中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 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）

* 说明：本实验硬件部分基于 prj3 实现的 MIPS 型处理器进行修改，所使用操作除 MUL 外均已在 prj3 中实现，故不特定对逻辑电路结构图和波形信号进行说明。RTL 代码段将分为直接乘法和 booth 算法两种实现方式对修改部分进行说明；软件仅对补充部分进行说明。

1、硬件部分

1.1 直接乘法

（1）状态机修改部分

```
//EX->IF:
        //REGIMM        2: opcode[5:0]=000001
        //I_branch      4: opcode[5:2]=0001
        //J             1: opcode[5:0]=000010
//EX->WB
        //R_Type        18: opcode[5:0]=6'b0 include JALR,JR //JR: rs->GPR[0]
```

```
          //I_calc        7: opcode[5:3]=001 include LUI
          //JAL           1: opcode[5:0]=000011
          //MUL           inst_MUL
//EX->LD
          //load          7: opcode[5:3]=100
//EX->ST
          //store         5: opcode[5:3]=101
EX: begin
          if((opcode[5:0]==6'b000001) | (opcode[5:2]==4'b0001) | (opcode[5:0]==6'b000010) )
                        next_state = IF;
          else if((~|opcode)| (opcode[5:3]==3'b001) | (opcode[5:0]==6'b000011)| inst_MUL)
                        next_state = WB;
          else if(opcode[5:3]==3'b100)
                        next_state = LD;
          else if(opcode[5:3]==3'b101)
                        next_state = ST;
          else
                        next_state = EX;
```

在状态机第二部分加入了 MUL 指令下从 EX 到 WB 的状态转移。

（2）乘法运算部分

```
//Signals connected to MUL
        wire inst_MUL;
        reg [31:0] MUL_A_tmp;
        reg [31:0] MUL_B_tmp;
        wire [63:0] MUL_Result;
        reg [63:0] MUL_Result_tmp;

...

//Signals connected to MUL
        assign inst_MUL = opcode==6'b011100 & func == 6'b000010;
        always@(posedge clk) begin
                if(rst)
                        MUL_A_tmp <= 32'b0;
                else if(current_state[3])
                        MUL_A_tmp <= RF_rdata1;
        end


        always@(posedge clk) begin
                if(rst)
                        MUL_B_tmp <= 32'b0;
                else if(current_state[3])
                        MUL_B_tmp <= RF_rdata2;
```

```
        end

        assign MUL_Result = MUL_A_tmp * MUL_B_tmp;

        always@(posedge clk) begin
                if(rst)
                        MUL_Result_tmp <= 64'b0;
                else if(current_state[4])
                        MUL_Result_tmp <= MUL_Result;
        end
```

与 ALU 操作数逻辑一致，在 ID 状态先将 MUL 的操作数进行寄存，在 EX 阶段将 MUL 运算结果进行寄存。该部分直接使用*实现乘法运算。

（3）寄存器相关部分

```
assign RF_wen = (current_state [8]) & //WB
                        ( inst_MUL |
                        (opcode[5:0]==6'b000000 & func[5]==1'b1)               |
                        (opcode[5:3]==3'b001 & (~&opcode[2:0]))               |
                        (opcode[5:3]==3'b100)                                 |
                        (opcode[5:0]==6'b000000 & func[5:3]==3'b000)          |
                        (opcode[5:0]==6'b000011)                              |
                        (opcode[5:0]==6'b000000 & func[5:0]==6'b001001)       |
                        (opcode[5:0]==6'b001111)                              |
                        ((opcode[5:0]==6'b000000 & func[5:1]==5'b00101) & ((func[0] &
(|RF_rdata2))|(~func[0] & (~|RF_rdata2)))) ;
        assign RF_waddr = inst_MUL                                    ? rd    :
                          (opcode[5:3]==3'b100 | opcode[5:3]==3'b001 )  ? rt    :
                          (opcode[5:0]==6'b000011)                    ? 5'd31 :
                          rd ;
        assign RF_wdata =  {32{inst_MUL}} & MUL_Result_tmp[31:0]   |
                           {32{(opcode[5:0]==6'b000000 & func[5:1]==5'b00101) & ((func[0] &
(|RF_rdata2))|(~func[0] & (~|RF_rdata2)))}}  & RF_rdata1           |
                           {32{opcode[5:3]==3'b100}}
                                      & load_data           | //define below
                           {32{opcode[5:0]==6'b000000 &
func[5:3]==3'b000}}                                                   &
Shifter_Result_tmp   |
                           {32{opcode[5:0]==6'b001111}}
                                      & {imm,16'b0}          |
                           {32{(~|opcode)&(func[5]|func[5:0]==6'b001001) |
opcode[5:3]==3'b001&(~&opcode[2:0]) | opcode[5:0]==6'b000011}}& ALU_Result_tmp       ;
```

在寄存器相关端口上添加了针对 MUL 指令的写使能信号、地址和数据的控制选择。

1.2 booth 算法

（1）状态机修改部分

```
//EX->IF:
        //REGIMM        2: opcode[5:0]=000001
        //I_branch      4: opcode[5:2]=0001
        //J             1: opcode[5:0]=000010
//EX->WB
        //R_Type        18: opcode[5:0]=6'b0 include JALR,JR //JR: rs->GPR[0]
        //I_calc        7: opcode[5:3]=001 include LUI
        //JAL           1: opcode[5:0]=000011
        //MUL           inst_MUL & MUL_done
//EX->LD
        //load          7: opcode[5:3]=100
        //EX->ST
        //store         5: opcode[5:3]=101
EX: begin
        if((opcode[5:0]==6'b000001) | (opcode[5:2]==4'b0001) | (opcode[5:0]==6'b000010) )
                next_state = IF;
        else if((~|opcode)| (opcode[5:3]==3'b001) | (opcode[5:0]==6'b000011))
                next_state = WB;
        else if(inst_MUL & MUL_done)
                next_state = WB;
        else if(opcode[5:3]==3'b100)
                next_state = LD;
        else if(opcode[5:3]==3'b101)
                next_state = ST;
        else
                next_state = EX;
```

（2）乘法运算部分

（2.1）mul.v

```
`timescale 10 ns / 1 ns
module mul(
        input clk,
        input rst,
        input [31:0] a,
        input [31:0] b,
```

```
        input run,
        output [63:0] result,
        output done
);
//booth 算法 将连续数位 1 转化为前后位的两个 1 相减 从而减少相加次数

        reg [2:0] i; //procedure
        reg [64:0] P;
        reg [31:0] a_tmp;
        reg [31:0] a_rev;// complement code
        reg [5:0] cnt;// count the time of loop
        reg isDone;
        reg inWork;

        always @ (posedge clk)
        begin
                if(rst)
                begin
                        i<=0;
                        P<=0;
                        a_tmp <=0;
                        a_rev <=0;
                        cnt <=0;
                        isDone <=0;
                        inWork <=0;
                end
                else if(run & ~inWork)//initial work
                begin
                        inWork <=1;
                        a_tmp <= a;
                        a_rev <= ~a +1'b1;
                        P <= {32'b0, b, 1'b0 };
                        i <= 1;
                        cnt <=0;
                end
                else if(run & inWork)
                begin
                        case(i)
                                //operate
                                1:begin
                                        if(cnt==32)
                                        begin
                                                cnt <=0;
                                                i <= 3;
```

```
                                        end
                                        else if( P[1:0] ==2'b00 | P[1:0] == 2'b11)
                                        begin
                                                P <= P;
                                                i <= 2;
                                        end
                                        else if( P[1:0] == 2'b01)
                                        begin
                                                P <= {P[64:33] + a_tmp , P[32:0]};
                                                i <= 2;
                                        end
                                        else if( P[1:0] == 2'b10)
                                        begin
                                                P <= {P[64:33] + a_rev , P[32:0]};
                                                i <= 2;
                                        end
                                end
                                //shift right
                                2:begin
                                        P <= {P[64] ,P[64:1]};
                                        cnt <= cnt +1'b1;
                                        i <=1;
                                end
                                //done flag
                                3:begin
                                        isDone <=1;
                                        i <= 4;
                                end
                                //refresh
                                4:begin
                                        isDone <=0;
                                        inWork <=0;
                                        i <=0;
                                end
                                default:
                                        i <=0;
                        endcase
                end
        end

        assign done =isDone;
        assign result = {64{done}} & P[64:1];

endmodule
```

简便起见，mul 模块中使用多周期实现 booth 算法。模块中设置输入输出端口 run 和 done 分别表示 booth 乘法器使能和结束信号。i 表示此时进行的运算操作，0 为初始态，1 为加减运算，2 位移位运算，3 为工作完成操作，4 为复位操作。a_tmp 和 a_rev 分别表示操作数 a 和 a 的补的寄存值。P 设置为 65 位，考虑操作结果位数及附加位。具体逻辑与 booth 算法逻辑相同。

（2.2）custom_cpu.v 相关端口

```
//Signals connected to MUL
        assign inst_MUL = opcode==6'b011100 & func == 6'b000010;
        assign MUL_run = current_state == EX & inst_MUL;
        always@(posedge clk) begin
                if(rst)
                        MUL_A_tmp <= 32'b0;
                else if(current_state[3])
                        MUL_A_tmp <= RF_rdata1;
        end

        always@(posedge clk) begin
                if(rst)
                        MUL_B_tmp <= 32'b0;
                else if(current_state[3])
                        MUL_B_tmp <= RF_rdata2;
        end

        mul mul_inst(
                .clk            (clk),
                .rst            (rst),
                .a              (MUL_A_tmp),
                .b              (MUL_B_tmp),
                .run            (MUL_run),
                .result         (MUL_Result),
                .done           (MUL_done)
        );
        always@(posedge clk) begin
                if(rst)
                        MUL_Result_tmp <= 64'b0;
                else if(current_state[4] & MUL_done) //EX
                        MUL_Result_tmp <= MUL_Result;
        end
```

逻辑与直接乘法基本一致，添加了 MUL_run 和 MUL_done 信号表示 booth 算法模块的启动和结束信号，使用模块例化代替*进行乘法运算。

2、软件部分（conv.c）

（1）卷积部分(convolution)

```
/* Fixed value for all loop */
        short FilterSize; // Filter include bias
        short InputSize;

        /* Reduce multiplications to speed things up, store the repeated result */
        short stride_x, stride_y; // x*S, y*S
        short FilterOuterAddr;  // Addr of head of Filter Channel
        short FilterInnerAddr;  // Addr of head of Filter in the Channel
        short InputAddr;        // Addr of head of a input volume
        short LineAddrInFilter; // Addr of head of the line in the Filter
        short LineAddrInInput;  // Addr of head of the line in the Input Volume

        /* Store the result of calculation temporarily, use int to avoid overflow */
        int temp;

        /* Store the offset of output*/
        short offset = 0;

        //Software will no care blank between vallue
        FilterSize = 1 + mul(weight_size.d2, weight_size.d3);
        InputSize = mul(input_fm_h, input_fm_w);

        // y always ahead to x, because use line as main sequence
        // put ahead x,y to save times of multiplication
        for (short no = 0; no < conv_size.d1; no++)
        { // num of output channel, each has a set of Filters
                FilterOuterAddr = mul(no, FilterSize); //ADDR of first item(bias) of filter
                for (short y = 0; y < conv_out_h; y++)
                { // height of output graph
                        stride_y = mul(y, stride); //num of raw
                        for (short x = 0; x < conv_out_w; x++)
                        {
                                stride_x = mul(x, stride); //num of column
                                temp = 0;
```

```
                                out[offset] = weight[FilterOuterAddr]; // Add bias to out
                                for (short ni = 0; ni < rd_size.d1; ni++)
                                {
                                        FilterInnerAddr = 1; //Consider bias
                                        InputAddr = mul(ni, InputSize);
                                        for (short ky = 0; ky < weight_size.d2; ky++)
                                        {
                                                LineAddrInFilter = mul(ky, weight_size.d3); //
width
                                                short ih = ky + stride_y - pad;
                                                LineAddrInInput = mul(ih, input_fm_w);//不考虑每行
填充的 8 字节
                                                for (short kx = 0; kx < weight_size.d3; kx++)
                                                {
                                                        short iw = kx + stride_x - pad;
                                                        //(kx+stride_x,...)is postion considering
pad
                                                        //(iw,ih) is real position ignoring pad,
consistent with memory
                                                        // the real point of origin has position
(pad,pad) considering pad
                                                        if (iw < 0 || ih < 0 || iw >= input_fm_w ||
ih >= input_fm_h)
                                                                continue; // not calculate padding
                                                        temp += mul(in[InputAddr + LineAddrInInput
+ iw] , weight[FilterOuterAddr + FilterInnerAddr +  LineAddrInFilter + kx]);
                                                }
                                        }
                                }
                                out[offset++] += (short)(temp >> FRAC_BIT);
                                /*
                                 *The temp bit is changed from 32 to 16
                                 *Since two 16-digit numbers are multiplied to produce 32 digits,
                                 *if the original decimal place is 10, the decimal place is 20
                                 *Therefore, after moving 10 bits to the right, the lowest 15 bits
are taken as the significant bits
                                 *The sign bit of the original multiplication of two numbers is taken
in the highest bit
                                 */
                        }
                }
        }
```

由于 convolution 的输出紧密排列，因此使用 offset 累加表示写位置以节省

乘法操作，FilterSize 和 InputSize 分别表示一个卷积核和一个输入特征图的大小。

在循环中，FilterOuterAddr 表示本次使用的卷积核第一个元素 bias 的位置(相对于weight 的偏移)，FilterInnerAddr 为卷积核内第一个权重值的位置(考虑 bias，位置为 1)，InputAddr 表示该输入图第一个元素的位置。

LineAddrInInput 和 LineAddrInFilter 分别表示所遍历到的该元素行首在输入图或卷积核当中的位置，在行主序的遍历过程中，可遍历每行的第一步对上述变量进行计算以减少乘法操作。

在具体计算中，kx 和 ky 表示卷积核中的横纵坐标，kx+stride_x 和 ky+stride_y 表示包含 pad 的输入特征图中的横纵坐标(但 pad 实际不占用内存)，iw 和 ih 表示输入特征图中该元素的真实坐标，即真实坐标的原点对应含 pad 输入特征图中坐标(pad,pad)

对于输出元素的计算，首先加上 bias 偏移量，然后特征图和卷积核对应位置按序进行相乘，此时使用 int 型中间变量 temp 进行暂存，以防止溢出和精度损失。累加完毕后，将 temp 右移小数位长度并进行类型转换，从而得到符合小数格式的 short 型变量，并将其加到 out 的对应元素上。

（2）pooling 部分

```
/* Fixed value for all loop */
short InputSize;
/* Reduce multiplications to speed things up, store the repeated result */
short stride_x, stride_y;
short InputAddr;
short LineAddrInInput;
short offset = 0;
//Treat Out by conv as input
InputSize = mul(input_fm_h, input_fm_w);
for (short no = 0; no < conv_size.d1; no++) //Channel of output
```

```
{
        InputAddr =mul(no, InputSize);
        for (short y = 0; y < pool_out_h; y++)
        {
                stride_y = mul(stride, y);
                for (short x = 0; x < pool_out_w; x++)
                {
                        stride_x = mul(stride, x);
                        //The minimum value of short type
                        short maxtmp = 0x8000;
                        for (short ky = 0; ky < KERN_ATTR_POOL_KERN_SIZE; ky++)
                        {
                                short ih = ky + stride_y - pad;
                                        //Address of head of line
                                        LineAddrInInput = mul(ih, input_fm_w);
                                for (short kx = 0; kx < KERN_ATTR_POOL_KERN_SIZE; kx++)
                                {
                                        short iw = kx + stride_x - pad;
                                        short curtmp;
                                        if (iw < 0 || ih < 0 || iw >= input_fm_w || ih >= input_fm_h)
                                                curtmp = 0;
                                        else
                                                curtmp = *(out + InputAddr + LineAddrInInput + iw);
                                        if (curtmp > maxtmp)
                                                maxtmp = curtmp;
                                }
                        }
                        out[offset++] = maxtmp;
                }
        }
}
```

此时将 convolution 所得结果当做 pooling 的输入，InputSize 表示输入大小，InputAddr 表示该组输入图首元素地址，LineAddrInInput 表示该遍历元素行首地址相对于该输入图的位置。

与 convolution 类似，在考虑 pad 后，有坐标 iw 和 ih 如代码所示。遍历过程中，设置 curtmp 表示当前遍历元素值,maxtmp 表示遍历元素所在方框内元素最大值,初始值为 0x8000,即 short 类型最小值。当 curtmp 大于 maxtmp 时将其替代，从而达到 Max Pooling 的效果。

（3）硬件加速器启动部分

```
#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
```

由于 start 位置权限为只写，done 位置权限为只读。将 start 位置赋值为 1 或者 0 即可启动或暂停硬件加速器，且可通过 done 位置是否等于 1 判断硬件加速是否结束。

（4）性能计数部分

```
Result res;
res.msec = 0;
bench_prepare(&res);

...

bench_done(&res);
printf("Total cycle: %d\n", res.msec);
```

调用 printf.h 和 printf.c 中所实现的 bench_prepare 和 bench_done 函数，即可从内存的对应位置获取 CPU 运行周期数，将其进行打印即可。

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）

问题 1：数据存放位置问题

开始时误解 PPT 中关于数据摆放位置的讲解，软件多余地考虑了硬件摆放中为了对齐使用的空白。在几个同学的讨论和测试中明白了软件和硬件存放位置的区别。

问题 2：数组设置问题

在某版代码中，首先使用数组存储卷积结果，再将其进行池化，结果出现了无法打印，运行超时的问题。之后将卷积结果同样存储至 out 位置，程序正确运行。猜测可能是由于内存有限或软件可见权限问题，设置一个较大的数组时可能出现内存不足、原数据摆放位置错误、中间数组不可访问等问题。

问题 3：硬件设计问题

在实现 booth 算法的过程中发现有多驱动、时序不匹配等问题。参照 emu 的报错日志进行修改，得到规范的写法。

问题 4：性能计数问题（未解决）

与几个完成该实验的同学讨论，发现性能计数结果无法打印。猜测可能是框架问题，有可能软件间协同存在问题、也可能进行性能计数时内存接口被占用导致无法访问周期数。

三、 对讲义中思考题（如有）的理解和回答

1、在软件算法中，如何避免出现溢出和精度损失

在 convolution 函数中需要对输入特征图和卷积核中的两个 short 类型数据进行乘法操作。为避免溢出和精度损失，使用 int 类型（即两倍 short 类型长度）进行乘法结果的暂存。由于 short 类型中有 10 位小数位，故所得 int 类型数据低 20 位表示小数。且乘法以补码形式进行，因此为得到符合 short 类型小数格

式的存储结果，将 int 型中间数据右移小数位长度（FRAC_BIT），然后转化为 short 即可实现截断。

2、不同实现方式的处理性能对比

测试任务及实现方式	时钟周期数
hw_conv（硬件加速器）	5744401
sw_conv（处理器、加减法代替乘法）	1263945240
sw_conv_mul（处理器，*实现乘法）	503569946
sw_conv_mul（处理器，booth 算法）	561432320

由上表可见，相比使用处理器直接运行软件算法，使用专门的硬件加速器将加快 2-3 个数量级，这体现了专用部件在实现具体操作上相比通用处理器的性能优势。

对比 sw_conv_mul 和 sw_conv 两个测试任务的时钟周期数，可见使用乘法指令将比用加减法直接实现乘法更加高效，性能将提高一倍以上。

对比使用*实现乘法和运用基于 booth 算法的多周期乘法器可见，时钟周期数并无明显差异，但从代码方面考虑硬件资源配置，使用 booth 算法将更加节约硬件资源。

特别的，本次实验所实现的 booth 乘法器使用多周期实现以模拟理论课所学的 booth 乘法过程，但并未充分发挥 booth 算法的性能优势，浪费了一定时钟周期数。笔者经过了解发现，当乘法指令较为密集的时候，可以考虑使用流水线阵列实现 booth 算法，将进一步提高 booth 算法的速度和资源优势。

四、 在课后，你花费了大约____5____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）

建议：

实验难度适中，可以考察一组卷积核中含多通道，从而增进同学对于多维的卷积算法与对应数据摆放及操作的理解。同时由于该实验大量涉及软件，可以在本实验中鼓励同学实现关于脚本、wrapper 等框架代码的补充修改，从而增进对整个项目运行过程的了解。

为方便同学进行代码测试，减少上板运行时间和资源消耗，建议项目提供只含软件的本地快速测试方法，帮助同学们定位错误。

致谢：

感谢常轶松老师和陈欲晓助教在硬件仿真和平台测试方面提供的帮助，感谢芦溶民助教对于讲义中歧义的讲解和及时修正，感谢陈远腾、徐步骥同学在本次测试软件上的帮助。