

武汉大学综合实验报告

机器学习数字集成电路设计

院系：物理科学与技术学院

专业：物理学（弘毅班）

报告人：王海飞 2019300001094

指导老师：常胜教授

二〇二二年 11 月 29 日

目录

1 前言	2
2 神经网络的基础算法	2
2.1 机器学习的概念和历史发展	2
2.2 神经网络的基本算法	3
2.3 本次实验中的神经网络结构	7
3 多层感知机的软件设计	7
3.1 前向传播的实现	7
3.2 反向传播的实现	8
3.3 MATLAB 深度神经网络工具箱	8
4 多层感知机的硬件设计	9
4.1 整体框架和设计思路	10
4.2 IP 核配置	10
4.3 性能分析	12
4.4 上板验证	13
5 拓展 1: 多层感知机的 HLS 实现	16
5.1 C Simulation	17
5.2 C Synthesis	17
5.3 Zynq 系统设计	18
5.4 PYNQ 调试	18
6 拓展 2: LeNet-5 的硬件加速	18
6.1 卷积神经网络介绍	18
6.2 LeNet-5 的架构	21
6.3 LeNet 的软件实现	22
6.4 LeNet 的硬件实现	22
7 总结	30
8 附录	33
8.1 多层感知机 MATLAB 代码	33
8.2 多层感知机 Verilog 代码	36
8.3 多层感知机上板验证代码	43
8.4 多层感知机 HLS 代码	44
8.5 LeNet-5 HLS 代码	47

1 前言

本报告围绕机器学习算法及其软硬件实现展开。各节内容如下：

- 第 2 节介绍机器学习的概念、历史发展和神经网络的基本算法。
- 第 3 节用 MATLAB 进行多层感知机的软件实现，包括前向传播预测和反向传播训练。
- 第 4 节用 Verilog 进行多层感知机的硬件加速，使用了 Xilinx 内置的 Block ROM，浮点计算等 IP 核，用流水线和加法树等方式提高了吞吐量。
- 第 5 节用 HLS 实现多层感知机的硬件加速。与 Verilog 等 HDL 相比，HLS 的开发速度要快得多，且仍能实现较高的性能。
- 第 6 节用 HLS 实现最早的卷积神经网络 LeNet-5，并针对硬件特点做了一些优化。对卷积层，采用了行缓存的优化策略；对全连接层，采用了循环交换的优化策略，实现了更高的吞吐量。此外，还使用了双缓冲技术节省数据拷贝时间；用 int8 量化减少了硬件资源消耗。
- 第 7 节总结了本次实验的流程，并对本次实验进行了评估。

本次实验的硬件平台为 PYNQ-Z2，使用的软件平台有：

- MATLAB R2022a
- Pytorch 1.13.0+cpu
- Vivado 2020.2
- Vitis HLS 2020.2
- PYNQ Jupyter Notebook

2 神经网络的基础算法

2.1 机器学习的概念和历史发展

机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。总体上，机器学习算法可以分为有监督学习，无监督学习，强化学习三种类型。有监督学习是给定已知数据和对应标签，学习如何预测未知数据的标签；无监督学习是给定已知数据，寻找数据中隐藏的结构；强化学习是给定数据，学习如何选择一系列行动，以最大化长期收益。

在 1980 年之前, 机器学习算法是零碎化的, 不成体系。但它们对整个机器学习的发展所起的作用不能被忽略, 列举部分如下:

- 线性判别分析。可以追溯到 1936 年。
- 贝叶斯分类器。起步于 1950 年代。
- logistic 回归。可以追溯到 1958 年。
- **感知器模型**。一种线性分类器, 可看作是人工神经网络的前身, 诞生于 1958 年, 但它过于简单, 甚至不能解决异或问题, 因此不具有实用价值, 更多的起到了思想启蒙的作用, 为后面的算法奠定了思想上的基础。
- k 均值算法。可以追溯到 1967 年。

从 1980 年开始, 机器学习才真正成为一个独立的方向。在这之后, 各种机器学习算法被大量的提出, 得到了快速发展, 列举部分如下:

- 决策树。1980 年代开始发展。
- **反向传播算法**。1986 年诞生。这是现在的深度学习中仍然被使用的训练算法, 奠定了神经网络走向完善和应用的基础。
- **卷积神经网络**。1989 年, LeCun 设计出了第一个真正意义上的卷积神经网络, 用于手写数字的识别, 这是现在被广泛使用的深度卷积神经网络的鼻祖。
- SVM 和 AdaBoost, 诞生于 1995 年。它们与神经网络是竞争关系, 并在此后十几年占据上分。

虽然真正意义上的人工神经网络诞生于 1980 年代, 反向传播算法和卷积神经网络也早就被提出, 但神经网络在很长一段时间内并没有得到大规模的成功应用, 在与 SVM 等机器学习算法的较量中处于下风。这是由于它受限于训练样本数和计算能力不足以及深层神经网络梯度消失问题。

情况的改变发生在 2012 年, Alex 网络的成功使得卷积神经网络卷土重来。此后, 深度神经网络蓬勃发展, 在语音识别、图像处理等方面取得了巨大成功。

2.2 神经网络的基本算法

2.2.1 单层感知机

单层感知机 (Perceptron) 的输入为 $x_i (i \in \{1, \dots, n\})$, 输出为一个布尔值 $y \in \{0, 1\}$ 。它具有 $n + 1$ 个参数: $w_i (i \in \{1, \dots, n\})$ 和 b 。输出和输入的关系为:

$$y = S \left(\sum_{i=1}^n w_i x_i + b \right), \quad (2.1)$$

其中 S 为单位阶跃函数，定义为

$$S(t) = \begin{cases} 1, & t > 0 \\ 0, & t \leq 0 \end{cases}. \quad (2.2)$$

一般把 w_i 叫做权重，把 b 叫做偏置。

直观来看，单层感知机用一条直线将样本分为两类，因此它可以处理线性边界，例如逻辑与，逻辑或。但是单层感知机无法处理非线性边界，例如异或，见图 (2.1)。现实世界中的分类任务大部分是非线性的，这限制了单层感知机的应用场景。

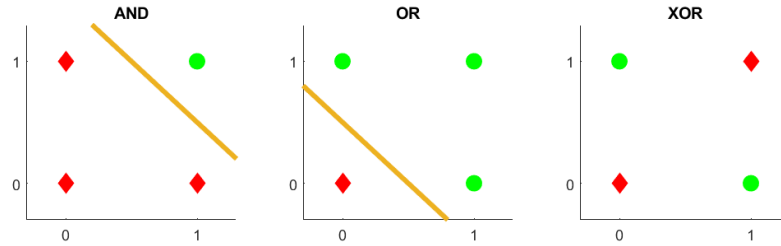


图 2.1: 单层感知机的线性边界可以处理逻辑与和逻辑或，但无法处理异或。

2.2.2 多层感知机

多层感知机 (Multilayer Perceptron, MLP) 它拥有多层的结构。它至少拥有一个隐含层，也就是总共至少三层。以拥有一个隐含层的多层感知机为例，它可以用如下映射定义：

$$\mathbb{R}^{n(1)} \xrightarrow{f(1)} \mathbb{R}^{n(2)} \xrightarrow{f(2)} \mathbb{R}^m \quad (2.3)$$

$$f_{(L)} : \mathbf{f}_{(L)}(\mathbf{x}) = \mathbf{h}_{(L)}(\mathbf{W}^{(L)}\mathbf{x} + \mathbf{b}^{(L)}), \quad (2.4)$$

其中 m 为输出的神经元个数； $\mathbf{W}^{(L)}$ 是第 L 层线性映射的矩阵， $\mathbf{b}^{(L)}$ 是 $\mathbb{R}^{n(L)}$ 中的向量，它们是可学习的参数； $\mathbf{h}_{(L)}$ 是第 L 层的“激活函数”。一个非线性的激活函数可以使多层感知机具有处理非线性边界的能力。常见的激活函数有 \tanh , ReLU , sigmoid 等。

也可以用带下标的符号表示：记 $z_j^{(L)}$ 为

$$z_j^{(L)} = \sum_{i=1}^{n^{(L-1)}} w_{ji}^{(L)} a_i^{(L-1)} + b_j^{(L)}, \quad (2.5)$$

$$(i \in \{1, \dots, n^{(L-1)}\}, j \in \{1, \dots, n^{(L)}\}),$$

则有

$$a_j^{(L)} = \mathbf{h}_{(L-1)}\left(z_j^{(L-1)}\right), \quad (2.6)$$

其中 $a_j^{(L)}$ 为第 L 层的激活值（即激活函数的输出值）。

多层感知机也叫全连接神经网络，指每一层神经元都和上一层的所有神经元连接。

多层感知机的预测过程是一层一层地往前计算，所以该过程也被称为前向传播（Forward Propagation）。这种网络也叫前馈神经网络（FNN），这与循环神经网络（RNN, Recurrent Neural Network）形成对比。RNN 中存在反馈通道，而 FNN 中不存在反馈通道，在推理时，所有数据前向传播。

2.2.3 梯度下降与反向传播

训练的过程，就是求出参数 W 和 b 使得预测误差尽可能小的过程。梯度下降（Gradient Descent）是常用的最小化误差函数的方法。对于一个前向传播过程 f ，定义误差函数（损失函数）为

$$J(\mathbf{x}) = \frac{1}{2m} \sum_{j=1}^m [f_j(\mathbf{x}) - y_j]^2 \quad (2.7)$$

记可学习的参数为 θ_i ，则梯度下降更新参数的公式为：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla J(\boldsymbol{\theta}) \quad (2.8)$$

其中 η 叫做学习率，可以根据训练情况进行人工调整，这种参数叫做超参数。式 2.8 也可以写成带下标的形式：

$$\Delta \theta_i = -\eta \frac{\partial J}{\partial \theta_i} \quad (2.9)$$

梯度下降的目的是将误差函数最小化。从式 2.8 可以看出，为了学习参数，关键就是求出损失函数对于可学习参数的梯度。梯度下降的示意图见图 2.2。

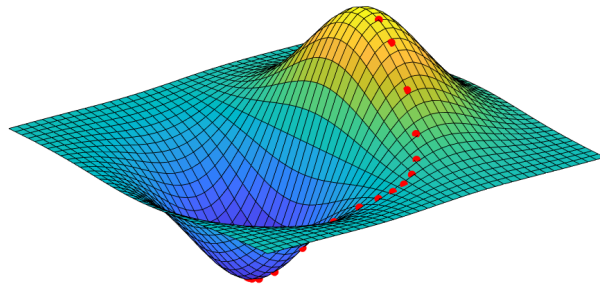


图 2.2: 梯度下降示意图

对于多层的神经网络，可以先求出最后一层神经元参数的梯度，再利用偏导数链式法则，一层一层地往前计算每一层的参数的梯度。这就是反向传播算法（Backpropagation）

的思想。具体以三层感知机为例，最后一层涉及的偏导数关系为：

$$\frac{\partial J}{\partial a_j^{(3)}} = \frac{1}{m}(a_j^{(3)} - y_j) \longleftarrow J(a_1^{(3)}, \dots, a_m^{(3)}) = \frac{1}{2p} \sum_{j=0}^{m-1} (a_j^{(3)} - y_j)^2, \quad (2.10a)$$

$$\frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} = h'_{(3)}(z_j^{(3)}) \longleftarrow a_j^{(3)} = h_{(3)}(z_j^{(3)}), \quad (2.10b)$$

$$\frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}} = a_i^{(2)} \longleftarrow z_j^{(3)} = \sum_{i=1}^{n^{(2)}} w_{ji}^{(3)} a_i^{(2)} + b_j^{(3)}, \quad (2.10c)$$

$$\frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} = 1 \longleftarrow z_j^{(3)} = \sum_{i=1}^{n^{(2)}} w_{ji}^{(3)} a_i^{(2)} + b_j^{(3)}, \quad (2.10d)$$

$$\frac{\partial z_j^{(3)}}{\partial a_i^{(2)}} = w_{ji}^{(3)} \longleftarrow z_j^{(3)} = \sum_{i=1}^{n^{(2)}} w_{ji}^{(3)} a_i^{(2)} + b_j^{(3)}. \quad (2.10e)$$

根据链式法则可得

$$\frac{\partial J}{\partial w_{ji}^{(3)}} = \frac{\partial J}{\partial a_j^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}} = \frac{1}{m}(a_j^{(3)} - y_j) h'_{(3)}(z_j^{(3)}) a_i^{(2)}, \quad (2.11a)$$

$$\frac{\partial J}{\partial b_j^{(3)}} = \frac{\partial J}{\partial a_j^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} = \frac{1}{m}(a_j^{(3)} - y_j) h'_{(3)}(z_j^{(3)}), \quad (2.11b)$$

$$\frac{\partial J}{\partial a_i^{(2)}} = \sum_j \frac{\partial J}{\partial a_j^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial z_j^{(3)}}{\partial a_i^{(2)}} = \sum_j \frac{1}{m}(a_j^{(3)} - y_j) h'_{(3)}(z_j^{(3)}) w_{ji}^{(3)}, \quad (2.11c)$$

其中式 2.11a 和式 2.11b 就是最后一层的参数的梯度；而式 2.11c 可以将梯度传播回上一层（即用它可以计算上一层参数的梯度）：

$$\frac{\partial J}{\partial w_{ji}^{(2)}} = \frac{\partial J}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial w_{ji}^{(2)}} \quad (2.12a)$$

$$\frac{\partial J}{\partial b_j^{(2)}} = \frac{\partial J}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial b_j^{(2)}} \quad (2.12b)$$

$$\frac{\partial J}{\partial a_i^{(1)}} = \sum_j \frac{\partial J}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial a_i^{(1)}} \quad (2.12c)$$

其中式 2.12a 和式 2.12b 是上一层参数的梯度；而式 2.12c 可以将梯度再往回传递一层。以此类推，层层反向传播，就可以将所有参数的梯度计算出来，然后每一层按照式 2.8 更新参数即可。

普通的梯度下降，也叫批量梯度下降（Batch Gradient Descent），每更新一次参数，就要用所有样本计算损失函数，较为耗时。随机梯度下降（Stochastic Gradient Descent）每

更新一次参数只用随机选取的一个样本来计算损失函数，这样可以节省时间，但是损失函数数值下降较慢。小批量梯度下降（Mini-Batch Gradient Descent）每次选取一批样本来计算损失函数，收敛速度和计算耗时介于批量梯度下降和随机梯度下降之间。

2.3 本次实验中的神经网络结构

本次实验的基础部分是实现一个简单的三层感知机，输入层有 9 个神经元，隐含层有 3 个神经元，输出层只有一个神经元，预期输出 $-1, 0, 1$ ，分别对应输入字母 Z, V, N 的 3x3 二值图像，见图 2.3。

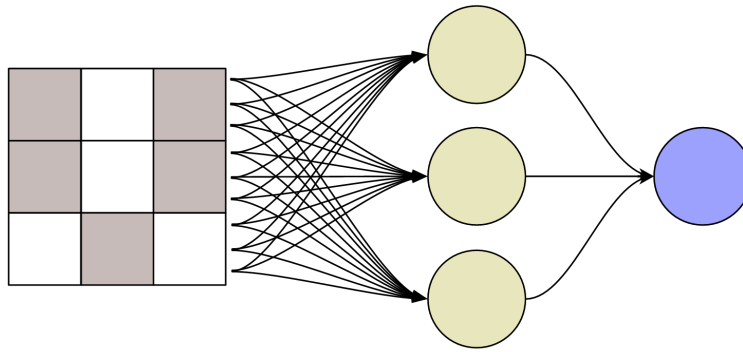


图 2.3: 实验所用三层神经网络的示意图

训练集和验证集相同，样本数为 3。由于此分类任务过于简单，隐含层和输出层都没有使用激活函数。

3 多层感知机的软件设计

下面分别用 MATLAB 训练第 2.3 节中的神经网络。为了节省位宽便于在硬件上使用更少资源，本次实现所用浮点数均为单精度（4 字节，32 位）而非双精度（8 字节，64 位）。

3.1 前向传播的实现

前向传播的公式为：

$$z_j^{(1)} = \sum_{i=1}^9 w_{ji}^{(1)} x_i + b_j^{(1)} \quad (3.1a)$$

$$z^{(2)} = \sum_{j=1}^3 w_j^{(2)} z_j^{(1)} + b^{(2)} \quad (3.1b)$$

MATLAB 实现见附录 8.1.1。

3.2 反向传播的实现

使用随机梯度下降。定义损失函数为

$$J(\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \mathbf{w}^{(2)}, b^{(2)}) = \frac{1}{2}(z^{(2)} - y)^2 \quad (3.2)$$

根据第 2.2.3 节的推导以及式 3.1，可得第二层参数的更新公式如下：

$$\Delta w_j^{(2)} = -\eta^{(2)} \sum_{k=1}^3 \frac{\partial J}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial w_j^{(2)}} = -\eta^{(2)} \frac{1}{3} \sum_{k=1}^3 (z_k^{(2)} - y_k) z_j^{(1)} \quad (3.3a)$$

$$\Delta b^{(2)} = -\eta^{(2)} \sum_{k=1}^3 \frac{\partial J}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial b^{(2)}} = -\eta^{(2)} \frac{1}{3} \sum_{k=1}^3 (z_k^{(2)} - y_k) \quad (3.3b)$$

$$\Delta z_j^{(1)} = -\eta^{(2)} \sum_{k=1}^3 \frac{\partial J}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial z_j^{(1)}} = -\eta^{(2)} \frac{1}{3} \sum_{k=1}^3 (z_k^{(2)} - y_k) w_j^{(2)} \quad (3.3c)$$

其中式 3.3c 将梯度通过偏导数链式法则传递回第一层。第一层的参数更新公式如下：

$$\Delta w_{ji}^{(1)} = -\eta^{(1)} \Delta z_j^{(1)} \frac{\partial z_j^{(1)}}{\partial w_{ji}^{(1)}} = -\eta^{(1)} \Delta z_j^{(1)} x_i \quad (3.4a)$$

$$\Delta b_j^{(1)} = -\eta^{(1)} \Delta z_j^{(1)} \frac{\partial z_j^{(1)}}{\partial b_j^{(1)}} = -\eta^{(1)} \Delta z_j^{(1)} \quad (3.4b)$$

式 3.3 和 3.4 中的 η_2 和 η_1 分别为第二层和第一层的学习率。

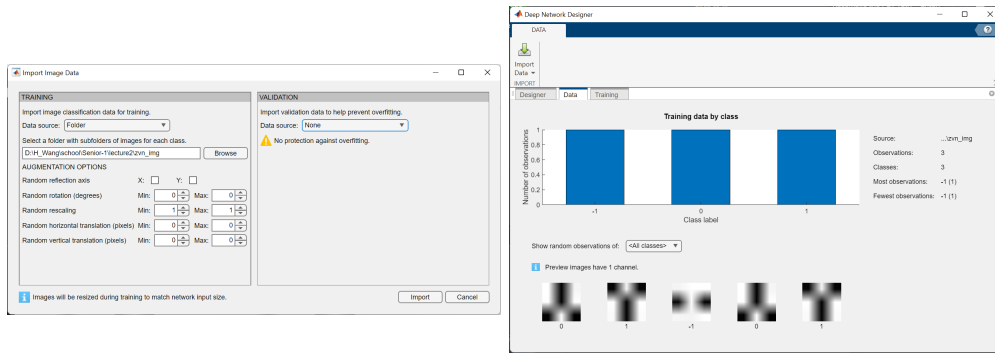
MATLAB 实现见附录 8.1.2。

3.3 MATLAB 深度神经网络工具箱

MATLAB 提供 Deep Network Designer，可以用于快速设计深度神经网络，见图 3.1。步骤如下：

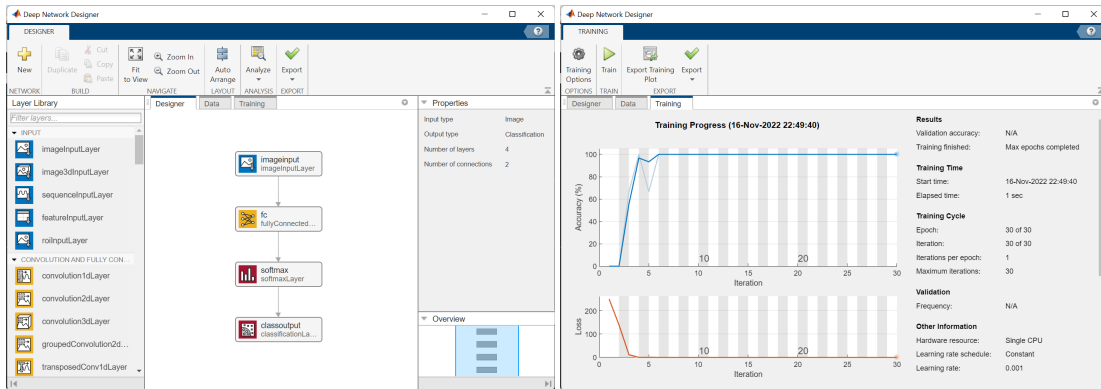
1. **制作训练集** 将 Z, V, N 保存为 png 图片格式，并分别至于名称为“-1”，“0”，“1”的文件夹中；
2. **打开工具箱** 在命令行输入 `deepNetworkDesigner` 打开深度神经网络工具箱；
3. **导入训练集** 选择“Data”标签，点击“Import Data”，输入文件夹路径导入训练集图片（见图 3.1a，由于没有验证集，“VALIDATION”栏的“Data source”选择“None”），然后即可查看训练集概况（见图 3.1b）；

4. **设计网络架构** 选择“Designer”标签，从左侧栏拖出合适的层（这里选择了一个全连接层和一个 Softmax 激活层）构成神经网络。可以选中层之后在右侧栏设置其参数，例如输出维度和个数；
5. **训练** 选择“Training”标签，点击“Training Options”为训练过程设置合适的超参数，然后点击“Train”即可开始训练。训练过程中可以实时查看 Accuracy 和 Loss；
6. **导出** 点击“Export”，将训练好的网络导出到工作区，重命名为 `net`。可以在工作区双击变量查看权重和偏置。以后可以在脚本中用 `predict(net, data)` 来进行预测。



(a) 导入训练集图片

(b) 查看训练集



(c) 设置网络结构

(d) 训练

图 3.1: MATLAB Deep Network Designer

4 多层感知机的硬件设计

本次实验使用的软件平台为 Vivado 2020.2，硬件平台为 PYNQ-Z2。

4.1 整体框架和设计思路

硬件设计的整体框架见图 4.1a，具体的数据流和控制流实现见图 4.1b。layer1 模块和 layer2 模块分别负责第一层网络和第二层网络的计算，顶层模块将二者连接起来。

第一层网络的计算用遍历 Block ROM 的方式实现，因此 layer1 模块中例化了 ROM。由于输入只有 -1 和 1 ，可以用浮点累加器将权重累加得到结果，输入 1 为加，输入 -1 为减。因此 layer1 模块中例化了浮点累加器。另外，需要计数器和其他逻辑来控制数据的流动。

第二层网络的计算，由于参数较少，不使用 Block ROM，而是直接将参数存储于寄存器中。为了将计算并行化，layer2 模块中并行地使用了三个浮点乘法器和一个四输入加法树（需要三个浮点加法器），见图 4.1b。

所有 Verilog 代码见附录 8.2。

4.2 IP 核配置

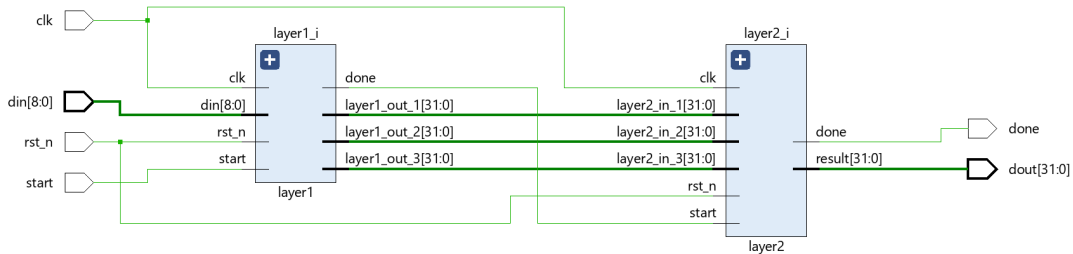
4.2.1 Block ROM

在 IP Catalog 中搜索 Block Memory Generator，进入配置界面（见图 4.2），选择单端口 ROM，设置位宽为 32，深度为 10，启用使能端口（Use ENA Pin），取消输出寄存器（Primitive Output Register）。如果使用输出寄存器，则延迟为 2 个周期，不使用则延迟为 1 个周期。

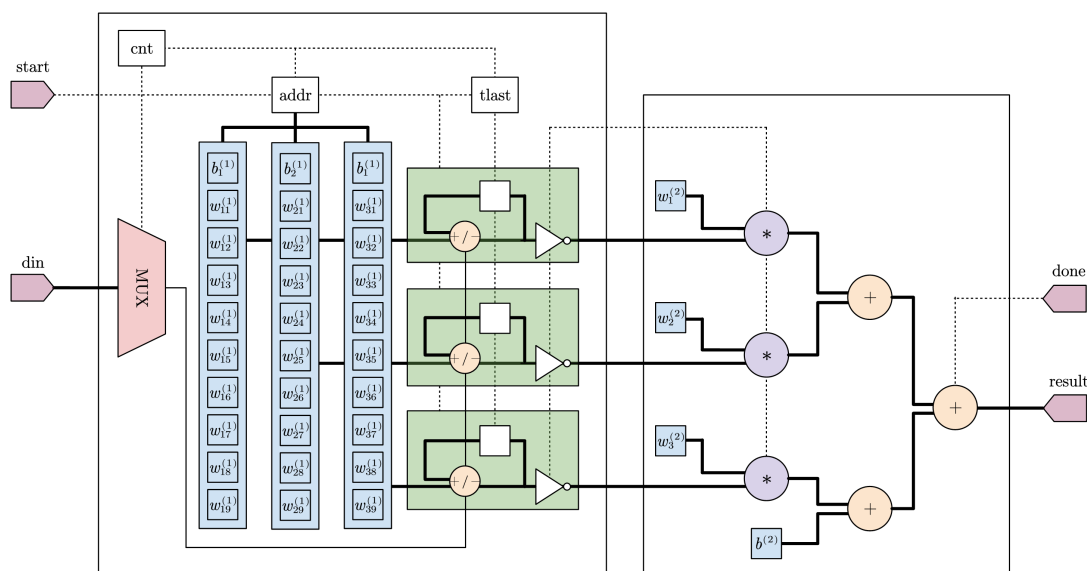
4.2.2 浮点累加器

首先在 IP Catalog 中搜索 Floating-point，进入配置界面，重命名为 `fp_acc`，然后选择“Both/Add/Subtract”处选择“Both”，此时会启用 `s_axis_operation` 接口，该接口最低位输入 0 可以将 IP 设置为加法，输入 1 则设置为减法，见图 4.3a。

浮点累加器的延迟选取了默认的最大值（22 个周期），流水线启动间隔为 1 周期，见图 4.3b。



(a) 整体框架示意图



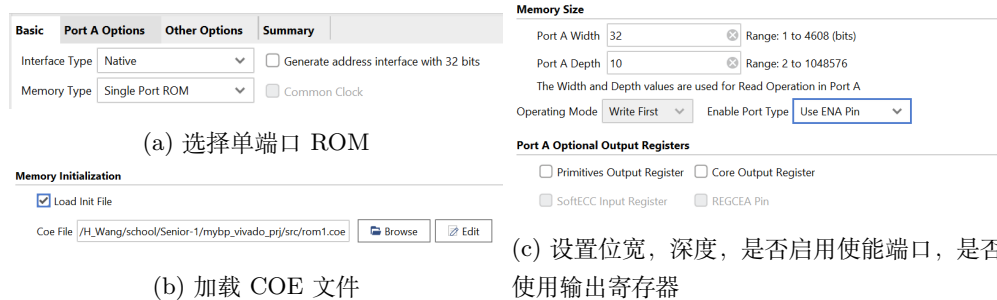
(b) 具体实现示意图

图 4.1: 硬件设计示意图

注意，累加器与加法器不同，它还需要提供 tlast 信号，以标记所累加的最后一个数据的时间。

4.2.3 浮点加法器

浮点加法器 IP 核命名为 `fp_add`，延迟使用默认最大值（11 个周期），流水线启动间隔为 1 个周期。在 “Both/Add/Subtract” 中选择 “Add”。其他设置与浮点累加器相同。

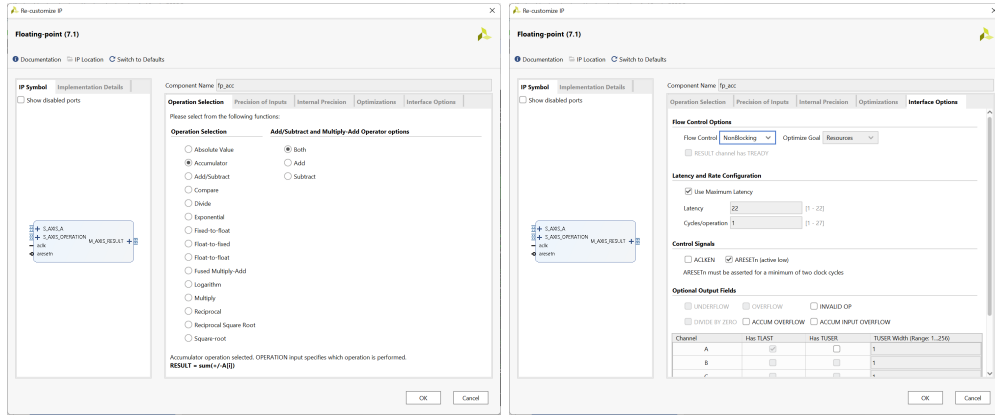


(a) 选择单端口 ROM

(b) 加载 COE 文件

(c) 设置位宽，深度，是否启用使能端口，是否使用输出寄存器

图 4.2: Block ROM IP 核配置



(a) 选择 “Accumulator” 和 “Both” 。(b) Flow Control 设置为 NonBlocking; LASS_AXIS_OPERATION 的最低位用来选择使用加法还是减法。

图 4.3: 浮点累加器 IP 核配置

4.2.4 浮点乘法器

浮点乘法器 IP 核命名为 `fp_mul`，延迟使用默认最大值（8 个周期），流水线启动间隔为 1 个周期。其他设置与浮点累加器相同。

4.3 性能分析

4.3.1 延迟和吞吐量

接下来计算延迟和吞吐量。浮点累加器的延迟为 $t_{\text{acc}} = 22$ 个周期，流水线启动间隔为 $\text{II} = 1$ 个周期，循环 $N = 10$ 次；浮点乘法器的延迟为 $t_{\text{mul}} = 8$ 个周期，流水线启动间隔为 $\text{II} = 1$ 个周期；浮点加法器的延迟为 $t_{\text{add}} = 11$ 个周期，流水线启动间隔为 $\text{II} = 1$ 个周期。因此总的延迟为

$$\begin{aligned}
 \text{Latency} &= [t_{\text{acc}} + (N - 1) \cdot \text{II}] + t_{\text{mul}} + 2 \cdot t_{\text{add}} \\
 &= (22 + 9) + 8 + 2 \cdot 11 \\
 &= 61
 \end{aligned} \tag{4.1}$$

个周期。注意到 ROM 的延迟为 $t_{\text{ROM}} = 1$ 个周期，但是可以让 ROM 提前一个周期加载好第一个数据从而消除读取 ROM 带来的延迟。

流水线启动间隔为

$$\text{Initiation Interval} = N \cdot \text{II} = 10 \tag{4.2}$$

个周期。延迟和流水线启动间隔的示意图见图 4.4。

4.3.2 仿真时序图

仿真代码见附录 8.2.2。时钟周期设置为 20ns；复位信号初始化为低电平，5 个周期之后被拉高；start 信号在复位变高之后的 5 个周期被拉高；此后，每 10 个周期循环输入 Z、V、N 三个字母的数据。仿真波形图见图 4.5。从图中可以看出累加器的延迟为 22 个周期（见 acc_result_tlast），乘法器的延迟为 8 个周期（见 mul_result_tvalid）以及加法器的延迟为 11 个周期（见 add_result_tvalid_1）。

4.3.3 资源使用情况

资源消耗情况见图 4.6。可以看出，一个单端口 Block ROM 只消耗 0.5 个 BRAM。这是因为 Xilinx 的一个 BRAM 可以作为单个 36kb RAM 使用，也可以作为两个 18Kb RAM 使用；浮点累加器，浮点乘法器，浮点加法器分别均使用 2 个 DSP，这是因为配置 IP 核时 Latency 均选取了默认的最大值。如果降低 Latency，则会消耗更多 DSP，但流水线启动间隔（Initiation Interval）不变。

4.4 上板验证

实验所用平台为 PYNQ-Z2，主控为 Zynq-7000 系列，内嵌 ARM Cortex-A9 双核，可以运行 Linux 操作系统。因此 Zynq SoC 可被分为 PS 端和 PL 端，PS 指 Processing System，含有 CPU 和 DDR，而 PL 指 Programmable Logic，即 FPGA 部分。在 Zynq 中，PS 端控制 PL 端运行，PL 端的时钟和复位都来自于 PS 端，二者一般通过 AXI 协议总线交流，AXI 互联模块既有硬核实现也有软核实现。

PYNQ 指 Python Productivity for Zynq。它是一个适用于 Zynq 器件的软件开发框架，通过高层次的封装，将底层硬件的细节与上层应用层分离，让使用者通过 Python 来使用和调试硬件 IP。需要注意，PYNQ 本身不能用来设计硬件，它只是用来调用硬件模块，使 IP 的调试与应用更为便捷。

本次实验采用 PS 端控制 PL 端 IP 的方式进行验证，使用 PYNQ Jupyter Notebook 中调用 IP 并展示结果。

4.4.1 搭建验证系统

为了搭建用于验证的 Zynq 系统，在 Vivado 的 Block Design 中添加 Zynq IP 核和 4 个 AXI GPIO IP 核，分别对应待验证 IP 核的四个端口。将 AXI GPIO IP 核与待验证的 IP 连接，其余 AXI 连线由 Vivado 自动完成，见图 4.7。

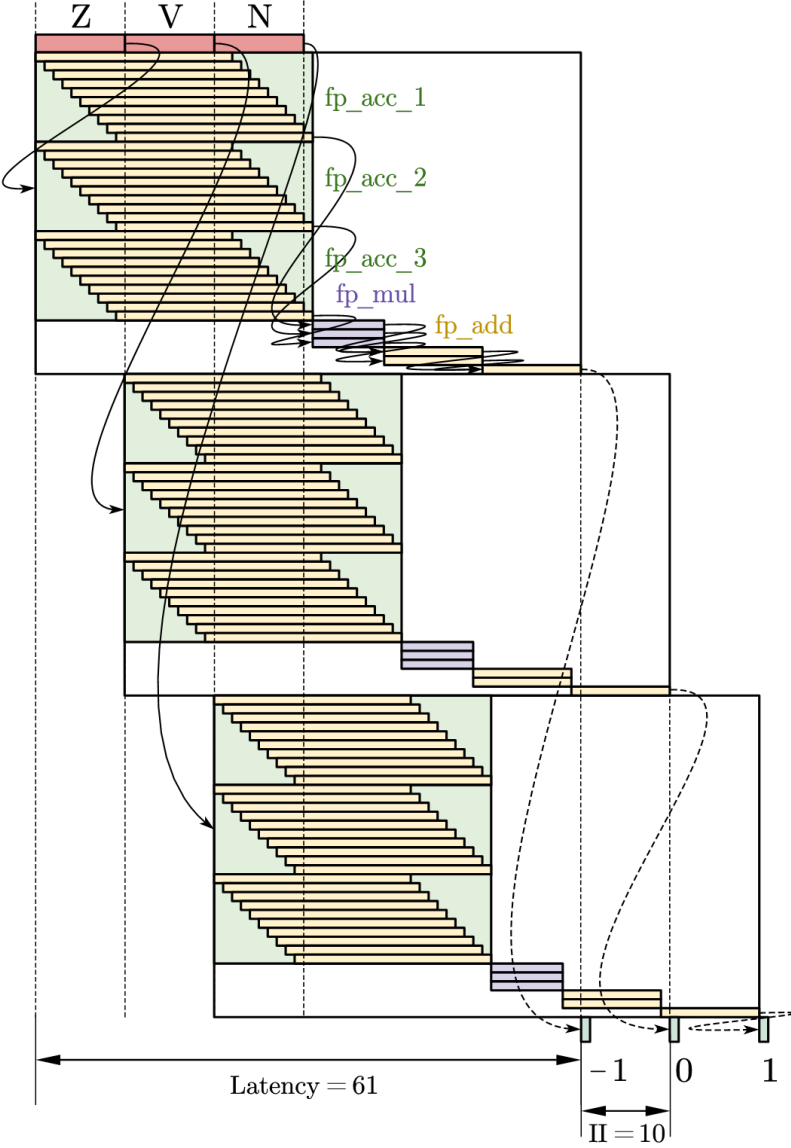


图 4.4: 延迟 (Latency) 和流水线启动间隔 (Initiation Interval, II)

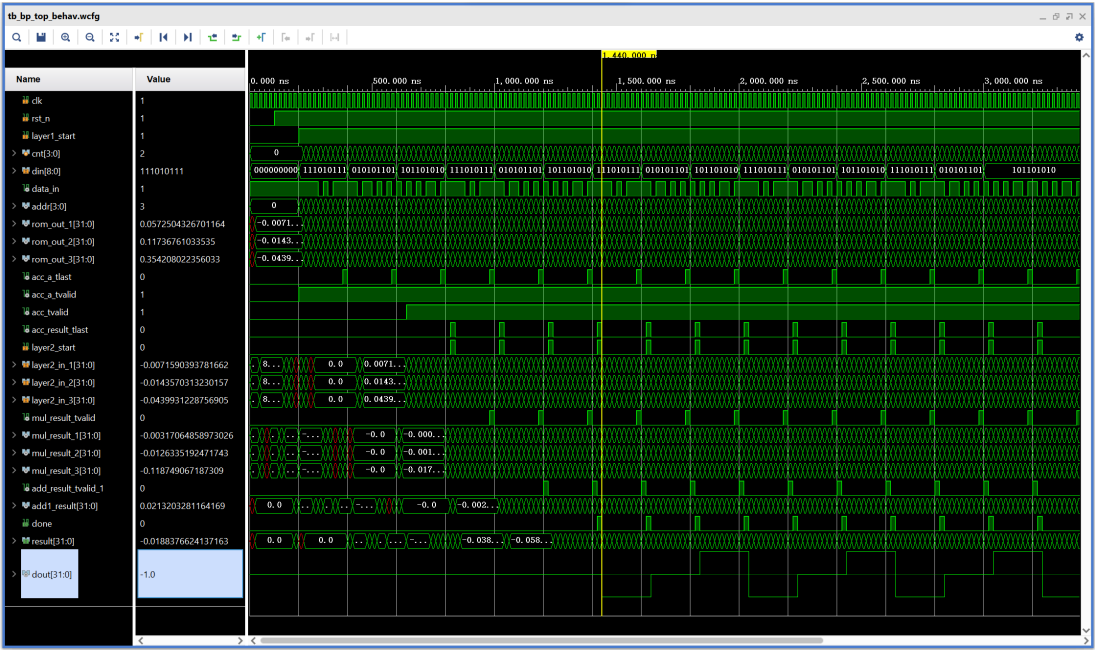


图 4.5: 仿真波形

Name	Slice LUTs (53200)	Slice Registers (106400)	Block RAM Tile (140)	DSPs (220)	Bonded IOB (200)	BUFGCTRL (32)
bp_top	3151	4220	1.5	18	45	1
layer1_i (layer1)	2221	2705	1.5	6	0	0
blk_rom_1_i (b	0	0	0.5	0	0	0
blk_rom_2_i (b	0	0	0.5	0	0	0
blk_rom_3_i (b	0	0	0.5	0	0	0
fp_acc_i1 (fp_a	736	900	0	2	0	0
fp_acc_i2 (fp_a	736	900	0	2	0	0
fp_acc_i3 (fp_a	736	900	0	2	0	0
layer2_i (layer2)	930	1515	0	12	0	0
fp_add_i1 (fp_	196	325	0	2	0	0
fp_add_i2 (fp_	196	325	0	2	0	0
fp_add_i3 (fp_	196	325	0	2	0	0
fp_mul_i1 (fp_	114	180	0	2	0	0
fp_mul_i2 (fp_	114	180	0	2	0	0
fp_mul_i3 (fp_	114	180	0	2	0	0

图 4.6: 资源消耗情况

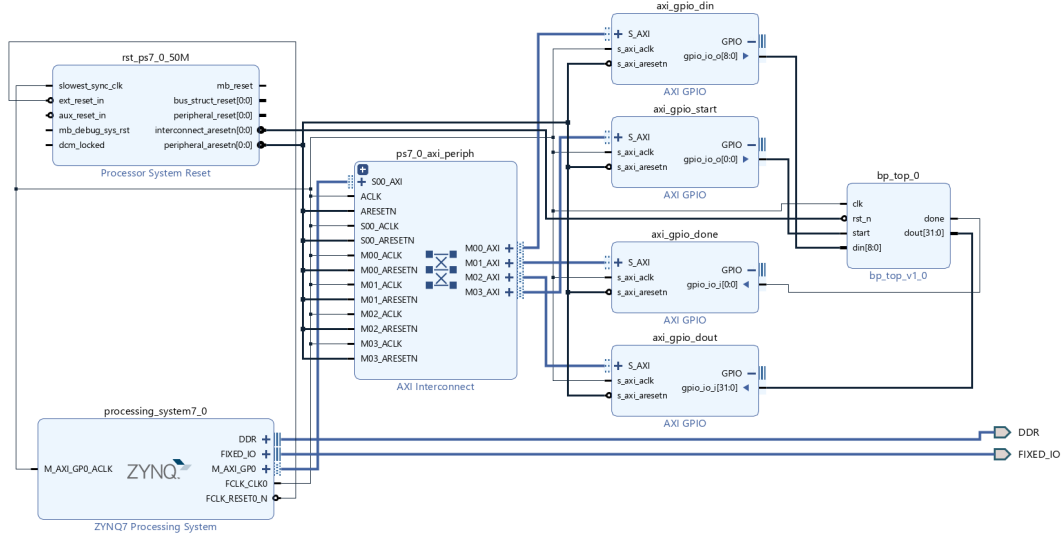


图 4.7: Zynq 系统设计

4.4.2 PYNQ Jupyter Notebook 设计

在 PYNQ Jupyter Notebook 中, 首先使用 `pynq.Overlay` 来下载比特流。在获取到 AxiGPIO 对象后, 使用 `read` 和 `write` 方法读写 AXI GPIO: 向 `gpio_din` 写入输入数据, 然后向 `gpio_start` 写入 1, 启动计算, 之后使用查询的方式判断计算是否完成, 计算完成后向 `gpio_start` 中写入 0, 停止计算, 然后从 `gpio_dout` 中读出计算结果。代码见附录 8.3。输出结果为:

```
Out [5]:
dout = -1.000000, elapsed time = 0.31 ms
dout = -0.000000, elapsed time = 0.16 ms
dout = 1.000000, elapsed time = 0.54 ms
```

可见输出结果正确。理论的计算时间只有 61 个时钟周期, 在 20ns 的周期下为 0.13us。实际耗时偏大可能是因为 PS 和 PL 是通过低性能的 GP 接口通信。GP 接口一般用于与低速外设 (例如 LED 灯、电机) 的通信。

5 拓展 1: 多层感知机的 HLS 实现

本次实验的 HLS 平台为 Vitis HLS 2020.2。HLS 指高层次综合 (High-Level Synthesis), 它将高级软件设计代码, 例如 C, C++, MATLAB, 转化为 Verilog, VHDL 等硬件描述语言 (HDL)。利用 HLS 可以大大缩短开发时间。

5.1 C Simulation

编写代码文件 `bp_top.cpp` 和 `bp_top.h` (见附录 8.2.1)。使用 Vitis HLS 的 `ap_int.h` 库提供的模板类 `ap_uint<9>`, 可以将 `din` 定义为 9 位无符号整数。由于输入和输出都只有一个数而非数组, 可以使用 `s_axilite` 接口协议。使用 `HLS PIPELINE` 编译指令可以将顶层函数流水线化, 用 `II=10` 将流水线启动间隔设定为 10 个周期。另外, 为了用 Block ROM 存储第一层参数, 需使用 `BIND_STORAGE` 编译指令。

编写 Testbench `tb_bp_top.cpp` (见附录 8.4.2), 分别将字母 Z、V、N 的数据输入给顶层函数, 查看输出结果。点击 C Simulation 进行 C++ 仿真。输出结果为:

```

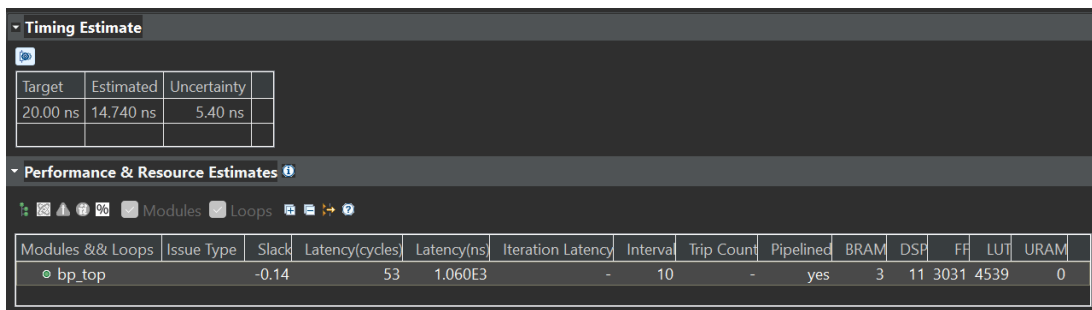
1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3   Compiling ../../src/tp_bp_top.cpp in debug mode
4   Compiling ../../src/layers.cpp in debug mode
5   Compiling ../../src/bp_top.cpp in debug mode
6   Generating csim.exe
7 z -> -1.000000
8 v -> -0.000000
9 n -> 1.000000
10 INFO: [SIM 1] CSim done with 0 errors.
11 INFO: [SIM 3] ***** CSIM finish *****

```

可见 C 仿真结果正确。

5.2 C Synthesis

点击 C Synthesis 进行高层次综合, 结果见图 5.1。



The screenshot shows the 'Timing Estimate' and 'Performance & Resource Estimates' sections of the Vitis synthesis report.

Timing Estimate

Target	Estimated	Uncertainty
20.00 ns	14.740 ns	5.40 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
bp_top		-0.14	53	1.060E3	-	10	-	yes	3	11	3031	4539	0

图 5.1: 高层次综合报告

可见延迟为 53 个周期, 流水线间隔为 10。还可以看到使用片上资源的数量。总体资源消耗情况比第 4 节中的 RTL 设计低一些, DSP 只用了 11 个, 而 RTL 设计用了 18 个。从 Schedule View 中可以看出延迟的降低是因为 HLS 使用了延迟更低的浮点 IP 核 (当然, 在 HDL 实现中也可以指定浮点 IP 核的计算延迟)。DSP 使用数量减少可能是因为 HLS 自动实现了 DSP 的复用。

5.3 Zynq 系统设计

用于调用 IP 的 Zynq 系统的 Block Design 见图 5.2。由于 HLS 导出的 IP 的接口都是 AXI 协议，使用自动连线即可，Vivado 会自动生成 AXI 互联软核。另外，由于实验只用到了 `s_axilite` 协议，没有用到 `m_axi` 协议，Zynq IP 核的设置保持默认含有 Master GP 接口即可，不需要启用 Slave HP 接口。

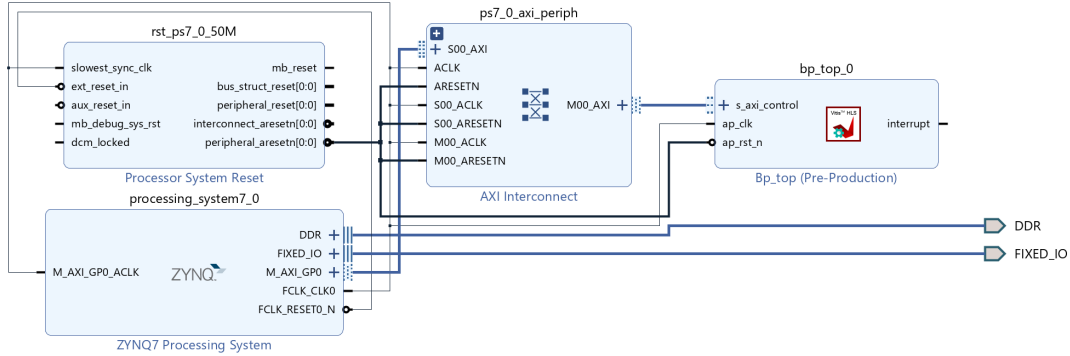


图 5.2: Zynq 系统的 Block Design

5.4 PYNQ 调试

与第 4.4 节类似，在 PYNQ Jupyter Notebook 中下载和调用 HLS 综合的 IP。代码见附录 8.4.3。输出结果为：

```
Out [10]:
dout = -1.000000, elapsed time = 0.0863 ms
dout = -0.000000, elapsed time = 0.0813 ms
dout = 1.000000, elapsed time = 0.0675 ms
```

可见结果正确，且耗时低于 RTL 设计。耗时更低可能是因为使用了 AXI 协议。

6 拓展 2: LeNet-5 的硬件加速

6.1 卷积神经网络介绍

卷积神经网络（Convolutional Neural Network, CNN）是一种区别于全连接神经网络的网络结构，它的主要操作是卷积，其特点是可以提取图像的局部特征。20 世纪 60 年代初，David Hubel 等在一篇论文中提出了 Receptive fields（感知野）的概念，是卷积神经网络的核心概念之一。1980 年，福岛邦彦提出了包含卷积层、池化层的神经网络。在此

基础上, 1998 年 Yann Lecun 提出了 LeNet-5。但由于在实际任务中表现不如 SVM 等算法, 且训练困难, 并未广泛应用。直到 2012 年, Hinton 组的 Alexnet 引入了 dropout 方法和新的深层结构, 大大提升了卷积神经网络的性能。此后, 卷积神经网络在工业界得到广泛的应用。

卷积神经网络与传统全连接神经网络 (多层感知机) 的不同之处在于引入了卷积层。卷积层 conv 的数学描述如下:

$$\text{conv}_K(I) = O: \quad O(i, j) = \sum_{x=0}^{m-1} \sum_{l=0}^{n-1} K(x, y) I(i + s_x x, j + s_y y), \quad (6.1)$$

$$K \in \mathbb{R}^m \times \mathbb{R}^n, \quad I \in \mathbb{R}^M \times \mathbb{R}^N, \quad O \in \mathbb{R}^{\lfloor \frac{M-m}{s_x} \rfloor + 1} \times \mathbb{R}^{\lfloor \frac{N-n}{s_y} \rfloor + 1}.$$

它将输入的 $M \times N$ 大小的图像输出为 $(\lfloor \frac{M-m}{s_x} \rfloor + 1) \times (\lfloor \frac{N-n}{s_y} \rfloor + 1)$ 大小的图像, 其中 s_x 、 s_y 分别是 x 和 y 方向上的步长 (Stride)。 K 叫做卷积核, 大小为 $m \times n$ 。通常有 $m = n$, $M = N$ 以及 $s_x = s_y$ 。卷积的示意图见图 6.1。

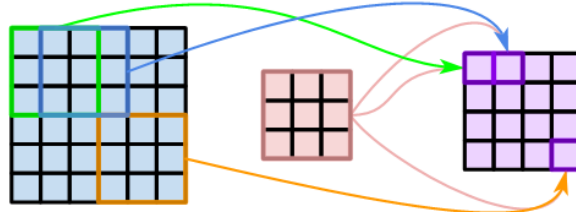


图 6.1: 卷积示意图

卷积的概念在卷积神经网络之前就已存在。在传统的计算机视觉和模式识别中, 卷积又被称为滤波器 (Filter)。一个简单的应用是边缘检测, 如下是两个分别可以检测垂直于 x 方向和 y 方向上的边缘的卷积核, 它们叫做 Sobel 算子:

$$\sigma_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \sigma_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (6.2)$$

在卷积神经网络中, 每一层的数据不只是一张图片, 而是有多张图片, 每一张图片称为一个通道 (Channel)。因此, 若输入通道为 C_i , 输出通道为 C_o , 则该层拥有 $C_i \times C_o$ 个卷积核。可见, 每一个卷积层的权重数据是四维的, 可以称为张量 (Tensor), 示意图见图 6.2。此时的公式如下:

$$\text{conv}_K(I) = O: \quad O(c_o, i, j) = \sum_{c_i=0}^{C_i-1} \sum_{c_o=0}^{C_o-1} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} K(c_o, c_i, x, y) I(c_i, i + s_x x, j + s_y y) \quad (6.3)$$

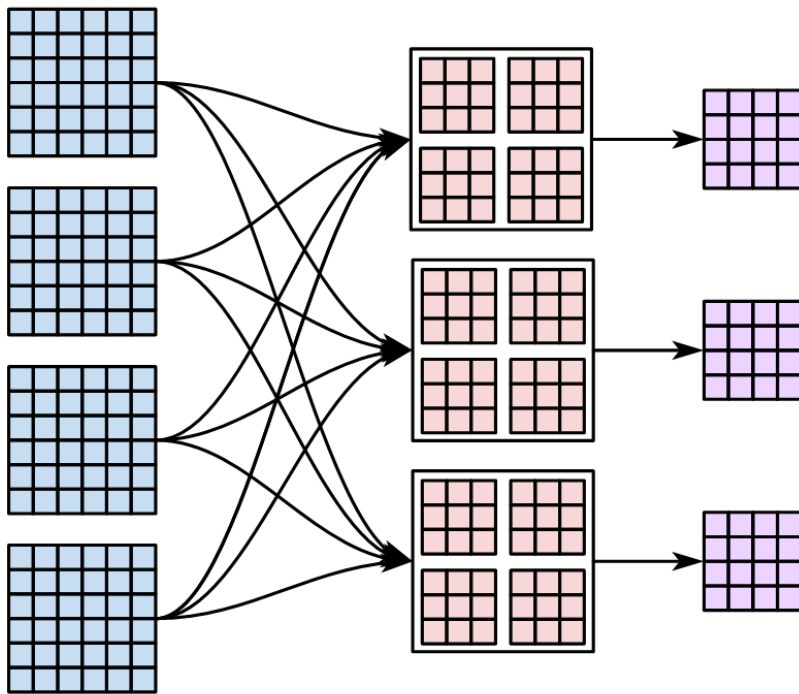


图 6.2: 输入通道为 3, 输出通道为 4, 一共需要 $3 \times 4 = 12$ 个卷积核

需要注意的是，在卷积层之后还有一个激活层，对图像中的每个点都应用一次激活函数。但是在不导致歧义的情况下，通常把卷积层和其后的激活层一起叫做卷积层。

卷积神经网络另外一个常用的层是池化（Pooling）层，也叫下采样（Downsampling）层。池化有平均池化（Average-Pooling），最大池化（Max-Pooling）等方式。一个 2×2 平均池化层可以等效为一个卷积层，其对应卷积核为：

$$\text{Average Pooling Kernal} = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}, \quad (6.4)$$

且步长为 2。可以看出，它是求一个 2×2 区域的最大值。最大池化则是求该区域的最大值。根据步长和池化区域大小的不同，还可将池化分为重叠池化和非重叠池化。

6.2 LeNet-5 的架构

LeNet 是最早的卷积神经网络。LeNet-5 的架构如图 6.3 所示。输入数据为 32×32 黑白（单通道）图像，经过步长为 1，输出通道为 6 的 5×5 卷积，其维度变为 $6 \times 28 \times 28$ ；经过 2×2 平均池化，其维度变为 $6 \times 14 \times 14$ ；再经过 16 通道 5×5 卷积和池化，其维度变为 $16 \times 5 \times 5$ ；最后通过两个分别含 120 和 84 个神经元的全连接层，通过 softmax 函数输出十种分类的概率。

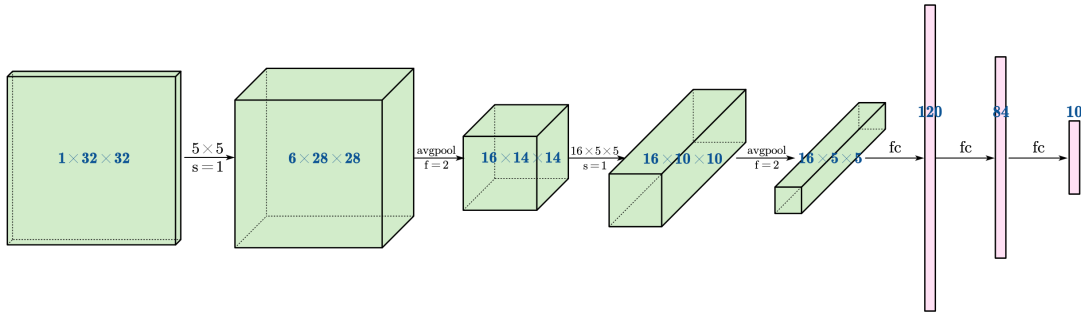


图 6.3: LeNet 的架构示意图

本次实验的网络较原 LeNet-5 有部分改动：

- 首先，原 LeNet-5 使用的激活函数为 sigmoid。但现在的深度卷积神经网络通常使用 ReLU 作为激活函数，它可以解决深层神经网络的梯度消失问题，因此本次实验实现的 LeNet-5 的激活函数使用 ReLU 而不是原本的 sigmoid；
- 其次，原 LeNet-5 使用的池化是平均池化，但对于深色背景的图片，最大池化的效果比平均池化更好，因此本次实验实现的 LeNet-5 使用最大池化；
- 最后，原 LeNet-5 的输出层使用径向基函数，但现在常用 softmax 作为多分类函数，效果更好，因此本次实验使用 softmax 作为输出层激活函数。

6.3 LeNet 的软件实现

6.3.1 Pytorch 训练

本次实验使用 Pytorch 框架来训练 LeNet-5。神经网络框架使用自动微分，计算图等技术，自动完成求形式导数和反向传播，使得用户只需设置样本和超参数、搭建网络框架即可训练，缩短了神经网络的开发周期。本次实验首先使用 `torchvision.datasets.MNIST` 下载并配置 MNIST 手写数字图像数据集，然后继承 `torch.nn.Module` 类搭建 LeNet-5 网络，最后定义损失函数和优化方法，在循环中完成前向传播和反向传播，从而实现训练。训练代码见 8.5.1。训练好的模型保存为 `model.pth`，可以用 `torch.load` 加载该模型，然后查看各层参数。

训练所得模型的准确率为 97.65%。

6.3.2 C++ 实现前向传播

由于 HLS 是将 C++ 代码编译为 HDL，因此先要将前向传播用 C++ 实现。把参数写入头文件中，写好卷积层、池化层和全连接层的函数，然后在顶层函数中搭建网络结构。另外，还需要 C++ Testbench 来调用顶层函数，测试功能是否正确。代码实现见附录 8.5.2。Testbench 输出结果为：

```

1 7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5
   1 2 4 4
2 Prediction of 50 pictures takes time 2002.00 ms
3 Accuracy = 100.00%
```

其中第一行是预测的 50 张图片的标签，第二行表示预测 50 张图片消耗的时间，为 2002 毫秒，第三行是预测 50 张图片的准确率。此代码在 Zynq-7020 的 PS 端 ARM Cortex A9 上编译运行。ARM Cortex A9 是嵌入式 CPU，比 PC 的 CPU 性能低。如果在 PC 上运行（实验的 PC 上位机 CPU 为 AMD Ryzen 5 5600H），只需要 50ms。之所以在 Zynq-7020 的 PS 端运行，是为了与之后的硬件加速器的结果做对比。

6.4 LeNet 的硬件实现

6.4.1 整体架构

Zynq Block Design 如图 6.4 所示。Zynq 的 PS 端不是直接将图片数据传输给加速器 IP 核，而是在 PS DDR 中开辟一块内存，用于存储输入图片的数据。然后，PS 端将输入图片在 DDR 中的地址通过 Zynq 的 Master GP 接口告诉 IP 核，IP 核再通过 Zynq 的 Slave GP 接口从 DDR 中获取图片的数据。这就是 `m_axi` 协议（即 AXI Master）的使用方式。如果使用 `axis` 协议（即 AXI Stream 协议），则是通过 DMA 来访问 DDR 内存。此

外, 还有 `s_axilite` 协议, 适合少量数据的传输, 不需要将数据先放在内存中, 而是 PS 端直接通过 Zynq 的 Slave GP 接口向 IP 核传输数据。

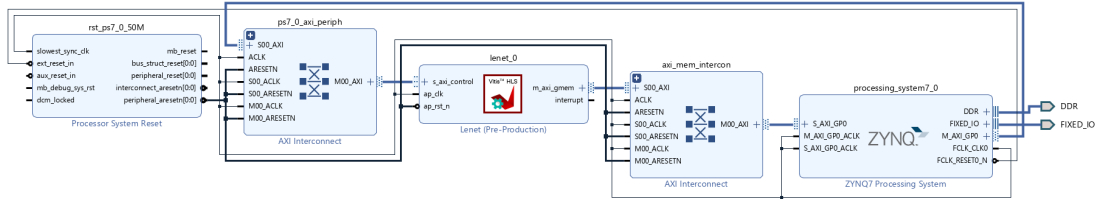


图 6.4: LeNet 的 Zynq Block Design

PL 端时钟是由 PS 端提供的, 可以在 Zynq IP 核里修改。本次实验使用的时钟周期为 10ns。

6.4.2 Baseline 实现

正确实现算法功能, 但没有针对算法特点做硬件优化的实现, 称为 Baseline 实现。第 6.3.2 节的代码就是 Baseline 实现, 其 HLS 综合报告见图 6.5。可以看出, 一个通道卷积操作的流水线 II 为 13, 吞吐量较低。第一层卷积的延迟为 150688 个周期, 第二层卷积的延迟为 76135 个周期, 第一层全连接层的延迟为 288015, 均较长。

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSH	FF	LUT	URAM
lenet	II Violation	-1.27	586124	5.861E6	-	586125	-	no	227	12	15127	12361	0
conv_relu_float_float_6ul_16ul_14ul_14ul_5ul_10ul_10ul_s	II Violation	-1.27	150689	1.507E6	-	150689	-	no	27	0	4547	2866	0
conv_oc		-	150688	1.507E6	9418	-	16	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	100	1.000E3	2	1	100	yes	-	-	-	-	-
conv_acc_0		-	9204	9.204E4	1534	-	6	no	-	-	-	-	-
conv_relu_0_conv_relu_1	II Violation	-	1421	1.421E4	135	13	100	yes	-	-	-	-	-
conv_acc_1_conv_acc_2		-	107	1.070E3	9	1	100	yes	-	-	-	-	-
conv_bias_0_conv_bias_1		-	107	1.070E3	9	1	100	yes	-	-	-	-	-
conv_relu_float_float_1ul_6ul_32ul_5ul_5ul_28ul_s	II Violation	-1.27	76135	7.610E5	-	76135	-	no	4	0	5326	2479	0
conv_oc		-	76134	7.610E5	12689	-	6	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	784	7.840E3	2	1	784	yes	-	-	-	-	-
conv_relu_0_conv_relu_1	II Violation	-	10312	1.030E5	134	13	784	yes	-	-	-	-	-
conv_acc_1_conv_acc_2		-	791	7.910E3	9	1	784	yes	-	-	-	-	-
conv_bias_0_conv_bias_1		-	791	7.910E3	9	1	784	yes	-	-	-	-	-
Loop 1		-	1024	1.024E4	1	1	1024	yes	-	-	-	-	-
preprocess_VITIS_LOOP_55_1		-	1092	1.092E4	70	1	1024	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	2362	2.362E4	13	2	1176	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	810	8.100E3	13	2	400	yes	-	-	-	-	-
VITIS_LOOP_215_1_VITIS_LOOP_216_2_VITIS_LOOP_217_3		-	403	4.030E3	4	1	400	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	288015	2.880E6	22	6	48000	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	60493	6.050E5	20	6	10080	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	5053	5.053E4	20	6	840	yes	-	-	-	-	-
max_idx	II Violation	-	21	210.000	5	2	9	yes	-	-	-	-	-

图 6.5: Baseline 实现 HLS 综合报告

PYNQ Jupyter Notebook 中的输出结果为:


```

Out [9]:
[7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4,
 0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4,
 4]
Elapsed time: 508.60 ms
accuracy = 100.00%

```

Baseline 实现的 509 毫秒与软件实现的 2000 毫秒相比，速度提升了约 293%。为了进一步设计的吞吐量，需要针对算法特点做一些优化，这将在接下来的几节里介绍。

6.4.3 优化 1: 卷积层——Line/Window Buffer

在硬件设计中，速度和面积的权衡非常重要。例如 CPU 片上的寄存器用的是 SRAM，速度快，但是占用面积大。而内存由于需要相对大量存储的原因，只能选用牺牲速度换取面积的 DRAM。

同理，在 FPGA 设计中，也要考虑速度与面积的权衡，其中一个重要的主题就是 BRAM（即 Block RAM）和寄存器的权衡。一方面，寄存器的可访问性比 BRAM 要高：BRAM 一个周期只能读 1 到 2 个数据（取决于单端口还是双端口），且数据有 1 到 3 个周期的延迟（第一个周期将地址传入，最快第二个周期才能获取到数据）；而寄存器的数据访问不受端口数量的限制（直接用导线扇出即可），也不需要延迟 1 个周期才能获取数据。但是另一方面，BRAM 的存储密度要远高于寄存器。因此，数据的存储选用 BRAM 还是寄存器，是速度与面积的权衡的一个体现，也是 FPGA 设计要考虑的一个重要问题。

在 HLS 设计中，Xilinx 的 `ARRAY_PARTITION` 编译指令就是用来处理此类问题的。如果使用 `complete` 参数，则将数组全部综合为寄存器，吞吐量大，但不适合数组很大的场景；如果使用 `factor=2` 的参数，则可以将本来用一个 BRAM 存储的数组拆分为两组，分别用两个 BRAM 存储，这样读端口数量就翻倍了，实现了更高的可访问性。

寄存器就是一组触发器（FF，Flip-Flop），这是因为触发器只有一位，而寄存器有多位。因此，上述数组的寄存器实现有时候也叫 FF 实现。另外，在 Verilog 中，一组寄存器也可以称为内存（Memory），但这些内存是用 FF 实现的，因此叫分布式内存（Distributed RAM），用来与块内存（Block RAM）做区分。

还有一种优化策略是，像 CPU 高速缓存那样，实现一个介于 BRAM 和寄存器之间的缓存，从而提高设计的吞吐量。卷积运算中的行缓存（Line Buffer）和窗缓存（Window Buffer）就是这样一个例子：如果将图像数据全部存储于 BRAM 中，由于读端口有限，一个周期只能读出两个数据，吞吐量太小；但如果全部用 FF 存储，占用面积又太大。Line Buffer 利用卷积操作的特点，可以实现面积与速度的均衡：假设卷积核大小为 3x3，那么可以将 3 行图像数据存到一个寄存器中（这个寄存器就是 Line Buffer），然后让卷积核滑过 Line Buffer（需要再将待卷积的数据从 Line Buffer 拷贝到 Window Buffer 中）。可以看出，实现 Line/Windows Buffer 之后，每个周期可以读出 9 个图像数据，如果卷积核大小

为 5x5，则一次性可以读出 25 个数据，这比 BRAM 的吞吐量要高得多。同时，与用 FF 存储整个图像相比，Line Buffer 只存储几行，面积又不至于太大。Line/Window Buffer 是硬件加速卷积运算的一个重要技术。

Line Buffer 和 Window Buffer 的示意图见图 6.6。

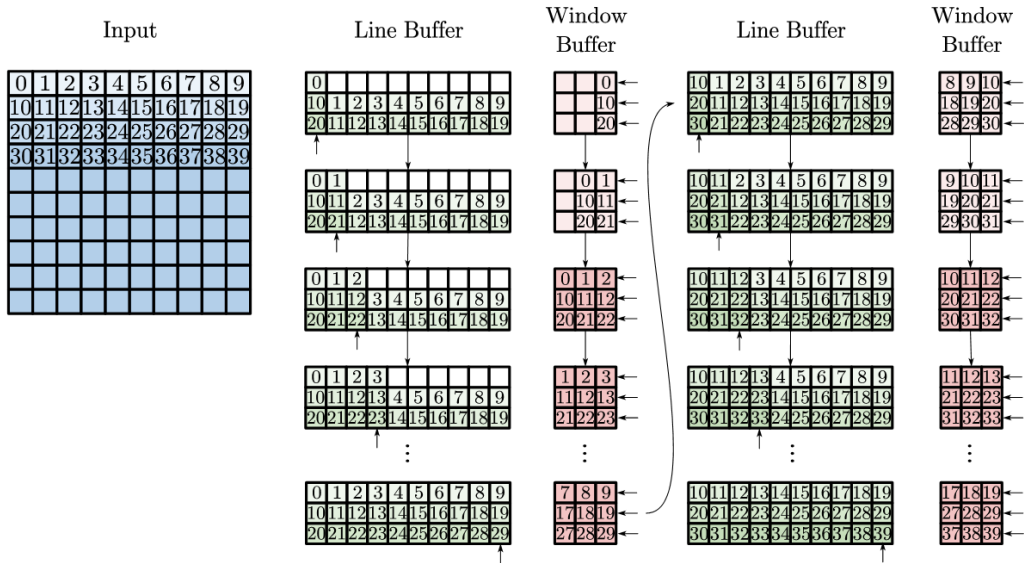


图 6.6: Line/Window Buffer 示意图

Line/Window Buffer 的实现见附录 8.5.3。实现 Line Buffer 之后的综合报告见图 6.7, PYNQ Jupyter Notebook 输出结果如下:

```
Out [11]: [7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4,
          0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4,
          4]
Elapsed time: 350.14 ms
```

从图 6.7 中可以看出，实现 Line/Window Buffer 之后，卷积操作的 II 减少到 1。第一层卷积操作的延迟由 150688 个周期下降到 21217 个周期，第二层卷积操作的延迟由 76135 个周期下降到 45889 个周期。50 张图片的推理时间由 509 毫秒下降到 350 毫秒。

但是，全连接层的流水线 II 仍然很高 (II=6)，全连接层的延迟较长，尤其是第一层，为 288015 个周期。因此下一节将针对全连接层进行优化。

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSR	FI	LU	URAM
lenet	II Violation	-1.19	426406	4.264E6	-	426407	-	no	227	127	31686	31286	0
conv_relu_float_float_1ul_6ul_32ul_32ul_5ul_5ul_28ul_28ul_s		-	21217	2.120E5	-	21217	-	no	4	0	10678	3617	0
conv_oc		-	21216	2.120E5	3536	-	6	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	784	7.840E3	2	1	784	yes	-	-	-	-	-
conv_initialize_line_buffer_0_conv_initialize_line_buffer_1		-	129	1.290E3	3	1	128	yes	-	-	-	-	-
conv_0_conv_1		-	1026	1.026E4	132	1	896	yes	-	-	-	-	-
conv_acc_1_conv_acc_2		-	791	7.910E3	9	1	784	yes	-	-	-	-	-
conv_bias_0_conv_bias_1		-	793	7.930E3	11	1	784	yes	-	-	-	-	-
conv_relu_float_float_6ul_16ul_14ul_14ul_5ul_5ul_10ul_10ul_s		-	45889	4.590E5	-	45889	-	no	27	0	7750	2846	0
conv_oc		-	45888	4.590E5	2868	-	16	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	100	1.000E3	2	1	100	yes	-	-	-	-	-
conv_acc_0		-	2652	2.652E4	442	-	6	no	-	-	-	-	-
conv_bias_0_conv_bias_1		-	109	1.090E3	11	1	100	yes	-	-	-	-	-
Loop 1		-	1024	1.024E4	1	1	1024	yes	-	-	-	-	-
preprocess_0_preprocess_1		-	1092	1.092E4	70	1	1024	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	2362	2.362E4	13	2	1176	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	810	8.100E3	13	2	400	yes	-	-	-	-	-
VITIS_LOOP_267_1_VITIS_LOOP_269_2_VITIS_LOOP_271_3		-	403	4.030E3	4	1	400	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	288015	2.880E6	22	6	48000	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	60493	6.050E5	20	6	10080	yes	-	-	-	-	-
fc_0_fc_1	II Violation	-	5053	5.053E4	20	6	840	yes	-	-	-	-	-
max_idx	II Violation	-	21	210.000	5	2	9	yes	-	-	-	-	-

图 6.7: 行缓存 (Line Buffer) 实现的综合报告

6.4.4 优化 2: 全连接层——循环交换

全连接层本质上是一个矩阵乘法，内层循环为一个累加操作，这个累加操作就是导致 II 较高，降低吞吐量的原因：每次累加操作必须等到上一次累加完成得到结果后才能进行，这样就无法利用加法器自身的流水线，使得流水线的 II 等于加法器的总延迟而不是 1 个周期。

为了消除每次累加结果对上一次累加结果的依赖性，可以考虑“循环交换 (Loop Interchange)”的优化方法。循环交换的示意图见图 6.8，其中 a—c 是未实现循环交换的示意图，d 是实现了循环交换的示意图。输入向量，矩阵和输出向量都存储在 BRAM 中，每个周期可以读写一个数据。可以看出，在未实现循环交换时，两次加法启动时间的间隔等于加法器的延迟；而在实现循环交换之后，两次加法启动时间的间隔可以减少到一个周期 (读 BRAM 需要一个周期)。

循环交换的实现见附录 8.5.3。实现了行缓存和循环交换之后的综合报告见图 6.9。在实现循环交换之前，全连接层的 II 等于 6 个周期，三个全连接层的延迟分别为 288015、60493、5053 个周期；实现循环交换之后，II 下降到 1 个周期，延迟分别降低到 48013、10091、850 个周期。

经过行缓存和循环交换的优化，整个网络的总延迟由 586124 个周期下降到 132262 个周期。在 PYNQ Jupyter Notebook 中的 50 张图片的预测结果如下，由只实现 Line/Window Buffer 的 350 毫秒进一步下降到 123 毫秒。

```
Out [10]:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4]
Elapsed time: 123.21 ms
```

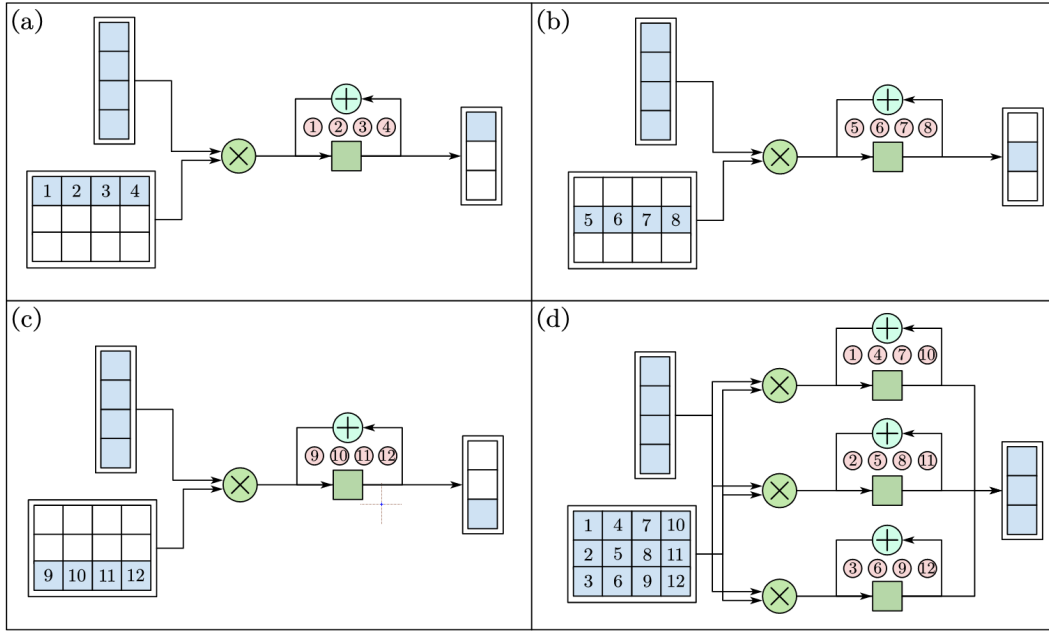


图 6.8: 循环交换 (Loop Interchange) 示意图。a—c 是未实现循环交换的示意图, d 是实现了循环交换的示意图。数字代表执行顺序。

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FH	LUT	URAM
lenet	II Violation	-1.27	132261	1.32366	-	132262	-	no	237	126	32708	31683	0
conv_relu_float_float_1ul_6ul_32ul_32ul_5ul_28ul_28ul_s		-	21217	2.120E5	-	21217	-	no	4	0	10678	3593	0
conv_oc		-	21216	2.120E5	3536	-	6	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	784	7.840E3	2	1	784	yes	-	-	-	-	-
conv_initialize_line_buffer_0_conv_initialize_line_buffer_1		-	129	1.290E3	3	1	128	yes	-	-	-	-	-
conv_0_conv_1		-	1026	1.026E4	132	1	896	yes	-	-	-	-	-
conv_acc_1_conv_acc_2		-	791	7.910E3	9	1	784	yes	-	-	-	-	-
conv_bias_0_conv_bias_1		-	793	7.930E3	11	1	784	yes	-	-	-	-	-
conv_relu_float_float_6ul_16ul_14ul_14ul_5ul_5ul_10ul_10ul_s		-	45889	4.590E5	-	45889	-	no	27	0	7750	2798	0
conv_oc		-	45888	4.590E5	2868	-	16	no	-	-	-	-	-
conv_initialize_acc_0_conv_initialize_acc_1		-	100	1.000E3	2	1	100	yes	-	-	-	-	-
conv_acc_0		-	2652	2.652E4	442	-	6	no	-	-	-	-	-
conv_initialize_line_buffer_0_conv_initialize_line_buffer_1		-	57	570.000	3	1	56	yes	-	-	-	-	-
conv_0_conv_1		-	270	2.700E3	132	1	140	yes	-	-	-	-	-
conv_acc_1_conv_acc_2		-	107	1.070E3	9	1	100	yes	-	-	-	-	-
conv_bias_0_conv_bias_1		-	109	1.090E3	11	1	100	yes	-	-	-	-	-
Loop 1		-	1024	1.024E4	1	1	1024	yes	-	-	-	-	-
preprocess_0_preprocess_1		-	1092	1.092E4	70	1	1024	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	2362	2.362E4	13	2	1176	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	II Violation	-	810	8.100E3	13	2	400	yes	-	-	-	-	-
VITIS_LOOP_270_1_VITIS_LOOP_272_2_VITIS_LOOP_274_3		-	403	4.030E3	4	1	400	yes	-	-	-	-	-
Loop 6		-	120	1.200E3	1	1	120	yes	-	-	-	-	-
fc_0_fc_1		-	48013	4.800E5	15	1	48000	yes	-	-	-	-	-
VITIS_LOOP_243_1		-	128	1.280E3	10	1	120	yes	-	-	-	-	-
Loop 9		-	84	840.000	1	1	84	yes	-	-	-	-	-
fc_0_fc_1		-	10091	1.010E5	13	1	10080	yes	-	-	-	-	-
VITIS_LOOP_243_1		-	92	920.000	10	1	84	yes	-	-	-	-	-
Loop 12		-	10	100.000	1	1	10	yes	-	-	-	-	-
fc_0	II Violation	-	850	8.500E3	21	10	84	yes	-	-	-	-	-
VITIS_LOOP_261_1		-	16	160.000	8	1	10	yes	-	-	-	-	-
max_idx	II Violation	-	21	210.000	5	2	9	yes	-	-	-	-	-

图 6.9: 实现了行缓存和循环交换之后的综合报告

6.4.5 优化 3: 双缓冲

双缓冲 (Double Buffer) 也叫乒乓操作, 它使用两组内存, 其中一组内存中的数据用于计算时, 另一组的内存开始拷贝下一次计算所需的数据。这样就用计算时间覆盖拷贝时间, 从而节省掉拷贝时间。双缓冲的示意图见图 6.10。

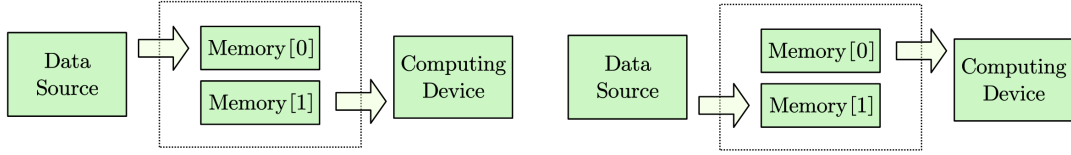


图 6.10: 双缓冲示意图

与 Line Buffer 不同, 双缓冲可以用软件实现, 这是因为图片数据存储在 PS 端的 DDR 中, PS 端只是将图片数据在 DDR 中的物理地址通过 Master GP 接口告知 IP 核, 然后 IP 核通过 Slave GP 接口从 DDR 中读取图片数据。

在本次实验中, 双缓冲可以运用在一批图片的预测上: 在上一张图片的数据给 IP 计算的同时, 开始拷贝下一张图片的数据。具体实现见附录 8.5.5。实现双缓冲前, 50 张图片的预测时间为 129.78ms, 实现双缓冲后, 50 张图片的预测时间为 122.90ms, 整体下降了约 7ms。

6.4.6 优化 4: int8 量化

查看训练好的模型的参数和激活值分布 (见图 6.11a), 发现大部分参数都集中在较小的范围, 用 32 位浮点数表示有一定“浪费”, 可以考虑用 8 位整数来存储大部分参数和数据, 从而减少模型体积, 这就是 int8 量化。一般用 int8 表示权重和激活值, 用 int32 表示偏置。这是因为权重和激活值的乘积用 int8 或者 int16 表示会溢出。本次实验采用较为简单的对称量化。设 $q_w \in \{-127, \dots, 127\}$ 为 w 的量化值, 则有

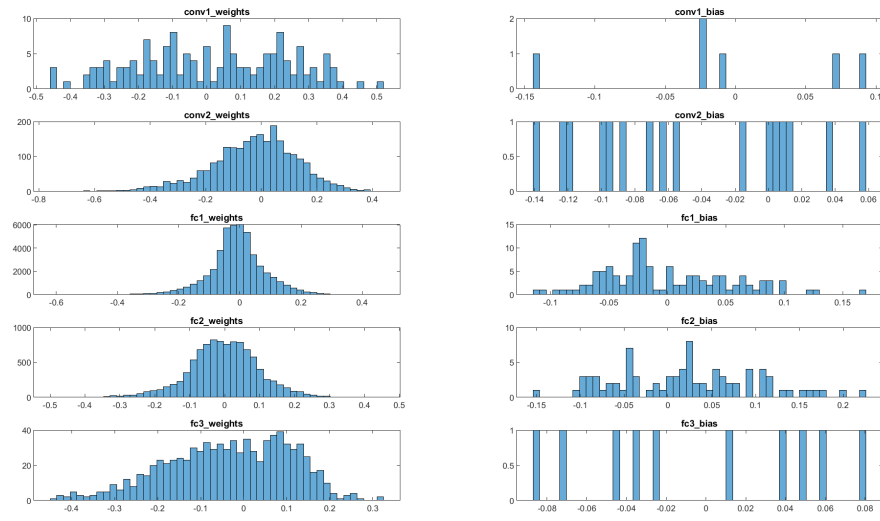
$$w = S_w q_w, \quad S_w = \frac{R_w}{127}, \quad (6.5)$$

$$q_w = \text{round}[\text{clamp}_{[-127, 127]}(w/S_w)], \quad (6.6)$$

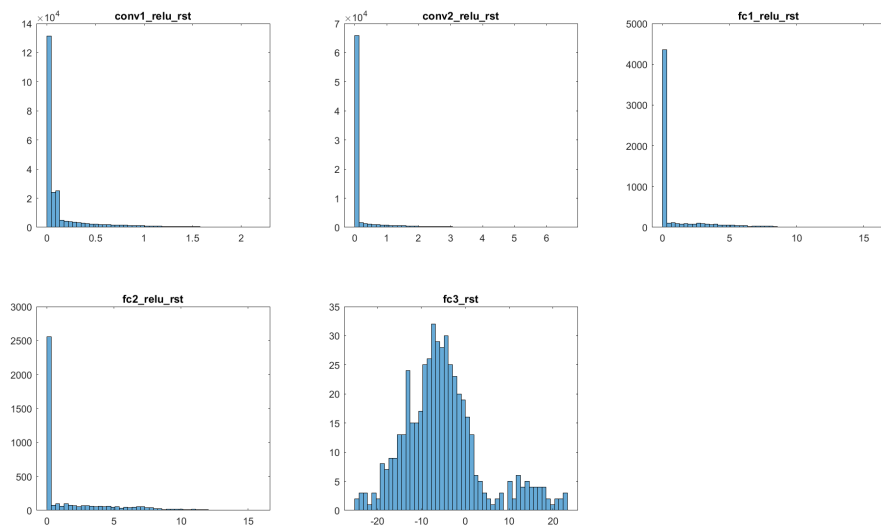
其中 clamp 函数定义如下:

$$\text{clamp}_{[m, n]}(x) = \begin{cases} n, & x > n \\ m, & x < m \\ x, & \text{otherwise} \end{cases}. \quad (6.7)$$

R_w 可以等于 $\max |w|$, 也可以小于 $\max |w|$ 。后者是为了防止离群值的影响, 将离群值直接截断。



(a) 参数的分布



(b) 激活值的分布

图 6.11: 参数和激活值的分布

记偏置的量化值为 q_b ，前一层激活值的量化值为 q_{a_1} ，后一层激活值的量化值为 q_{a_2} ，它们的缩放比率分别为 S_b 、 S_{a_1} 、 S_{a_2} ，则有

$$a_2 = \sum w a_1 + b, \quad (6.8)$$

$$S_{a_2} q_{a_2} = S_w S_{a_1} \sum q_w q_{a_1} + S_b q_b. \quad (6.9)$$

为了使 $\sum q_w q_{a_1}$ 与 q_b 直接相加，令

$$S_b = S_w S_{a_1}, \quad (6.10)$$

则有

$$q_{a_2} = \frac{S_w S_{a_1}}{S_{a_2}} \left(\sum q_w q_{a_1} + q_b \right) \quad (6.11)$$

$$= M \left(\sum q_w q_{a_1} + q_b \right). \quad (6.12)$$

现在除浮点数 M 以外的计算都可以通过整数完成，而 M 本身可以通过乘以整数 M_0 再右移 n 位来近似， M_0 满足 $2^{-n} M_0 \approx M$ 。这样，所有操作都可以用 int8 和 int32 完成。由于需要知道激活值的分布范围 S_{a_1} ，因此需要少量的数据来做前向推理进行校准。这种量化方式叫做训练后静态量化 (Post-Training Static Quantization)。激活值的分布见图 6.11b。

本次实验的量化是手动实现的，并未用到深度学习框架。代码见附录 8.5.4。实现量化后的综合报告见图 6.12。可以看出，资源使用大幅度降低，使用的 FF 由 32708 个减少到 9076 个，LUT 由 31683 个减少到 12653 个，DSP 由 126 个降低到 47 个，BRAM 由 237 个降低到 88 个。量化后，精度下降为 96.875%。不过，量化后耗时并未降低，这是因为之前在用浮点数计算时已经通过各种方法将流水线间隔优化到了 1 个周期。

6.4.7 性能比较

四种实现的资源消耗对比见表 1，延迟对比见图 6.14。

7 总结

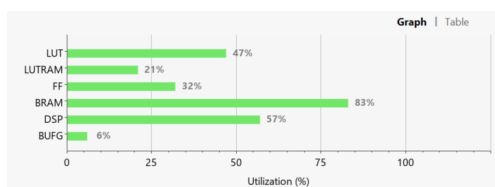
本次实验对一个简单的三层感知机和卷积神经网络 LeNet 进行了硬件加速。对于多层感知机，首先基于 MATLAB 用反向传播算法完成了多层感知机的训练，然后分别用 HDL 和 HLS 实现了硬件加速，其中用到了流水线和并行化方法。HDL 和 HLS 的资源利用率互有高低，较为相近。对于 LeNet，考虑到开发时间有限，使用了 HLS：首先用 pytorch 训练模型，其次用 C++ 做了前向传播的 Baseline 实现（速度较软件约提升 4 倍），然后针对卷积层和全连接层的特点分别做了行缓存和循环交换的优化手段，降低延迟并提升了

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(μs)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
lenet	II Violation	-	138796	1.388E6	-	138797	-	no	88	47	9076	12653	0
conv_relu_unsigned_char_int_signed_char_int_1ul_6ul_32ul_5ul_5ul_28ul_28ul_s	-	-	20407	2.040E5	-	20407	-	no	4	16	2839	3339	0
conv_relu_signed_char_int_signed_char_int_6ul_16ul_14ul_14ul_5ul_10ul_10ul_s	-	-	33393	3.340E5	-	33393	-	no	2	16	2414	2874	0
Loop 1	-	-	1024	1.024E4	1	1	1024	yes	-	-	-	-	-
preprocess_0_preprocess_1	-	-	1025	1.025E4	3	1	1024	yes	-	-	-	-	-
Loop 3	-	-	4704	4.704E4	1	1	4704	yes	-	-	-	-	-
Loop 4	-	-	4704	4.704E4	1	1	4704	yes	-	-	-	-	-
scale0_scale1_scale2	-	-	4707	4.707E4	5	1	4704	yes	-	-	-	-	-
Loop 6	-	-	1176	1.176E4	1	1	1176	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	-	-	1179	1.179E4	5	1	1176	yes	-	-	-	-	-
Loop 8	-	-	1600	1.600E4	1	1	1600	yes	-	-	-	-	-
Loop 9	-	-	1600	1.600E4	1	1	1600	yes	-	-	-	-	-
scale0_scale1_scale2	-	-	1605	1.605E4	7	1	1600	yes	-	-	-	-	-
Loop 11	-	-	400	4.000E3	1	1	400	yes	-	-	-	-	-
maxpooling_0_maxpooling_1_maxpooling_2	-	-	403	4.030E3	5	1	400	yes	-	-	-	-	-
Loop 13	-	-	400	4.000E3	1	1	400	yes	-	-	-	-	-
VITIS_LOOP_237_1_VITIS_LOOP_239_2_VITIS_LOOP_241_3	-	-	402	4.020E3	4	1	400	yes	-	-	-	-	-
Loop 15	-	-	120	1.200E3	1	1	120	yes	-	-	-	-	-
Loop 16	-	-	120	1.200E3	1	1	120	yes	-	-	-	-	-
Loop 17	-	-	120	1.200E3	1	1	120	yes	-	-	-	-	-
fc_0_fc_1	-	-	48006	4.800E5	8	1	48000	yes	-	-	-	-	-
VITIS_LOOP_210_1	-	-	121	1.210E3	3	1	120	yes	-	-	-	-	-
scale0	-	-	121	1.210E3	3	1	120	yes	-	-	-	-	-
Loop 21	-	-	84	840.000	1	1	84	yes	-	-	-	-	-
Loop 22	-	-	84	840.000	1	1	84	yes	-	-	-	-	-
Loop 23	-	-	84	840.000	1	1	84	yes	-	-	-	-	-
fc_0_fc_1	-	-	10084	1.010E5	6	1	10080	yes	-	-	-	-	-
VITIS_LOOP_210_1	-	-	85	850.000	3	1	84	yes	-	-	-	-	-
scale0	-	-	87	870.000	5	1	84	yes	-	-	-	-	-
Loop 27	-	-	10	100.000	1	1	10	yes	-	-	-	-	-
Loop 28	-	-	10	100.000	1	1	10	yes	-	-	-	-	-
fc_0	II Violation	-	840	8.400E3	10	10	84	yes	-	-	-	-	-
VITIS_LOOP_228_1	-	-	10	100.000	2	1	10	yes	-	-	-	-	-
max_idx	-	-	10	100.000	2	1	9	yes	-	-	-	-	-

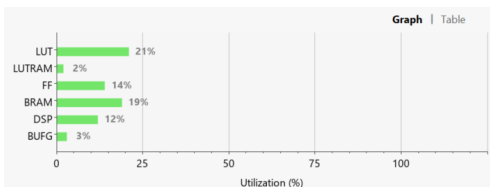
图 6.12: int8 量化后的综合报告

Implementation	FF	LUT	DSP	BRAM
Baseline	15127	12361	12	227
Line Buffer	31686	31286	127	227
Line Buffer & Loop Interchange	32708	31863	126	237
Line Buffer & Loop Interchange & Int8 quantization	9076	12653	47	88

表 1: 资源消耗对比



(a) 量化前



(b) 量化后

图 6.13: 量化前后的资源消耗比例

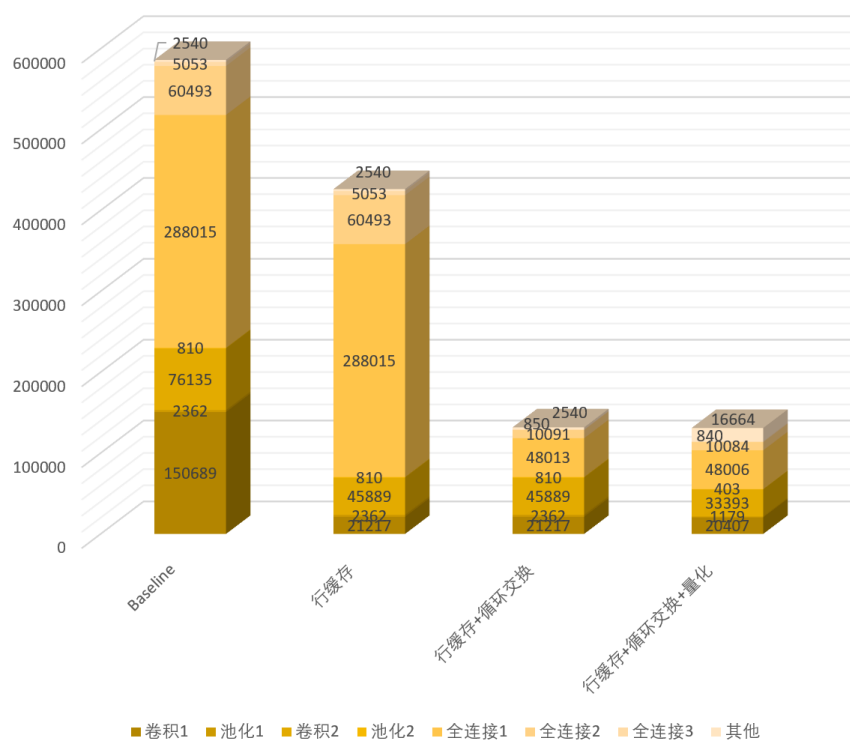


图 6.14: 优化前后延迟对比

吞吐量（速度较软件约提升 17 倍）。此外，还利用了双缓冲技术省去数据拷贝时间；并使用 int8 量化，将资源消耗降低了约 67%，准确度由 97.65% 下降到 96.88%。

本次实验的改进空间如下：

1. 多层感知机的训练集与测试集相同，样本数只有 3。不过，正因为该分类任务较为简单，所以可以作为很好的理解和练习硬件加速的工具。未来可以考虑用 HDL 实现更加复杂的多层感知机。
2. LeNet-5 的硬件加速只使用了 HLS，没有使用 HDL。未来可以用 HDL 实现 LeNet-5，并对比两者性能。
3. 本次实验的量化采用的是对称量化。可以使用更复杂的非对称量化来确保更高的准确率。
4. 本次实验的系统搭建采用了性能较低的 GP 接口。对于实际大量数据的流处理，一般使用 HP 高性能接口，并采用 AXI Stream 协议，从每一张图片第 5 行数据开始传输时就开始 5x5 卷积（而不是等到一张图片传输完成后再进行卷积），进一步提高吞吐量。

8 附录

8.1 多层感知机 MATLAB 代码

8.1.1 前向传播的 MATLAB 实现

代码 8.1: MATLAB 前向传播实现

```

1  % lecture 1
2  % 2022.10.12
3
4  clc; clear;
5
6  load bptrandata.mat
7  Z = [1, 1, 1, -1, 1, -1, 1, 1, 1];
8  V = [1, -1, 1, 1, -1, 1, -1, 1, -1];
9  N = [-1, 1, -1, 1, -1, 1, 1, -1, 1];
10
11 Zlabel = (Z * weight1 + bias1) * weight2' + bias2;
12 Vlabel = (V * weight1 + bias1) * weight2' + bias2;
13 Nlabel = (N * weight1 + bias1) * weight2' + bias2;
14 fprintf("Zlabel = %.2f \n", Zlabel);
15 fprintf("Vlabel = %.2f \n", Vlabel);
16 fprintf("Nlabel = %.2f \n", Nlabel);

```

其中 bptrandata.mat 存储了已经提供好的权重和偏置数据。代码运行输出为：

```
Zlabel = -1.00
Vlabel = 0.00
Nlabel = 1.00
```

8.1.2 反向传播的 MATLAB 实现

代码 8.2: 反向传播的 MATLAB 实现

```
1 % lecture 2
2 % 2022.10.21
3
4 clear; clc;
5
6 %% Hyper-parameters
7 lr1 = 0.005; % learning rate of layer 1
8 lr2 = 0.001; % learning rate of layer 2
9
10 batch_size = 100;
11 max_batch_num = 1000;
12
13 n2 = 3; % number of neurons of layer 2
14 n1 = 9; % number of neurons of layer 1
15
16 loss_threshold = 1e-6; % threshold of loss
17
18 %% Training data
19 y = [-1, 0, 1];
20 x = [1, 1, 1, -1, 1, -1, 1, 1, 1;
21      1, -1, 1, 1, -1, 1, -1, 1, -1;
22      -1, 1, -1, 1, -1, 1, 1, -1, 1]';
23
24 %% Parameter Initialization
25
26 weight1 = single(randn([n2, n1]));
27 bias1 = single(randn([n2, 1]));
28 weight2 = single(randn([1, n2]));
29 bias2 = single(randn(1));
30
31 %% training
32 for i = 1:max_batch_num
33     loss = 0;
34     tic;
35     for j = 1:batch_size
36         % (1) randomly choose a data
37         k = randi(3);
38
39         % (2) forward propagation
40         z1 = weight1 * x(:, k) + bias1; % (n2, n1) * (n1, 1) + (n2, 1) -> (n2, 1)
```

```

41     z2 = weight2 * z1 + bias2; % (1, n2) * (n2, 1) + (1) -> (1)
42     loss = loss + (z2 - y(k)) ^ 2 / (2 * batch_size);
43
44     % (3) backpropagation
45     % (3.1) compute gradients
46     gradient_weight2 = mean(z2 - y(k)) * z1'; % (1) * (1, n1) -> (1, n1)
47     gradient_bias2 = mean(z2 - y(k)); % (1)
48     gradient_z1 = mean(z2 - y(k)) * weight2'; % (1) * (1, n2) -> (1, n2)
49     gradient_weight1 = gradient_z1 * x(:, k)'; % (n2, 1) * (1, n1) -> (n2, n1)
50     gradient_bias1 = gradient_z1; % (n2, 1)
51
52     % (3.2) gradient descent
53     weight2 = weight2 - lr2 * gradient_weight2;
54     bias2 = bias2 - lr2 * gradient_bias2;
55     weight1 = weight1 - lr1 * gradient_weight1;
56     bias1 = bias1 - lr1 * gradient_bias1;
57 end
58 toc;
59
60 % (4) display batch number
61 fprintf("Trained %d batches, loss = %.6f\n", i, loss)
62
63 % (5) judge convergence
64 if (loss < loss_threshold)
65     break;
66 end
67 end
68
69 %% evaluate
70 rst = weight2 * ((weight1 * x) + bias1) + bias2;
71 disp("The result of forward propagation is:");
72 disp(rst);

```

运行结果为:

```

Elapsed time is 0.003915 seconds.
Trained 1 batches, loss = 2.663801
Elapsed time is 0.001138 seconds.
Trained 2 batches, loss = 0.036581
Elapsed time is 0.000769 seconds.
Trained 3 batches, loss = 0.001222
Elapsed time is 0.000650 seconds.
Trained 4 batches, loss = 0.000056
Elapsed time is 0.000851 seconds.
Trained 5 batches, loss = 0.000004
Elapsed time is 0.000621 seconds.
Trained 6 batches, loss = 0.000000
The result of forward propagation is:
-1.0005   -0.0006    1.0000

```

8.2 多层感知机 Verilog 代码

8.2.1 Verilog 设计代码

代码 8.3: 顶层模块 bp_top.v

```

1 module bp_top #(
2     parameter WIDTH = 32
3 )
4     input clk,
5     input rst_n,
6     input start,
7     input [8:0] din,
8     output done,
9     output reg [WIDTH-1:0] dout
10 );
11
12 wire layer1_done;
13
14 wire [WIDTH-1:0] layer1_out_1;
15 wire [WIDTH-1:0] layer1_out_2;
16 wire [WIDTH-1:0] layer1_out_3;
17 layer1 layer1_i (
18     .clk          (clk          ),
19     .rst_n        (rst_n        ),
20     .start        (start        ),
21     .din          (din          ),
22     .done         (layer1_done ),
23     .layer1_out_1 (layer1_out_1),
24     .layer1_out_2 (layer1_out_2),
25     .layer1_out_3 (layer1_out_3)
26 );
27
28 wire layer2_start = layer1_done;
29 wire [WIDTH-1:0] result;
30 layer2 layer2_i (
31     .clk          (clk          ),
32     .rst_n        (rst_n        ),
33     .start        (layer2_start),
34     .layer2_in_1  (layer1_out_1),
35     .layer2_in_2  (layer1_out_2),
36     .layer2_in_3  (layer1_out_3),
37     .result       (result       ),
38     .done         (done         )
39 );
40
41 always @(posedge clk or negedge rst_n) begin
42     if (~rst_n)
43         dout <= {WIDTH{1'b0}};
44     else begin

```

```

45     if (done)
46         dout <= result;
47     else
48         dout <= dout;
49     end
50 end
51
52 endmodule

```

代码 8.4: 第一层网络模块 layer1.v

```

1  module layer1 #(
2      parameter WIDTH = 32
3  )(
4      input          clk          ,
5      input          rst_n        ,
6      input          start        ,
7      input [8:0]    din          ,
8      output         done         ,
9      output [WIDTH-1:0] layer1_out_1 ,
10     output [WIDTH-1:0] layer1_out_2 ,
11     output [WIDTH-1:0] layer1_out_3
12 );
13
14 // (0) detect posedge of `start` to allow for ROM latency
15 reg start_reg;
16 always @(posedge clk or negedge rst_n) begin
17     if (~rst_n)
18         start_reg <= 1'b0;
19     else
20         start_reg <= start;
21 end
22 wire start_posedge = start & ~start_reg;
23
24 // (1) counter
25 reg [3:0] cnt;
26 wire [3:0] cnt_last = 4'd9;
27 always @(posedge clk or negedge rst_n) begin
28     if (~rst_n)
29         cnt <= 4'd0;
30     else begin
31         if (cnt == cnt_last | start_posedge)
32             cnt <= 4'd0;
33         else if (start)
34             cnt <= cnt + 1'b1;
35         else
36             ;
37     end
38 end
39

```

```

40 // (2) rom_out
41 // (2.1) rom enable, address amd output
42 // wire ena = start & (cnt != 4'd9);
43 wire ena = 1'b1;
44 wire [3:0] addr = (~start) ? 4'd0 :
45     (cnt < cnt_last) ? (cnt + 1'b1) :
46     (cnt == cnt_last) ? 4'd0 : 4'd0;
47 wire [WIDTH-1:0] rom_out_1;
48 wire [WIDTH-1:0] rom_out_2;
49 wire [WIDTH-1:0] rom_out_3;
50
51 // (2.2) rom instantiation
52 blk_rom_1 blk_rom_1_i(
53     .addra    (addr),
54     .clka     (clk),
55     .douta    (rom_out_1),
56     .ena      (ena)
57 );
58 blk_rom_2 blk_rom_2_i(
59     .addra    (addr),
60     .clka     (clk),
61     .douta    (rom_out_2),
62     .ena      (ena)
63 );
64 blk_rom_3 blk_rom_3_i(
65     .addra    (addr),
66     .clka     (clk),
67     .douta    (rom_out_3),
68     .ena      (ena)
69 );
70
71 // (3) floating point accumulator
72
73 // (3.1) slave ports
74 wire acc_a_tvalid = start;
75 wire acc_a_tlast = (cnt == cnt_last);
76 wire data_in = ((cnt == 4'd0) & 1'b1) | ((cnt > 4'd0) & din[cnt - 1'b1]);
77 wire opcode = {5'd0, ~data_in};
78
79 // (3.2) master ports
80 wire acc_tvalid_1, acc_tvalid_2, acc_tvalid_3;
81 wire acc_tvalid = acc_tvalid_1 & acc_tvalid_2 & acc_tvalid_3;
82 wire acc_result_tlast_1, acc_result_tlast_2, acc_result_tlast_3;
83 wire acc_result_tlast = acc_result_tlast_1 & acc_result_tlast_2 & acc_result_tlast_3;
84 assign done = acc_tvalid & acc_result_tlast;
85
86 // (3.3) fp_acc instantiation
87 fp_acc fp_acc_i1 (
88     .aclk          (clk),
89     .aresetn       (rst_n),

```

```

90         .s_axis_a_tvalid      (acc_a_tvalid),
91         .s_axis_a_tdata      (rom_out_1),
92         .s_axis_a_tlast      (acc_a_tlast),
93         .s_axis_operation_tvalid (start),
94         .s_axis_operation_tdata (opcode),
95         .m_axis_result_tvalid  (acc_tvalid_1),
96         .m_axis_result_tdata  (layer1_out_1),
97         .m_axis_result_tlast  (acc_result_tlast_1)
98     );
99
100     fp_acc fp_acc_i2 (
101         .aclk                  (clk),
102         .aresetn               (rst_n),
103         .s_axis_a_tvalid      (acc_a_tvalid),
104         .s_axis_a_tdata      (rom_out_2),
105         .s_axis_a_tlast      (acc_a_tlast),
106         .s_axis_operation_tvalid (start),
107         .s_axis_operation_tdata (opcode),
108         .m_axis_result_tvalid  (acc_tvalid_2),
109         .m_axis_result_tdata  (layer1_out_2),
110         .m_axis_result_tlast  (acc_result_tlast_2)
111     );
112
113     fp_acc fp_acc_i3 (
114         .aclk                  (clk),
115         .aresetn               (rst_n),
116         .s_axis_a_tvalid      (acc_a_tvalid),
117         .s_axis_a_tdata      (rom_out_3),
118         .s_axis_a_tlast      (acc_a_tlast),
119         .s_axis_operation_tvalid (start),
120         .s_axis_operation_tdata (opcode),
121         .m_axis_result_tvalid  (acc_tvalid_3),
122         .m_axis_result_tdata  (layer1_out_3),
123         .m_axis_result_tlast  (acc_result_tlast_3)
124     );
125
126     endmodule

```

代码 8.5: 第二层网络模块 layer2.v

```

1  module layer2 # (
2      parameter WEIGHT2_1 = 32'hbd834eb1,
3      parameter WEIGHT2_2 = 32'hbe05a318,
4      parameter WEIGHT2_3 = 32'hbeccdcc3,
5      parameter BIAS2 = 32'hbd1ed05e,
6      parameter WIDTH = 32
7  )
8  (
9      input      clk      ,
10     input      rst_n    ,

```



```

11     input          start          ,
12     input [WIDTH-1:0] layer2_in_1 ,
13     input [WIDTH-1:0] layer2_in_2 ,
14     input [WIDTH-1:0] layer2_in_3 ,
15     output         done           ,
16     output [WIDTH-1:0] result
17 );
18
19 wire mul_result_tvalid_1, mul_result_tvalid_2, mul_result_tvalid_3;
20 wire mul_result_tvalid = mul_result_tvalid_1 & mul_result_tvalid_2 & mul_result_tvalid_3;
21 wire [31:0] mul_result_1, mul_result_2, mul_result_3;
22 fp_mul fp_mul_i1 (
23     .aclk                (clk),
24     .aresetn              (rst_n),
25     .s_axis_a_tvalid      (start),
26     .s_axis_a_tdata       (WEIGHT2_1),
27     .s_axis_b_tvalid      (start),
28     .s_axis_b_tdata       (layer2_in_1),
29     .m_axis_result_tvalid (mul_result_tvalid_1),
30     .m_axis_result_tdata  (mul_result_1)
31 );
32
33 fp_mul fp_mul_i2 (
34     .aclk                (clk),
35     .aresetn              (rst_n),
36     .s_axis_a_tvalid      (start),
37     .s_axis_a_tdata       (WEIGHT2_2),
38     .s_axis_b_tvalid      (start),
39     .s_axis_b_tdata       (layer2_in_2),
40     .m_axis_result_tvalid (mul_result_tvalid_2),
41     .m_axis_result_tdata  (mul_result_2)
42 );
43
44 fp_mul fp_mul_i3 (
45     .aclk                (clk),
46     .aresetn              (rst_n),
47     .s_axis_a_tvalid      (start),
48     .s_axis_a_tdata       (WEIGHT2_3),
49     .s_axis_b_tvalid      (start),
50     .s_axis_b_tdata       (layer2_in_3),
51     .m_axis_result_tvalid (mul_result_tvalid_3),
52     .m_axis_result_tdata  (mul_result_3)
53 );
54
55 wire add_result_tvalid_1, add_result_tvalid_2;
56 wire [WIDTH-1:0] add1_result, add2_result;
57
58 fp_add fp_add_i1 (
59     .aclk                (clk),
60     .aresetn              (rst_n),

```

```

61         .s_axis_a_tvalid      (mul_result_tvalid_1),
62         .s_axis_a_tdata      (mul_result_1),
63         .s_axis_b_tvalid      (mul_result_tvalid_2),
64         .s_axis_b_tdata      (mul_result_2),
65         .m_axis_result_tvalid  (add_result_tvalid_1),
66         .m_axis_result_tdata  (add1_result)
67     );
68
69     fp_add fp_add_i2 (
70         .aclk                  (clk),
71         .aresetn               (rst_n),
72         .s_axis_a_tvalid      (mul_result_tvalid_3),
73         .s_axis_a_tdata      (mul_result_3),
74         .s_axis_b_tvalid      (1'b1),
75         .s_axis_b_tdata      (BIAS2),
76         .m_axis_result_tvalid  (add_result_tvalid_2),
77         .m_axis_result_tdata  (add2_result)
78     );
79
80     fp_add fp_add_i3 (
81         .aclk                  (clk),
82         .aresetn               (rst_n),
83         .s_axis_a_tvalid      (add_result_tvalid_1),
84         .s_axis_a_tdata      (add1_result),
85         .s_axis_b_tvalid      (add_result_tvalid_2),
86         .s_axis_b_tdata      (add2_result),
87         .m_axis_result_tvalid  (done),
88         .m_axis_result_tdata  (result)
89     );
90
91     endmodule

```

8.2.2 Verilog 仿真代码

代码 8.6: Verilog 仿真代码

```

1  `timescale 1ns / 100ps
2
3  module tb_bp_top();
4
5  parameter WIDTH = 32;
6  parameter T = 20;
7
8  reg [8:0] data_z;
9  reg [8:0] data_v;
10 reg [8:0] data_n;
11 reg clk;
12 reg rst_n;
13 reg start;

```

```

14 reg [8:0] din;
15
16 wire [WIDTH-1:0] dout;
17 wire done;
18 reg [1:0] i;
19 reg [2:0] j;
20
21 initial begin
22     data_z <= 9'b111010111;
23     data_v <= 9'b010101101; // from lsb to msb
24     data_n <= 9'b101101010; // from lsb to msb
25     clk <= 1'b1;
26     rst_n <= 1'b0;
27     start <= 1'b0;
28     din <= {WIDTH{1'b0}};
29     #(5 * T)
30     rst_n <= 1'b1;
31     #(5 * T)
32     start <= 1'b1;
33     for (j = 0; j < 5; j = j + 1) begin
34         for (i = 0; i < 3; i = i + 1) begin
35             if (i == 0)
36                 din <= data_z;
37             else if (i == 1)
38                 din <= data_v;
39             else
40                 din <= data_n;
41             #(10 * T);
42         end
43     end
44 end
45
46 reg [3:0] k;
47 initial k = 0;
48 always #(T) begin
49     if (done) begin
50         k = k + 1;
51         if (k == 3 * 5)
52             $stop;
53     end
54 end
55
56 always #(T/2) clk <= ~clk;
57
58 bp_top bp_top_i (
59     .clk      (clk ) ,
60     .rst_n    (rst_n) ,
61     .start    (start) ,
62     .din      (din ) ,
63     .dout     (dout ) ,

```

```

64         .done      (done )
65     );
66
67 endmodule

```

8.3 多层感知机上板验证代码

In [1]:

```

import pynq
from pynq.lib import AxiGPIO
from pynq import Overlay
import time
import struct

overlay = Overlay("./bp_gpio.bit")

```

In [2]:

```

for ip in overlay.ip_dict:
    print(ip)

```

Out [2]:

```

axi_gpio_din
axi_gpio_done
axi_gpio_dout
axi_gpio_start
processing_system7_0

```

In [3]:

```

gpio_done = overlay.axi_gpio_done.channel1
gpio_din = overlay.axi_gpio_din.channel1
gpio_dout = overlay.axi_gpio_dout.channel1
gpio_start = overlay.axi_gpio_start.channel1

```

In [4]:

```

data_z = 0x1d7
data_v = 0xad
data_n = 0x16a
data_in = [data_z, data_v, data_n]

```

In [5]:

```

def int2hex2float(din):
    # convert raw to float
    din = hex(din)[2:]
    din = struct.unpack('!f', bytes.fromhex(din))[0]
    return din

for i in range(3):
    # data in

```

```

In [5]:
    gpio_din.write(data_in[i], 0x1ff)
    # start IP
    gpio_start.write(0x1, 0x1)
    st = time.time()
    while (gpio_done.read() == 0x0):
        pass
    et = time.time()
    # stop IP
    gpio_start.write(0x0, 0x1)
    # get result
    result = gpio_dout.read()
    # display result
    result = int2hex2float(result)
    print("dout = %.6f, elapsed time = %.2f ms" % (result, (et - st) * 1000)
        )

```

```

Out [5]:
    dout = -1.000000, elapsed time = 0.31 ms
    dout = -0.000000, elapsed time = 0.16 ms
    dout = 1.000000, elapsed time = 0.54 ms

```

8.4 多层感知机 HLS 代码

8.4.1 C++ 实现

代码 8.7: bp_top.h

```

1  #ifndef _BP_TOP_H_
2  #define _BP_TOP_H_
3
4  #include <ap_int.h>
5
6  typedef ap_uint<9> DIN;
7  typedef ap_uint<10> DATAIN;
8  typedef float DOUT;
9  typedef float BIAS_WEIGHT;
10
11 const BIAS_WEIGHT bias_weight_1_1[10] = {-0.0071590394, 0.057250433, -0.00063881744,
    0.054453917, -0.060536921, 0.061717860, -0.062706791, -0.0029062936, 0.057055391,
    0.0011056951,};
12 const BIAS_WEIGHT bias_weight_1_2[10] = {-0.014357031, 0.11736761, -0.0062056831,
    0.11763309, -0.12337402, 0.12403638, -0.12338469, -0.0050607617, 0.11447403,
    -0.0033660980,};
13 const BIAS_WEIGHT bias_weight_1_3[10] = {-0.043993123, 0.35420802, -0.013432572,
    0.35347047, -0.38314989, 0.38265964, -0.38031217, -0.014360869, 0.35383368,
    -0.014122883,};
14 const BIAS_WEIGHT weight_2_1 = -0.064114936;
15 const BIAS_WEIGHT weight_2_2 = -0.13050497;

```

```

16 const BIAS_WEIGHT weight_2_3 = -0.40012178;
17 const BIAS_WEIGHT bias_2 = -0.038772933;
18
19 DOUT bp_top(DIN din);
20
21 #endif

```

代码 8.8: bp_top.cpp

```

1  #include "bp_top.h"
2  #include <cstdio>
3
4  DOUT bp_top(DIN din)
5  {
6  #pragma HLS PIPELINE II=10
7  #pragma HLS interface s_axilite port=din
8  #pragma HLS interface s_axilite port=return
9  #pragma HLS BIND_STORAGE variable=bias_weight_1_1 type=rom_1p impl=bram
10 #pragma HLS BIND_STORAGE variable=bias_weight_1_2 type=rom_1p impl=bram
11 #pragma HLS BIND_STORAGE variable=bias_weight_1_3 type=rom_1p impl=bram
12     DATAIN data_in = ((DATAIN) din << 1) + (DATAIN) 1;
13     DOUT acc1 = 0;
14     DOUT acc2 = 0;
15     DOUT acc3 = 0;
16
17     acc: for (int i = 0; i < 10; i++) {
18         acc1 += (data_in & 0x1) ? bias_weight_1_1[i] : -bias_weight_1_1[i];
19         acc2 += (data_in & 0x1) ? bias_weight_1_2[i] : -bias_weight_1_2[i];
20         acc3 += (data_in & 0x1) ? bias_weight_1_3[i] : -bias_weight_1_3[i];
21         data_in = data_in >> 1;
22     }
23
24     DOUT dout = (acc1 * weight_2_1 + acc2 * weight_2_2) + (acc3 * weight_2_3 + bias_2);
25     return dout;
26 }

```

8.4.2 C++ Testbench

代码 8.9: tb_bp_top.cpp

```

1  #include <cstdio>
2  #include "bp_top.h"
3
4  int main()
5  {
6     DIN data_z = 0x1d7;
7     DIN data_v = 0x0ad;
8     DIN data_n = 0x16a;
9

```

```

10     DOUT dout;
11
12     dout = bp_top(data_z);
13     printf("z -> %.6f\n", dout);
14     dout = bp_top(data_v);
15     printf("v -> %.6f\n", dout);
16     dout = bp_top(data_n);
17     printf("n -> %.6f\n", dout);
18
19     return 0;
20 }

```

8.4.3 PYNQ Jupyter Notebook

代码 8.10: 用 PYNQ Jupyter Notebook 下载和调用第 5 节 IP 并展示结果

In [1]:

```

from pynq import Overlay
import numpy as np
import time
import struct

overlay = Overlay("mybp_hls_maxi.bit")
bp_top = overlay.bp_top_0
reg_map = bp_top.register_map
print(reg_map)

```

Out [1]:

```

RegisterMap {
  CTRL = Register(AP_START=0, AP_DONE=0, AP_IDLE=1, AP_READY=0, RESERVED_1
    =0, AUTO_RESTART=0, RESERVED_2=0),
  GIER = Register(Enable=0, RESERVED=0),
  IP_IER = Register(CHANO_INT_EN=0, CHAN1_INT_EN=0, RESERVED=0),
  IP_ISR = Register(CHANO_INT_ST=0, CHAN1_INT_ST=0, RESERVED=0),
  ap_return = Register(ap_return=0),
  din = Register(din=write-only, RESERVED=write-only)
}

```

In [2]:

```

CTRL = reg_map.CTRL.address
dout = reg_map.ap_return.address
din = reg_map.din.address

```

In [3]:

```

data_z = 0x1d7
data_v = 0xad
data_n = 0x16a

```



```

18         transform = transforms.Compose([
19             transforms.Resize((32,32)),
20             transforms.ToTensor(),
21         ],
22         download = True)
23
24
25 test_dataset = torchvision.datasets.MNIST(root = './data',
26     train = False,
27     transform = transforms.Compose([
28         transforms.Resize((32,32)),
29         transforms.ToTensor(),
30     ]),
31     download = True)
32
33 train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
34     batch_size = batch_size,
35     shuffle = False)
36
37
38 test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
39     batch_size = batch_size,
40     shuffle = False)
41
42
43 # define the network
44 class LeNet5(nn.Module):
45     def __init__(self, num_classes):
46         super(LeNet5, self).__init__()
47         self.layer1 = nn.Sequential(
48             nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
49             nn.ReLU(),
50             nn.MaxPool2d(kernel_size = 2, stride = 2))
51         self.layer2 = nn.Sequential(
52             nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
53             nn.ReLU(),
54             nn.MaxPool2d(kernel_size = 2, stride = 2))
55         self.fc = nn.Linear(400, 120)
56         self.relu = nn.ReLU()
57         self.fc1 = nn.Linear(120, 84)
58         self.relu1 = nn.ReLU()
59         self.fc2 = nn.Linear(84, num_classes)
60
61     def forward(self, x):
62         out = self.layer1(x)
63         out = self.layer2(out)
64         out = out.reshape(out.size(0), -1)
65         out = self.fc(out)
66         out = self.relu(out)
67         out = self.fc1(out)

```

```

68         out = self.relu1(out)
69         out = self.fc2(out)
70         return out
71
72     # create the model
73     model = LeNet5(num_classes).to(device)
74
75     # loss function
76     cost = nn.CrossEntropyLoss()
77
78     # optimizer with the model parameters and learning rate
79     optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
80
81     # defined to print how many steps are remaining when training
82     total_step = len(train_loader)
83
84
85     if __name__ == "__main__":
86         # Start training
87         print("Start Training ...")
88
89         for epoch in range(num_epochs):
90             for i, (images, labels) in enumerate(train_loader):
91                 images = images.to(device)
92                 labels = labels.to(device)
93
94                 # forward pass
95                 outputs = model(images)
96                 loss = cost(outputs, labels)
97
98                 # backward and optimize
99                 optimizer.zero_grad()
100                 loss.backward()
101                 optimizer.step()
102
103                 if (i+1) % 400 == 0:
104                     print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
105                             .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
106
107         # save the model
108         torch.save(model, "model.pth")
109
110         # test the model
111         with torch.no_grad():
112             correct = 0
113             total = 0
114             for images, labels in test_loader:
115                 images = images.to(device)
116                 labels = labels.to(device)
117                 outputs = model(images)

```

```

118         _, predicted = torch.max(outputs.data, 1)
119         total += labels.size(0)
120         correct += (predicted == labels).sum().item()
121
122     print('Accuracy of the network on the 10000 test images: {} %'.format(100 * correct
        / total))

```

8.5.2 前向传播的 C++ Baseline 实现

代码 8.12: 卷积等算子的 C++ 模板实现 op.h

```

1  #ifndef _OP_H_
2  #define _OP_H_
3
4  #include <cstdint>
5  #include <cstdlib>
6
7  namespace mylenet { // to avoid INPUT macro mixing with windows.h
8      typedef uint8_t INPUT;
9      typedef float WEIGHT;
10     typedef float BIAS;
11     typedef float DATA;
12 } // namespace mylenet
13
14 template <typename I, typename D, size_t H, size_t W>
15 void preprocess(const I (&input)[H][W], D (&output)[1][H][W]) {
16     preprocess:
17         for (size_t i = 0; i < H; i++) {
18             for (size_t j = 0; j < W; j++) {
19                 output[0][i][j] = input[i][j] / 255.;
20             }
21         }
22 }
23
24 template <typename T>
25 T relu(T x) {
26     if (x > 0)
27         return x;
28     else
29         return 0;
30 }
31
32 template <typename D, typename W, size_t KH, size_t KW>
33 D _conv_kernal(const D (&input)[KH][KW], const W (&kernal)[KH][KW]) {
34     D tmp = 0;
35     for (size_t i = 0; i < KH; i++) {
36         for (size_t j = 0; j < KW; j++) {
37             tmp += input[i][j] * kernal[i][j];
38         }

```

```

39     }
40     return tmp;
41 }
42
43 template <typename D, typename W, typename B, size_t IH, size_t IW, size_t KH, size_t KW,
44           size_t OH, size_t OW>
45 void _conv2d(
46     const D (&input)[IH][IW],
47     const W (&weights)[KH][KW],
48     D (&output)[OH][OW])
49 {
50     D convoluted[KH][KW];
51     conv_relu_0:
52     for (int i = 0; i < OH; i++) {
53         conv_relu_1:
54         for (int j = 0; j < OW; j++) {
55             conv_relu_2:
56             for (int k = 0; k < KH; k++) {
57                 conv_relu_3:
58                 for (int l = 0; l < KW; l++) {
59                     convoluted[k][l] = input[i + k][j + l];
60                 }
61             }
62             output[i][j] = relu(_conv_kernal<D, W, B, KH, KW>(convoluted, weights));
63         }
64     }
65
66     template <typename D, typename W, typename B, size_t IC, size_t OC, size_t IH,
67             size_t IW, size_t KH, size_t KW, size_t OH, size_t OW>
68     void conv_relu(const D (&input)[IC][IH][IW], const W (&weights)[OC][IC][KH][KW],
69                  const B bias[OC], D (&output)[OC][OH][OW]) {
70         D _conv_out[OH][OW];
71         D _conv_acc[OH][OW];
72
73     conv_oc:
74         for (size_t i = 0; i < OC; i++) {
75             conv_initialize_acc_0:
76             for (size_t k = 0; k < OH; k++) {
77                 conv_initialize_acc_1:
78                 for (size_t l = 0; l < OW; l++) {
79                     _conv_acc[k][l] = 0;
80                 }
81             }
82             conv_acc_0:
83             for (size_t j = 0; j < IC; j++) {
84                 _conv2d<D, W, B, IH, IW, KH, KW, OH, OW>(input[j], weights[i][j],
85                                                            _conv_out);
86             }
87             conv_acc_1:
88             for (size_t k = 0; k < OH; k++) {

```

```

88         conv_acc_2:
89             for (size_t l = 0; l < OW; l++) {
90                 _conv_acc[k][l] += _conv_out[k][l];
91             }
92         }
93     }
94     conv_bias_0:
95     for (size_t k = 0; k < OH; k++) {
96         conv_bias_1:
97         for (size_t l = 0; l < OW; l++) {
98             output[i][k][l] = _conv_acc[k][l] + bias[i];
99         }
100     }
101 }
102 }
103
104 template <typename D, size_t C, size_t IH, size_t IW, size_t OH, size_t OW>
105 void maxpooling(const D (&input)[C][IH][IW], size_t pool_size, size_t stride,
106                D (&output)[C][OH][OW]) {
107     D tmp, ele;
108     maxpooling_0:
109     for (size_t c = 0; c < C; c++) {
110         maxpooling_1:
111         for (size_t i = 0; i < OH; i++) {
112             maxpooling_2:
113             for (size_t j = 0; j < OW; j++) {
114                 tmp = 0;
115                 maxpooling_3:
116                 for (size_t k = 0; k < pool_size; k++) {
117                     maxpooling_4:
118                     for (size_t l = 0; l < pool_size; l++) {
119                         ele = input[c][i * stride + k][j * stride + l];
120                         tmp = (ele > tmp) ? ele : tmp;
121                     }
122                 }
123                 output[c][i][j] = tmp;
124             }
125         }
126     }
127 }
128
129 template <typename D, typename W, typename B, size_t IL, size_t OL>
130 void fc_relu(D (&input)[IL], const W (&weights)[OL][IL], const B (&bias)[OL],
131             D (&output)[OL]) {
132     D tmp = 0;
133     fc_0:
134     for (size_t nout = 0; nout < OL; nout++) {
135         tmp = 0;
136         fc_1:
137         for (size_t nin = 0; nin < IL; nin++) {

```

```

138         tmp += weights[nout][nin] * input[nin];
139     }
140     output[nout] = relu<D>(tmp + bias[nout]);
141 }
142 }
143
144 template <typename D, typename W, typename B, size_t IL, size_t OL>
145 void fc(D (&input)[IL], const W (&weights)[OL][IL], const B (&bias)[OL],
146         D (&output)[OL]) {
147     D tmp = 0;
148 fc_0:
149     for (size_t nout = 0; nout < OL; nout++) {
150         tmp = 0;
151         fc_1:
152         for (size_t nin = 0; nin < IL; nin++) {
153             tmp += weights[nout][nin] * input[nin];
154         }
155         output[nout] = tmp + bias[nout];
156     }
157 }
158
159 template <typename D, size_t C, size_t H, size_t W, size_t L>
160 void flatten(D (&din)[C][H][W], D (&dout)[L]) {
161     size_t l = 0;
162     for (size_t c = 0; c < C; c++) {
163         for (size_t h = 0; h < H; h++) {
164             for (size_t w = 0; w < W; w++) {
165                 dout[l++] = din[c][h][w];
166             }
167         }
168     }
169 }
170
171 template <typename D, size_t L>
172 size_t max_idx(const D (&rst)[L]) {
173     D tmp = rst[0];
174     size_t idx = 0;
175 max_idx:
176     for (size_t i = 1; i < L; i++) {
177         if (rst[i] > tmp) {
178             tmp = rst[i];
179             idx = i;
180         }
181     }
182     return idx;
183 }
184
185 #endif

```

代码 8.13: 网络结构 net.cpp

```

1  #include "op.h"
2  #include "param.h"
3
4  using namespace mylenet;
5
6  int LeNet(INPUT (&img)[32][32]) {
7      // preprocess
8      DATA img_preprocessed[1][32][32] = {0};
9      preprocess(img, img_preprocessed);
10
11     // compute
12     DATA conv1_relu_rst[6][28][28];
13     conv_relu(img_preprocessed, conv1_weights, conv1_bias, conv1_relu_rst);
14
15     // (2) pooling1
16     DATA pool1_rst[6][14][14];
17     maxpooling(conv1_relu_rst, 2, 2, pool1_rst);
18
19     // (3) conv2 and relu
20     DATA conv2_relu_rst[16][10][10];
21     conv_relu(pool1_rst, conv2_weights, conv2_bias, conv2_relu_rst);
22
23     // (4) pool2
24     DATA pool2_rst[16][5][5];
25     maxpooling(conv2_relu_rst, 2, 2, pool2_rst);
26
27     // (5) flatten
28     DATA flattened[400];
29     flatten(pool2_rst, flattened);
30
31     // (6) fc1
32     DATA fc1_relu_rst[120];
33     fc_relu(flattened, fc1_weights, fc1_bias, fc1_relu_rst);
34
35     // (7) fc2
36     DATA fc2_relu_rst[84];
37     fc_relu(fc1_relu_rst, fc2_weights, fc2_bias, fc2_relu_rst);
38
39     // (8) fc3
40     DATA final_rst[10];
41     fc(fc2_relu_rst, fc3_weights, fc3_bias, final_rst);
42
43     return max_idx(final_rst);
44 }

```

代码 8.14: 测试文件 tb_net.cpp

```

1  #include "tbutils.h"
2  #include "op.h"
3  #include "net.h"

```

```
4 #include <string>
5
6 #ifdef _WIN32
7 #include <windows.h>
8 DWORD st, et;
9 #endif
10
11 #ifdef __linux__
12 #include <sys/time.h>
13 timeval st, et;
14 #endif
15
16 int main()
17 {
18     float _img[32][32] = {0};
19     mylenet::INPUT img[32][32] = {0};
20     int rst;
21     std::string str = "./imgs/img_";
22     std::string img_path = "./imgs/img_";
23
24     #ifdef _WIN32
25     DWORD elapsed_time = 0;
26     #endif
27     #ifdef __linux__
28     double elapsed_time = 0;
29     #endif
30
31     int rst_vec[50];
32
33     for (int i = 0; i < 50; i++) {
34         img_path = str + std::to_string(i) + ".txt";
35         txt2arr(img_path, _img);
36         copy_arr(img, _img);
37
38         #ifdef _WIN32
39         st = GetTickCount();
40         #endif
41         #ifdef __linux__
42         gettimeofday(&st, NULL);
43         #endif
44
45         rst = lenet(img);
46         rst_vec[i] = rst;
47
48         #ifdef _WIN32
49         et = GetTickCount();
50         elapsed_time += et - st;
51         #endif
52         #ifdef __linux__
53         gettimeofday(&et, NULL);
```



```

54     elapsed_time += (et.tv_sec - st.tv_sec) * 1000 + (et.tv_usec - st.tv_usec) / 1000;
55     #endif
56
57     std::cout << rst << " ";
58 }
59 std::cout << std::endl;
60
61 std::cout << "\nPrediction of 50 pictures takes time "<< elapsed_time << " ms\n";
62
63 int gold[50] = {7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5,
64               4, 0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4, 4};
65 int coincide = 0;
66 for (int i = 0; i < 50; i++) {
67     if (rst_vec[i] == gold[i]) {
68         coincide++;
69     }
70 }
71
72 std::cout << "Accuracy = " << coincide / 50. * 100. << "%" << std::endl;
73
74 return 0;
75 }

```

代码 8.15: 测试文件用到的函数的模板实现 tb_utils.h

```

1  #ifndef _TB_UTILS_H_
2  #define _TB_UTILS_H_
3
4  #include "op.h"
5  #include <iostream>
6  #include <fstream>
7  #include <string>
8
9  template<typename T, size_t M, size_t N>
10 void disp_arr(T (&arr)[M][N])
11 {
12     for (size_t i = 0; i < M; i++) {
13         for (size_t j = 0; j < N; j++) {
14             std::cout << arr[i][j] << ' ';
15         }
16         std::cout << std::endl;
17     }
18 }
19
20 template<typename T, size_t M, size_t N>
21 void txt2arr(std::string filename, T (&arr)[M][N])
22 {
23     std::ifstream infile;
24     infile.open(filename);
25     for (size_t i = 0; i < M; i++) {

```

```

26     for (size_t j = 0; j < N; j++) {
27         infile >> arr[i][j];
28     }
29 }
30 infile.close();
31 }
32
33 template<typename D, typename S, size_t M, size_t N>
34 void copy_arr(D (&dst)[M][N], const S (&src)[M][N])
35 {
36     for (size_t i = 0; i < M; i++) {
37         for (size_t j = 0; j < N; j++) {
38             dst[i][j] = src[i][j];
39         }
40     }
41 }
42 #endif

```

8.5.3 HLS 优化

代码 8.16: 用 Line/Window Buffer 优化卷积运算

```

1  template <typename D, typename W, typename B, size_t IH, size_t IW, size_t KH,
2      size_t KW, size_t OH, size_t OW>
3  void _conv2d(const D (&input)[IH][IW], const W (&weights)[KH][KW],
4      D (&output)[OH][OW])
5  {
6      static D window_buffer[KH][KW] = {0};
7      static D line_buffer[KH][IW] = {0};
8
9      conv_initialize_line_buffer_0:
10     for (size_t i = 0; i < KH - 1; i++) {
11         conv_initialize_line_buffer_1:
12         for (size_t j = 0; j < IW; j++) {
13             line_buffer[i + 1][j] = input[i][j];
14         }
15     }
16
17     conv_0:
18     for (size_t i = KH - 1; i < IH; i++) {
19         conv_1:
20         for (size_t j = 0; j < IW; j++) {
21             conv_shift_window_buffer_0:
22             for (size_t k = 0; k < KH; k++) {
23                 conv_shift_line_buffer:
24                 if (k == KH - 1) {
25                     line_buffer[k][j] = input[i][j];
26                 } else {
27                     line_buffer[k][j] = line_buffer[k + 1][j];

```

```

28         }
29         conv_shift_window_buffer_1:
30         for (size_t l = 0; l < KW - 1; l++) {
31             window_buffer[k][l] = window_buffer[k][l + 1];
32         }
33         window_buffer[k][KW - 1] = line_buffer[k][j];
34     }
35     if (j >= KW - 1) {
36         output[i - KH + 1][j - KW + 1] =
37             _conv_kernal<D, W, B, KH, KW>(window_buffer, weights);
38     }
39 }
40 }
41 }

```

代码 8.17: 用循环交换优化全连接层流水线

```

1 template <typename D, typename W, typename B, size_t IL, size_t OL>
2 void fc(D (&input)[IL], const W (&weights)[OL][IL], const B (&bias)[OL], D (&output)[OL])
3 {
4     D out_tmp[OL] = {0};
5     fc_0:
6     for (size_t nin = 0; nin < IL; nin++) {
7         fc_1:
8         for (size_t nout = 0; nout < OL; nout++) {
9             out_tmp[nout] += weights[nout][nin] * input[nin];
10        }
11    }
12    for (size_t nout = 0; nout < OL; nout++) {
13        output[nout] = out_tmp[nout] + bias[nout];
14    }
15 }

```

8.5.4 int8 量化代码

代码 8.18: int8 量化

```

1 import torch
2
3 # load model
4 model = torch.load("model.pth")
5
6 # load params
7 conv1_weights = model.layer1[0].weight.data # torch.Size([6, 1, 5, 5])
8 conv1_bias    = model.layer1[0].bias.data   # torch.Size([6])
9 conv2_weights = model.layer2[0].weight.data # torch.Size([16, 6, 5, 5])
10 conv2_bias    = model.layer2[0].bias.data   # torch.Size([16])
11 fc1_weights   = model.fc.weight.data       # torch.Size([120, 400])
12 fc2_weights   = model.fc1.weight.data      # torch.Size([84, 120])

```

```

13 fc3_weights      = model.fc2.weight.data # torch.Size([10, 84])
14 fc1_bias         = model.fc.bias.data # torch.Size([120])
15 fc2_bias         = model.fc1.bias.data # torch.Size([84])
16 fc3_bias         = model.fc2.bias.data # torch.Size([10])
17
18 # quant
19 import numpy as np
20
21 # quantization
22 s_x = 1 / 255
23
24 # manually set activation range by looking at the histogram
25 conv1_rst_max = 2
26 conv2_rst_max = 6
27 fc1_rst_max = 15
28 fc2_rst_max = 15
29
30 # manually set weight range by looking at the histogram
31 conv1_weights_max = 0.5
32 conv2_weights_max = 0.45
33 fc1_weights_max = 0.4
34 fc2_weights_max = 0.3
35 fc3_weights_max = 0.4
36 # or set weight range by maximum, subject to outliers
37 # conv1_weights_max = torch.max(torch.abs(conv1_weights))
38 # conv2_weights_max = torch.max(torch.abs(conv2_weights))
39 # fc1_weights_max = torch.max(torch.abs(fc1_weights))
40 # fc2_weights_max = torch.max(torch.abs(fc2_weights))
41 # fc3_weights_max = torch.max(torch.abs(fc3_weights))
42
43 # conv1
44 s_conv1_weights = conv1_weights_max / 127
45 s_conv1_bias = s_conv1_weights * s_x
46
47 conv1_weights_q = torch.round(conv1_weights / s_conv1_weights).to(torch.int8)
48 conv1_bias_q = torch.round(conv1_bias / s_conv1_bias).to(torch.int8)
49
50 # conv2
51 s_conv2_weights = conv2_weights_max / 127
52 s_conv1_a = conv1_rst_max / 127
53 s_conv2_bias = s_conv2_weights * s_conv1_a
54
55 conv2_weights_q = torch.round(conv2_weights / s_conv2_weights).to(torch.int8)
56 conv2_bias_q = torch.round(conv2_bias / s_conv2_bias).to(torch.int8)
57
58 # fc1
59 s_fc1_weights = fc1_weights_max / 127
60 s_conv2_a = conv2_rst_max / 127
61 s_fc1_bias = s_fc1_weights * s_conv2_a
62

```

```

63 fc1_weights_q = torch.round(fc1_weights / s_fc1_weights).to(torch.int8)
64 fc1_bias_q = torch.round(fc1_bias / s_fc1_bias).to(torch.int8)
65
66 # fc2
67 s_fc2_weights = fc2_weights_max / 127
68 s_fc1_a = fc1_rst_max / 127
69 s_fc2_bias = s_fc2_weights * s_fc1_a
70
71 fc2_weights_q = torch.round(fc2_weights / s_fc2_weights).to(torch.int8)
72 fc2_bias_q = torch.round(fc2_bias / s_fc2_bias).to(torch.int8)
73
74 # fc3
75 s_fc3_weights = fc3_weights_max / 127
76 s_fc2_a = fc2_rst_max / 127
77 s_fc3_bias = s_fc3_weights * s_fc2_a
78
79 fc3_weights_q = torch.round(fc3_weights / s_fc3_weights).to(torch.int8)
80 fc3_bias_q = torch.round(fc3_bias / s_fc3_bias).to(torch.int8)
81
82 m_conv1 = s_conv1_weights * s_x / s_conv1_a
83 m_conv2 = s_conv2_weights * s_conv1_a / s_conv2_a
84 m_fc1 = s_fc1_weights * s_conv2_a / s_fc1_a
85 m_fc2 = s_fc2_weights * s_fc1_a / s_fc2_a
86
87 def multiply(N, M, P):
88     for n in range(1, N):
89         result = M * P
90         Mo = round(2 ** n * M)
91         approx_result = (Mo * P) >> n
92         error = result - approx_result
93         if (error < 1 and error > -1):
94             print("n = %d, Mo = %d, approx = %f, error = %f" %
95                   (n, Mo, approx_result, error))
96             return n, Mo
97
98 p_conv1 = round(conv1_rst_max / (s_conv1_weights * s_x))
99 p_conv2 = round(conv2_rst_max / (s_conv2_weights * s_conv1_a))
100 p_fc1 = round(fc1_rst_max / (s_fc1_weights * s_conv2_a))
101 p_fc2 = round(fc2_rst_max / (s_fc2_weights * s_fc1_a))
102
103 N = 31
104 n_conv1, m0_conv1 = multiply(N, m_conv1, p_conv1)
105 n_conv2, m0_conv2 = multiply(N, m_conv2, p_conv2)
106 n_fc1, m0_fc1 = multiply(N, m_fc1, p_fc1)
107 n_fc2, m0_fc2 = multiply(N, m_fc2, p_fc2)
108
109 tensor2txt("./param_quant/conv1_weights", conv1_weights)
110 tensor2txt("./param_quant/conv1_bias", conv1_bias)
111 tensor2txt("./param_quant/conv2_weights", conv2_weights)
112 tensor2txt("./param_quant/conv2_bias", conv2_bias)

```

```

113 tensor2txt("./param_quant/fc1_weights", fc1_weights )
114 tensor2txt("./param_quant/fc2_weights", fc2_weights )
115 tensor2txt("./param_quant/fc3_weights", fc3_weights )
116 tensor2txt("./param_quant/fc1_bias", fc1_bias )
117 tensor2txt("./param_quant/fc2_bias", fc2_bias )
118 tensor2txt("./param_quant/fc3_bias", fc3_bias )

```

8.5.5 PYNQ 调用 IP 核

In [1]:

```

from pynq import Overlay
from pynq import allocate
from pynq import DefaultIP
import cv2
import time
import numpy as np
import struct
from array import array
from os.path import join

# load raw mnist data
class MnistDataLoader(object):
    def __init__(self, training_images_filepath, training_labels_filepath,
                 test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath, labels_filepath):
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected 2049, got {}'.format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected 2051, got {}'.format(magic))
            image_data = array("B", file.read())
            images = []
            for i in range(size):
                images.append([0] * rows * cols)
            for i in range(size):
                img = np.array(image_data[i * rows * cols:(i + 1) * rows * cols]

```

In [1]:

```

    ])
    img = img.reshape(28, 28)
    images[i][:] = img

    return images, labels

def load_data(self):
    x_train, y_train = self.read_images_labels(self.
        training_images_filepath, self.training_labels_filepath)
    x_test, y_test = self.read_images_labels(self.test_images_filepath,
        self.test_labels_filepath)
    return (x_train, y_train), (x_test, y_test)

```

In [2]:

```

input_path = './'
training_images_filepath = join(input_path, 'train-images-idx3-ubyte/train-
    images.idx3-ubyte')
training_labels_filepath = join(input_path, 'train-labels-idx1-ubyte/train-
    labels.idx1-ubyte')
test_images_filepath = join(input_path, 't10k-images-idx3-ubyte/t10k-images.
    idx3-ubyte')
test_labels_filepath = join(input_path, 't10k-labels-idx1-ubyte/t10k-labels.
    idx1-ubyte')

```

In [3]:

```

mnist_dataloader = MnistDataloader(training_images_filepath,
    training_labels_filepath, test_images_filepath, test_labels_filepath)
(x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()

# upsample to 32x32
images_50 = []
for img in x_test[0:50]:
    img = np.array(img, dtype='u1')
    images_50.append(cv2.resize(img, dsize=(32, 32), interpolation=cv2.
        INTER_LINEAR_EXACT))

```

In [4]:

```

class LeNetDriver(DefaultIP):
    # Driver for LeNet IP
    bindto = ["xilinx.com:hls:lenet:1.0"]
    def __init__(self, description):
        super().__init__(description=description)
        self.img_in = 0x18
        self.ap_ctrl = 0x00
        self.ap_return = 0x10
        self.img_size = 32
        self.img_in_buf_1 = allocate(shape=(self.img_size, self.img_size),
            dtype='u1')
        self.img_in_buf_2 = allocate(shape=(self.img_size, self.img_size),

```

In [4]:

```

dtype='u1')

def predict(self, img_in):
    self.write(self.img_in, self.img_in_buf_1.physical_address)
    np.copyto(self.img_in_buf_1, np.uint8(img_in))

    self.write(self.ap_ctrl, 0x01)
    while self.read(self.ap_ctrl) == 0x01:
        pass

    return self.read(self.ap_return)

# double buffer implementation for batch prediction
def batch_predict(self, img_in):
    flag = False
    output = np.zeros(len(img_in), dtype=np.int)

    self.write(self.img_in, self.img_in_buf_1.physical_address)
    np.copyto(self.img_in_buf_1, np.uint8(img_in[0]))

    for i in range(1, len(img_in)):
        self.write(self.ap_ctrl, 0x01)
        current_buf = self.img_in_buf_1 if flag else self.img_in_buf_2
        self.write(self.img_in, current_buf.physical_address)
        np.copyto(current_buf, np.uint8(img_in[i]))
        while self.read(self.ap_ctrl) == 0x01:
            pass
        output[i-1] = self.read(self.ap_return)
        flag = not flag

    self.write(self.ap_ctrl, 0x01)
    while self.read(self.ap_ctrl) == 0x01:
        pass
    output[len(img_in)-1] = self.read(self.ap_return)

    return output

```

In [5]:

```

overlay = Overlay("lenet_hls_lb_fc_10ns.bit")

lenet = overlay.lenet_0

# double buffer implementation
st = time.time()
output = lenet.batch_predict(images_50)
et = time.time()
print(output)
print("Elapsed time: %.2f ms" % ((et - st) * 1000))

```


Out [5]:

```
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4]
Elapsed time: 122.90 ms
```

In [6]:

```
output = []
elapsed = 0
# without double buffer
for img in images_50:
    st = time.time()
    rst = lenet.predict(img)
    et = time.time()
    elapsed = elapsed + (et - st) * 1000
    output.append(rst)

print(output)
print("Elapsed time: %.2f ms" % elapsed)

gold = [7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6,
        5, 4, 0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1,
        2, 4, 4]

coincide = 0
for i in range(50):
    if (output[i] == gold[i]):
        coincide += 1
accuracy = coincide / 50
print("accuracy = %.2f %" % (accuracy * 100))
```

Out [6]:

```
[7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4,
 0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4,
 4]
Elapsed time: 129.78 ms
accuracy = 100.00 %
```

In [7]:

```
# quantized
overlay = Overlay("mylenet_fixed.bit")

lenet = overlay.lenet_0

st = time.time()
output = lenet.batch_predict(images_50)
et = time.time()
print(output)
print("Elapsed time: %.2f ms" % ((et - st) * 1000))
```

```
In [7]: coincide = 0
        for i in range(50):
            if (output[i] == gold[i]):
                coincide += 1
        accuracy = coincide / 50
        print("accuracy = %.2f %" % (accuracy * 100))
```

```
Out [7]: [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
          1 2 8 1 7 4 2 3 5 1 2 9 4]
Elapsed time: 119.42 ms
accuracy = 96.00 %
```