的数据,为了运算方便,我们先进行整数运算,然后再根据偏移量进行右移,从而得到我们所需要的浮点数结果。

关于池化算法的实现,整体思路和卷积相似,只不过池化的作用是为了去掉卷积算法输出特征图中不重要的样本,进一步减少参数数量。本实验的池化算法是 Max Pooling 算法,池化方法示例如下:

对 $N \times N$ 的特征图取最大值,作为采样后的样本值

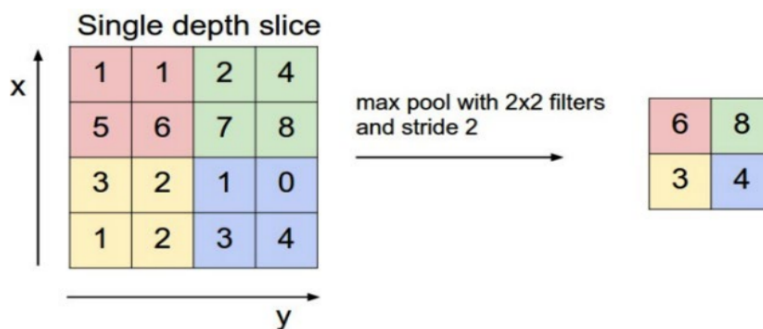


图 5: 池化举例

从图中可知,我们进行池化的方法是将输入图像划分成若干小块,然后在每个部分中选取最大值作为该部分的输出值,根据 stride 的值决定输出块大小。

```
1 //TODO: Please add your implementation here
2
3 /*Max Pooling算法, 对N×N的特征图取最大值, 作为采样后的样本值*/
4
5 /*定义输入数组、输出数组*/
6 //池化中的输入数组即为卷积的输出数组Output[oc][oh][ow]
7 typedef short (*IN_TYPE)[conv_size.d1][conv_size.d2][conv_size.d3];
8 //池化的输出数组是取完最大值并更新后的结果
9 typedef short (*OUT_TYPE)[conv_size.d1][pool_out_h][pool_out_w];
10 IN_TYPE in_array = (IN_TYPE)(in + input_offset);
11 OUT_TYPE out_array = (OUT_TYPE)(out + output_offset);
12
13 //类似卷积的for循环
14 for(int no=0;no<conv_size.d1;no++){ //输出特征图数量
15     for(int y=0;y<pool_out_h;y++){ //输出特征图的行数
16         int ybase = mul(y, stride) - pad; //乘法外移
17         for(int x=0;x<pool_out_w;x++){ //输出特征图的列数
18             int xbase = mul(x, stride) - pad; //乘法外移
19             short max = 0x8000; //short类型表示的最小值
20             for(int ky=0;ky<KERN_ATTR_POOL_KERN_SIZE;ky++){
21                 int ih = ky + ybase;
22                 for(int kx=0;kx<KERN_ATTR_POOL_KERN_SIZE;kx++){
23                     int iw = kx + xbase;
24                     if(iw>=0 && iw<input_fm_w && ih>=0 && ih<input_fm_h){ //条件判断
25                         if(max < (*in_array)[no][ih][iw]){ //大小判断
26                             max = (*in_array)[no][ih][iw];
27                         }
28                     }
29                 }
30             }
31             (*out_array)[no][y][x] = max; //将其中的元素值替换为最大值
32         }
33     }
34 }
```

图 6: pooling 函数代码实现

上图为 pooling() 的具体代码,主要部分也是一个多重 for 循环,该循环主要做的事情是找寻每个小块中的最大值,然后将块中的元素值替换为各自块的最大值。

然后是关于硬件加速器控制软件,我们不需要自己来实现硬件加速器,只需要进行控制即可,因此该部分很简单。

❑ MIPS/RISC-V功能型处理器按I/O外设方式访问加速器 —— 基地址为0x6003_0000

偏移地址	寄存器名	访问方式	说明
0x0	START (32-bit)	只写 (W)	START[0]: 加速器启动位 写1表示启动加速器; 写0表示停止加速器
0x8	DONE (32-bit)	只读 (R)	DONE[0]: 加速器工作状态 读出1表示加速器已完成卷积运算

❑ 功能型处理器上对硬件加速器的软件控制流程

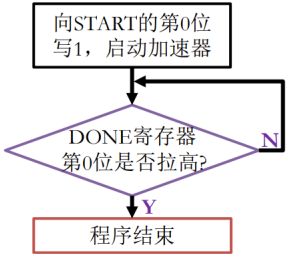


图 7: 硬件加速器控制软件流程

```
1  #ifdef USE_HW_ACCEL
2  void launch_hw_accel()
3  {
4      volatile int* gpio_start = (void*)(GPIO_START_ADDR);
5      volatile int* gpio_done = (void*)(GPIO_DONE_ADDR);
6
7      //TODO: Please add your implementation here
8      // 启动硬件加速器
9      *gpio_start = 1;
10     // 等待硬件加速器完成
11     while (*gpio_done == 0);
12     // 硬件加速器完成
13     *gpio_start = 0;
14 }
15 #endif
```

图 8: launch_hw_accel 函数代码实现

最后,软件部分需要实现性能计数器的打印,直接复用 perf_cnt 相关的接口函数即可:

```
1  int result = comparing();
2  bench_done(&res);
3  printf("Cycle Count:      %u\n", res.msec);
4  printf("Instruction Count: %u\n", res.InstructionCount);
5  printf("benchmark finished\n");
6
7  if (result == 0) {
8      hit_good_trap();
9  } else {
10     nemu_assert(0);
11 }
```

图 9: main 函数中复用性能计数器相关代码

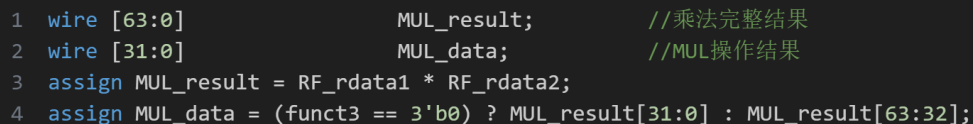
- 为定制处理器增加乘法指令。

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	

MUL performs an XLEN-bit×XLEN-bit multiplication and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and signed×unsigned multiplication respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] *rdh*, *rs1*, *rs2*; MUL *rdl*, *rs1*, *rs2* (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

图 10: RISC-V MUL 指令

MUL 指令的作用是 32bit*32bit, 然后将结果存储在一个 64bit 的寄存器中, 不过在本实验中我们只需要获取乘法的低 32 位结果即可。



```

1 wire [63:0]          MUL_result;          //乘法完整结果
2 wire [31:0]          MUL_data;            //MUL操作结果
3 assign MUL_result = RF_rdata1 * RF_rdata2;
4 assign MUL_data = (funct3 == 3'b0) ? MUL_result[31:0] : MUL_result[63:32];

```

图 11: custom_cpu 中 mul 指令的实现

这里我没有使用理论课上讲的 bush 算法,而是直接使用 * 符号来表示乘法,效率会相对较低,可以进一步优化。

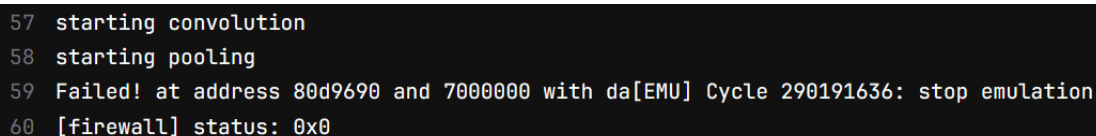
二、 实验过程中遇到的问题、对问题的思考过程及解决方法(比如 RTL 代码中出现的逻辑 bug,逻辑仿真和 FPGA 调试过程中的难点等)

- 关于讲义中说明不要额外定义新的数组的问题。

讲义中说明在写卷积和池化函数的时候,不要定义额外的数组,我在一开始也没有定义额外的数组,但是感觉代码较为难以理解,不利于编写者编写代码,也不利于阅读者进行阅读,于是我加上了输入和输出的相关数组,以便于编写和阅读。

- cpu 相关问题。

在 debug 的过程中我发现我的 fpga 一直有错误,只能跑过硬件加速的测试,不能跑过另外两个测试,而且错误都是相同的错误,我在使用 fpga 加速仿真之后,发现是在指令 add 的时候出现的问题,但是该条指令本身没有问题,由于没有金标准来让我往前推算是从哪里最先开始出错的,所以我只能进行尝试修改,以达到找出代码中 bug 出现的位置。在修改完 conv.c 的代码之后,还是出现了相同的问题,然后我推测可能是 cpu 时序出现了问题,导致最后传递给仿真的时候某些数据结果和标准不同。



```

57 starting convolution
58 starting pooling
59 Failed! at address 80d9690 and 7000000 with da[EMU] Cycle 290191636: stop emulation
60 [firewall] status: 0x0

```

图 12: fpga 加速仿真出现的问题

然后我修改了 PC 相关的时序逻辑之后,发现错误解决了,但是在和助教验收的时候,助教说是否进行我做的修改并不会造成什么影响,然后推测真正的错误不在此处,但是助教在检查完寄存器代码之后也并未发现其他错误,而我也没有检查出 cpu 除了时序逻辑之外的组合逻辑有什么问题。因此此问题还有待进一步找寻并修改。


```

1 //PC寄存器的时序逻辑电路
2 always @(posedge clk)
3 begin
4     if (rst)                                //rst同步的高电平复位信号
5         PC <= 32'b0;
6     else if(current_state == EX)
7         PC <= Jump ? Jump_addr: (Branch ? Branch_addr: PC_next);
8     else if(Instruction_current == 32'b0 && current_state == ID)
9         PC <= PC_next;

```

图 13: 为 PC 寄存器增加一条 NOP 指令跳转的操作

三、 对讲义中思考题的理解和回答

- 思考题 1: 如果使用边界填充, 算法应如何修改。

在卷积算法中, 使 i_h 和 i_w 变量减去 pad , 使得它们的枚举起点考虑了因边界存在而带来的偏移。因此如果需要边界填充的话, 只需要修改 pad 值即可。

池化算法中方法相同。

- 思考题 2: 在软件算法实现中, 需考虑如何避免出现溢出和精度损失。

由于我们使用整数进行运算, 然后再进行移位操作将其转化为浮点数, 所以我们如果在乘法和求和过程中进行移位操作的话会导致末尾数据的丢失, 因此我们可以在整个 `for` 循环外面对求和之后的结果进行移位操作, 这样可以避免相应问题。

四、 实验所耗时间

在课后, 你花费了大约 10 小时完成此次实验。

五、 实验思考与心得

- 关于不同实现方法的性能差异。


```

47 Passed!
48 Cycle Count:                2194634895
49 Instruction Count:  282710832
50 benchmark finished
51 time 21954.04ms

47 Passed!
48 Cycle Count:                76988673
49 Instruction Count:  4517149
50 benchmark finished
51 time 777.06ms

47 Cycle Count:                1998028
48 Instruction Count:  122441
49 benchmark finished
50 time 32.24ms

```

图 14: 三种方式的性能计数器结果

上图从上到下分别为 sw_conv,sw_conv_mul,hw_conv 三种测试的相应性能计数器及运行时间结果, 经过分析后得到的性能比较如下图:

sw_conv	时钟周期数	指令数	运行时间(ms)
数值	2194634895	282710832	21954.04
sw_conv_mul	时钟周期数	指令数	运行时间(ms)
数值	76988673	4517149	777.06
hw_conv	时钟周期数	指令数	运行时间(ms)
数值	1998028	122441	32.24
sw_conv_mul对比sw_conv	27.51	61.59	27.25
hw_conv对比sw_conv	1097.4	2307.96	679.96
hw_conv对比sw_conv_mul	37.53	35.89	23.1

图 15: 性能比较结果

由上图可以看出,使用 MUL 指令对性能有较大提升,而使用硬件加速可以进一步较大地提升性能。

• 实验心得

关于本实验本身的难度并不大,难点在于 debug 的过程,由于选做实验中的波形都没有金标准进行对比,因此很难根据波形往前追溯到自己最开始错误的地方,因为很多报错并不是实际出错的位置,导致修改起来十分麻烦,希望以后在选做实验中可以提供金标准的波形。

另外本实验的 pad 值宏定义为 0,即实际上没有使用到边界填充的算法,希望可以增加对边界填充的相关测试,从而让同学们更好的了解本实验。