中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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。

1 实验设计及说明

本次实验项目"深度学习算法与硬件加速器"分为三个主要阶段:

1. **阶段 I: 软件实现与优化** — 在定制的 MIPS/RISC-V 处理器上, 使用 C 语言实现 2D 卷积和池化这两种深度学习核心算法。重点在于理解算法原理、数据在内存中的布局, 并通过软件优化技术(如代码外提、强度削减)提升其在无硬件乘法指令的处理器上的执行效率。

2. **阶段 II: 硬件指令扩展** — 为定制处理器增加硬件乘法指令 (`MUL`), 并评估该硬件支持对软件算法执行性能带来的变化。

3. **阶段 III: 硬件加速器集成** — 利用实验项目提供的专用卷积池化硬件加速器, 编写控制软件通过 I/O 地址空间访问来启动和管理加速器,实现算法的硬件卸载。

本报告将重点阐述阶段 I 和阶段 II 的设计、实现、遇到的问题以及性能分析。

1.1 软件算法实现与优化

核心任务是实现 `convolution()` 和 `pooling()` 两个函数。算法基于 16-bit 定点数进行运算, 输入图像为 28×28 单通道,卷积核为 5×5,共 20 组,最终输出 20 个通道的 12×12 特征图。

1.1.1 初步实现与性能瓶颈

根据讲义中 2D 卷积算法的伪代码, 我首先实现了一个功能直接、未优化的版本。该版本在多层嵌套循环中直接通过 `mul()` 函数计算各层级的索引地址。在实验框架中,纯软件版本的 `mul()` 函数通过一系列更基础的指令模拟乘法, 其执行开销巨大。因此, 这种在循环中大量调用 `mul()` 进行地址计算的实现, 导致纯软件版本执行严重超时,无法在规定时间内完成。

Listing 1: 未优化代码执行超时日志

```
starting convolution
time 1000217.90ms
custom cpu running time out
```

```
./software/workload/ucas-cod/benchmark/simple_test/dnn_test/riscv32/elf/sw_conv failed
Hit bad trap
```

1.1.2 软件优化:从循环乘法到强度削减的演进

为了解决初始版本因过多调用 `mul()` 函数而导致的超时问题,我采用了一系列系统性的优化策略,其核心思想是最大限度地减少循环内部,尤其是嵌套最深层循环中 `mul()` 函数的调用次数。这个过程可以分为几个清晰的演进步骤:

步骤一:识别性能瓶颈——初始的循环结构 首先,我们回顾未优化代码的核心循环结构,它在计算地址索引时充满了乘法。

Listing 2: 性能瓶颈:未优化循环中的多重乘法

```
// 仅为示意,非完整代码
for (no = 0; no < wr_size.d1; no++) {
    for (ni = 0; ni < rd_size.d1; ni++) {
        // ...
        for (y = 0; y < conv_out_h; y++) {
            for (x = 0; x < conv_out_w; x++) {
                // 每次循环都重新计算输出索引,包含两次乘法
                int output_index = mul(no, output_size_aligned) + mul(y, conv_out_w) + x;

                for (ky = 0; ky < weight_size.d2; ky++) {
                    for (kx = 0; kx < weight_size.d3; kx++) {
                        // 每次循环都计算输入坐标,包含两次乘法
                        int input_y = mul(y, stride) + ky;
                        int input_x = mul(x, stride) + kx;

                        // 索引计算中再次包含乘法
                        int input_index = mul(ni, input_size_aligned) + mul(input_y, RD_SIZE_D3) +
                            input_x;
                        // ...
                    }
                }
            }
        }
    }
}
```

在纯软件模式下,上述代码中的每一个 `mul()` 都意味着一次高昂的函数调用。一个 $20 \times 1 \times 12 \times 12 \times 5 \times 5$ 的卷积计算,会导致数百万甚至更多的 `mul()` 调用,这是性能的致命瓶颈。

步骤二:代码外提与一级强度削减 优化的第一步是应用代码外提和强度削减。我识别出那些随循环变量线性变化的乘法,并将其转换为加法。

- **外层循环 (no 和 ni):** 我引入了 `output_no_offset` 和 `input_ni_offset` 等变量。这些变量在进入循环前初始化,并在每次循环迭代结束时通过加法更新。例如,`output_no_offset` 在 `no` 循环的末尾通过 `+= output_size_aligned` 来更新,避免了在内层循环中反复计算 `mul(no, output_size_aligned)`。

- **中层循环 (y):** 同样地,我引入了 `output_y_offset` 和 `y_stride`。`output_y_offset` 在 `y` 循环末尾通过 `+= conv_out_w` 更新,避免了 `mul(y, conv_out_w)`。`y_stride` 则通过 `+= stride` 更新,避免了 `mul(y, stride)`。

经过这一步,代码演变为:

Listing 3: 演进过程:应用一级强度削减后的循环

```
// 优化思路示意
int output_no_offset = 0;
for (no = 0; no < wr_size.d1; no++) {
    // ...
    int output_y_offset = 0;
    int y_stride = 0;
    for (y = 0; y < conv_out_h; y++) {
        // ...
        // 输出索引现在是 O(1) 计算
        int output_index = output_no_offset + output_y_offset + x;

        for (ky = 0; ky < weight_size.d2; ky++) {
            // 输入y坐标计算仍然需要加法
            int input_y = y_stride + ky;
            // 但行偏移计算仍然需要乘法
            int input_row_offset = mul(input_y, RD_SIZE_D3);
            // ...
        }
        output_y_offset += conv_out_w;
        y_stride += stride;
    }
    output_no_offset += output_size_aligned;
}
```

这一步已经显著减少了乘法次数,但最内层的循环中仍然存在 `mul(input_y, RD_SIZE_D3)` 这样的乘法。

步骤三:彻底优化——最终实现 最终的实现将强度削减应用到了所有层级。我为每个循环变量都维护了相应的累加偏移量,使得每次迭代中地址的计算几乎完全是加法操作。

- **引入 `x_stride`:** 在 `x` 循环中也引入 `x_stride` 变量,通过 `x_stride += stride` 来更新,避免了 `mul(x, stride)`。
- **保留必要的乘法:** 最终,只有两种乘法被保留下来:
    1. **核心乘加 (MAC) 操作**: `sum += mul(in[...], weight[...])`。这是卷积算法的本质,无法消除。
    2. **动态行偏移计算**: 在 `ky` 循环内部,由于 `input_y` 的值是动态变化的 (`y_stride + ky`),其对应的行偏移量 `mul(RD_SIZE_D3, input_y)` 难以通过简单的累加实现(因为它不是对一个固定增量的累加)。虽然理论上可以进一步优化,但这会引入更复杂的控制逻辑,考虑到其带来的性能提升与复杂度的权衡,在此处保留这个乘法是合理的。同理,`mul(WEIGHT_SIZE_D3, ky)` 也被保留。

这个过程将一个性能极差的实现,逐步演化为一个高效的、几乎纯加法运算的地址计算模型。最终的代码在**??**中有详细展示。这个优化过程不仅解决了超时问题,也让我对"代码外提"和"强度削减"这两个编译优化中的经典技术有了远比课堂上更直观和深刻的认识。

1.1.3 优化后的核心代码实现

经过优化后，绝大多数用于地址计算的乘法都被消除，仅保留了最核心的卷积操作（输入像素值与权重值的乘法）以及少数难以避免的动态行偏移计算。

卷积函数关键代码 Listing 4 展示了优化后 convolution 函数的核心循环部分。

Listing 4: 优化后的 convolution 函数核心循环

```
// --- Convolution Main Loops ---
// Iterate over each output channel (filter).
for (no = 0; no < wr_size.d1; no++) {
    int bias = weight[bias_index]; // Fetch bias for the current filter

    // Since RD_SIZE_D1 is 1, `ni` loop iterates only once.
    // Offsets are reset for each filter.
    input_ni_offset = 0;
    weight_ni_offset = 0;
    for (ni = 0; ni < rd_size.d1; ni++) {
        output_y_offset = 0; // Reset row offset for each input channel
        y_stride      = 0; // Reset strided y-coord for each input channel
        for (y = 0; y < conv_out_h; y++) {
            x_stride = 0; // Reset strided x-coord for each output row
            for (x = 0; x < conv_out_w; x++) {
                int sum = 0; // Accumulator for the current output pixel
                int output_index = output_no_offset + output_y_offset + x;

                if (ni == 0) {
                    out[output_index] = bias; // Initialize with bias
                }

                // --- Core Multiply-Accumulate (MAC) Loop ---
                for (ky = 0; ky < weight_size.d2; ky++) {
                    int input_y = y_stride + ky - pad;
                    // Pre-calculate row offsets to reduce `mul` in the innermost loop
                    int input_input_y_offset = mul(RD_SIZE_D3, input_y);
                    int weight_input_y_offset = mul(WEIGHT_SIZE_D3, ky);
                    for (kx = 0; kx < weight_size.d3; kx++) {
                        int input_x = x_stride + kx - pad;

                        if (input_y >= 0 && input_y < input_fm_h && input_x >= 0 &&
                            input_x < input_fm_w) {
                            // Calculate flattened indices
                            int input_index =
                                input_ni_offset + input_input_y_offset + input_x;
                            // The `+ 1` skips the bias value
                            int weight_index = weight_no_offset + weight_ni_offset +
                                               weight_input_y_offset + kx + 1;
                            // Perform the essential multiplication
                            sum += mul(in[input_index], weight[weight_index]);
                        }
```

```
                }
            }
            x_stride += stride; // Strength reduction for x
            // Add quantized sum to the (bias-initialized) output pixel
            out[output_index] += sum >> FRAC_BIT;
        }
        // Strength reduction for y
        output_y_offset += conv_out_w;
        y_stride      += stride;
    }
    // Update offsets for the next input channel
    input_ni_offset += input_size_aligned;
    weight_ni_offset += weight_size_aligned;
  }
  // Update base offsets for the next output filter
  bias_index      += weight_size_aligned;
  output_no_offset += output_size_aligned;
  weight_no_offset += weight_size_grouped;
}
```

该函数通过六层嵌套循环实现 2D 卷积。最外层循环 (no) 遍历每个输出通道(滤波器)。对于每个滤波器,先获取其偏置 (bias)。由于本实验输入通道数 (ni) 为 1,内层 ni 循环仅执行一次。接下来的 y 和 x 循环遍历输出特征图的每个像素点。对于每个输出点,首先用 bias 初始化,然后通过最内层的 ky 和 kx 循环,将卷积核滑过输入图像的对应区域,执行乘加 (MAC) 操作,并将结果累加到 sum 中。最后,对 sum 进行定点量化(右移 FRAC_BIT 位)并加到已初始化的输出像素上。整个过程中,各级地址偏移量均通过累加(强度削减)进行更新,以避免不必要的乘法。

池化函数关键代码 Listing 5 展示了优化后 pooling 函数的核心循环。

Listing 5: 优化后的 pooling 函数核心循环

```
// --- Pooling Main Loops ---
// Iterate over each feature map (channel).
for (no = 0; no < conv_size.d1; no++) {
  // Calculate base offset for the current input feature map
  unsigned current_channel_input_base = mul(no, mul(input_fm_h, input_fm_w));
  oy_offset = 0; // Reset y-stride offset for each feature map
  for (int oy = 0; oy < pool_out_h; oy++) {
    ox_offset = 0; // Reset x-stride offset for each output row
    for (int ox = 0; ox < pool_out_w; ox++) {
      short max = -32768; // Initialize max to SHRT_MIN

      // --- Find Max Value in Pooling Window ---
      for (int ky = 0; ky < KERN_ATTR_POOL_KERN_SIZE; ky++) {
        int input_y = oy_offset + ky - pad;
        // Calculate row offset for the input feature map
        input_y_offset = mul(input_y, input_fm_w);
        for (int kx = 0; kx < KERN_ATTR_POOL_KERN_SIZE; kx++) {
          int input_x = ox_offset + kx - pad;
```

```
                // Boundary check
                if (input_y >= 0 && input_y < input_fm_h && input_x >= 0 &&
                    input_x < input_fm_w) {
                    int input_index =
                        current_channel_input_base + input_y_offset + input_x;
                    if (in[input_index] > max) {
                        max = in[input_index];
                    }
                }
            }
        }
        out[output_offset] = max; // Write max value to output
        output_offset++;
        ox_offset += stride; // Strength reduction for x
    }
    oy_offset += stride; // Strength reduction for y
  }
}
```

该函数通过五层嵌套循环实现 Max Pooling。最外层循环 (no) 遍历每个特征图通道。内层的 oy 和 ox 循环遍历池化后输出特征图的每个位置。对于每个输出位置,初始化一个最大值变量 max 为最小值。然后,通过最内层的 ky 和 kx 循环,在输入特征图的对应 2×2 窗口内寻找最大像素值,并更新 max。窗口扫描完成后,将找到的 max 值写入一维展开的输出数组中。同样,输入坐标的计算也采用了强度削减来优化性能。

1.2 硬件乘法指令 (MUL) 实现

为评估硬件乘法对性能的提升,我在定制的 RISC-V 处理器中添加了 MUL 指令的支持。

1.2.1 ALU 设计修改

我分析了 RISC-V 指令集,发现其中并没有 MIPS 指令集中的 NOR 指令。因此,我安全地复用了原先为 NOR 指令预留的 ALU 操作码 3'b101 来实现乘法运算。在 ALU 模块 (alu.v) 中,我添加了对该操作码的判断,并直接使用 Verilog 的 * 运算符来实现 32 位乘法。尽管这会生成一个复杂的组合逻辑电路,但其优势是在一个时钟周期内即可得到乘法结果。

Listing 6: ALU 中添加乘法功能的关键逻辑

```
// note: when using dnn_accelerator, switch nor to mul because there's no nor inst in riscv32
assign Result =
   (ALUop == 3'b000) ? (A & B) :
   (ALUop == 3'b001) ? (A | B) :
   (ALUop == 3'b010) ? (A + B) :
   (ALUop == 3'b100) ? (A ^ B) :
   (ALUop == 3'b101) ? (A * B) : // MUL operation using reused opcode
   (ALUop == 3'b110) ? sub_result[`DATA_WIDTH-1:0] :
   (ALUop == 3'b011) ? {{(`DATA_WIDTH-1){1'b0}}, sub_carryout} :
   (ALUop == 3'b111) ? {{(`DATA_WIDTH-1){1'b0}}, slt_result} :
   {`DATA_WIDTH{1'bx}};
```

1.2.2 译码逻辑修改

相应地，在 CPU 的译码单元中，我定义了新的本地参数 `ALU_MUL`，并修改了 R 型指令的译码逻辑，使其能够识别 `MUL` 指令（`opcode=0110011`, `funct3=000`, `funct7=0000001`）并生成对应的 ALU 操作码。为了区分 `ADD` 和 `MUL`（它们共享相同的 `funct3`），我利用了 `funct7` 字段的第 0 位作为判断依据。

Listing 7: IDU 中添加 `MUL` 指令译码的关键逻辑

```
// ALU Operation (alu_op) Selection Logic
assign alu_op =
    is_R ? // R-Type instructions (opcode: 0110011)
        (funct3 == 3'b000 && funct7[0] ? ALU_MUL     : // MUL (funct7[0] indicates MUL)
         funct3 == 3'b000 ? (funct7_5 ? ALU_SUB : ALU_ADD) : // ADD/SUB
         // ... other R-type instructions
                        ALU_XXX) :
    // ... other instruction types
    ALU_XXX;
```

2 实验结果与分析

经过软件优化和硬件指令扩展，我分别对硬件加速器 (`hw_conv`)、纯软件优化版 (`sw_conv`) 和使用硬件乘法指令的软件优化版 (`sw_conv_mul`) 进行了测试。

2.1 性能对比分析

下表总结了三种实现方式在 FPGA 上运行的性能计数器数据。

表 1: 三种实现方式性能对比

性能指标	硬件加速器 (hw_conv)	软件加乘法指令 (sw_conv_mul)	纯软件 (sw_conv)
执行时间 (ms)	**117.14**	**4378.95**	**234319.93**
总周期数	**1,032,910**	**428,208,936**	**1,946,775,199**
指令数	10,307	5,406,546	329,178,209
访存指令数 (Ld+St)	3,477	663,293	2,471,944
ALU 操作数	10,312	4,513,673	196,770,090
IW Stalls	929,195	359,852,678	545,745,747
RDW Stalls	42,289	44,363,896	105,424,737

从 Table. 1 中可以得出以下结论：

1. **硬件加速器 vs 软件实现**：硬件加速器的性能遥遥领先，其执行时间（117ms）和总周期数（约 100 万）相比任何软件实现都有数量级的优势。与最优的软件实现（软件 +MUL 指令，4379ms）相比，性能提升了约 **37.6 倍**。这充分体现了专用硬件在执行特定计算密集型任务（如卷积）时的巨大效率优势。硬件加速器通过并行计算和流水线化的数据通路，将复杂的算法固化为电路，避免了取指、译码等通用处理开销，并大大减少了总的访存次数。

2. **硬件乘法指令的效果**：对比“软件 +MUL 指令”和“纯软件”两种实现，硬件 `MUL` 指令带来了显著的性能提升。执行时间从约 234 秒锐减到约 4.4 秒，性能提升了约 **53.5 倍**。从性能计数器看，总周期数减少了约

78%。最核心的区别在于指令数和 ALU 操作数：纯软件版本需要执行高达 3.29 亿条指令来模拟乘法，而硬件乘法版仅需 540 万条指令。这表明，对于乘法密集型应用，提供一条单周期的硬件乘法指令是至关重要的，它可以极大地减少指令数量和执行周期。

3. **软件优化的重要性**：回顾 Section 1.1 中提到的优化过程，如果没有通过代码外提和强度削减将软件中的 `mul()` 调用次数降至最低，即使有硬件 `MUL` 指令，程序性能也无法达标，甚至纯软件版本会直接超时。这证明了算法层面的优化与硬件层面的支持相辅相成，缺一不可。

4. **性能瓶颈分析**：在两个软件版本中，IW Stalls（指令等待）和 RDW Stalls（写回数据冒险等待）都占据了大量的周期。这表明即使经过优化，处理器流水线仍然因为数据依赖和内存访问延迟而频繁暂停，这在后续的 Cache 设计实验中将是重点优化的方向。

3 实验中遇到的问题与思考

本次实验过程中的主要挑战集中在软件算法的优化和数据类型的正确处理上。

3.1 软件性能优化过程

如 Section 1.1 所述，最初版本的软件实现由于在多层循环中大量使用 `mul()` 函数进行地址计算，导致纯软件模拟乘法时严重超时。这让我深刻认识到，在没有硬件原生支持的情况下，高开销操作（如乘法）的调用频率是性能的致命瓶颈。

解决这个问题的过程是一个系统性的优化过程。我逐层分析了循环结构，识别出那些不随当前循环变量变化的“循环不变量”，并将它们的计算（如 `output_size_aligned`）提到循环之外。更关键的是，对于随循环变量线性变化的地址偏移量，我应用了“强度削减”技术，用代价低廉的加法累积来代替代价高昂的乘法。例如，`y_stride` 在 `y` 循环中通过 `y_stride += stride` 更新，而不是每次都计算 `mul(y, stride)`。这个过程不仅解决了超时问题，也让我对编译器优化中的经典技术有了更直观的理解。

3.2 定点数运算中的溢出和精度损失问题

在 `convolution` 函数中，累加器 `sum` 的类型可以定义为 `int`。同时，两个 16-bit 定点数（`short` 类型）相乘的结果也使用 32-bit 的 `int` 来存储，这样，在整个累加过程中不会发生溢出。在所有乘积累加完成后，再进行 `>> FRAC_BIT` 的定点量化移位，这样，可以避免精度损失。

3.3 对讲义中思考题的回答

问题：不考虑边界填充的算法，如果使用边界填充，应如何修改？

回答：如果使用边界填充 (Padding)，算法需要在计算输入坐标时进行调整。当前的代码通过 `pad` 变量来处理，在计算 `input_y` 和 `input_x` 时，是`... - pad`。这实际上是假设输入图像在逻辑上已经被填充，而我们访问的是这个逻辑大图上的坐标。一个支持 Padding 的算法修改方式大概形式如下：

1. **调整输入坐标计算：**计算输入坐标时，应先减去 padding 值，即 `input_y_coord = y * stride + ky - pad`。这正是当前代码的实现方式，它将输出网格上的点 `(y, x)` 映射回带有 padding 的逻辑输入图像上的坐标。

2. **修改边界检查逻辑：**在进行 `if (input_y >= 0 && input_y < input_fm_h ...)` 的边界检查时，如果计算出的 `input_y` 或 `input_x` 落在了原始图像之外但在 padding 区域之内，那么参与乘加运算的输入值应为 0。当前代码通过边界检查直接跳过了这些区域的计算，这等效于“零填充”(Zero-Padding)，因为 `sum` 不会增加任何值。如果需要支持其他填充方式，则需要在此处添加更复杂的逻辑。

因此，我的代码实现隐式地支持零填充。若要使其更明确或支持其他填充方式，可以在边界检查的 `else` 分支中处理填充值，而不是简单地跳过。

4 实验所耗时间

在课后，我花费了大约 15 小时完成此次实验。其中，大部分时间消耗在理解讲义内容以及提升算法效率上。

5 实验总结与心得体会

本次实验项目“深度学习算法与硬件加速器”，通过三个层层递进的阶段，让我对软硬件协同设计的重要性有了极为深刻和具体的认识。首先，在纯软件实现阶段，我直面了在无原生硬件支持下，通过软件模拟高开销操作（乘法）所带来的巨大性能瓶颈。通过对卷积和池化算法进行细致的代码外提和强度削减优化，我成功地将一个严重超时的程序变得可以运行，这不仅锻炼了我的算法分析和优化能力，也让我亲身体会到了编译器优化技术的威力与必要性。同时，处理 16-bit 定点数运算中潜在的溢出和精度损失问题，也加深了我对底层数据表示和嵌入式编程严谨性的理解。

其次，在为定制处理器添加硬件 `MUL` 指令的阶段，我体验了从指令集扩展、ALU 硬件修改到译码逻辑更新的全过程。性能测试结果戏剧性地展示了单条硬件指令所能带来的巨大飞跃——超过 50 倍的性能提升。这让我清晰地看到，针对特定应用领域的计算特性，在硬件层面提供原生支持是提升性能最直接、最有效的方式之一。

最后，通过对比硬件加速器的性能，我理解了专用硬件（ASIC/FPGA 加速器）为何在深度学习等领域不可或缺。硬件加速器通过高度并行和流水化的架构，将算法逻辑固化为硬件电路，完全绕过了通用处理器的取指-译码-执行循环，从而实现了极致的能效比和性能。三者性能的巨大差异——纯软件的分钟级、软件 +MUL 的秒级、硬件加速器的毫秒级——生动地诠释了从通用计算到专用计算的性能演进路径。

总而言之，本次实验是一次完整的软硬件协同设计之旅。我不仅掌握了 2D 卷积和池化的算法实现，更重要的是，我学会了如何从系统层面分析性能瓶颈，并分别在软件算法、处理器指令集和专用硬件三个不同层次上进行针对性的设计与优化。这段经历极大地提升了我的系统思维和工程实践能力，为我未来从事计算机体系结构或高性能计算领域的研究与开发打下了坚实的基础。

6 致谢

感谢宁彦祯同学和宋俊仪同学，他们的真知灼见给了我很大的启发；感谢朱徐塬学长及各位助教，您们的帮助对我有非常大的指导意义。