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	计算机视觉
设计题目	基于FPGA的CNN网络实现
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	苏敬勇

目录

1 概述

本项目基于 FPGA 和 Bluespec HDL 实现硬件卷积神经网络推理加速。

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。

2 理论推导

见附件。

3 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，仿真验证也比较简单。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。

3.1 神经网络结构设计

3.1.1 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 *
```

```python
class FcNet(nn.Module):
  def __init__(self) -> None:
    super().__init__()
    self.name = "fc"
    self.fc1 = nn.Linear(784, 64)
    self.fc2 = nn.Linear(64, 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, 64]          50,240
            Linear-2                   [-1, 10]             650
================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.19
Estimated Total Size (MB): 0.20
----------------------------------------------------------------
```

此网络的总参数量为 50890，其中第一层的参数量为 50240，第二层的参数量为 650，总大小只有 0.19MiB。

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：

```
/home/chiro/.conda/envs/cumcm/lib/python3.10/site-
packages/torch/utils/cpp_extension.py:325: UserWarning:

                                !! WARNING !!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!
Your compiler (c++) is not compatible with the compiler Pytorch was
built with for this platform, which is g++ on linux. Please
use g++ to to compile your extension. Alternatively, you may
compile PyTorch from source using c++, and then you can also use
c++ to compile your extension.

See https://github.com/pytorch/pytorch/blob/master/CONTRIBUTING.md for
help
with compiling PyTorch from source.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!

                                !! WARNING !!

  warnings.warn(WRONG_COMPILER_WARNING.format(
data (1000, 1, 28, 28) target (1000,)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 64]          50,240
            Linear-2                   [-1, 10]             650
================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
----------------------------------------------------------------
```

```
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.19
Estimated Total Size (MB): 0.20
----------------------------------------------------------------
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.290047
Train Epoch: 0 [6400/60000 (11%)]       Loss: 0.845264
Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.269843
Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.466920
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.267038
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.468555
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.373619
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.413304
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.195239
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.500714

Test set: Average loss: 0.2826, Accuracy: 9207/10000 (92%)

Train Epoch: 1 [0/60000 (0%)]   Loss: 0.134922
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.258774
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.587307
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.359384
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.133505
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.303560
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.316389
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.321140
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.242984
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.300242

Test set: Average loss: 0.2257, Accuracy: 9420/10000 (94%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.083718
Train Epoch: 2 [6400/60000 (11%)]       Loss: 0.185606
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.226405
Train Epoch: 2 [19200/60000 (32%)]      Loss: 0.039246
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.413483
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.314556
```

```
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.114459
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.275238
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.341927
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.132406

Test set: Average loss: 0.2542, Accuracy: 9292/10000 (93%)
```

可以看到，经过 3 轮训练（`epochs = 3`），网络的准确率达到了 93% 以上。
虽然这个准确率不是很高，但是这个网络的参数量只有 0.19MiB，而且训练过程中的内存占用也很小，这对于实验来说是非常有利的。

3.1.2 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
```

同时，在训练过程中，使用 `QPyTorch` 进行训练，这样可以在训练过程中对模型参数进行定点量化，从而进一步提升推理模型效果。

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：

```python
  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。

在实现 `q_forward()` 之后，进一步进行了模型量化过程测试，实现了如下的量化全连接层：

```
def manual_fc_layer(layer, x):
  dtype = np.__dict__[Q_TYPE]
  valid_bits = int(Q_TYPE[3:])
  dtype2 = np.__dict__[f'int{valid_bits * 2}']
  x0 = x if isinstance(x, np.ndarray) else
float2fixv(x.clone().detach().numpy(), Q_BITS, dtype=dtype)
  w = float2fixv(layer.weight.clone().detach().numpy(), Q_BITS,
dtype=dtype)
  b = float2fixv(layer.bias.clone().detach().numpy(), Q_BITS,
dtype=dtype)
  r0 = [dtype2(0) for _ in range(len(b))]
  for i in range(len(x0)):
```

```
    for j in range(len(b)):
      mul_raw = dtype2(dtype2(x0[i]) * dtype2(w[j][i]))
      mul = dtype2(mul_raw >> Q_BITS)
      r0[j] = dtype2(r0[j] + mul)
      if i == j:
        r0[i] = dtype2(r0[i] + b[i])
  r0 = dtype(r0)
  return r0
```

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
...
```

3.1.3 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。

由于两层以及以上的神经网络对 FPGA 过于复杂，无法综合，所以这里暂时只实现了一层 CNN。要实现更多层 CNN 在代码层面上只需要添加几行即可，不过需要更强的 FPGA 设备和更好的综合算法。

在本项目中，使用了 1 层卷积层和 1 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, 2, 3, 1)
    self.fc1 = nn.Linear(338, 10)

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

```
// (1, 28, 28) -> (2, 26, 26)
Layer#(
    Vector#(1, Vector#(28, Vector#(28, ElementType))),
    Vector#(2, Vector#(26, Vector#(26, ElementType)))
  ) conv1 <- mkConvLayer("conv1");
// (2, 26, 26) -> (2, 13, 13)
Layer#(
  Vector#(2, Vector#(26, Vector#(26, ElementType))),
  Vector#(2, Vector#(13, Vector#(13, ElementType)))
) max_pooling1 <- mkMaxPoolingLayer;
// flatten -> relu -> (338, )
Layer#(Vector#(338, ElementType), Vector#(338, ElementType)) relu1 <-
mkReluLayer;
Layer#(Vector#(338, ElementType), Vector#(10, ElementType)) fc1 <-
mkFCLayer("cnn", "fc1");
Layer#(Vector#(10, ElementType), ResultType) softmax <-
mkSoftmaxLayer;
```

同时也实现了用 `numpy` 手动实现的定点小数卷积层推导:

```
def manual_conv_layer(layer, x):
  dtype = np.__dict__[Q_TYPE]
  valid_bits = int(Q_TYPE[3:])
  dtype2 = np.__dict__[f'int{valid_bits * 2}']
  x0 = x if isinstance(x, np.ndarray) else
float2fixv(x.clone().detach().numpy(), Q_BITS, dtype=dtype)
  if len(x0.shape) == 3:
    x0 = x0.reshape((1, *x0.shape))
  w = float2fixv(layer.weight.clone().detach().numpy(), Q_BITS,
dtype=dtype)
  b = float2fixv(layer.bias.clone().detach().numpy(), Q_BITS,
dtype=dtype)
  kernel_size = 3
  out_size = (x0.shape[2] + 1 - kernel_size, x0.shape[3] + 1 -
kernel_size)
  output_channel = w.shape[0]
  out = np.zeros((output_channel, *out_size), dtype=dtype2)
  for c in range(output_channel):
    for i in range(out_size[0]):
      for j in range(out_size[1]):
        xx = x0[0, :, i:i+kernel_size, j:j+kernel_size].flatten()
        ww = w[c, :].flatten()
        rr = dtype2(0)
        for xi in range(len(xx)):
          mul_raw = dtype2(dtype2(xx[xi]) * dtype2(ww[xi]))
          mul = dtype2(mul_raw >> Q_BITS)
          rr = dtype2(rr + mul)
        rr = dtype(rr)
        out[c, i, j] = rr + b[c]
  return out
```

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, 2, 26, 26]              20
            Linear-2                  [-1, 10]           3,390
================================================================
Total params: 3,410
Trainable params: 3,410
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.01
Estimated Total Size (MB): 0.03
----------------------------------------------------------------
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.256816
Train Epoch: 0 [6400/60000 (11%)]       Loss: 0.625938
Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.294960
Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.438405
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.333763
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.649237
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.192731
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.253268
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.076433
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.579675

Test set: Average loss: 0.1848, Accuracy: 9436/10000 (94%)

Train Epoch: 1 [0/60000 (0%)]   Loss: 0.224390
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.097651
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.237935
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.116756
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.207009
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.126870
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.159335
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.228187
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.171130
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.407692

Test set: Average loss: 0.1732, Accuracy: 9463/10000 (95%)
```

```
Train Epoch: 2 [0/60000 (0%)]   Loss: 0.178725
Train Epoch: 2 [6400/60000 (11%)]       Loss: 0.213457
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.101777
Train Epoch: 2 [19200/60000 (32%)]      Loss: 0.114195
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.250721
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.185595
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.065680
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.099127
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.211035
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.090882

Test set: Average loss: 0.1850, Accuracy: 9434/10000 (94%)

save model to cnn.pt
Test Manual: Accuracy: 4/4 (100%)

cnn conv1.weight torch.Size([2, 1, 3, 3]) max 0.5546875 min
-0.55859375 int16 max 65519 int16 min 1 ../data/cnn-conv1.weight.hex
restore diff mean 0.0 max 0.0 min 0.0
cnn conv1.bias torch.Size([2]) max -0.49609375 min -1.2890625 int16
max 65409 int16 min 65206 ../data/cnn-conv1.bias.hex
restore diff mean 0.0 max 0.0 min 0.0
cnn fc1.weight torch.Size([10, 338]) max 2.0390625 min -4.8085938
int16 max 65535 int16 min 0 ../data/cnn-fc1.weight.hex
restore diff mean 0.0 max 0.0 min 0.0
cnn fc1.bias torch.Size([10]) max 3.0625 min -2.296875 int16 max 65471
int16 min 311 ../data/cnn-fc1.bias.hex
restore diff mean 0.0 max 0.0 min 0.0
```

`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。

3.2 模型推理过程

在硬件实现的推理流程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

3.2.1 网络权重等参数如何存储和读取

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。

此外，由于卷积层的参数较少，而且其读取并非一个连续的过程，因此在实现中将卷积层的数据直接生成为 Bluespec 代码，硬编码到 `Vector::Vector#()` 中。

```
def dump_bsv(model, vector: bool = False, conv_only: bool = True,
max_dim: int = 0, fixed: bool = False):
  valid_bits = int(Q_TYPE[3:])
  name = model.name.lower()
  with open(GEN_PATH + name + ".bsv", "w") as f:
    f.write(f"// generated file\npackage {name};\n\nimport
Vector::*;\nimport Config::*;\n\n")
    def get_layer_name(original: str) -> str:
      return (name + "_" + original).replace(".", "_").replace("-",
"_")
    for key in model.state_dict():
      if conv_only and "conv" not in key:
        continue
      data = model.state_dict()[key]
      if max_dim > 0 and len(data.shape) > max_dim:
        if max_dim == 1:
          data = data.reshape([-1])
        else:
          data = data.reshape([*data.shape[:max_dim - 1], -1])
```

```
      if fixed:
        data = np.array([float2fix(x, Q_BITS) for x in
data.flatten()], dtype=Q_TYPE).reshape(data.shape)
      f.write(f"// model {name} key {key} \n")

      def write_function_header(typ: str, typ2=None, typ3="a"):
        typ2 = typ2 if typ2 is not None else typ
        f.write(f"function {typ2} {get_layer_name(key)}();\n")
        f.write(f"  {typ} {typ3};\n")

      if vector:
        if len(data.shape) == 1:
          typ = f"Vector#({data.shape[0]}, ElementType"
          write_function_header(typ)
          for i in range(data.shape[0]):
            f.write(f"a[{i}]={data[i]};")
          f.write("\n")
        elif len(data.shape) == 2:
          typ = f"Vector#({data.shape[0]}, Vector#({data.shape[1]},
ElementType)"
          write_function_header(typ)
          for i in range(data.shape[0]):
            for j in range(data.shape[1]):
              f.write(f"a[{i}][{j}]={data[i][j]};")
            f.write("\n")
        elif len(data.shape) == 4:
          typ = f"Vector#({data.shape[0]}, Vector#({data.shape[1]},
Vector#({data.shape[2]}, Vector#({data.shape[3]}, ElementType)))"
          write_function_header(typ)
          for i in range(data.shape[0]):
            for j in range(data.shape[1]):
              for k in range(data.shape[2]):
                for l in range(data.shape[3]):
                  f.write(f"a[{i}][{j}][{k}][{l}]={data[i][j][k]
[l]};")
              f.write("\n")
      else:
```

```
        if len(data.shape) == 1:
          write_function_header(f"ElementType a[{data.shape[0]}]",
f"ElementType[]", "")
          for i in range(data.shape[0]):
            f.write(f"a[{i}]={data[i]};")
          f.write("\n")
        elif len(data.shape) == 2:
          write_function_header(f"ElementType a[{data.shape[0]}]
[{data.shape[1]}]", f"ElementType[][]", "")
          for i in range(data.shape[0]):
            for j in range(data.shape[1]):
              f.write(f"a[{i}][{j}]={data[i][j]};")
            f.write("\n")
        elif len(data.shape) == 4:
          write_function_header(f"ElementType a[{data.shape[0]}]
[{data.shape[1]}][{data.shape[2]}][{data.shape[3]}]", f"ElementType[]
[][][]", "")
          for i in range(data.shape[0]):
            for j in range(data.shape[1]):
              for k in range(data.shape[2]):
                for l in range(data.shape[3]):
                  f.write(f"a[{i}][{j}][{k}][{l}]={data[i][j][k]
[l]};")
              f.write("\n")
      f.write(f"  return a;\n")
      f.write(f"endfunction\n\n")

    f.write("endpackage\n")
```

生成的代码如下:

```
// generated file
package cnn;

import Vector::*;
import Config::*;

// model cnn key conv1.weight
```

```
function ElementType[][] cnn_conv1_weight();
  ElementType a[2][9] ;
a[0][0]=0.24524441361427307;a[0][1]=0.1674749106168747;a[0]
[2]=-0.2524031698703766;a[0][3]=0.15313813090324402;a[0]
[4]=0.55595862865448;a[0][5]=0.004632969852536917;a[0]
[6]=-0.5603101849555969;a[0][7]=-0.06770122051239014;a[0]
[8]=0.45140358805656433;
a[1][0]=-0.19808626174926758;a[1][1]=0.1607075333595276;a[1]
[2]=0.2862326204776764;a[1][3]=0.331738144159317;a[1]
[4]=0.1785387098789215;a[1][5]=-0.2291257083415985;a[1]
[6]=0.0023327621165663004;a[1][7]=-0.11739463359117508;a[1]
[8]=-0.22275766730308533;
  return a;
endfunction

// model cnn key conv1.bias
function ElementType[] cnn_conv1_bias();
  ElementType a[2] ;
a[0]=-1.290930986404419;a[1]=-0.4963105022907257;
  return a;
endfunction

endpackage
```

3.2.2 矩阵相乘过程

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。

3.2.3 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。

3.2.4 卷积计算过程

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

`mkConvLayer` 定义如下：

输出、输入都是 3D `Vector`。

```
// input shape: (input_channels, input_lines, input_lines)
// output shape: (output_channels, output_lines, output_lines)
module mkConvLayer#(parameter String layer_name)(Layer#(in, out))
// now assuming that stride == 1
  provisos (
    Bits#(in, input_bits),
    Bits#(out, output_bits),
    Mul#(TMul#(output_size, SizeOf#(ElementType)), output_channels,
output_bits),
    Mul#(TMul#(input_size, SizeOf#(ElementType)), input_channels,
input_bits),
    Mul#(input_lines, input_lines, input_size),
    Mul#(output_lines, output_lines, output_size),
    Add#(output_lines, 2, output_lines_2),
    // 3D vectors required
    PrimSelectable#(in, Vector::Vector#(input_lines, Vector::Vector#
(input_lines, ElementType))),
    PrimSelectable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, ElementType))),
    Add#(output_lines, kernel_size, TAdd#(input_lines, 1)),
    Mul#(kernel_size, kernel_size, kernel_size_2),
    // Mul#(kernel_size_2, SizeOf#(ElementType), kernel_size_2_bits),
    // Add#(kernel_size, 0, 3),
    PrimUpdateable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, ElementType)))
  );
// ...
endmodule
```

其中,

```
    // 3D vectors required
    PrimSelectable#(in, Vector::Vector#(input_lines, Vector::Vector#
(input_lines, ElementType))),
    PrimSelectable#(out, Vector::Vector#(output_lines, Vector::Vector#
(output_lines, ElementType))),
```

这一部分规定了输出的维度也需要是 3D。

本层将计算过程大致划分为以下流水阶段：

1. `im2col`：将输入的 3D 矩阵中需要与 Kernel 进行运算的部分重新排列为 1D 的向量，以便后续计算
2. `calculate_mul_col`：计算每一个 `col` 与 Kernel 的乘积
3. `calculate_acc_col`：计算相乘后的结果的累加和，并加上偏置
4. `cols_to_output`：将 `cols` 重新组合为一个矩阵，组合完成后送入下一层

3.2.5 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
```

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：

```
// (1, 28, 28) -> (2, 26, 26)
Layer#(
    Vector#(1, Vector#(28, Vector#(28, ElementType))),
    Vector#(2, Vector#(26, Vector#(26, ElementType)))
  ) conv1 <- mkConvLayer("conv1");
// (2, 26, 26) -> (2, 13, 13)
Layer#(
  Vector#(2, Vector#(26, Vector#(26, ElementType))),
  Vector#(2, Vector#(13, Vector#(13, ElementType)))
) max_pooling1 <- mkMaxPoolingLayer;
// flatten -> relu -> (338, )
Layer#(Vector#(338, ElementType), Vector#(338, ElementType)) relu1 <-
mkReluLayer;
Layer#(Vector#(338, ElementType), Vector#(10, ElementType)) fc1 <-
mkFCLayer("cnn", "fc1");
Layer#(Vector#(10, ElementType), ResultType) softmax <-
mkSoftmaxLayer;
```

然后再通过 FIFO 进行互相连接：

```
rule put_data if (totalPut + 1 < maxTotal);
  let d <- input_data.get;
  match {.target, .data} = d;
  let target_int = elementToInt(target);
  conv1.put(vec(data));
  ResultType t = truncate(pack(target_int));
  targets.enq(t);
  totalPut <= totalPut + 1;
endrule

rule put_data_max_pooling1;
  let out <- conv1.get;
  max_pooling1.put(out);
endrule

rule put_data_relu1;
```

```
  let out <- max_pooling1.get;
  relu1.put(flatten3(out));
endrule

rule put_data_fc1;
  let out <- relu1.get;
  fc1.put(out);
endrule

rule put_data_softmax;
  let out <- fc1.get;
  softmax.put(out);
endrule

rule get_data_softmax;
  let data <- softmax.get;
  let target = targets.first;
  targets.deq;
  $write("[cnt=%x] Got target: %d, pred: %d, ", cnt, target, data);
  if (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，一次计算时长约为 800 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 结果呈现

由于当前仅有仿真环境，硬件设备上的输入输出并不方便，所以仅进行了仿真环境中的实验。

```
→ chiro@chiro-pc  ~/programs/bsv-cnn git:(master) ✗ make
// ...编译输出...
Elaborated module file created: mkTb.ba
All packages are up to date.
bsc +RTS -Ksize -RTS -steps-max-intervals 10000000 -p
+:/home/chiro/programs/bsv-cnn -p +:/home/chiro/programs/bsv-cnn/gen -
sim -e mkTb -o sim.out
Warning: Command line: (S0073)
  Duplicate directories were found in the path specified with the -p
flag.
  Only the first occurrence will be used. The duplicates are:
    /home/chiro/programs/bsv-cnn
Bluesim object created: mkTb.{h,o}
Bluesim object created: model_mkTb.{h,o}
Simulation shared library created: sim.out.so
Simulation executable created: sim.out

./sim.out -m 10000000 > /dev/stdout
Hello CNN

// ...一些输出...
[cnt=000bd83e] Got target:  5, pred:  5, correct
[cnt=000bdb52] Got target:  5, pred:  5, correct
[cnt=000bde66] Got target:  9, pred:  9, correct
[cnt=000be17a] Got target:  3, pred:  3, correct
[cnt=000be48e] Got target:  4, pred:  4, correct
[cnt=000be7a2] Got target:  3, pred:  5, wrong
[cnt=000beab6] Got target:  5, pred:  5, correct
[cnt=000bedca] Got target:  4, pred:  7, wrong
[cnt=000bf0de] Got target:  7, pred:  7, correct
[cnt=000bf3f2] Got target:  2, pred:  2, correct
[cnt=000bf706] Got target:  2, pred:  2, correct
[cnt=000bfa1a] Got target:  2, pred:  2, correct
```

```
[cnt=000bfd2e] Got target:  9, pred:  9, correct
[cnt=000c0042] Got target:  3, pred:  3, correct
[cnt=000c0356] Got target:  7, pred:  7, correct
Stopping, total:         999, correct:         930, accuracy:
 93 %

make: [Makefile:11: bsv-FC] 错误 1 (已忽略)
```

`make` 开始仿真测试，读取测试集并逐个测试输出，记录正确的值，并计算 Accuracy。

5 总结

本项目使用 Bluespec SystemVerilog 语言实现一个简单的 CNN 模型，并在 FPGA 上进行推理。通过对 CNN 的各层进行抽象，我们可以很方便地定义一个在 FPGA 上进行推理的 CNN 模型。在实验中，我们使用一个简单的两层 CNN 网络，在训练过程中进行定点量化，然后对一个简单的 MNIST 数据集进行测试，得到了 93% 的 Accuracy，与使用 PyTorch 推理得到的结果基本一致。