北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

基于 FPGA 的 CIFAR-10 仿真任务
技术报告

目录

一、前言及系统方案

本次的小学期任务中，我们小组选择了 CIFAR-10 分类任务，最终目标是将神经网络算法部署至 FPGA 芯片上，在硬件层面实现对彩色图像的多分类任务。

如今，图像识别算法已经有了很充足的发展，即使提供的图片仅有 32*32 的大小，也有 resnet152 这样复杂的模型能将识别的准确率提升到 90%以上，但这些模型往往参数量巨大，在服务器上尚且能够表现出色，但图像识别的具体应用场景，如无人驾驶、智能农业机器人等，显然不可能为每个终端部署一台高性能服务器，因此，将图像分类算法固化在芯片上无疑是具有重要的意义的：通过在终端直接处理数据，可以极大减少终端与服务器间的通讯成本，同时增加了终端的工作效率。

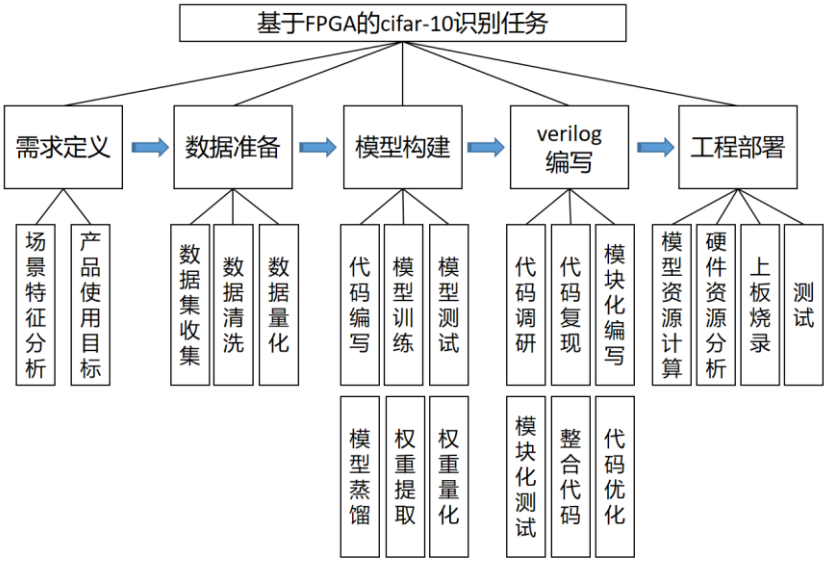

图 1.1 技术路线图

通过对任务总体流程的分析评估，我们将整个任务逐层拆解，最终确定了一条技术路线，如图 1.1 所示。我们的分工安排，也正是根据这条技术路线展开的。

此外，我们的项目已经开源在了 https://github.com/Robin-WZQ/CNN-FPGA，其中包含我们项目所有的代码，以及配套的说明文件。

二、数据准备

CIFAR-10 数据集中的图片均为 3 通道 32×32 像素，包含 10 个类别。在本次

任务中，我们将所有图片均转换为了灰度图，灰度值为 RGB 三通道像素值的平均值，如图 2.1 所示。

图 2.1 RGB 图转换为灰度图的示例

此步操作的主要目的是与后续 verilog 部署的代码对齐，同时也能降低参数量、计算量。CIFAR10 数据集可在以下地址中下载：

http://www.cs.toronto.edu/~kriz/cifar.html

三、智能系统训练

在模型的选择上，我们使用了非常轻量的 LeNet-5 模型，其结构及参数量等详细信息如图 3.1 所示。

层标识符	网络层	描述	输入量	参数	计算量
-	输入层	(28, 28, 1)	-	-	-
layer1	Conv2d	out_channels=16, kernel_size=5, stride=1	28×28×1=784	5×5×1×16+16=416	28×28×1×16×5×5=313,600
	ReLU	-	24×24×16=9,216	-	-
	MaxPool2d	kernel_size=2, stride=2	24×24×16=9,216	-	-
layer2	Conv2d	out_channels=32, kernel_size=5, stride=1	12×12×16=2,304	5×5×16×32+32=12,832	12×12×16×32×5×5=1,843,200
	ReLU	-	8×8×32=2,048	-	-
	MaxPool2d	kernel_size=2, stride=2	8×8×32=2,048	-	-
fc1	Linear	out_features=120	4×4×32=512	512×120+120=61,560	61,440
	ReLU	-	120	-	-
fc2	Linear	out_features=84	120	120×84+84=10,164	10,080
	ReLU	-	84	-	-
classifier	Linear	out_features=10	84	84×10+10=850	840

图 3.1 LeNet-5 模型详细信息

针对此项任务，我们部署了模型蒸馏算法来训练模型，模型蒸馏的概念如图 3.2 所示。

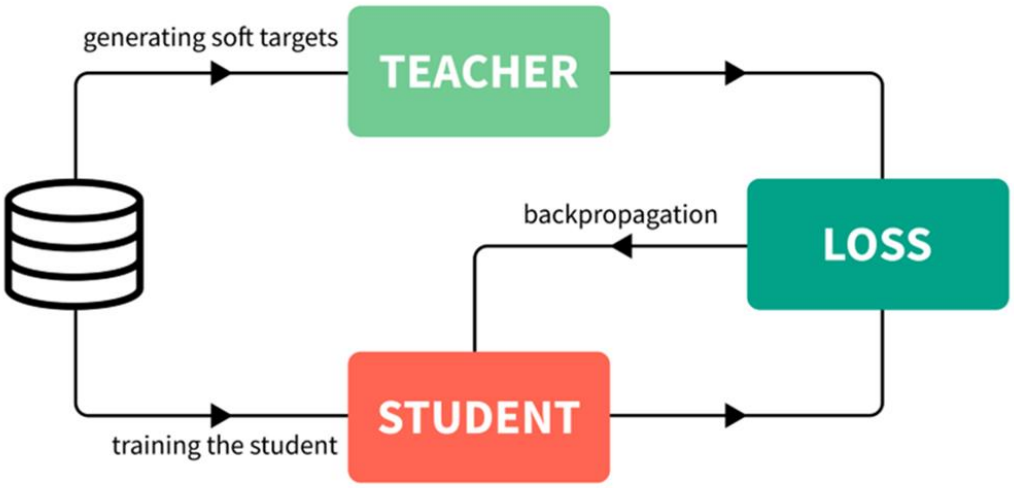

图 3.2 模型蒸馏概念图

模型蒸馏算法可以描述如下：

1. 训练一个较为复杂且效果较好的模型作为 Teacher model。
2. 使用 Teacher model 对训练集图片进行推理，将输出结果经过 softmax 计算得到 soft label。
3. 在训练 Student model 时，使 loss 的计算包含两部分，一部分为模型实际标签也就是 hard label，另一部分为从 Teacher model 处得到的 soft label。

在细节上，我们在蒸馏时，将 loss 函数计算公式设计为了

$$loss = \alpha(\mathbf{CE}(pred, y)) + (1-\alpha)(\mathbf{KL}(pred/T, soft\_label/T))$$

其中, α 和 T 是超参数，分别为权重系数和蒸馏温度。 CE 为交叉熵损失函数，KL 为 KL 散度损失函数。

在具体设置上，在训练 Teacher model 时，所采用的设置为 epochs=50；lr=0.001；batch_size=128；对输入图片先变换到 224×224 像素，再归一化，再以均值 0.449，标准差 0.226 进行 normalize。在训练 Student model 时，所采用的设置为 epochs=100；lr=0.001；batch_size=128；α=0.3；T=7；对输入图片先归一化，再以均值 0.4734，标准差 0.2507 对图片进行 normalize。

我们以 LeNet-5 为 Student model，尝试使用了多种 ResNet 架构的模型作为 Teacher model，模型参数量并将结果进行了横向对比，实验结果如图 3.3 所示。

teacher_model	student_model	teacher_score	student_score
/	Lenet5	/	64.5%
Resnet18	Lenet5	72.3%	63.2%
Resnet50	Lenet5	92.3%	63.4%
Resnet152	Lenet5	94.7%	67.4%

图 3.3 模型蒸馏实验对比结果

四、图像和权重量化

为了将图像和训练好的模型权重导入到 verilog 实现的神经网络中，我们需要对数据进行量化。

4.1 图像量化

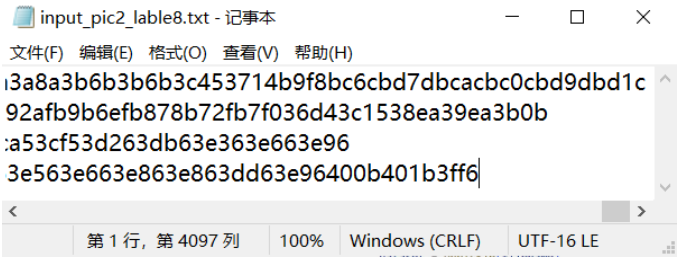

图 4.1 图像量化结果展示

数据量化的主要工作是将数据集中 uint8 类型的原始图像转化为 IEEE754 半精度浮点数类型的数据，以满足实际上板的需要。

由于我们只需要在板上进行神经网络的推理，所以此处以一张图像为例，介绍数据量化的过程：

1. 读取原始数据集的测试集部分”cifar-10-python\cifar-10-batches-py\test_batch”，得到数据集的元组 dataset={“data”:{…}, “labels”,{…}}。其中 dataset[”data”]的格式是(10000, 3072)，代表 10000 张图像，数据类型是 uint8，每张图像顺序存储 RGB 通道的子图，每个子图按照行优先的方式存储，即 3072=3*32*32；dataset[“labels”]的格式是(10000, 1)，代表 10000 张图像对应的标签；

2. 选择第 id 张图像进行处理，将其变形为数据类型为 FP32 的(3, 32, 32)的 tensor；
3. 为了将输入与训练模型保持一致，对图像进行如下灰度化+正则化的变换，得到数据类型为 FP32 的(1, 32, 32)的 tensor；
4. 将数据类型为 FP32 图像转为数据类型为 IEEE754 半精度浮点数的图像，半精度浮点数总共 16 位，按顺序由 1 位符号位+5 位阶码+10 位尾数组成。在这里，我们需要的输出的结果是长度为 32*32*4 的符号串，4 表示 16 位二进制转为十六进制的长度。

4.2 权重量化

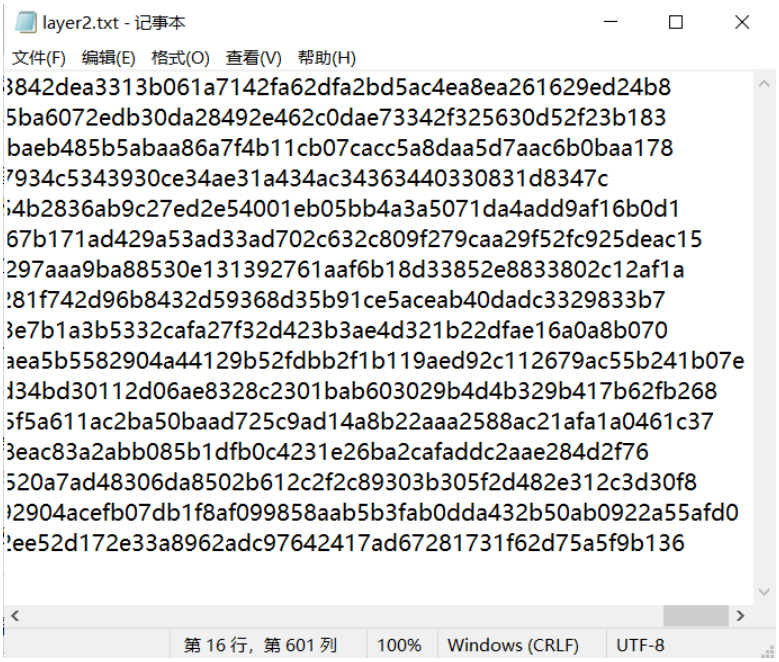

图 4.2 模型权重量化结果展示

权重量化的主要工作是将提取出的数据类型为 FP32 的权重文件转换为对应网络层需要的 IEEE754 半精度浮点数/单精度浮点数，以满足实际上板的需要。

在权重的量化中，由于 verilog 实现的神经网络的结构和需要不同，我们对卷积层的参数和全连接层的参数分别进行处理。

1. 实例化神经网络，读取已保存的权重；
2. 根据参数层的名称进行不同的处理。

4.2.1 卷积层

- Input

- scale: (OutCls, InCls, KnSize, KnSize)
- dtype: FP32

• Output

- scale: (OutCls, InCls*KnSize*KnSize)
- dtype: IEEE754 半精度浮点数

4.2.2 全连接层

• Input

- `scale: (InNodes, OutNodes)`
- dtype: FP32

• Output

- scale: (OutNodes*InNodes, 1)
- dtype: IEEE754 半精度浮点数

五、Verilog 仿真

针对 verilog 仿真我们可以从 3 个角度进行分析，从微观单元上看，整个模型是由各种计算单元构成，包括卷积滑窗、池化、激活函数等等；从宏观结构上看，模型可以采用层次化设计，它主要是卷积和全连接的组合；从功能整合上看，在实现了各个微观单元后，我们可以逐步构成宏观结构，实现最终的模型仿真。

5.1 卷积结构

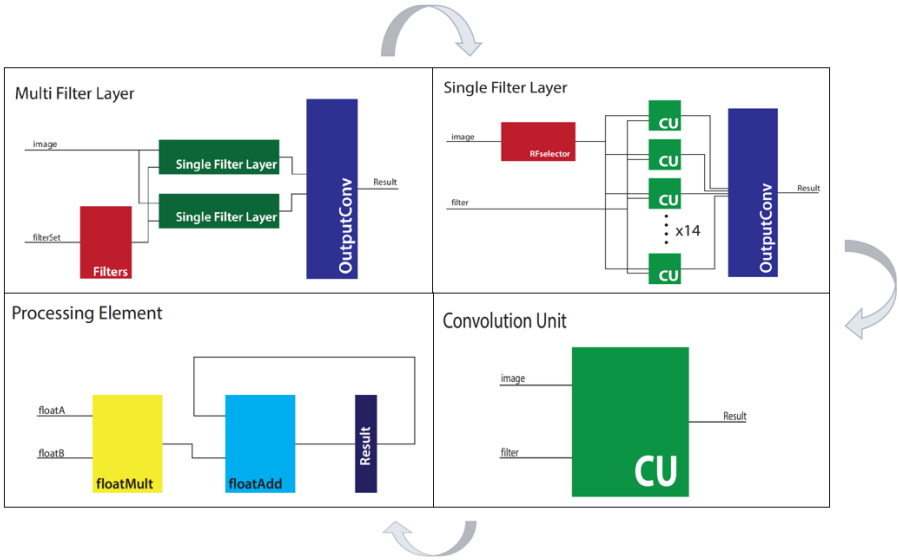

图 5.1 卷积结构说明

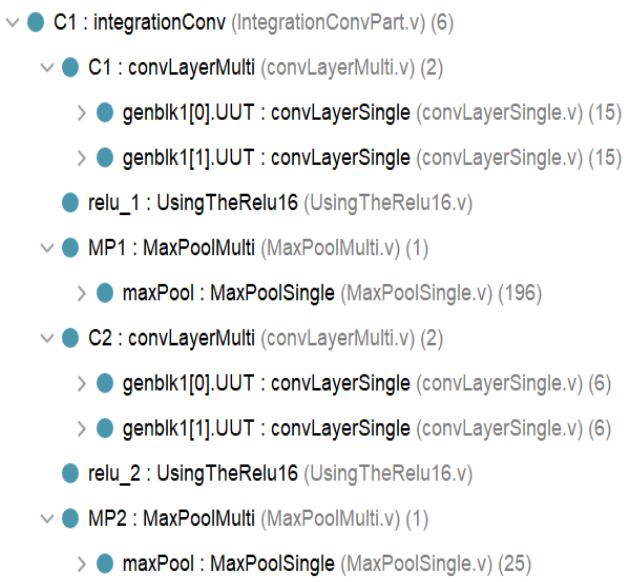

图 5.2 卷积代码结构

图 5.1 和图 5.2 分别展示了我们卷积的结构设计思路和代码结构，该模块采用层次化设计思想，从最复杂的多卷积单元到最简单的处理单元 PE。卷积的本质，就是多个乘法器和加法器的叠加。

可配置参数：

名称	说明	默认值
DATA_WIDTH	数据位宽	16
ImgInW	输入图像的宽度	32
ImgInH	输入图像的高度	32
Conv1Out	第一层卷积输出	28
MvgP1out	最大池化输出	14
Conv2Out	第二层卷积输出	10
MvgP2out	最大池化输出	5
Kernel	卷积核的大小	5
DepthC1	第一层卷积核数量	6
DepthC2	第二层卷积核数量	16

输入输出：

名称	类型	说明	长度
CNNinput	input	输入的图像，数据从左上至右下排列，每一个像素值用半精度浮点数表示	ImgInW × ImgInH × DATA_WIDTH
Conv1F	input	第一层卷积核权值，从第一个卷积核左上开始，到最后一个卷积核右下，每一个值用半精度浮点数表示	Kernel × Kernel × DepthC1× DATA_WIDTH
Conv2F	input	第二层卷积核权值，从第一个卷积核左上开始，到最后一个卷积核右下，每一个值用半精度浮点数表示	DepthC2 × Kernel × Kernel × DepthC1 × DATA_WIDTH
iConvOutput	output	输出的特征图	MvgP2out × MvgP2out × DepthC2 × DATA_WIDTH

图 5.3 卷积单元参数设置

图 5.3 展示了我们卷积的相关参数，该模块共包含 10 个可变参数，可按需改变数据位宽、输入图片大小等。

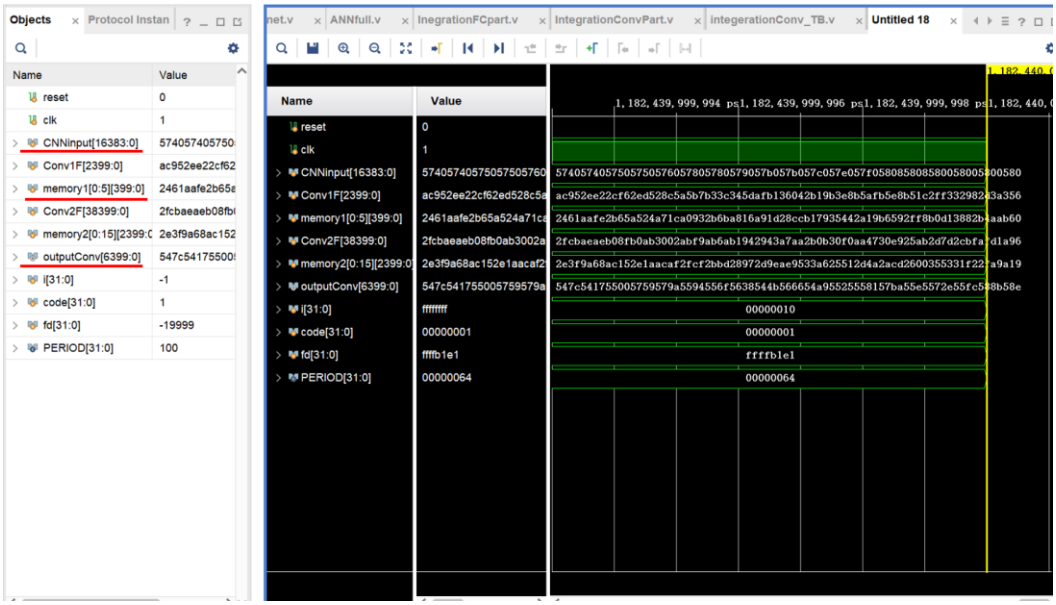

图 5.4 卷积仿真结果展示

最终的仿真结果如图 5.4 所示，在输入了图像、卷积层参数后，模块得到了正确的输出。

5.2 最大值池化结构

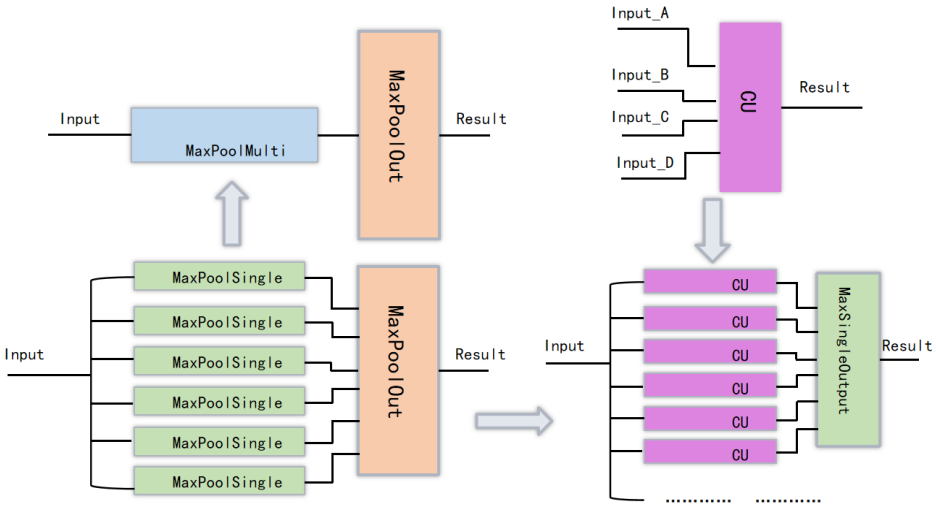

图 5.5 最大值池化结构说明

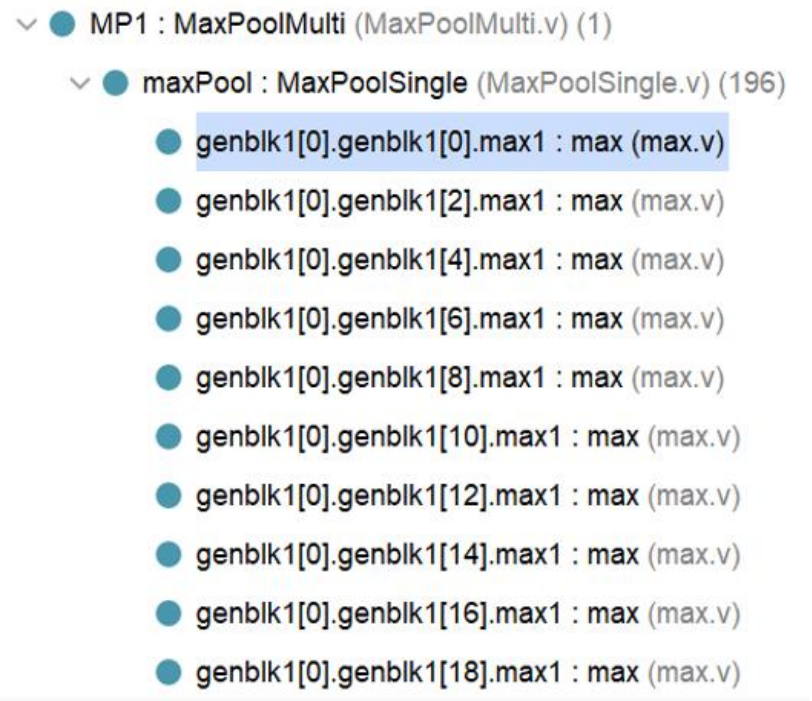

图 5.6 最大值池化代码结构

图 5.1 和图 5.2 分别展示了我们卷积的结构设计思路和代码结构，该模块同样采用层次化设计思想，从最复杂的多通道池化结构到最简单的 4 位数比较器。而

最大值池化的本质上，就是多个大小比较器的叠加。

5.3 ReLu 结构

本次项目的激活函数我们采用了 ReLu 函数，ReLu 函数的表达式如下。之所以采用 ReLu 函数作为激活函数而没有采用其他函数诸如 TanH、Sigmoid 等函数，是因为 ReLu 的运算简单，速度更快，而且效果几乎没有差别。

$$ReLu(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

对于上述的数学公式我们用如下代码实现对每一个数的处理：

```
if (input_fc[DATA_WIDTH*i-1+DATA_WIDTH] == 1'b1) begin // 负数直接赋值为0
    output_fc[DATA_WIDTH*i+:DATA_WIDTH] = 0;
end else begin
    output_fc[DATA_WIDTH*i+:DATA_WIDTH] = input_fc[DATA_WIDTH*i+:DATA_WIDTH];
end
```

图 5.7 展示了 ReLu 的结构，顶层为 UsingTheReLu16 模块，这个模块的功能是将半精度浮点数的输入流按顺序进行并行处理，每个半精度浮点数由一个运算单元实现，参见图 5.7 下半部分的仿真结果。

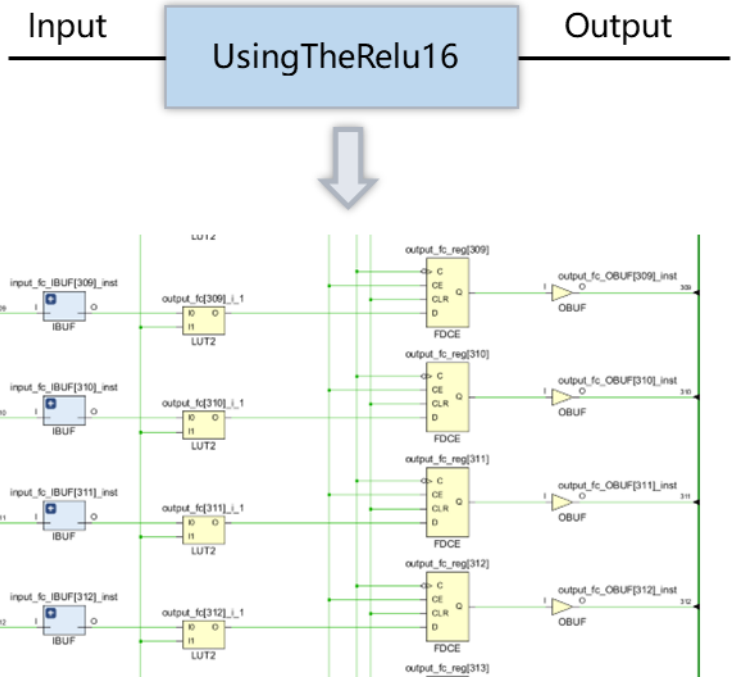

图 5.7 ReLu 结构说明

5.4 全连接结构

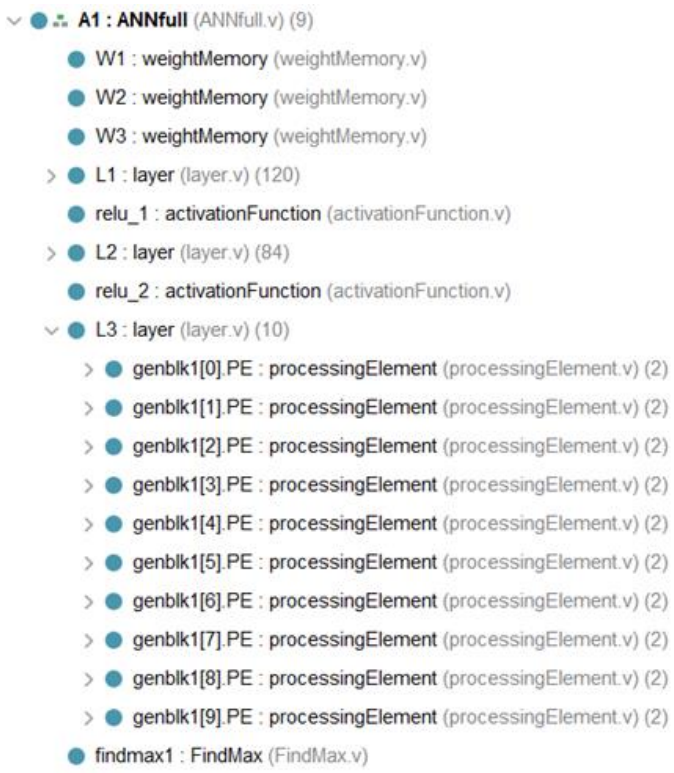

图 5.8 全连接代码结构

图 5.8 展示了我们整个全连接部分的代码结构，对于一个全连接层，它具有与输出节点数相同的 PE。每一个 PE 通过乘加器进行加权求和，求得对应输出节点的结果。

可配置参数：

名称	说明	默认值
DATA_WIDTH	数据位宽	32
INPUT_NODES_L1	第一层输入节点数	400
INPUT_NODES_L2	第二层输入节点数	120
INPUT_NODES_L3	第三层输入节点数	84
OUTPUT_NODES	输出节点数	10

输入输出：

名称	类型	说明	长度
input_ANN	input	全连接层的输入，用单精度浮点数表示	DATA_WIDTH*INPUT_NODES_L1
output_ANN	output	预测的标签值，cifar-10为10分类，需要用4位二进制表示	4

图 5.9 全连接单元参数设置

图 5.9 展示了我们卷积的相关参数，该模块共包含 5 个可变参数，可按需改变数据位宽、每一层的节点数。

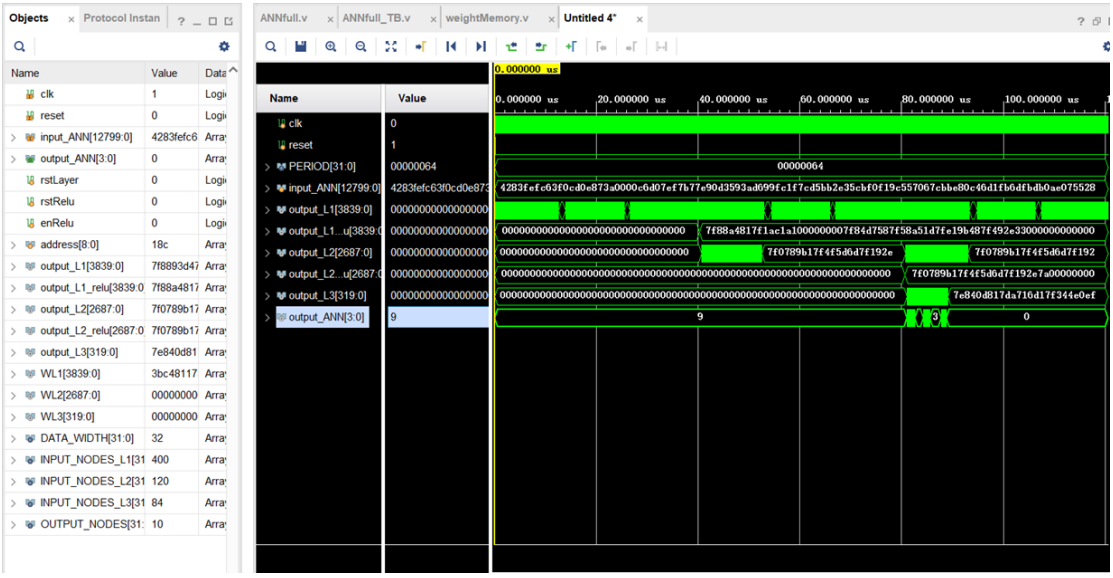

图 5.10 全连接仿真结果展示

最终的仿真结果如图 5.10 所示，在读取了权重后，模块根据输入得到了正确的输出。

5.5 输出选择

在网上可找到的参考项目里，大多使用了 softmax 层来完成输出选择的工作，但是经过调研之后，我们发现 softmax 层仍有许多不足之处。

首先，观察 softmax 层公式：

$$Softmax(zi) = \frac{e^{zi}}{\sum_{c=1}^{C} e^{zc}}$$

我们可以看到 softmax 层的实现主要依赖于指数函数的仿真实现，由于 vivado 不可能像在计算机里那样直接进行高精度浮点数计算，常用的方法就是将指数函数进行泰勒展开，通过高阶展开逐步逼近目标精度，但我们在将数据进行量化处理的时候，势必会造成精度的损失，加上网上对泰勒展开的阶数并没有明确的说法，也就是说，展开阶数过少，必定导致精度不足，误差过大，而展开阶数过多，则会极大地增加工作量。

此外，由于 softmax 层实际上并不改变数据之间的逻辑大小，因此我们在推理阶段中，完全可以直接将 softmax 层舍去。全连接层计算完成后，输出的结果为 32*10 的数组，即 32 位宽的 10 个概率值。由于数据量并不大，我们只需要直接对各个数据大小进行比较即可。

这里我采用的方法是，对 FC 层提供的 10 个数据，在高位直接添加 4bit 宽的 label，为方便上板时观察输出结果，label 的取值从 0001 排至 1010，分别对应分类任务中的 1-10 类。

而在实际排序的时候，我们只需要比较真正的数据部分，通过快速排序的方法，就可以很快地挑选出概率值最大的数据。随后将该数据的 label 输出即可完成数据选择任务。

5.6 精度转换

整合时遇到的第二个问题是数据精度不同步的问题，卷积层在构建时是基于 16 位的数据位宽，而全连接层的算子使用的数据位宽则为 32 位，因此在将卷积层与全连接层链接在一起前，还需要加一层精度转换层，将 16 位的卷积结果扩

展到 32 位。

对于半精度浮点数而言，转成单精度浮点数后正负并不会改变，因此符号位可以直接继承，半精度浮点数里，阶码共占 5 位，而单精度浮点数里阶码共占 8 位，因此在转换过程中，需要先将半精度浮点数的阶码扩充至 8 位，然后-15+127，即加上 112（01110000），尾数部分由于无需考虑偏移量，因此只需将原本 10 位的尾数扩充至 23 位即可。

六、完整的 verilog 项目结构

图 6.1 完整的 verilog 项目结构

正如前面所述，整体采用层次化、模块化设计思想。其中，包含 3 个单元：卷积单元、精度转换单元以及全连接单元。

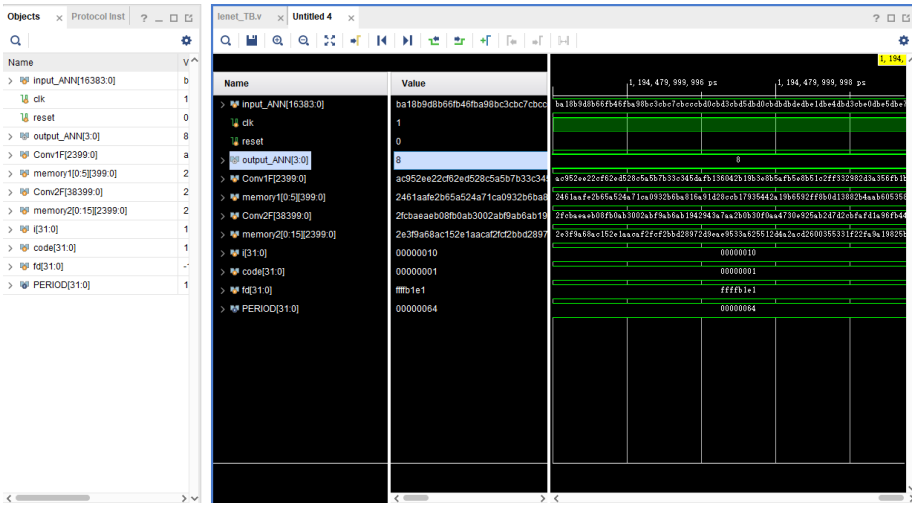

图 6.2 最终结果展示

最终项目的结果如图 6.2 所示。

七、部署测试

7.1 PS+PL 尝试

最初我们考虑的是在电脑屏幕上以文字的方式输出结果，于是在搜索后选定了 PS+PL 端的模式进行上板。而大部分相关的教程都是使用 HDL 的高级语言，一开始造成了一些理解上的困难。后来发现不管是哪种语言，都是要生成 IP 核，然后与 zynq-7000 内核相连，因此按照相同的方式将我们的代码封装成了 IP 核，并连接。连接成功后生成 sdk 文件，打开后与开发板相连，写入程序存入 sd 卡中，接下来就可以运行程序，sdk 中也会显示对应的输出结果。通过一个简单的 helloworld 程序证明了该种方法的可行性。但是在连接这一步上出现了大大小小的问题，比如端口连接报错或者是输入不成功，搜索后也没有结果，无奈只能放弃这种方式。

7.2 纯 PL 端实现

按照之前做的亮灯 demo，我们开始思考如何将输入和输出对应到开发板上。输入可以直接在 verilog 代码里嵌入，更换图片则重新写入。原本的输出就是一个 4 位 2 进制，正好可以与开发板上的小灯相对应，只需要修改 0000 时的对应让它更方便观察即可。于是我们再次对 verilog 仿真代码进行了调整，并且直接

把网络权重嵌入，但又遇到了一些问题。

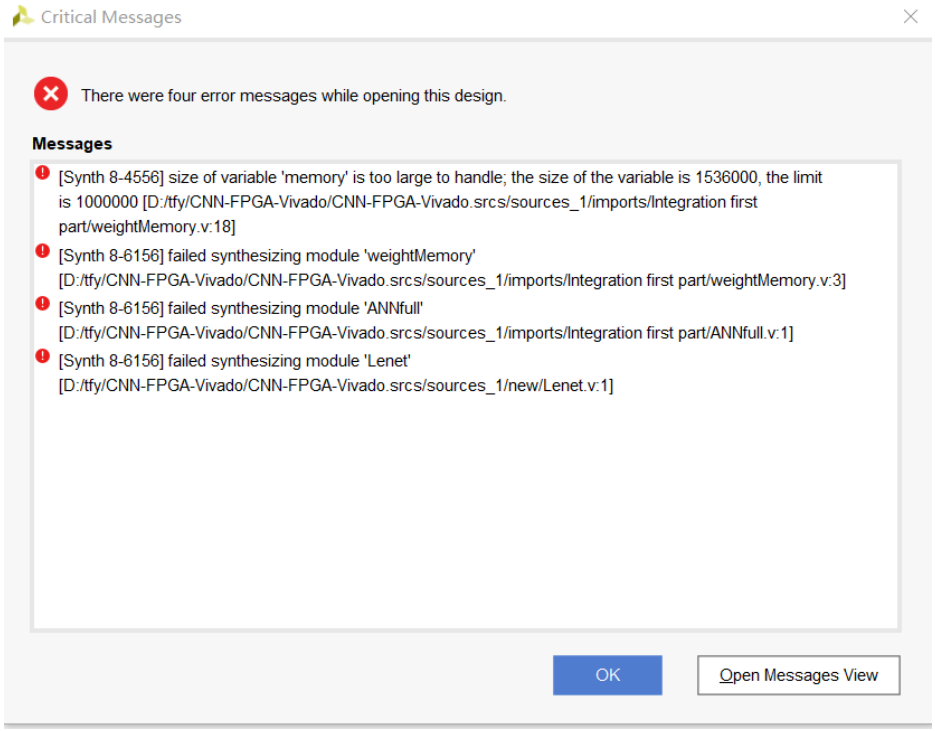

图 7.1 报错展示

出现了数组无法设置特别大导致无法成功运行等问题（如图 7.1 所示），因为存在 1000000 的限制，出于时间限制，纯 PL 端的仿真代码未达到最终完善效果，会在后续进行进一步调试。

八、总结

虽然我们的项目参考了 Gtihub 上的开源项目，但我们也做出了许多改进，这里再总结一下我们的工作内容：

1. 原项目目标为手写数字识别，我们进一步拓展，研究其在 cifar-10 数据集上的效果；
2. 据此，我们基于 pytorch 独立设计了网络，并使用蒸馏得到更好的结果；
3. 在模型结构上进行修改，包括删除第三层卷积，增加了一层全连接等；
4. 在卷积层使用了 relu 替代 tanh（自编代码）；
5. 使用 maxpool 替代 avgpool（自编代码）；
6. 修改了源码中的一些错误（如 reset 命名错误，数据位宽错误等）；

7. 改变了全连接层的输入维度、输出维度；
8. 编写了卷积层的 testbench，并通过了仿真；
9. 自编 16 转 32 位转换器以及对应 tesetbench 代码；
10. 原项目并未实现整个网络的结构，我们整合了整个网络并成功进行仿真；
11. 上板进行了测试；
12. 编写了中文注释，方便阅读;
13. 删除了 softmax，自定义了更高效的选择器；
14. 提供了 pytroch 的训练源码、模型以及配套的量化代码，方便移植。

附录（源码清单）

- cifar-10-torch
  - cifar-source # 源代码文件夹
    - cifar-10-python：数据集文件夹
    - res：模型存储文件夹
    - distilled_lenet5_best.pt：蒸馏得到的 Lenet5 模型参数文件
    - train.py：Teacher model 训练代码
    - distill.py：模型蒸馏训练代码
    - test.py：验证评估代码
    - save_params.py：参数导出代码
    - models.py：模型结构代码
    - Lenet5_parameters_cifar.txt：导出模型参数文件
  - README.md # 说明文件
- CNN-FPGA-Vivado # 包含 testbench 和源码，以及整个工程文件
  - CNN-FPGA-Vivado.src # 项目源码
    - sim_1/new # testbench 源码
    - sources_1/new # 工程源码
    - CNN-FPGA-Vivado.xpr # 项目工程
- quantification
  - cifar-10-python # 数据集
  - distilled_lenet5_best.pt # 训练好的模型
  - input_pic2_label8.txt # 数据转换例子
  - quantification_img.py # 输入图像量化代码
  - quantification_para.py # 模型权重量化代码
- weight
  - classifier.txt # 全连接第 3 层权重
  - fc1.txt # 全连接第 1 层权重
  - fc2.txt # 全连接第 2 层权重
  - layer1.txt # 卷积第 1 层权重

- layer2.txt # 卷积第 2 层权重