

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 宋俊仪 学号 2023K8009929018 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、设计说明

1 乘法指令的添加

讲义中要求 ALU 添加乘法指令, 为了尽量不改变之前的代码逻辑和 ALUop 译码逻辑, 我做了如下设计:

由于原本 ALU 中已有 8 种操作, 而且 RISC-V 指令集中并没有直接使用 NOR 的指令要求, 所以之前的为 Prj3 所写的 ALU 中的 NOR 指令是“无用”的, 因此我选择了替换策略, 即: 将原来的 ALU 中的 NOR 指令替换为乘法 MUL 指令。这样我的 ALUop 设计就几乎不会有改变。通过查询相关资料, 得到了 RISC-V 乘法相关指令的指令码:(见图 1)

指令	funct7	funct3	opcode	描述
<code>mul</code>	0000001	000	0110011	乘法, 取结果低位
<code>mulh</code>	0000001	001	0110011	有符号乘法, 取结果高位
<code>mulhu</code>	0000001	011	0110011	无符号乘法, 取结果高位
<code>mulhsu</code>	0000001	010	0110011	混合符号乘法, 取结果高位

图 1: MUL 相关指令码

本次实验我们只需用到最基础的 `mul` 就可以完成, 于是根据查询到的指令码, 我在 ALU 中添加的 MUL 逻辑和 `turbo_cpu` 中的 ALUop 的译码逻辑如下(见图 2, 图 3):

```

wire [31:0]    and_result;
wire [31:0]    or_result;
wire [32:0]    add_result;
wire [31:0]    xor_result;
wire [31:0]    mul_result; //由于RISC-v不用nor，所以将之前的nor改为mul
assign mul_result = A*B;

```

图 2: ALU 中的 MUL 运算

```

assign ALU_op = (Opcode == 7'b0010011 | Opcode == 7'b0110011) && funct_length3 == 3'b111 ? 3'b000 :
  (Opcode == 7'b0010011 | Opcode == 7'b0110011) && funct_length3 == 3'b110 ? 3'b001 :
  (Opcode == 7'b0010011 | Opcode == 7'b0110011) && funct_length3 == 3'b100 ? 3'b100 :
  (Opcode == 7'b0010011 | Opcode == 7'b0110011) && funct_length3 == 3'b011 ||
  (Opcode == 7'b1100011 & funct_length3[2] & funct_length3[1]) ? 3'b011 :
  ((Opcode == 7'b0010011 | Opcode == 7'b0110011) && funct_length3 == 3'b010) ||
  (Opcode == 7'b1100011 & funct_length3[2] & ~funct_length3[1]) ? 3'b111 :
  (Opcode == 7'b0110011 && funct_length7[5] == 1'b1) ||
  (Opcode == 7'b1100011 & ~funct_length3[2]) ? 3'b110 :
  (Opcode == 7'b0110011 && funct_length7[0] == 1'b1) ? 3'b101 :
  3'b010 ;

```

图 3: ALUop 译码逻辑

2 卷积算法设计

• 卷积原理的理解

卷积的过程可以举一个例子来理解——侦探拿着这个放大镜在照片上扫描：1. 对准位置：侦探将“红色汽车”放大镜对准照片的左上角。2. 匹配打分：放大镜下的区域，和“红色汽车”的模板有多像？他会给出一个分数。如果这片区域看起来很像一辆红色汽车（比如有红色、有车轮形状），分数就很高。如果只是一片绿色的草地，分数就很低，甚至是负分。这个“打分”的过程，就是代码里的“乘加运算”。放大镜（卷积核）里的每个点都有一个权重值，它会和照片上对应像素的颜色值相乘，然后把所有的乘积加起来，得到最终的分数。3. 记录分数：侦探在一张新的、空白的“地图”上，在他刚刚检查过的位置，写下这个分数。这张新地图就是输出特征图。4. 移动放大镜：侦探将放大镜向右移动一小步（这个步长即 Stride），然后重复第 2 步和第 3 步。5. 扫描全图：他会一行一行地、从左到右、从上到下地用放大镜扫描整张照片，直到把所有位置的分数都记录在新地图上。

• 卷积代码逻辑

根据上面的分析和理解，卷积的代码设计如下（见图 4）：

```

1  选定一个输出通道 no (选定一个放大镜)
2  |
3  +--> 对于输入通道 ni = 0, 1, ...
4  |
5  +--> 选定一个输出像素 (y, x) (在新地图上找一个位置)
6  |
7  +--> 如果是第一个输入通道 (ni == 0), 则将偏置(bias)写入 out(no, y, x)
8  |
9  +--> 初始化累加器 unfixed_res = 0
10 |
11 +--> 对于卷积核内的每一点 (ky, kx) (透过放大镜看)
12 | |
13 | | +--> 找到输入图像对应的点 (ih, iw)
14 | | +--> unfixed_res += in(ni, ih, iw) * weight(no, ni, ky, kx)
15 | |
16 | | +--> out(no, y, x) += unfixed_res >> FRAC_BIT (将分数累加到输出像素)
17

```

图 4: 卷积的逻辑设计

由于不可以自定义数组,所以我采用如下方式来进行寻址,Convolution 核心循环计算代码设计如下,六层循环计算的代码注释在图中(见图 5):

```

1  // 遍历每一个输出通道 (或每一个卷积核)
2  for (no = 0; no < weight_size.d0; no++) {
3      // 遍历每一个输入通道
4      for (ni = 0; ni < rd_size.d1; ni++) {
5          // 遍历输出特征图的每一个像素 (y, x)
6          for (y = 0; y < conv_size.d2; y++) {
7              for (x = 0; x < conv_size.d3; x++) {
8                  int unfixed_res = 0; // 用于累加乘积的高精度中间结果
9
10                 // 在处理第一个输入通道时,加载偏置项(bias)
11                 // 假设权重布局为 [bias, w0, w1, ...], 每个卷积核权重部分大小为 (d2*d3), 偏置占1个short
12                 // 因此整个核(带偏置)的大小为 (d2*d3 + 1)
13                 if (ni == 0) {
14                     *out + output_offset + no*conv_size.d2*conv_size.d3 + y*conv_size.d3 + x =
15                         *(weight + no*weight_size.d1*(weight_size.d2*weight_size.d3+1)); /* 加载偏置 */
16                 }
17
18                 // 遍历卷积核
19                 for (ky = 0; ky < weight_size.d2; ky++) {
20                     // 计算当前卷积核位置对应输入特征图上的行号 (ih)
21                     int ih = ky + mul(y, stride) - pad;
22                     int offset = mul(ky, weight_size.d3); // 预计算卷积核内的行偏移
23                     for (kx = 0; kx < weight_size.d3; kx++) {
24                         // 计算当前卷积核位置对应输入特征图上的列号 (iw)
25                         int iw = kx + mul(x, stride) - pad;
26
27                         // 边界检查: 确保计算区域在输入特征图的有效范围内
28                         if (iw >= 0 && iw < input_fm_w &&
29                             ih >= 0 && ih < input_fm_h)
30                         {
31                             // 乘加累算 (MAC)
32                             // 权重地址计算: 跳过偏置项, 所以 +1
33                             unfixed_res += mul(
34                                 *(in + input_offset + ni*rd_size.d2*rd_size.d3 + ih*rd_size.d3 + iw),
35                                 *(weight + no*weight_size.d1*(weight_size.d2*weight_size.d3+1) + ni*(weight_size.d2*weight_size.d3+1) + offset + kx + 1)
36                             );
37                         }
38                     }
39                 }
40                 // 将累加结果进行定点数转换(右移FRAC_BIT位), 并加到输出像素上
41                 *out + output_offset + no*conv_size.d2*conv_size.d3 + y*conv_size.d3 + x += unfixed_res >> FRAC_BIT; /* 定点数转换 */
42             }
43         }
44     }
45 }

```

图 5: 卷积算法代码实现

3 池化算法设计

• 池化原理的理解

接着刚刚的卷积算法的例子，现在有了一张记录着“红色汽车”可能位置的“热力图”。但是这张图可能太大了，而且信息有些冗余。比如，放大镜在车头给了一个高分，在旁边的车门又给了一个高分。其实这都属于同一辆车，他不需要这么精确的位置。他只需要知道“大概在这个区域有一辆车”就行了。这时他需要做一次总结归纳，这个过程就是池化。

• 池化代码逻辑


遍历输出特征图的过程首先通过 `no` 索引外层循环遍历每个输出特征图，然后使用 `y` 和 `x` 索引的第二层和第三层循环遍历每个输出特征图的每一个元素。在每个池化窗口的开始，初始化 `max` 为 `SHORT_MIN`，以表示当前池化窗口的最大值。接下来，内层循环通过 `ky` 和 `kx` 索引遍历池化窗口的每一个元素。计算池化窗口内元素在输入特征图中的位置 `ih`，并检查计算出的 `ih` 和 `iw` 是否在输入特征图的有效范围内。在有效范围内，对池化窗口内的每个元素，找出最大值。最后，将池化窗口中找到的最大值 `max` 保存到输出特征图对应的位置。池化的核心代码逻辑如下（见图 6）：

```
1 // 遍历每一个通道
2 for (no = 0; no < conv_size.d1; no++) {
3     // 遍历池化输出的每一个位置 (y, x)
4     for (y = 0; y < pool_out_h; y++) {
5         for (x = 0; x < pool_out_w; x++) {
6             // 初始化最大值为 short 类型的最小值
7             short max = SHORT_MIN;
8             // 遍历池化核
9             for (ky = 0; ky < KERN_ATTR_POOL_KERN_SIZE; ky++) {
10                // 计算当前池化核位置对应输入特征图上的行号 (ih)
11                int ih = ky + mul(y, stride) - pad;
12                for (kx = 0; kx < KERN_ATTR_POOL_KERN_SIZE; kx++) {
13                    // 计算当前池化核位置对应输入特征图上的列号 (iw)
14                    iw = kx + mul(x, stride) - pad;
15
16                    // 边界检查：确保计算区域在输入特征图的有效范围内
17                    if (iw >= 0 && iw < input_fm_w &&
18                        ih >= 0 && ih < input_fm_h)
19                    {
20                        // 获取池化窗口内的像素值
21                        short tmp = *(in + input_offset + no*conv_size.d2*conv_size.d3 + ih*conv_size.d3 + iw);
22                        // 如果当前值大于记录的最大值，则更新最大值
23                        if (max < tmp) {
24                            max = tmp;
25                        }
26                    }
27                }
28            }
29            // 将池化窗口内的最大值写入到输出位置
30            // 注意输出是紧凑存储的，地址计算使用 pool_out_h 和 pool_out_w
31            *(out + output_offset + no*pool_out_h*pool_out_w + y*pool_out_w + x) = max;
32        }
33    }
34 }
```

图 6: 池化算法代码实现

4 硬件加速器的驱动

硬件加速器部分的设计不需要我们完成，这里只需要根据要求完成它的驱动程序即可，具体代码和注释见下图（见图 7）：



```

1  #ifdef USE_HW_ACCEL
2  // 启动硬件加速器并等待其完成
3  void launch_hw_accel()
4  {
5      // 定义指向MMIO寄存器的 volatile 指针
6      // volatile 关键字防止编译器优化对这些地址的读写操作
7      volatile int* gpio_start = (void*)(GPIO_START_ADDR);
8      volatile int* gpio_done = (void*)(GPIO_DONE_ADDR);
9
10     // 启动硬件加速器
11     // 通过位操作将启动寄存器的最低位置为1，同时保持其他位不变
12     *gpio_start = (*gpio_start & 0xfffffffffe) | 0x1;
13
14     // 轮询等待硬件完成
15     // 不断读取完成状态寄存器，直到其最低位变为1
16     while ((*gpio_done & 0x1) == 0)
17         ;
18
19     // 停止硬件加速器（或清除启动信号）
20     // 将启动寄存器的最低位清零
21     *gpio_start = *gpio_start & 0xfffffffffe;
22 }
23 #endif

```

图 7: 硬件加速器驱动代码实现

5 性能计数器实现

在 conv.c 文件中, 添加 printf.h 头文件, 并在代码中添加和我在 Prj4 中 bench.c 实现的性能计数器一样的代码, 实现周期和指令计数器, 代码如下(见图 8):

```

1 // 初始化性能计数器结构体
2 Result res;
3 res.msec      = 0;
4 res.ins       = 0;
5
6 // 准备性能计数（例如，清零计数器）
7 bench_prepare(&res);
8
9 // 结束性能计数（例如，读取计数器值）
10 bench_done(&res);
11 printf("=====\n");
12 printf("PERFORMANCE COUNTERS\n");
13 printf("=====\n");
14
15 // 打印性能结果
16 printf("TOTAL CYCLES      : %u\n", res.msec      ); // 总时钟周期
17 printf("TOTAL INSTRUCTIONS : %u\n", res.ins       ); // 总执行指令数
18
19 printf("=====\n");
20

```

图 8: 计数器代码实现

二、 实验中的难点及实验结果

1 实验中的难点

本实验中,我花费了大量时间学习理解卷积和池化算法,并借鉴了学长的代码思路。所以问题并不是很多,但是我发现我一开始所写的代码六层循环的复杂度较高,导致运行时间较长。后来我通过应用我在汇编语言课程上学习到的相关知识,发现可以通过将卷积和池化的代码逻辑进行优化,减少循环的层数和复杂度。

这里我主要实验了对循环部分的“代码外提”、“循环展开”、“强度削弱”等手段来实现优化的。对池化算法同样可以进行相关优化,下面仅以卷积算法为例。具体代码及相关注释如下(见图 9):

```

1  for (int no = 0; no < output_ch; no++) {
2      // 强度削弱: 获取当前输出通道的基址指针
3      short *p_out_channel = out + output_offset + no * output_ch_size;
4
5      // 遍历每一个输入通道
6      for (int ni = 0; ni < input_ch; ni++) {
7          // 强度削弱: 获取当前输入通道和权重的基址指针
8          const short *p_in_channel = in + input_offset + ni * input_ch_size;
9          // 权重指针跳过偏置 (+1)
10         const short *p_weight_kernel = weight + no * weight_per_output_ch_size + ni * weight_per_input_ch_size + 1;
11
12         // 遍历输出特征图的行
13         for (int y = 0; y < output_h; y++) {
14             // 强度削弱: 获取输出行的指针
15             short *p_out_row = p_out_channel + y * output_w;
16             const int start_ih = mul(y, stride) - pad;
17
18             // ---- 4. 循环展开 (x 循环) ----
19             int x = 0;
20             // 处理可以被展开的部分
21             for (; x <= output_w - UNROLL_FACTOR; x += UNROLL_FACTOR) {
22                 // 为展开的循环准备多个累加器, 减少数据依赖
23                 int acc[UNROLL_FACTOR] = {0};
24
25                 const int start_iw_base = mul(x, stride) - pad;
26
27                 // 遍历卷积核
28                 for (int ky = 0; ky < kernel_h; ky++) {
29                     const int ih = start_ih + ky;
30
31                     // 边界检查: 如果整行都在有效范围外, 直接跳过
32                     if (ih < 0 || ih >= input_fm_h) continue;
33
34                     // 强度削弱: 获取输入行和权重行的指针
35                     const short *p_in_row = p_in_channel + ih * input_fm_w;
36                     const short *p_weight_row = p_weight_kernel + ky * kernel_w;
37
38                     for (int kx = 0; kx < kernel_w; kx++) {
39                         // 对展开的每个点进行计算
40                         for (int i = 0; i < UNROLL_FACTOR; ++i) {
41                             const int iw = start_iw_base + mul(i, stride) + kx;
42                             // 边界检查
43                             if (iw >= 0 && iw < input_fm_w) {
44                                 acc[i] += mul(p_in_row[iw], p_weight_row[kx]);
45                             }
46                         }
47                     }
48                 }
49             }
50         }
51     }
52 }

```

图 9: 卷积代码优化

2 实验结果

通过对比我的周期计数器和指令计数器在同一个测试程序下的运行结果, 我发现使用乘法指令后, 以及使用硬件加速器后, 我的周期计数器和指令计数器都大大减少了, 其性能提升是数量级上的提升。具体的结果如下(见图 11, 图 12, 图 13):

```

45 starting convolution
46 starting pooling
47 =====
48 PERFORMANCE COUNTERS
49 =====
50 TOTAL CYCLES          : 1582515280
51 TOTAL INSTRUCTIONS    : 293536932
52 =====
53 Passed!
54 benchmark finished
55 time 15853.48ms

```

图 10: 未使用 MUL 以及硬件加速器运行结果

```

45 starting convolution
46 starting pooling
47 =====
48 PERFORMANCE COUNTERS
49 =====
50 TOTAL CYCLES          : 31860267
51 TOTAL INSTRUCTIONS    : 5333555
52 =====
53 Passed!
54 benchmark finished
55 time 346.54ms
56 reset: before MMIO access...
57 reset: MMIO accessed

```

图 11: 使用 MUL 但未使用硬件加速器运行结果

```

43 reset: before MMIO access...
44 reset: MMIO accessed
45 Launching task...
46 =====
47 PERFORMANCE COUNTERS
48 =====
49 TOTAL CYCLES          : 723664
50 TOTAL INSTRUCTIONS    : 69249
51 =====
52 Passed!
53 benchmark finished
54 time 36.22ms
55 reset: before MMIO access...
56 reset: MMIO accessed

```

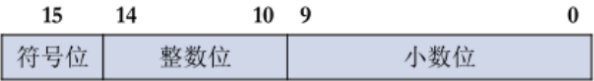
图 12: 使用 MUL 以及硬件加速器运行结果

三、思考题和实验总结

1 思考题

□ 输入图像、卷积核、输出特征图均使用**16-bit定点数**表示

- 即使用整数来表示小数，具体格式如下



- 在C语言里可以用short类型来表示定点数
 - 用整数运算代替小数运算
- 输入图像和卷积核的数据文件已按16-bit定点数格式存放数据
- 在软件算法实现中，需考虑如何避免出现溢出和精度损失
 - 乘法、加法运算的中间结果可使用32-bit定点数来表示
 - 请同学们思考如何扩展？

第一步：扩展与乘法

在计算每一个输出点时，我们先初始化一个 32 位的累加器，并将其清零。然后，我们取出第一个 16 位的输入值和第一个 16 位的卷积核值。当它们相乘时，我们允许结果自然地扩展到 32 位，并将这个完整的、未经任何损失的 32 位结果放入累加器中。

第二步：在 32 位空间内累加

接下来，我们进行第二次乘法，同样得到一个 32 位的结果，然后将它加到我们 32 位的累加器上。我们重复这个过程，直到所有相关的乘法和加法都完成。

第三步：收缩与量化

现在我们有 32 位的最终结果。但是，我们的任务要求输出一个 16 位的数。所以，我们需要把这个 32 位的结果“收缩”回 16 位。这个过程必须小心处理，分为两步：1. 调整精度：32 位的结果比 16 位结果拥有更多的小数位。我们需要通过右移操作来丢弃掉多余的、精度最低的小数位，使它的小数位数与目标 16 位格式对齐。这是一种有控制的、可接受的精度调整。2. 处理溢出：在调整完精度后，这个数的整数部分仍然可能比 16 位格式所能容纳的要大。因此，我们必须进行一次“溢出”检查：如果这个数大于 16 位能表示的最大值，就把它强行设置为最大值。如果它小于 16 位能表示的最小值，就把它强行设置为最小值。如果它恰好在 16 位的表示范围内，则直接转换。

2 实验总结

通过本次实验，我将理论知识与动手实践紧密结合，对深度学习中的核心算法及硬件加速技术有了更为深刻和具象的认识。这不仅是一次对课堂理论的验证，更是一场对工程实现能力的全面锤炼。从零开始实现卷积与池化算法，到亲手设计并驱动一个专用的硬件加速器，我深刻体会到了算法、软件与硬件之间相辅相成的紧密联系。

卷积与池化算法的实现，让我对深度学习的基石单元——特征提取，有了前所未有的直观理解。在编写卷积算法时，我反复调试嵌套循环、处理步长(stride)与填充(padding)的边界条件，并对内存访问模式进行优化，以求最大化计算效率。同样，在实现池化层(无论是最大池化还是平均池化)时，我直面了采样过程中信息筛选的本质。这个过程锻炼了我将抽象数学模型转化为高效、健壮代码的能力，特别是在资源有限的嵌入式环境中，如何平衡性能与精度显得尤为重要。

硬件加速器的设计与实现是本次实验中另一项令我获益匪浅的挑战。通过为卷积运算设计并行的乘加(MAC)阵列，我理解了硬件何以能够突破 CPU 的串行瓶颈，实现数量级的性能飞跃。我学习了如何编写控制逻辑来调度数据流，并通过软件驱动程序向加速器送入待处理的特征图和权重，再取回计算结果。当性能测试显示，使用硬

件加速器后，卷积和池化算法的执行时间得到数十倍的缩短时，我真切地感受到了硬件加速在现代计算，尤其是在人工智能应用中的决定性力量。

总而言之，这次实验使我在深度学习算法、硬件体系结构以及软硬件协同设计等多个维度上获得了显著的成长。我不仅掌握了卷积、池化等关键算法的底层实现细节，更具备了设计和应用硬件加速器的实践技能。这个过程培养了我严谨的工程思维和解决复杂问题的能力，这些宝贵的经验无疑将对我未来的学习和研究工作产生深远而积极的影响。

四、 实验所耗时间

在课后，你花费了大约 25(学习算法 5 小时,代码完成 12 小时,完成实验报告 6 小时) 小时完成此次实验。

致谢:感谢韩初晓同学在我的学习卷积和池化上的帮助和朱徐塬助教在实验 debug 过程中给予的指导