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。

1. 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。

1. 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。

1. 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。

1. 在课后，你花费了大约\_\_\_\_5\_\_\_\_小时完成此次实验。
2. 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。