



lenet 实验报告

郭兵

目录

1 代码仿真及原理分析	1
1.1 Lenet 结构	1
1.2 lenet 总体结构	1
1.3 controller 模块分析	2
1.4 conv_in1 模块分析	3
1.5 RELU 模块	5
1.6 max_pool 模块	5
1.7 conv_in4 模块	6
1.8 digit_produce 模块	7
2 代码可综合设计及上板验证	9
2.1 资源消耗	9
2.2 时序分析	9
2.3 功率消耗	10
2.4 上板验证	10
2.5 实际效果	12
3 改进思路	13

1 代码仿真及原理分析

1.1 Lenet 结构

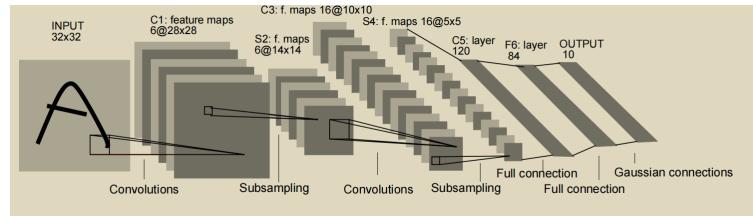


图 1: lenet 网络结构

如图 1 所示, LeNet 共分为 7 层, 分别是: C1 卷积层,S2 池化层,C3 卷积层,S4 池化层,C5 卷积层,F6 全连接层,OUTPUT 全连接层。

但本实验并非标准的 lenet, 本实验使用的 lenet 结构如下:

- C1 卷积层: 其输入矩阵为 32x32, 卷积核为 5x5x4, 步长为 1
- S2 池化层: 输入特征图矩阵为 28x28x4, 采样区域为 4x4, 步长为 4
- F3 全连接层: 输入特征图矩阵为 7x7x4, 卷积核为 7x7x4x10, 输出为包含 10 个元素的一维向量 (表示 10 种不同数字的概率, 值越大概率越大)

1.2 lenet 总体结构

lenet 项目总体结构如图 2 所示, 每次计算完成后, 都要将特征图数据存入 sram 中, 然后 ctrl 模块产生坐标读出数据供下一级使用。

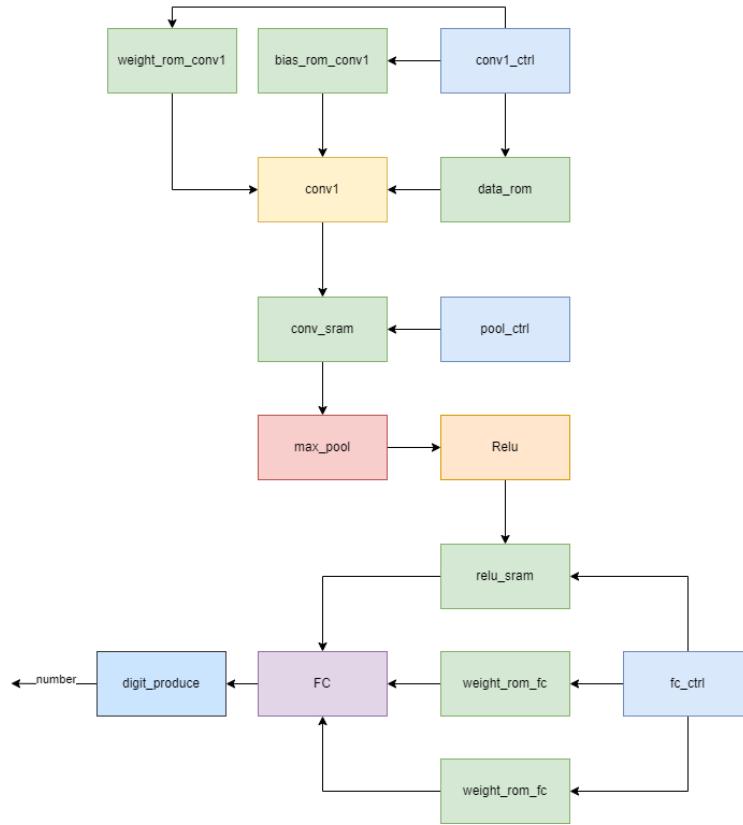


图 2: lenet 项目结构

1.3 controller 模块分析

信号	作用
go(input)	整体卷积操作开始进行信号
first_data(output)	一次卷积操作开始信号
last_data(output)	一次卷积结束信号
aa_bias(output)	bias 的坐标
aa_data(output)	data 的坐标
aa_weight(output)	weight 的坐标
cena(output)	rom 读使能信号
ready(output)	整体卷积操作完成信号

表 1: controller 信号作用

该模块算是一个抽象功能模块, 其主要功能就是产生相应的坐标及卷积结束信号。C1 卷积层输入为尺寸 32×32 的图片, 通过 6 个 5×5 的卷积核与输入图像卷积生成 $28 \times 28 \times 6$ 的特征图, 首先列出表 1 信号, 之后均会以下列信号来展开讲解。

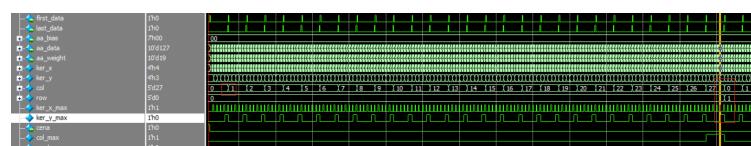


图 3: 一次卷积操作

如图 3 所示, 当 first_data 信号升高后, 一次卷积操作开始, 生成相应的卷积坐标, 由于使用的 5x5 的卷积核, 且输入图片为 32x32, 故可看到数据卷积坐标生成为 0,1,2,3,4,32,33..... 对应的正是这一次要被卷积的数据坐标, 一次卷积结束后(即 last_data 信号拉高), col 计数器加 1, 说明完成了输出图坐标 (0,0) 的卷积 (相对于输出来说), 准备生成输出图坐标的 (1,0) 的卷积结果, 当 col 计数器为 27 时且 last_data 信号升高, 此时 row 计数器加 1, 说明输出图的第 0 行卷积结果已经生成完成, 准备第 1 行卷积结果生成, 此时被卷积的数据坐标将从 1,2,3,4,5,33,34..... 开始, 也就是坐标比之前加了 1, 如图 3 所示, 输入为 5x5 的图像, 输出为 3x3 的图像, 卷积核大小为 3x3, 卷积核顺序扫描输入图片, 并作点积, 然后生成输出图片, 该原理可同样类比到 32 x32 图片卷积。

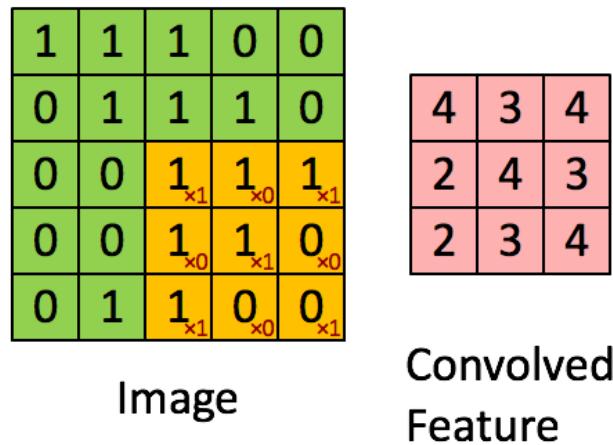


图 4: 卷积原理

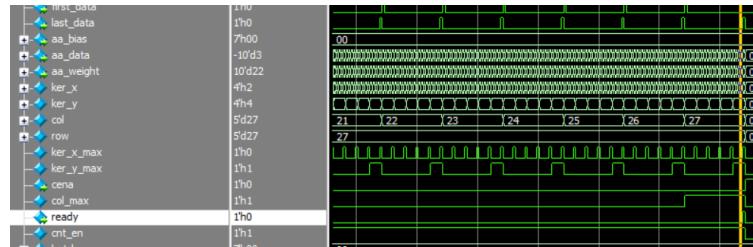


图 5: 卷积结果

当 32x32 图的卷积结果生成完毕后, 如图 5 所示, 此时 row 为 27, 且 last_data 信号为高电平, 此时 ready 信号升高, 说明卷积完成。

至于为什么输入 32x32 的图片输出 28x28 的图片, 其计算公式为 $size_o = size_i - size_{kernel} + 1$

1.4 conv_in1 模块分析

之前的 controller 作用为生成卷积坐标, 而该模块就是对这些坐标的数据进行 MAC 计算, 该模块包含了 conv_in1_acc 模块, 而 conv_in1_acc 又包含了 mac 模块, 故只对该模块核心功能讲解。

信号	作用
aa_en(input)	数据有效信号
aa_first_data(input)	一次卷积操作开始信号
aa_last_data(input)	一次卷积结束信号
image(input WD*INPUT_NUM)	特征图数据
bias(input WD_BIAS*OUTPUT_NUM)	bias 的值
weight(input WD*INPUT_NUM*OUTPUT_NUM)	weight 的值
q(output WD*OUTPUT_NUM)	输出特征图数据
q_en(output)	输出数据有效信号

表 2: conv_in1 信号作用

首先是模块信号的作用, 如表 2 所示。

该模块的核心代码如图 6 所示, 可以看到本 lenet 网络的第一个卷积层使用了 4 个 5x5 的卷积核, 故第一次卷积会生成四张特征图。

```

genvar i;
generate
    for (i=0; i<OUTPUT_NUM; i=i+1) begin : acc_in1
        conv_in1_acc #((
            .INPUT_NUM (INPUT_NUM),
            .SHIFT     (SHIFT)
        )) u_conv_in1_acc(
            .clk,
            .rst_n,
            .en,
            .first_data,
            .last_data,
            .image_in,
            .bias_in,
            .weight_in,
            .q_en,
            .q
        );
    end
endgenerate

```

图 6: 核心代码

当 controller 模块生成坐标后, 存放 weight,bias,data 数据的 rom 会读出相应坐标的数据(延迟一个周期), 然后该模块正好收到数据, 对相应的图像数据进行 mac 计算, 接下来以卷积核 0 作为例子, 来讲解具体操作。

如图 7 所示, 红框圈住的部分为一个图像数据 8'h05, 首先列出一次卷积操作公式如下:

$$data_o = data_i * weight + bias \quad (1)$$

由图 7 可看出 bias 为 12'f3a, weight 为 8'h3b, 故卷积结果为:

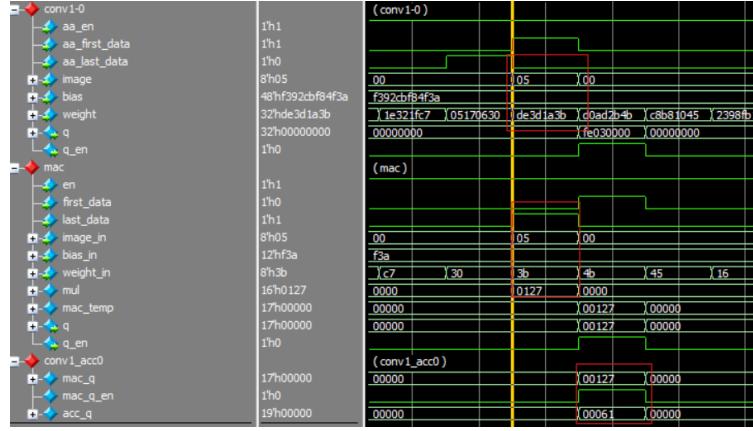


图 7: 卷积结果

$$data_o = signed(8'h05 * 8'h3b) + signed(12'f3a) = 8'h61 \quad (2)$$

很显然与图 7 的 acc_q 信号相同, 而该信号则是 conv1 卷积层的

1.5 RELU 模块

该模块作用是引入非线性变换, 使得神经网络可以学习更复杂的模式和特征。它的主要优点是计算简单、不存在梯度消失问题, 并且能够加速收敛和提高模型的泛化能力, 其公式如下:

$$f(x) = max(0, x) \quad (3)$$

relu 函数的仿真结果如图 8 所示, 可以看到当输入为 32'hff02ffff, 输出为 32'h000020000, 也就是将小于 0 的数据改为 0, 大于等于 0 的数据不变。

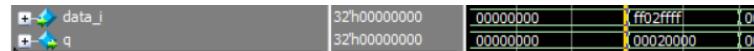


图 8: relu 仿真

1.6 max_pool 模块

接下来数据就进入池化层, 池化操作也有一个控制模块, 由于之前讲解过, 故不过多描述, 需要注意的是, 池化层的控制模块只需要输出特征图的坐标就可以, 而不需要去输出权重和 bias 的坐标。

本模块采样区域为 4x4, 其步长为 4, 故 28x28x4 的特征图经过池化后变为 7x7x4 的特征图, 本次使用的是最大池化, 其基本原理图可如图 9 所示, 该示例的特征图为 4x4, 采样区域为 2x2, 步长为 2, 故会将特征图高度减半, 宽度减半变为 2x2, 只看第一个采样区域, 其中的数字为 1,5,6,1, 根据最大池化, 只选择采样区域最大值, 故选择 6, 其他采样区域同理。

MAX POOLING

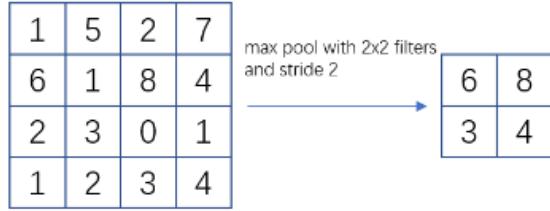


图 9: max_pooling

我们看一组池化操作仿真, 如图 10 所示, 由于采样区域为 4×4 的矩阵, 故一次操作是在 16 个数据中做比较, 我们只看特征图 4(特征图总共 4 张, 特征图 1 对应的是 $\text{data_in}[7:0]$, 依次类推), 将该矩阵还原为:

```
00 00 00 00
fe fd fe 01
fb f8 ff 09
f8 f7 07 0f
```

故根据最大池化原理显然可知其输出为 $8'h0f$, 这也与图 10 的 q 结果一致。



图 10: max_pooling 仿真

最后上代码讲解, 如图 11 所示,max_temp 当第一个数据来的时候赋值为第一个数据, 之后如果有大于它的数据, 将大于他的数据赋值给本身, 这样当 last_data 信号到来时,max_temp 中存放的就是最大的数据。

```
genvar i;
generate
    for (i=0; i<INPUT_NUM; i=i+1) begin : max_pool_num
        always @ (posedge clk or negedge rst_n)
            if (!rst_n)
                max_temp[(i+1)*WD-1:i*WD] <= 0;
            else begin
                if (!len)
                    max_temp[(i+1)*WD-1:i*WD] <= 0;
                else begin
                    if (first_data)
                        max_temp[(i+1)*WD-1:i*WD] <= d_in[(i+1)*WD-1:i*WD];
                    else if ($signed(d_in[(i+1)*WD-1:i*WD]) > $signed(max_temp[(i+1)*WD-1:i*WD]))
                        max_temp[(i+1)*WD-1:i*WD] <= d_in[(i+1)*WD-1:i*WD];
                end
            end
        end
    endgenerate
```

图 11: max_pooling 代码

1.7 conv_in4 模块

该模块也即是全连接层, 当池化操作完成后, 经过 relu 函数, 然后送入全连接层, 此时全连接层的输入特征图尺寸为 $7 \times 7 \times 4$, 经过全连接层后输出为 10 个向量, 其中最大的值对应的

坐标即代表识别的数字。

具体操作为首先将 $7 \times 7 \times 4$ 的向量与 $7 \times 7 \times 4$ 的权重内积, 生成 1 个输出, 而总共有 10 组 $7 \times 7 \times 4$ 的权重, 分别对应了 0,1,2,3,4,5,6,7,8,9, 故本次操作会生成 10 个结果, 也即是 10 个数字的预测结果。

由图 12 不难看出, 总共输出了 10 组 q , 每组 q 的值代表了对应数字的预测值。

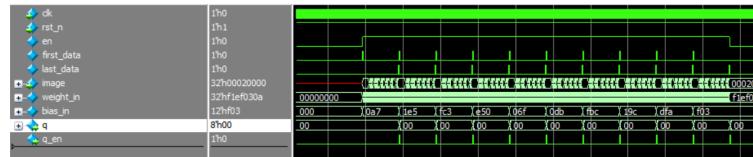


图 12: fc 层仿真

1.8 digit_produce 模块

该模块主要功能就是将全连接层送入的 10 个数据进行比较(该比较操作和最大池化相似, 不过需要保存最大值对应的坐标), 值最大的 q 对应的下标就是识别的数字。如图 13 所示, 当 ready_temp 升高时, 一次识别操作结束, 此时查看 digit 的值为 7, 故识别的数字为 7。

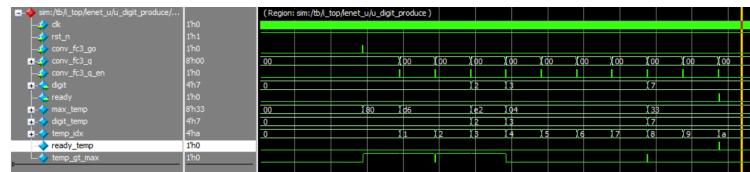


图 13: digit_produce 仿真

如图 14 所示, 当 conv_fc3_q_en 为高电平时, temp_idx 进行 +1, 当 conv_fc3_q_en (数据有效信号)和 temp_gt_max (输入数据比暂存数据大)均为高电平时, digit_temp 赋值为此时输入数据 q 的下标, 当 ready_temp 升高时, 此时对应的 digit 就是识别数字, 至此完成主要模块讲解。

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        max_temp <= 0;
    else begin
        if (conv_fc3_go)
            max_temp <= 1 << (^WD-1);
        else begin if (conv_fc3_q_en)
            if (temp_gt_max)
                max_temp <= conv_fc3_q;
        end
    end
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        temp_idx <= 0;
    else if (conv_fc3_go)
        temp_idx <= 0;
    else if (conv_fc3_q_en)
        temp_idx <= temp_idx + 1;
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) digit_temp <= 0;
    else if (conv_fc3_go) digit_temp <=0;
    else if (conv_fc3_q_en && temp_gt_max) digit_temp <= temp_idx;
end
```

图 14: digit_produce 代码

2 代码可综合设计及上板验证

2.1 资源消耗

由于本实验在 FPGA 上综合, 故资源消耗主要包含 LUT,FF,BRAM 等。其资源消耗如图 15 所示。

Resource	Estimation	Available	Utilization %
LUT	1378	53200	2.59
FF	329	106400	0.31
BRAM	2.50	140	1.79
IO	4	125	3.20
BUFG	1	32	3.13

图 15: 资源消耗

2.2 时序分析

由图 16 可以看出其最坏 $T_{setup} = 8.151\text{ns}$, 时序较为充裕, T_{Hold} 同样大于 0, 故设计的时序相对比较好。

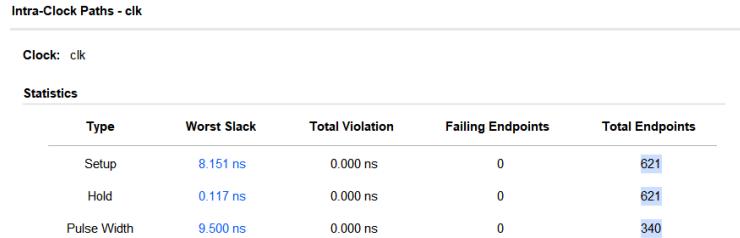


图 16: clk 时序

由图 17 可以看出最长路径是 data_rom->mac 模块, 具体信号为 data_rom 读出的数据到 conv1 的 mac 模块的 mac_temp 信号。该路径对应的 setup 时间就是最坏路径的时间, 由公式:

$$T \geq t_{cq} + t_{logic} + t_{su} - t_{skew} \quad (4)$$

故可知若增加 setup 的 slack, 我们可以减少 t_{logic} , 也就是可以在两模块之间加入 reg 寄存器, 这样虽然会优化时序, 也会增加 lenet 计算时间, 需要去权衡考虑。

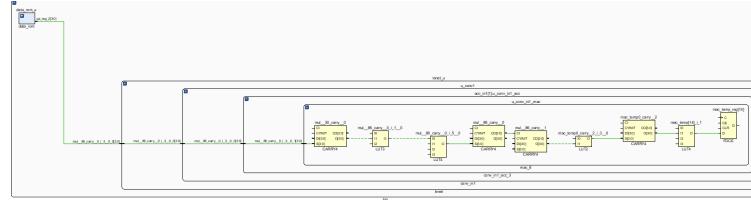


图 17: 最长路径

可对时序做出以下优化, 这会优化布局布线, 从而减小组合逻辑延时。

- 将 MAC 模块操作映射到 XILINX 的 DSP 模块
- 将存放权重和 bias 的 rom 改为 FPGA 的 BRAM

2.3 功率消耗

如图 18 所示, 本实验功耗主要为板子的静态功耗, 为 0.106W, 动态功耗中 logic 和 signal 功耗占主导, 分别为 0.005W 和 0.004W。

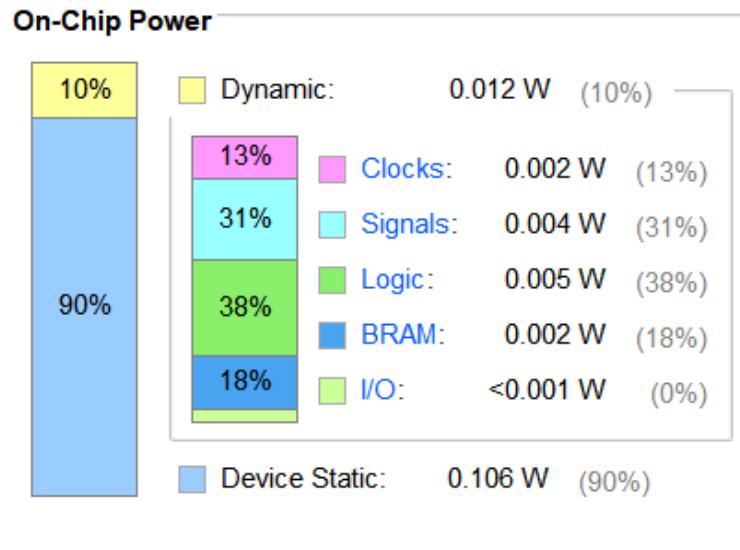


图 18: 最长路径

2.4 上板验证

实验给出的代码为不可综合的代码, 具体体现为 data_rom 模块, 故只需将 data_rom 模块改为 ROM IP 即可, 如图 19 所示, 只需加入 xilinx 的 bram ip 的 rom 模块即可。

```

`include "global.v"

`module data_rom(
    input                      clk,
    input                      rst_n,
    input [9:0]                 aa,
    input                      cena,
    output [^WD-1:0] qa
);
/*
//32*32*8
*/
`rom_32x32_8 rom(
    // A is write port
    .clka(clk),
    .ena(!cena),
    // .web(1'b0),
    .addr(a),
    // .dinb(8'd0),
    .douta(qa)
);
/*
*/
blk_mem_gen_0 data_rom (
    .clka(clk),      // input wire clka
    .ena[!cena],     // input wire ena
    .addr(aa),       // input wire [9 : 0] addr
    .douta(qa)      // output wire [7 : 0] douta
);
/*
// reg [^WD-1:0] mem [0:1023];
// initial
// $readmemh("input_pic",mem);

// always @ (posedge clk)
// if(!cena)
//     qa=mem[aa];
*/
endmodule

```

图 19: data_rom 代码

本次实验采用 zynq7020, 其代码上板验证的框架如图 20 所示。lenet 启动信号由按键控制, 当按键按下时, 启动 lenet, 之后计算完成后, 通过串口发送至上位机。

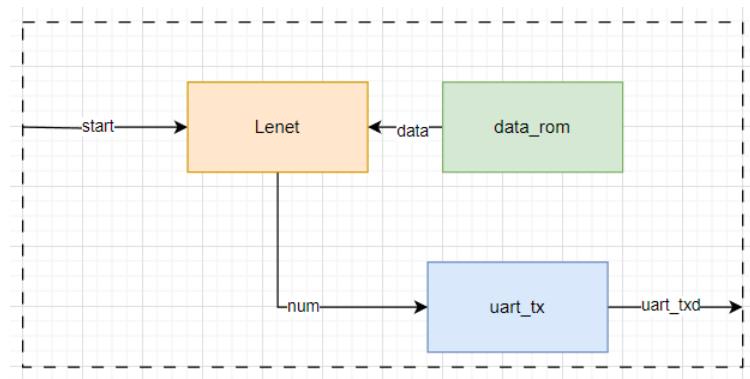


图 20: 上板整体框架

2.5 实际效果

如图 21,22 所示, 开发板按下按键 key0,lenet 开始计算, 计算完成后, 向串口发送数据, 上位机显示数据为 07, 也就是识别的数字为 7。

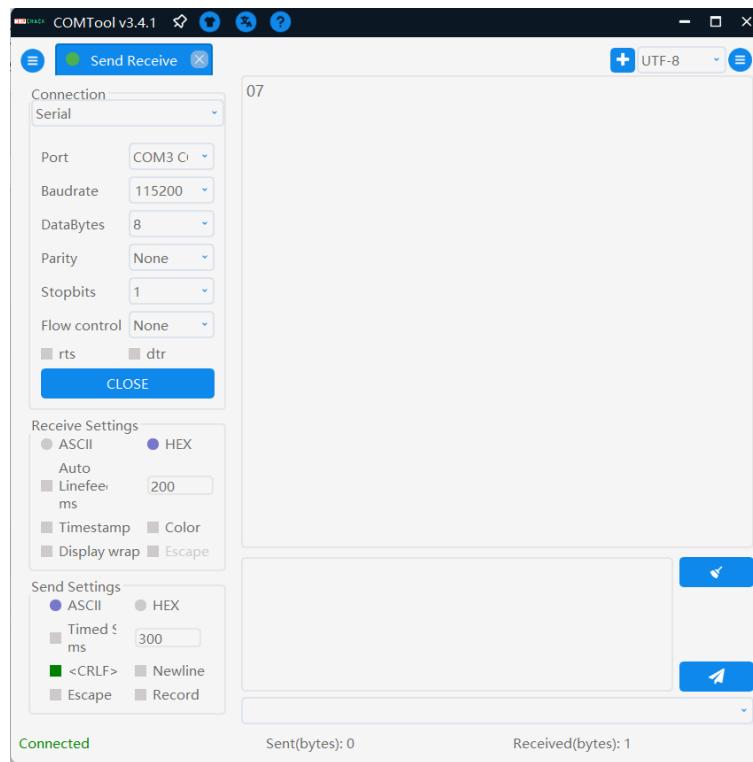


图 21: 串口接收数据

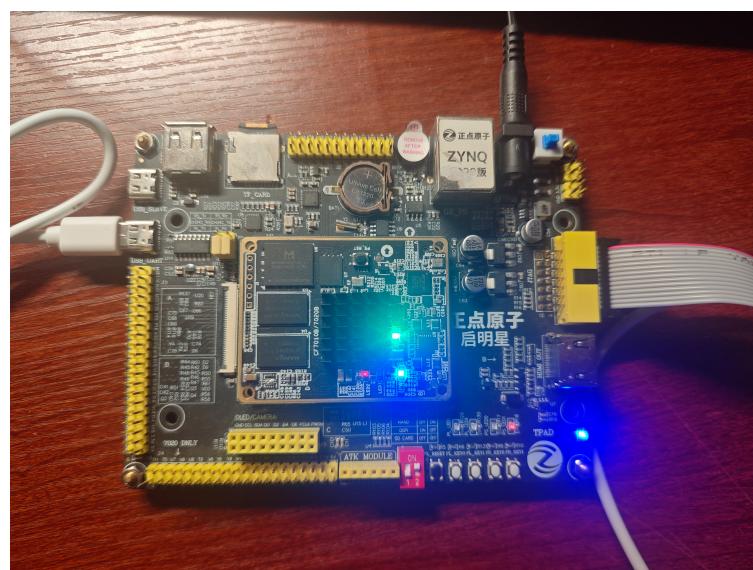


图 22: 开发板运行图

3 改进思路

lenet 主要的操作有卷积操作, 池化操作, relu 激活函数操作, 目前最流行的加速器方案就是 CPU+DSA, 故本项目也可采用该方案, 下面介绍开源 DNN 加速器 Gemini, 为本项目改进方案提供思路, 具体架构如图 23 所示, gemini 通过协处理器的方式挂载在 CPU 上。

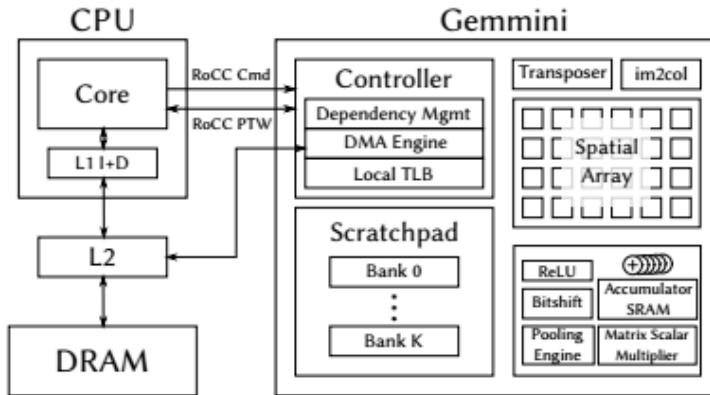


图 23: gemini 系统架构

先阐述该项目的特点:

- 可以对深度学习架构进行系统评估。具体来说, gemini 提供了灵活的硬件模板、多层次软件堆栈和集成的 SoC 环境。
- 使用基于 fpga 的性能测量和商用 ASIC 合成流程对 gemmini 生成的加速器进行严格的评估, 以进行性能和效率分析。与最先进的商业深度神经网络加速器相比, gemini 生成的加速器提供了相当的性能。
- gemini 基础设施能够实现运行 DNN 工作负载的 soc 的系统加速器协同设计, 包括 DNN 加速器的有效虚拟地址转换方案的设计和共享缓存层次结构中内存资源的供应。

现代 DNN 加速器加速单元有两种: 一是 systolic 的方式, TPU 就是这个做法, 二是采用 parallel vector units, 如 Brainwave 和 NVDLA。图 24 为两种不同架构对比。

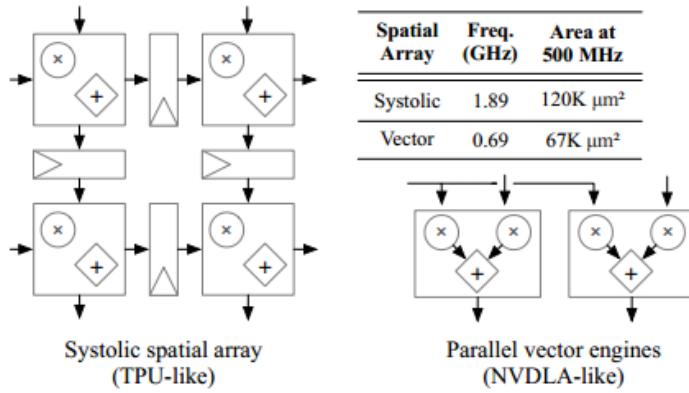


图 24: 两种不同架构对比

编程支持:gemini 提供了一个多层次的软件流来支持不同的编程场景。在高层, gemini 包含一个按钮软件流, 它读取 ONNX 文件格式的 DNN 描述, 并生成将运行它们的软件二进制文件, 将尽可能多的内核映射到 gemini 生成的加速器上。或者, 在较低的层次上, 生成的加速器也可以通过 C/ c++ api 编程, 并为常见的 DNN 内核调优函数。为了实现高性能, 这些函数必须针对不同的硬件实例进行不同的调优, 这取决于 scratchpad 的大小和其他参数。因此, 每当产生一个新的加速器时, gemini 还会生成一个附带的头文件, 其中包含各种参数, 例如空间数组的尺寸、支持的数据流以及包含的计算块(例如池化、im2col 或 transposition blocks)

数据分段和映射: 在运行时, 基于层输入的维度和加速器实例化的硬件参数, gemini 使用启发式方法将每次迭代中移动到草稿簿中的数据量最大化。gemini 在运行时计算循环块大小, 这些块大小决定了在执行平铺矩阵乘法、卷积、残差加法等内核时, 在 DRAM、L2 和 scratchpad 之间移动的时间和数量。如果程序员愿意, 低级 API 还允许他们手动设置每个内核的 tile 大小。

虚拟内存支持: 除了编程接口, gemini 还通过提供虚拟内存支持使加速器编程更容易。这对于希望避免手动地址转换的程序员和希望研究现代加速器中虚拟内存支持的研究人员都很有用。

所以通过总结 gemini 的做法, 对 lenet 优化的方案如下:

- 为 lenet 提供编程支持, 且支持不同层级的编程。
- 将 mac 模块替换为图 23 的两种方法之一。
- 动态计算循环块的大小, 允许程序员配置硬件算子。

此外, 还有可以采用存内计算技术对 lenet 进行优化。