哈尔滨工业大学（深圳）

Harbin Institute of Technology (shenzhen)

课程设计报告

学年学期	2023春
课程名称	嵌入式计算
设计题目	一种简单的工业视觉计算加速方案
学生学号	200110619
学生姓名	梁鑫嵘
学院	计算机科学与技术
年级专业	计算机科学与技术
任课老师	张春慨

评语:

报告得分:	代码得分:	总分:
	评阅老师签字:	年 月

说明

1. 本设计报告为程序设计及应用课程期末课程项目一部分。
2. 程序设计及应用课程期末总分 100 分,其中设计报告 60 分,代码 40 分。
3. 本设计报告中至少应包含项目总体功能设计、各模块的详细说明和项目特点（优点）,也可以适当添加其他相关内容。
4. 对于关键代码,可以粘贴到设计报告相应位置中,但请务必保证设计报告的工整。
5. 设计报告字数不得少于 2500 字。
6. 完整代码、数据文件及可执行文件请用单独的压缩文件进行提交。

目录

1 概述

本项目基于 FPGA 和嵌入式软核或硬核处理器，实现一种简单的工业视觉计算加速方案，主要通过新型硬件描述语言设计硬件电路，计划将大部分模型计算过程直接在硬件上实现，以提高计算速度、降低功耗和延迟，从而在嵌入式系统中实现工业视觉算法的计算加速。

本项目其实是本人在学习一门新的硬件描述语言的过程中的一个练习项目，其中有许多不足和尚未实现之处，请老师谅解。

关键词：FPGA、IP 核、工业视觉、计算加速、Bluespec

2 项目的背景和意义

在这个 AI 技术飞速发展的时代，工业视觉技术也在不断发展，工业视觉技术的发展对于工业生产的自动化、智能化、信息化有着重要的意义。
而一个能够运行工业视觉算法的嵌入式系统，可以在工业生产中发挥重要的作用，比如在工业生产中，可以通过工业视觉技术对产品进行检测，从而提高产品的质量，减少人工检测的成本，提高生产效率。

但是，工业视觉技术的发展也面临着一些问题，比如工业视觉技术的计算量大，计算速度慢、对系统性能要求较高，这些问题都会影响工业视觉技术的发展，所以，如何提高工业视觉技术的计算速度，是一个值得研究的问题。

以市场上常用的工业视觉设备为例，要实现安全帽检测、工作服检测、工作人员路径跟踪、工作人员是否越界等，一般使用神经网络算法进行目标分类、目标检测、目标跟踪等，这些算法的计算量大，计算速度慢，对系统性能要求较高；同时一些工业视觉的应用场景也对计算实时性要求较高，例如 SMT 贴片机的贴片过程中，需要对贴片机的贴片过程进行实时检测，记录并判断拜访位置和精度是否符合要求，以保证贴片的质量，这就对工业视觉技术的计算速度和实时性提出了更高的要求。

在这些应用场景中，许多只需要比较小比较简单的网络就可以完成，如安全帽工作服监测等，这些情况下可以使用更低的计算资源来完成，但是，市场上的工业视觉设备一般都是使用通用的处理器来完成，造成了计算资源的浪费。同时，对于贴片机等大型快速工控设备的质量检测这样的场景，又需要快速的相应时间和可控的延迟时间，使用通用处理器的方式很难满足这些要求。

为了解决这些实际问题，本项目探索了利用 FPGA 和嵌入式软核或硬核处理器，实现一种简单的工业视觉计算加速方案，主要通过新型硬件描述语言设计硬件电路，计划将大部分模型计算过程直接在硬件上实现，以提高计算速度、降低功耗和延迟，从而在嵌入式系统中实现工业视觉算法的计算加速。

3 项目的功能介绍

在设计中，本项目主要分软件、硬件两部分，其中软件部分主要是在嵌入式系统中运行的程序，硬件部分主要是在 FPGA 中运行的程序。

软件部分负责将外部数据输入到神经网络计算推理 IP 中，然后等待计算完成并得到返回的结果，最后将结果输出到外部。
硬件部分负责接收软件部分传入的数据，通过硬件电路计算，然后将计算结果返回给软件部分。

由于本项目是一个练习项目，且本人手上暂且没有 FPGA 和工业相机等实验设备，所以本项目中的硬件部分只实现了一个简单的神经网络计算推理 IP，且暂未实现计划中的软件部分，只对软件部分的模型框架进行了设计。在软件部分的说明中，可以给出对应的伪代码。

本项目当前主要工作重点和创新创造主要在硬件部分，通过新型 HDL（Hardware Description Language，硬件描述语言）的高级特性快速地描述一个神经网络计算推理 IP，并且进行了快速仿真验证。

4 项目的设计细节

在项目实现和验证过程中，主要使用了 MNIST 这一数据集。一是此数据集对于神经网络的计算量较小，可以在较短的时间内完成仿真验证；二是此数据集的数据较为简单，可以快速地进行仿真验证。完成 MNIST 数据集的仿真验证后，可以很方便地变换网络结构和数据集，进行更复杂的仿真验证。

MNIST 数据集包含 60000 个训练样本和 10000 个测试样本，每个样本都是一个 28*28 的灰度图像，图像中的每个像素点的灰度值在 0-255 之间，代表了图像中的颜色深浅。每个样本都有一个标签，标签的值在 0-9 之间，代表了图像中的数字。

4.1 神经网络结构设计

4.1.1 全连接网络设计

在项目实现中，首先进行了简单的网络设计。为了尽量降低开发和验证过程中的复杂度，本项目的神经网络结构设计尽量简单。

在网络结构设计中，首先设计了一个简单的全连接神经网络，该网络包含一个输入层、一个隐藏层和一个输出层，其中输入层包含 784 个神经元，隐藏层包含 32 个神经元，输出层包含 10 个神经元。在网络结构设计中，为了简化网络结构，本项目中的神经元都是使用相同的激活函数，即 ReLU 函数。不过为了让逻辑更加简单，ReLU 激活函数是可选的。

在 PyTorch 中，此全连接神经网络代码如下：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from config import *

class FcNet(nn.Module):
  def __init__(self) -> None:
    super().__init__()
    self.name = "fc"
    self.fc1 = nn.Linear(784, 32)
    self.fc2 = nn.Linear(32, 10)

  def forward(self, x):
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    # x = F.relu(x)
    x = self.fc2(x)
    output = F.log_softmax(x, dim=1)
    return output
```

为其添加了一个名为 `name` 的属性，用于之后快速生成并导出模型权重等数据到对应文件中。

通过 `torchsummary` 工具，可以得到此网络的结构信息：

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
```

```
================================================================
            Linear-1                    [-1, 32]          25,120
            Linear-2                    [-1, 10]             330
================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.10
Estimated Total Size (MB): 0.10
----------------------------------------------------------------
```

此网络的总参数量为 25450，其中第一层的参数量为 25120，第二层的参数量为 330，总大小只有 0.10MiB。

运行代码中的 `run.py`，对此网络进行训练，训练过程和结果如下：

```
(cumcm) ➜ chiro@chiro-pc  ~/programs/bsv-cnn/model git:(master) ✗
python run.py
data (1000, 1, 28, 28) target (1000,)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [-1, 32]          25,120
            Linear-2                    [-1, 10]             330
================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.10
Estimated Total Size (MB): 0.10
----------------------------------------------------------------
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.390919
```

```
Train Epoch: 0 [6400/60000 (11%)]       Loss: 0.694435
Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.284620
Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.606888
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.494829
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.831313
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.275505
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.432781
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.305147
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.837039

Test set: Average loss: 0.3851, Accuracy: 8961/10000 (90%)

Train Epoch: 1 [0/60000 (0%)]   Loss: 0.591371
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.753268
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.307875
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.078126
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.369558
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.534010
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.723705
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.317592
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.443172
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.356221

Test set: Average loss: 0.4522, Accuracy: 8733/10000 (87%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.367605
Train Epoch: 2 [6400/60000 (11%)]       Loss: 0.418987
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.424415
Train Epoch: 2 [19200/60000 (32%)]      Loss: 0.512402
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.890818
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.615039
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.463552
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.405713
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.514904
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.452697

Test set: Average loss: 0.3784, Accuracy: 9021/10000 (90%)
```

可以看到，经过 3 轮训练（`epochs = 3`），网络的准确率达到了 90%。虽然这个准确率不是很高，但是这个网络的参数量只有 0.10MiB，而且训练过程中的内存占用也很小，这对于嵌入式设备来说是非常有利的。

4.2 模型的简单量化

由于 FPGA 上的硬件资源有限，尤其是用于浮点计算的 DSP 等，因此需要对模型进行简单的量化，尽量减少浮点运算。

模型量化有多种方案，PyTorch、TensorFlow 等框架都有对应的量化方案，但是这些方案大都比较复杂，或是不同层之间使用不同的量化方案，或是需要添加额外的训练或推理过程，或是需要额外的工具支持。考虑到项目工作量、时间限制和硬件条件限制，项目中选择了一种简单的量化方案，即将模型中的浮点小数参数转换为定点小数参数，同时忽略模型整体的偏移、缩放等信息。之后会说明这样量化的合理性。

设量化后小数的位数为 N，其中小数部分占 N 位，符号位占 1 位。对于每个浮点参数，可以将其乘以 2^N，然后取整，即可得到定点小数参数。在推理过程的定点数相乘计算中，将计算结果除以 2^N，即可得到对应的定点乘法计算结果。在这个过程中，所有的乘法都可以用移位操作来代替，从而提高了计算效率。

表示为 Python 代码如下：

```python
# 浮点数转换为定点数
def float2fix(float_num: float, decimal_bit: int) -> int:
    fix_num = int(float_num * (2 ** decimal_bit))
    return fix_num


# 定点数转换为浮点数
def fix2float(fix_num: int, decimal_bit: int) -> float:
    float_num = fix_num * 1.0 / (2 ** decimal_bit)
    return float_num
```

以量化过程分类的话，此量化属于 PTQ（Post-training Quantization，训练后量化），即在训练过程中不对模型进行量化，而是在训练结束后对模型进行量化。这种量化方式的优点是简单，缺点是量化后的模型精度可能会有所下降。

有了模型数据的定点量化方案，还需要评估定点量化的参数以及初步估计其量化误差。

在 Python 类 `FcNet` 中添加了如下函数：

```
  def q_forward(self, x):
    def calc(d, n=Q_BITS):
      return ((d * (2 **
n)).to(torch.__dict__[Q_TYPE]).to(torch.float32)) / (2 ** n)
    x = calc(x)
    x = torch.flatten(x, 1)
    x = calc(x)
    x = self.fc1(x)
    x = calc(x)
    x = self.fc2(x)
    x = calc(x)
    output = F.log_softmax(x, dim=1)
    return output
```

这个函数的作用是将输入数据和模型参数转换为定点数，然后进行推理过程，在推理过程中不断将数据重新量化到定点数，最后输出结果。这样就可以粗略得到量化后的模型的输出结果。不过这样的评估仍然是不够准确的，其在推理过程中的矩阵相乘等计算仍然是浮点数计算，不一定能够反应全部使用定点计算的情况。

项目中在模型数据量化和导出逻辑后，做了简单的量化误差测试，将量化导出的数据重新加载到模型中，并使用 `q_forward()` 替换 `forward()`，比较两者的输出结果。

```
fc fc1.weight torch.Size([32, 784]) max 1.553686 min -1.8483458 int32
max 4294967287 int32 min 8 ../data/fc-fc1.weight.hex
restore diff mean -1.0933673e-08 max 9.5320866e-07 min -9.52743e-07
fc fc1.bias torch.Size([32]) max 0.07604257 min -0.16778715 int32 max
4294958541 int32 min 7307 ../data/fc-fc1.bias.hex
restore diff mean 9.636278e-08 max 9.331852e-07 min -9.275973e-07
fc fc2.weight torch.Size([10, 32]) max 0.2685824 min -0.2566016 int32
max 4294967189 int32 min 1203 ../data/fc-fc2.weight.hex
restore diff mean -4.848989e-08 max 9.4622374e-07 min -9.49949e-07
fc fc2.bias torch.Size([10]) max 0.798702 min -1.0660744 int32 max
4294592164 int32 min 251099 ../data/fc-fc2.bias.hex
restore diff mean -4.0233136e-08 max 6.2584877e-07 min -7.1525574e-07
Test Q: Accuracy: 9021/10000 (90%)
```

以上是数据导出到 `int32` 过程和量化误差测试的部分输出。行 `restore diff...` 表示使用 `q_forward()` 进行推理得到的结果与 `forward()` 的结果输出之间的差的平均值、极值等。可以看到量化误差很小，而且量化后的模型的准确率仍然很高，说明这种简单的量化方案是有可行性的。

参数导出为小数占 20 位，符号位占 1 位的 32 位有符号定点数（`Q_TYPE = "int32", Q_BITS = 20`）。
在实现过程中同样测试了其他的量化参数，在 `int8` 量化下全连接模型效果并不好，`int16` 下表现不够稳定，于是仅使用 `int32` 继续进行后续的实验。

模型数据被分层导出到 `fc-fc1.xxx.hex` 文件中，`xxx` 为 `weight` 或 `bias`，分别表示权重和偏置，导出格式为 `.hex` 文件，每行表示一个 Block Memory 的一行数据，为大写的 8 位十六进制数。`fc-fc1.bias.hex` 的部分文件内容：

```
00013778
00009687
00011FFE
FFFE5439
FFFF7A3F
...
```

4.3 卷积神经网络设计

类似于全连接网络，卷积神经网络也可以使用定点数进行量化。在本项目中，使用了 2 层卷积层和 2 层全连接层的卷积神经网络，其结构如下：

```
class CNNNet(nn.Module):
  def __init__(self) -> None:
    super().__init__()
    self.name = "cnn"
    self.conv1 = nn.Conv2d(1, 8, 3, 1)
    self.conv2 = nn.Conv2d(8, 4, 3, 1)
    self.fc1 = nn.Linear(4*16*3*3, 32)
    self.fc2 = nn.Linear(32, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = F.relu(x)
    x = self.conv2(x)
    x = F.relu(x)
    x = F.max_pool2d(x, 2)
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    output = F.log_softmax(x, dim=1)
    return output

  def q_forward(self, x):
    def calc(d, n=Q_BITS):
      return ((d * (2 **
n)).to(torch.__dict__[Q_TYPE]).to(torch.float32)) / (2 ** n)
    x = calc(x)
    x = self.conv1(x)
    x = calc(x)
    x = F.relu(x)
    x = calc(x)
    x = self.conv2(x)
    x = calc(x)
```

```
        x = F.relu(x)
        x = calc(x)
        x = F.max_pool2d(x, 2)
        x = calc(x)
        x = torch.flatten(x, 1)
        x = calc(x)
        x = self.fc1(x)
        x = calc(x)
        x = F.relu(x)
        x = calc(x)
        x = self.fc2(x)
        x = calc(x)
        output = F.log_softmax(x, dim=1)
        return output
```

网络结构、训练过程、数据导出过程:

```
(cumcm) ➜ chiro@chiro-pc  ~/programs/bsv-cnn/model git:(master) ✗
python run.py
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 8, 26, 26]              80
            Conv2d-2           [-1, 4, 24, 24]             292
            Linear-3                 [-1, 32]          18,464
            Linear-4                 [-1, 10]             330
================================================================
Total params: 19,166
Trainable params: 19,166
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.07
Estimated Total Size (MB): 0.14
----------------------------------------------------------------
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.318179
Train Epoch: 0 [6400/60000 (11%)]       Loss: 0.377542
```

```
Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.202414
Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.162691
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.050427
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.294064
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.060565
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.147821
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.088254
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.167153

Test set: Average loss: 0.0833, Accuracy: 9747/10000 (97%)

Train Epoch: 1 [0/60000 (0%)]   Loss: 0.056826
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.110928
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.048150
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.018540
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.068210
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.056555
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.110055
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.021205
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.252945
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.094724

Test set: Average loss: 0.0755, Accuracy: 9763/10000 (98%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.016985
Train Epoch: 2 [6400/60000 (11%)]       Loss: 0.079642
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.008232
Train Epoch: 2 [19200/60000 (32%)]      Loss: 0.130380
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.240502
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.207409
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.008487
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.009750
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.042786
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.035889

Test set: Average loss: 0.0670, Accuracy: 9805/10000 (98%)
```

```
cnn conv1.weight torch.Size([8, 1, 3, 3]) max 0.34368476 min
-0.6340126 int32 max 4294943368 int32 min 23284 ../data/cnn-
conv1.weight.hex
restore diff mean 1.6321428e-07 max 9.23872e-07 min -8.940697e-07
cnn conv1.bias torch.Size([8]) max -0.071760215 min -0.7841277 int32
max 4294892050 int32 min 4294145079 ../data/cnn-conv1.bias.hex
restore diff mean 4.2561442e-07 max 6.556511e-07 min 3.7252903e-08
cnn conv2.weight torch.Size([4, 8, 3, 3]) max 0.6175755 min -1.7718568
int32 max 4294966222 int32 min 1701 ../data/cnn-conv2.weight.hex
restore diff mean 1.0383721e-07 max 9.3411654e-07 min -9.4622374e-07
cnn conv2.bias torch.Size([4]) max 0.29474875 min 0.007723984 int32
max 309066 int32 min 8099 ../data/cnn-conv2.bias.hex
restore diff mean -2.469169e-07 max -1.1920929e-07 min -4.4703484e-07
cnn fc1.weight torch.Size([32, 576]) max 1.9494749 min -2.6695695
int32 max 4294967236 int32 min 26 ../data/cnn-fc1.weight.hex
restore diff mean 6.9132405e-08 max 9.5181167e-07 min -9.52743e-07
cnn fc1.bias torch.Size([32]) max 0.74299103 min -0.93857664 int32 max
4294930366 int32 min 5583 ../data/cnn-fc1.bias.hex
restore diff mean 1.9356958e-07 max 9.313226e-07 min -8.34465e-07
cnn fc2.weight torch.Size([10, 32]) max 0.66478866 min -1.2915374
int32 max 4294964495 int32 min 263 ../data/cnn-fc2.weight.hex
restore diff mean 1.0145395e-07 max 9.23872e-07 min -9.49949e-07
cnn fc2.bias torch.Size([10]) max 0.4453562 min -0.5563072 int32 max
4294723504 int32 min 44237 ../data/cnn-fc2.bias.hex
restore diff mean -2.1383167e-07 max 9.23872e-07 min -9.23872e-07
Test Q: Accuracy: 9805/10000 (98%)
```

`run.py` 同样对测试用数据集进行了数据量化和数据导出，导出为 `data/test_input.[data/target].hex` 文件，其中 `data/test_input.data.hex` 文件为输入数据，`data/test_input.target.hex` 文件为输入数据对应的标签。其中导出的数据经过 `torch.utils.data.DataLoader` 加工，进行随机打乱并数据归一化，然后选择其中的一部分测试集导出为 `.hex` 文件。

4.4 硬件部分

硬件部分主要需要解决以下问题：

1. 网络权重等参数如何存储和读取
2. 如何计算定点数的乘法和加法
3. 如何计算矩阵相乘过程
4. 如何输入输出

4.4.1 矩阵相乘过程

在上文已经提到，模型数据导出为了 `.hex` 文件格式，其实就是为了便于 Verilog 读取。Verilog 读取 `.hex` 的方式是调用 `$readmemh` 函数，该函数的作用是从文件读取文本格式的十六进制字符串数据并存储到指定的寄存器中，并且在综合时会被综合为 Block Memory 等利于存储的格式。

在矩阵计算的过程中，利用硬件的并行性和矩阵计算的特性，可以进行计算加速。

以全连接神经网络的第一层为例，输入数据为 28×28 的矩阵，矩阵中每个值都是 `int32` 的 32 位浮点数，其中小数部分占 20 位。

将输入图像拉平（`flatten()`），成为 1×784 的矩阵，与本层权重 W 相乘，$W.shape = 784 \times 32$，相乘过程为：

1. 从矩阵 W 中选择一列，形状为 784×1
2. 与输入数据向量每个对应位置元素相乘
3. 将相乘结果相加，得到一个值
4. 回到 1. 直到 32 列全部被计算

在这样的计算过程中，可以发现，每次选择权重矩阵中的一列，然后进行挨个相乘后相加运算，各列之间是并行计算的关系。即，我们可以并行地同时选择权重矩阵中的 32 列，并进行 784 次相乘相加，最后得到一列 32 个元素的向量，即得到本层矩阵乘法计算结果。

4.4.2 偏置相加过程

不仅是模型的权重数据，模型的偏置数据也被写入 Block Memory。为了降低硬件复杂度，偏置数据同样只能每个周期读一次。

为了在规定时间内将偏置读取并添加到上述矩阵乘法结果中，并尽量少地占用硬件逻辑，可以将偏置的读、数据加和过程，与矩阵运算过程融合。具体如下：

1. 在矩阵乘法读取每一行元素数据的时候，同时读取当前“读指针”对应的偏置数据
2. 如果当前时钟周期能够获取到这一列的对应偏置，则提早地将其加入求和过程
3. 当矩阵乘法算法完成，偏置已经被加入到对应位置，结果就是本层神经网络输出

于是，本层在每周期只读取一次权重和偏置数据的情况下，只需要矩阵长边（本例子中为 784）那么多时钟周期（忽略 FIFO 时钟延迟），即可得到本层神经网络计算结果。

4.4.3 代码实现细节

定义每层神经网络的输入输出接口如下：

```
interface Layer#(type in, type out);
  method Action put(in x);
  method ActionValue#(out) get;
endinterface
```

- `put` 是一个 `Action method`，有效执行时会将一个 `in` 类型的数据添加到此层的 FIFO 输入缓冲中以便进一步计算。
- `get` 是一个返回 `ActionValue` 的方法，有效执行时将 FIFO 输出缓冲中的数据取出传递给下一层网络。只有输出缓冲区有数据此方法才有效。

全连接层

定义全连接层模块如下：

```
module mkFCLayer#(parameter String layer_name)(Layer#(in, out))
  provisos(
    Bits#(out, lines_bits),
    Bits#(in, depth_bits),
    Mul#(lines, 32, lines_bits),
    Mul#(depth, 32, depth_bits),
    PrimSelectable#(in, Int#(32)),
    PrimSelectable#(out, Int#(32)),
    PrimWriteable#(Reg#(out), Int#(32)),
    Add#(TLog#(lines), a__, TLog#(depth))
  );
// ...
endmodule
```

- `in` / `out` 为泛型的输入输出类型
- `lines` 定义此层的权重行数，如 `fc1` 则 `lines = 32`
- `depth` 定义此层的权重列数，如 `fc1` 则 `depth = 784`
- `PrimSelectable#(*, Int#(32))` 定义输入输出都是一维向量格式
- `Add#(TLog#(lines), a__, TLog#(depth))` 规定 `depth >= lines`，以简化逻辑

神经网络数据加载模块接口定义如下：

```
interface LayerData_ifc#(type td, type lines, type depth);
  method ActionValue#(Bit#(TMul#(lines, SizeOf#(td)))) getWeights();
  method ActionValue#(td) getBias();
  method Action weightsStart();
  method Action biasStart();
  method Bit#(TAdd#(TLog#(depth), 1)) getWeightsIndex();
  method Bit#(TAdd#(TLog#(lines), 1)) getBiasIndex();
  method Bool weightsDone();
  method Bool biasDone();
endinterface
```

- `getWeights` 获取当前读取到的权重数据
- `getBias` 获取当前获取到的偏置数据
- `weightsStart` 设置权重读指针开始递增

- `biasStart` 设置偏置读指针开始递增。权重和偏置指针是不同的，尽管它们在本设计中几乎同步
- `getWeightsIndex`、`getBiasIndex` 获取当前指针值，其为上一周期获取到的值的对应指针
- `weightsDone`、`biasDone` 两个指针是否分别已经完成一轮数据读取

神经网络数据加载模块定义如下：

```
module mkLayerData#(parameter String model_name, parameter String
layer_name)(LayerData_ifc#(td, lines, depth))
    provisos (
      Bits#(td, sz),
      Literal#(td),
      Log#(depth, depth_log),
      Log#(lines, lines_log),
      Mul#(lines, sz, lines_bits)
    );
// ...
endmodule
```

则在 `mkFCLayer` 中可以这样定义这个数据读取模块：

```
LayerData_ifc#(Int#(32), lines, depth) data <- mkLayerData("fc",
layer_name);
```

`mkFCLayer` 中管理两个 FIFO，一个负责读入数据的缓冲，一个负责读出数据的缓冲，二者容量都是 2，有满、空信号引出。

```
  FIFOF#(in) fifo_in <- mkFIFOF;
  FIFOF#(out) fifo_out <- mkFIFOF;
```

`mkFCLayer` 大致有以下几个状态：

1. 等待 `fifo_in` 数据输入缓冲
2. `fifo_in` 非空，设置读取模块的两个读指针开始从 0 递增
3. 乘加权重数据，并加上对应的偏置数据
4. 偏置数据处理完毕，乘加上剩下的权重数据
5. 两个指针走完，将计算结果 `tmp` 压入 `fifo_out`，清除寄存器状态并返回 1.

`mkFCLayer` 其他特殊处理：

1. 为了尽量保证乘加的精度，`tmp` 中每个元素位宽设置为 64 位，尽量减少数值溢出，在数据压入 `fifo_out` 时才转换回 32 位
2. 由于规定 `depth >= lines`，所以不需要处理只加 `bias` 的情况

由于代码较长，不完整贴出。

ReLU 层

ReLU 层只需要一个 `fifo_out` 进行数据管理。

它将所有小于 0 的输入数据置为 0，其他不变。

```
module mkReluLayer(Layer#(in, out))
  provisos (
    Bits#(in, input_bits),
    Mul#(input_size, 32, input_bits),
    PrimSelectable#(in, Int#(32)),
    Bits#(out, output_bits),
    Mul#(output_size, 32, output_bits),
    PrimSelectable#(out, Int#(32)),
    Add#(input_bits, 0, output_bits),
    PrimUpdateable#(out, Int#(32))
  );

  FIFO#(out) fifo_out <- mkFIFO1;

  method Action put(in x);
    out y;
    for (Integer i = 0; i < valueOf(input_size); i = i + 1) begin
      if (x[i] < 0) y[i] = 0;
      else y[i] = x[i];
    end
    fifo_out.enq(y);
  endmethod

  method ActionValue#(out) get;
    fifo_out.deq;
```

```
    return fifo_out.first;
  endmethod

endmodule
```

Softmax 层

标准的 Softmax 层应当将输出归一化，使之结果和为 1。这里为了节省硬件逻辑，只是选择出值最大的下标并输出。严格来说，这个应该可以算是 ArgMax 层。

```
module mkSoftmaxLayer(Layer#(in, out))
  provisos (
    Bits#(in, input_bits),
    Mul#(input_size, 32, input_bits),
    PrimSelectable#(in, Int#(32)),
    Bits#(out, output_bits),
    PrimIndex#(out, a__)
  );

  FIFO#(out) fifo_out <- mkFIFO1;

  method Action put(in x);
    out y = unpack('0);
    // just `hard' max
    for (Integer i = 0; i < valueOf(input_size); i = i + 1)
      if (x[i] > x[y]) y = fromInteger(i);
    fifo_out.enq(y);
  endmethod

  method ActionValue#(out) get;
    fifo_out.deq;
    return fifo_out.first;
  endmethod

endmodule
```

卷积层

卷积层 `mkConvLayer` 的逻辑与 `mkFCLayer` 类似，不同点有：

1. 可以通过 `img2col` 的方式，将需要与卷积核进行运算的部分重新排列，进一步提高并行性
2. 由于多了一层卷积，可以说输入输出的维度也是不同的，需要保持信息的二维特征

由于本项目中卷积层实现尚未完全，所以暂时不可用。在后续会继续完善卷积部分。

`mkConvLayer` 定义如下：

```
module mkConvLayer#(parameter String layer_name)(Layer#(in, out))
// now assuming that stride == 1
  provisos (
    Bits#(in, input_bits),
    Mul#(input_size, 32, input_bits),
    Mul#(input_lines, input_lines, input_size),
    Bits#(out, output_bits),
    Mul#(TMul#(output_size, 32), output_channels, output_bits),
    Mul#(output_lines, output_lines, output_size),
    // 2D vectors required
    PrimSelectable#(in, Vector::Vector#(input_lines, Int#(32))),
    PrimSelectable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, Int#(32)))),
    Add#(output_lines, kernel_size, TAdd#(input_lines, 1)),
    Mul#(kernel_size, kernel_size, kernel_size_2),
    Mul#(kernel_size_2, 32, kernel_size_2_bits),
    Add#(kernel_size, 0, 3),
    PrimUpdateable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, Int#(32))))
  );
// ...
endmodule
```

其中，

```
    // 2D vectors required
    PrimSelectable#(in, Vector::Vector#(input_lines, Int#(32))),
    PrimSelectable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, Int#(32)))),
```

这一部分规定了输出的维度也需要是 2D。

```
  rule set_data_in;
    data_in <= fifo_in.first;
  endrule

  Wire#(Vector#(output_lines, Vector#(output_lines, Bit#
(kernel_size_2_bits)))) cols <- mkWire;
  rule bind_cols;
    Vector#(output_lines, Vector#(output_lines, Bit#
(kernel_size_2_bits))) cols_ = unpack('0);
    for (Integer i = 0; i < valueOf(output_lines); i = i + 1) begin
      for (Integer j = 0; j < valueOf(output_lines); j = j + 1) begin
        cols_[i][j] = {
          pack(data_in[i][j]),
          pack(data_in[i][j + 1]),
          pack(data_in[i][j + 2]),
          pack(data_in[i + 1][j]),
          pack(data_in[i + 1][j + 1]),
          pack(data_in[i + 1][j + 2]),
          pack(data_in[i + 2][j]),
          pack(data_in[i + 2][j + 1]),
          pack(data_in[i + 2][j + 2])
        };
      end
    end
    cols <= cols_;
  endrule
```

这部分在规定 `kernel_size == 3` 的情况下，将 28×28 的二维图像散列到 $26 \times 26 \times 3 \times 3$ 的小区域序列中，以便后续计算中卷积核（权重）直接与 `col` 部分相乘。

但是，由于图像位数和列数等过宽，超过了 Bluespec 编译器的处理能力，它在有限的时间内并不能将这一 `img2col` 逻辑转换到硬件。所以后续的计算等暂未进行。编译过程：

```
→ chiro@chiro-pc  ~/programs/bsv-cnn git:(master) ✗ make verilog-CNN
ROOT=/home/chiro/programs/bsv-cnn /home/chiro/programs/bsv-
cnn/bsvbuild.sh -v mkTb CNN.bsv 10000000
top module: mkTb
top file  : CNN.bsv
print simulation log to: /dev/stdout

// 已经尽可能延长可计算时间，增大栈空间
bsc +RTS -Ksize -RTS -steps-max-intervals 10000000 -verilog -g mkTb -u
CNN.bsv
checking package dependencies
compiling CNN.bsv
code generation for mkTb starts
Warning: "Prelude.bs", line 584, column 27: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `Prelude.PrimIndex~Prelude.Integer~32'. The current number of steps
is
  100000. Next warning at 200000 steps. Elaboration terminates at
  1000000000000 steps.
  During elaboration of the body of rule `bind_cols' at "Layers.bsv",
line
  211, column 8.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Prelude.bs", line 1329, column 9: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `Prelude.Ord~Prelude.Integer'. The current number of steps is
200000. Next
  warning at 300000 steps. Elaboration terminates at 1000000000000
steps.
  During elaboration of `tmp' at "Layers.bsv", line 233, column 13.
```

```
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Prelude.bs", line 3090, column 0: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `primFix'. The current number of steps is 300000. Next warning at
400000
  steps. Elaboration terminates at 1000000000000 steps.
  During elaboration of `tmp' at "Layers.bsv", line 233, column 13.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Prelude.bs", line 3090, column 0: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `primFix'. The current number of steps is 400000. Next warning at
500000
  steps. Elaboration terminates at 1000000000000 steps.
  During elaboration of the body of rule `start' at "Layers.bsv", line
235,
  column 8.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Array.bsv", line 208, column 23: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `primExtract'. The current number of steps is 500000. Next warning
at 600000
  steps. Elaboration terminates at 1000000000000 steps.
  During elaboration of the body of rule `start' at "Layers.bsv", line
235,
  column 8.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Prelude.bs", line 421, column 6: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
```

```
  `Prelude.Ord~Prelude.Integer'. The current number of steps is
600000. Next
  warning at 700000 steps. Elaboration terminates at 1000000000000
steps.
  During elaboration of the body of rule `start' at "Layers.bsv", line
235,
  column 8.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.
Warning: "Prelude.bs", line 1329, column 9: (G0024)
  The function unfolding steps interval has been exceeded when
unfolding
  `Prelude.Ord~Prelude.Integer'. The current number of steps is
700000. Next
  warning at 800000 steps. Elaboration terminates at 1000000000000
steps.
  During elaboration of the body of rule `acc' at "Layers.bsv", line
243,
  column 8.
  During elaboration of `conv1' at "CNN.bsv", line 9, column 92.
  During elaboration of `mkTb' at "CNN.bsv", line 7, column 8.

// 之后过程无法结束
```

在后续会继续完善 2D 的卷积层。

整体网络连接

有了上述代码结构，我们可以很方便地定义一个神经网络的各层：

```
Layer#(Vector#(784, Int#(32)), Vector#(32, Int#(32))) fc1 <-
mkFCLayer("fc1");
// 可选的 ReLU，与训练一致
// Layer#(Vector#(32, Int#(32)), Vector#(32, Int#(32))) relu1 <-
mkReluLayer;
Layer#(Vector#(32, Int#(32)), Vector#(10, Int#(32))) fc2 <-
mkFCLayer("fc2");
Layer#(Vector#(10, Int#(32)), Int#(32)) softmax <- mkSoftmaxLayer;
```

然后再通过 FIFO 进行互相连接:

```
rule put_data;
  let d <- input_data.get;
  Tuple2#(Int#(32), Vector::Vector#(784, Int#(32))) d_pack =
unpack(d);
  match {.target, .data} = d_pack;
  let real_target = target >> q_bits();
  fc1.put(data);
  targets.enq(real_target);
endrule

// rule put_data_relu1;
//   let out <- fc1.get;
//   relu1.put(out);
// endrule

rule put_data_fc2;
  let out <- fc1.get;
  fc2.put(out);
endrule

rule put_data_softmax;
  let out <- fc2.get;
  softmax.put(out);
endrule

rule get_data_softmax;
  Int#(32) real_data <- softmax.get;
  Int#(32) target = targets.first;
  targets.deq;
  $write("[cnt=%x] Got target: %d, pred: %d, ", cnt, target,
real_data);
  if (real_data == target) begin
    $display("correct");
    correct <= correct + 1;
  end else begin
```

```
    $display("wrong");
  end
  total <= total + 1;
endrule
```

通过不同 FIFO 的缓冲，我们构建起了一个流水线结构的推理模型。实际延迟由计算时间最长的一层决定。

在本例子中，一次计算时长为 $784 + 1 + 32 + 1 + 1 = 819$ 周期，如果在 50Mhz 的外设主频下运行，可以在 17us 内得出推理结果。

4.5 软件部分

软件部分由于进度、硬件等原因暂时未实现，以下仅讨论设想中的实现方式和原理。

在嵌入式系统上较为常见的计算机指令集架构有 ARM、RISC-V、C-Sky 等，大多数采用的 IO 访问方式都是地址总线空间映射，即将一个设备的控制寄存器、数据空间等映射到总线上的一段地址空间内，以总线对应地址读写的方式对其进行管理访问。本项目设计中也将把这个“神经网络计算 IP 模块”映射到总线地址空间中。

嵌入式系统中常见的片内总线有 AXI、APB、PCIe 等，本项目设计中预期使用 AXI 总线对其进行访问和控制。

设计本模块的交互控制寄存器接口如下：

说明	起始地址	长度	权限
输入数据	0x1f001000	0xb60	写
设置目的地址	0x1f002000	0x8	写
开始计算	0x1f002004	0x1	写

由于使用了 DMA 方式获取数据，还需要一条连接到 CPU 或中断控制器的中断信号线。

大致流程：

```
int32_t input[28][28];
int32_t output;

/// 读取摄像头数据，并进行一定的预处理，写入 input[][]
memcpy(0x1f001000, input, sizeof(input));
*(uint64_t*)(0x1f002000) = &output;
*(uint8_t*)(0x1f002004) = 1;
/// 设置中断，并等待中断触发返回

printf("output is %d\n", output);
```

5 结果呈现

由于当前仅有仿真环境，所以仅进行了仿真环境中的实验。

```
→ chiro@chiro-pc  ~/programs/bsv-cnn git:(master) ✗ make FC
ROOT=/home/chiro/programs/bsv-cnn /home/chiro/programs/bsv-
cnn/bsvbuild.sh -bs mkTb FC.bsv 10000000
top module: mkTb
top file  : FC.bsv
print simulation log to: /dev/stdout

maximum simulation time argument: -m 10000000

bsc +RTS -Ksize -RTS -steps-max-intervals 10000000 -sim -g mkTb -u
FC.bsv
checking package dependencies
compiling FC.bsv
code generation for mkTb starts
Elaborated module file created: mkTb.ba
All packages are up to date.
bsc +RTS -Ksize -RTS -steps-max-intervals 10000000 -sim -e mkTb -o
sim.out

/// 输出

[cnt=00046865] Got target:           2, pred:           8, wrong
[cnt=00046b78] Got target:           9, pred:           0, wrong
```

```
[cnt=00046e8b] Got target:           0, pred:           2, wrong
[cnt=0004719e] Got target:           0, pred:           0, correct
[cnt=000474b1] Got target:           8, pred:           2, wrong
[cnt=000477c4] Got target:           0, pred:           0, correct
[cnt=00047ad7] Got target:           0, pred:           6, wrong
[cnt=00047dea] Got target:           5, pred:           2, wrong
[cnt=000480fd] Got target:           9, pred:           2, wrong
[cnt=00048410] Got target:           1, pred:           2, wrong
[cnt=00048723] Got target:           3, pred:           4, wrong
[cnt=00048a36] Got target:           0, pred:           0, correct
[cnt=00048d49] Got target:           9, pred:           2, wrong
[cnt=0004905c] Got target:           8, pred:           9, wrong
[cnt=0004936f] Got target:           8, pred:           2, wrong
Stopping, total:         381, correct:          78, accuracy:
 20 %
```

`make FC` 开始仿真测试，读取测试集并逐个测试输出，记录正确的值。

由于实现上的问题，或是精度上的问题，硬件上全连接层尚且存在问题，得到的结果明显是有问题的。后续会进一步改进。

6 嵌入式系统总结

作为一门偏“硬”的课程，《嵌入式计算》的课程内容量大，对于基础知识的要求也高。在本学期的学习中，我对于嵌入式系统的相关知识有了一些了解。非常感谢老师（和助教们）的付出。没有你们就没有这门课程，也就没有我对嵌入式系统的更深入的了解。愿这门课程越开越好，也希望更多同学来选修学习。