基于 FPGA 的 CNN 图像分类系统
数字逻辑设计项目报告

计 91　刘程华　刘松铭

清华大学计算机系

日期：2021 年 6 月 30 日

目录

1 项目简介

1.1 项目目标

近年来，人工智能开始显示出其在为人们的生活服务方面越来越大的潜力，但是，其庞大的计算需求使其很难在功能较弱的计算工具中实现。与 CPU 相比，FPGA 具有并行机制，因此速度更快。与 GPU 相比，FPGA 的功耗更低，消耗的能源更少。根据上述背景，我们最初的想法是在 FPGA 上实现卷积神经网络，利用 FPGA 的强大并行特性加速神经网络的计算，并将加速结果与基于 CPU 的神经网络进行对比。但跟助教、老师交流后，发现有两个明显的问题：1、FPGA 资源有限，即使有并行机制，其加速效果未必能比得上 CPU；2、单纯做硬件加速欠缺展示性。于是我们打算对选题进行调整。经过反复与助教的交流后，我们最终听从助教的建议，将“加速”改为“展示”，搭建一个基于 FPGA 的卷积神经网络，并将隐藏层结果输出到屏幕上进行展示。

一句话概括，我们希望基于 FPGA 实现卷积神经网络，并且利用神经网络搭建一个图像分类系统，为用户展示输入图像的分类结果以及神经网络隐藏层的计算中间结果。

1.2 模块划分

我们要完成的事情主要有两件：一是将 CNN 实现在 FPGA 上，确保 CNN 的正确运行，二是将图片输入 FPGA，操控 CNN 完成分类并将结果输出到屏幕上。一种很自然的想法就是分别按照这两件事对模块进行划分。

神经网络我们单独作一个大模块来实现，分为输入层，隐藏层和输出层。神经网络的权重预先在电脑上训练好，转为二进制文件后通过 Quartus 写入片内。

图像输入通过 SD 卡实现。我们预先使用读卡器在 SD 卡写入图像，然后在将 SD 卡插入板上进行图像读写。图像的展示使用 VGA 实现。由于 VGA 渲染需要缓存，因此我们也在 RAM 为 VGA 预留了缓存。值得注意的是，由于我们选取的数据集（下面会详细介绍）的图像大小比较小，且神经网络的权重参数的大小也比较小，因此可以直接使用 RAM 当缓存使用。

本项目有两个重要的接口：输入图像与 CNN 之间的接口以及 CNN 输出与 VGA 之间的接口。对于前者，我们采用 FIFO + valid 的方法。一旦 SD 卡读出了图像，便将 valid 拉高，以流的形式将数据传入 CNN，CNN 用一个 FIFO 接收输出。对于后者，我们采用 Mux + valid 的方法，一旦神经网络得到了稳定的结果，便将 valid 拉高，外界根据用户的指令选择对应的内容输入给 VGA 展示。值得注意的是，SD 卡、神经网络、VGA 的时序都是不同的，需要分别单独编写控制代码。

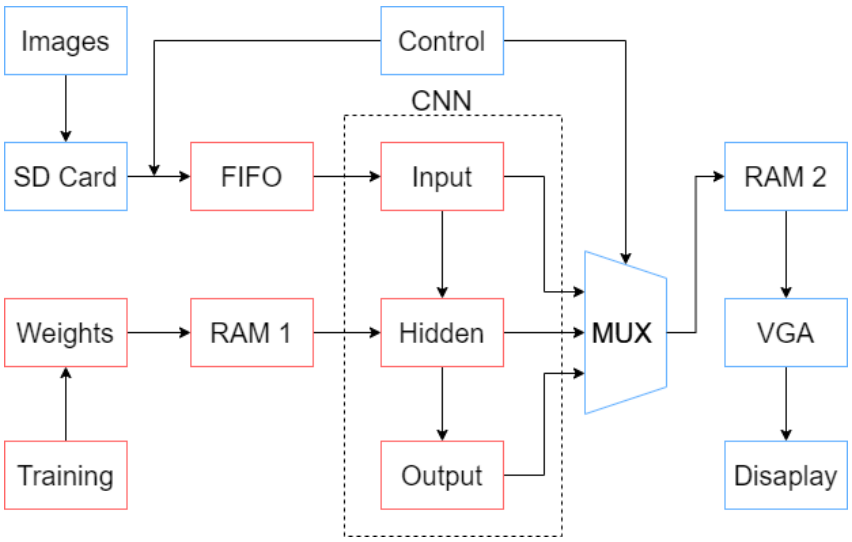

图 1: 模块划分

1.3 分工规划

接口：控制接口朝箭头正边沿进入下一层；

刘程华：蓝色部分

刘松铭：红色部分

这么分工主要原因是 CNN 模块和 SD 卡、VGA 等外设的耦合度比较低，时序相对独立并且接口非常清晰。刘松铭同学负责正确地实现 CNN，确保输入正确的情况下能得到正确的输出。刘程华同学则负责单独调式两个外设，确保两个外设的正确运行。所有模块的联合调试则由两个人共同完成，因为两个人对各自部分的接口比较熟悉，也较能清楚对方是否遵循了接口规定。此外，这么划分也利于各自仿真，CNN 的仿真只需要在电脑上输入图像数据集观察仿真结果，而外设则可以分别单独调试。

1.4 完成情况

各个模块均已完成编写、仿真、调试与上板测试。

2 神经网络结构设计

在设计基于 FPGA 的卷积神经网络之前，我们首先设计了基于 CPU 的卷积神经网络，并且在 PC 上完成了网络结构的定义和权重训练。

2.1 数据集选择

由于 FPGA 资源有限，只能支持结构比较小的网络，相对应地，图像的输入尺寸也不能太大。综合考虑种种因素，我们最终选用 CIFAR - 10 数据集 (图2)。CIFAR - 10 数据集是图像分类研究中使用最为广泛的数据集之一，包含十种不同的类别 60000 张 32×32 像素的彩色图像。十

种不同的类别分别是飞机、汽车、鸟、猫、鹿、狗、青蛙、马、轮船和卡车，每个类别有 6000 张图像。该数据集中图片的大小适合我们的 FPGA 做运算，展示起来也有较好的效果。

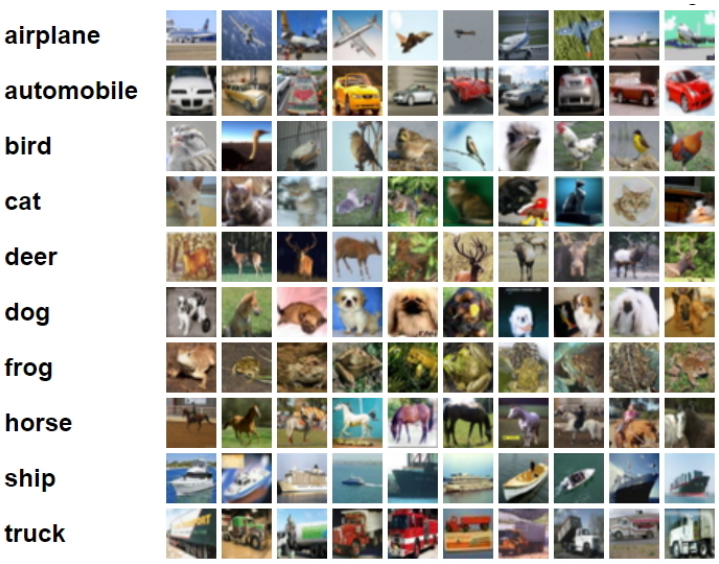

图 2: CIFAR - 10 数据集

2.2 网络结构

卷积神经网络的结构定义如下：

(a) Conv2d(3, 4, 5, 2, bias=False)，输入为 3 通道 RGB，输出为 4 张特征图，卷积核为 5×5，步长为 2。

(b) AvgPool2d(10, 10)，非重叠均值池化，大小为 10×10。

(c) Conv2d(4, 10, 1, bias=False)，输入为 4 张特征图，输出为 10 个分类结果，卷积核为 1×1，步长为 1，目的是做特征升维。

我们的 CNN 由两层卷积和一层池化层构成，对于 CIFAR - 10（$3 \times 32 \times 32$），对应的参数数量为：

Layer (type)	Output Shape	Param
Conv2d-1	[1,4,14,14]	300
AvgPool2d-2	[1,4,1,1]	0
Conv2d-3	[1,10,1,1]	40

表 1: 参数信息

我们的卷积神经网络总共有 340 个参数，在 PyTorch 中默认类型为 float32，估计网络的总大小为 0.02MB，我们的 FPGA 最大为 4MB，预计是能够较好的运行的。

2.3 超参数选择和模型训练

我们使用 SGD 作优化器，设置学习率为 0.001，momentum 为 0.9，进行 20 轮随机梯度下降的训练。得到的模型在 10000 张测试集中有 29% 的准确率。特别地，硬件实现的 CNN 在 Verilator 上仿真也能达到近 30% 的准确率。考虑到我们的网络较小，这个结果还是令人满意的。

3 各模块设计

在这个小节中，我们将简要介绍我们各个模块的设计实现思路。

3.1 VGA 控制

VGA 控制模块按照 VGA 显示器的通信协议，以频率 50MHz 为显示器提供输出的 RGB 像素值。按照 VGA 通信协议，控制器从上到下、从左到右逐行扫描屏幕各点像素。

具体的设计方法，我们参考网络学堂中的 VGA 控制程序例程（800×600，clk 50MHz ）。通过行、列计数器分别进行计数，完成对于屏幕的逐行扫描。

- 初始界面：逻辑绘制（逻辑化简）
- 输入界面：像素 1x1 ⇒ 16x16（位运算）
- 隐藏层界面：像素 1x1 ⇒ 32x32 （位运算）
- 控制系统：根据 ram 完成信号和拨码开关设置

3.2 SD 卡图片加载

3.2.1 测试文件的数据结构

我们将大量的展示图片存放在 SD 中，在展示过程中 FPGA 从 SD 卡中得到图片数据并做运算。我们的图片数据为 CIFAR - 10 的二进制版本 ("http://www.cs.toronto.edu/ kriz/cifar-10-binary.tar.gz")。包含文件 data_batch_1.bin , data_batch_2.bin , ... , data_batch_5.bin 以及 test_batch.bin。这些文件的格式为 “<1 x 标签 > <3072 x 像素 >”，第一个字节是第一张图片的标签，它是 0-9 范围内的数字，0-9 顺次对应的类别和图2中从上到下的类别相同。接下来的 3072 个字节是图像像素的值。前 1024 个字节是红色通道值，接下来的 1024 个是绿色，最后的 1024 个是蓝色。这些值以行优先顺序存储，因此前 32 个字节是图像第一行的红色通道值。每个文件都包含 10000 个这样的 3073 字节 “行” 的图像，尽管没有任何行分隔行。因此，每个文件的长度应恰好为 30730000 字节。我们选用 test_batch.bin 作为我们的展示文件。

我们还要做的是对 test_batch.bin 文件所在的具体位置进行定位，使用 WinHex 软件我们得到该文件的起始位置在扇区 8448/31108096，偏移地址 420000，结束位置在扇区 68467/31108096，偏移地址 35055375。

3.2.2 SD 卡接口

SD 卡支持两种接口，SDIO 和 SPI。其中，SDIO 接口比较复杂且封闭，SPI 则比较自由且资料相对较多，所以我们主要研究的是通过 SPI 来读取 SD 卡中的信息。SD 卡分为标准容量的

SDSC 卡 (小于等于 2GB) 、大容量的 SDHC 卡（2GB ~ 32 GB) 以及更大容量的 SDXC 卡 (32 GB ~ 2 TB), 我们根据的助教建议采取 SDHC 卡。

SD 卡外形和接口图如图 3 所示，SD 卡由 9 个引脚与外部通信，支持 SPI 和 SDIO 两种模式，不同模式下，SD 卡引脚功能描述如下表所示。

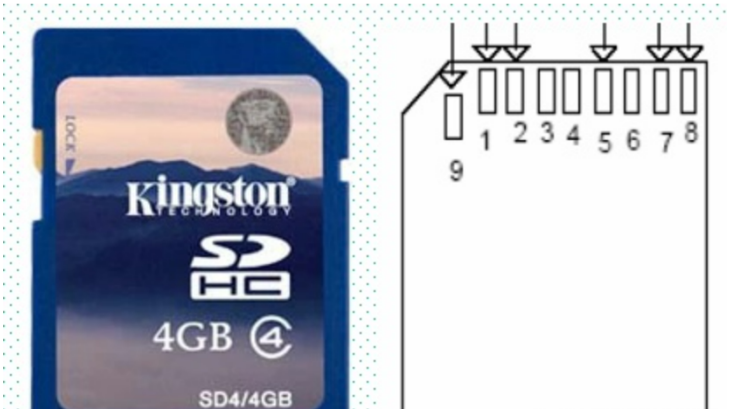

图 3: SD 卡接口

针脚	1	2	3	4	5	6	7	8	9
SD 卡模式	CD/DAT3	CMD	VSS	vCC	CLK	VSS	DATO	DAT1	DAT2
SPI 模式	CS	MOSI	VSS	vCC	CLK	VSS	MISO	NC	NC

SPI 是一种同步串行总线，它根据时钟相位、锁存数据的时机可以分为 $2 \times 2 = 4$ 种模式。SD 卡支持的是模式 0, 即时钟上升沿锁存数据，下降沿发送数据。此外，SPI 收发数据时，最高有效位（MSB）先被处理。SPI 使用了四条数据线:

- SCLK 时钟，主机产生
- SS 或 CS 从机选择，一般是低有效
- MOSI 或 DI 主机发送，从机接收
- MISO 或 DO 从机发送，主机接收，需要上拉电阻

本次实现中，我们使用状态机来产生 SPI 总线的时钟、控制信号以及收发数据，为了方便状态机的设计，我们将产生时钟以及发送和接收 SPI 总线的数据作为了子过程（subroutine），方便其他状态调用。

3.2.3 SD 卡初始化

为了读取 SD 卡中的数据，首先要对 SD 卡进行上电初始化，图 4 是 SD 卡规范中提供的初始化流程。

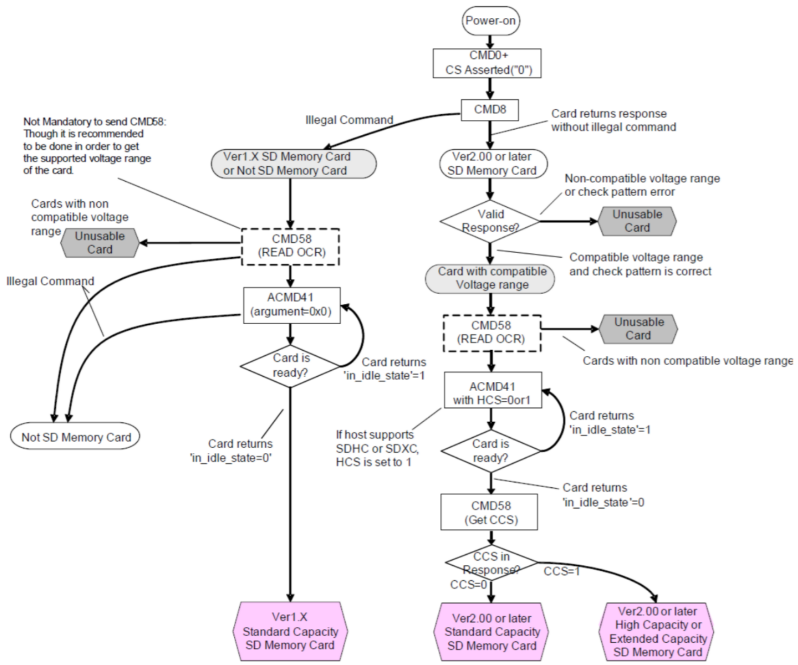

图 4: SPI 模式初始化流程

因为在上电初期，电压的上升过程据 SD 卡组织的计算约合 64 个 CLK 周期才能到达 SD 卡的正常工作电压，这段时间这个叫做 Supply ramp up time，其后的 10 个 CLK 是为了与 SD 卡同步，之后开始 CMD0 的操作，严格按照此项操作，一定没有问题。因此，在 S_INIT 状态下，要发送 10 个字节，才进入等待状态。所以 SD 卡的初始化，需要先给予至少 74 个 CLK 后，这是“Power on”过程需要注意的。

上电后发送 CMD0，对卡进行软复位，之后发送 CMD8 命令，用于区分 SD 卡 2.0。CMD 命令长度固定为 6 个字节，格式见下表，发送时只需要正确构造即可。SPI 模式不会验证命令的 CRC，但还需要发送。SD 卡规范中规定 CMD8 命令的 CRC 校验一直打开，故 CMD8 命令也需要设置正确的 CRC（硬编码即可）。本模块用到的命令的具体参数设置可参见：

“http://wolverine.caltech.edu/referenc/SDSDv17.pdf”

Bit position	47	46	[45 : 40]	[39 : 8]	[7 : 1]	0
Width (bits)	1	1	6	32	7	1
Value	0′	11′	*x*	*x*	*x*	’1’
Description	start bit	transmission bit	command index	argument	CRC7	end bit

3.2.4 SD 卡读操作

SD 卡单块数据块的读取流程是：发送 CMD16 指令，设置数据块大小，等待 CMD 响应（R1）。然后发送 CMD17 指令开始读取数据块，等待 CMD 响应 R1。读完一个数据块的数据完成单块的读取。其中，CMD16 设置数据块的大小，一般为 512 字节，此设置直接决定 SD 卡的块大小，SD 卡默认的块大小自动失效，但对于高容量 SD 卡，块大小固定为 512 字节，不受此指令影响。

对于多块数据的读取流程（图5）差别在于我们用 CMD18 指令来开始读取数据块（响应为 R1），读取 N 个数据块后发送 CMD12 指令（响应 R1）结束多块数据的读取。CMD16、17、18、

12 的指令说明如下 (命令编号，参数，响应，缩写，说明)：

- CMD16；[31:0], 数据地址；R1；R1SET_BLOCKLEN；用于设置 SD 卡的块大小
- CMD17；[31:0], 数据地址；R1；EAD_SINGLE_BLOCK；带一个参数，表示要读的数据块的首地址，快大小由 CMD16 设置
- CMD18；[31:0], 数据地址；R1；READ_MULTIPLE_BLOCK；连续读取多数据块数据知道主机发送 CMD12 指令
- CMD12；[31:0] 无效；R1b；STOP_TRANSMISSION；强制结束当前 SD 的数据传输

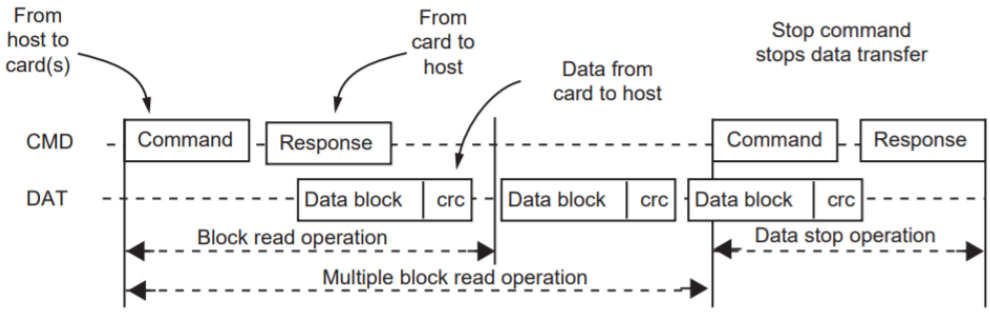

图 5: SD 卡多块读操作

3.2.5 测试文件数据结构的进一步改进

我们最初的方案是将包含 10000 张图片的二进制文件放入 SD 卡中整体通过扇区进行读写。但实际实践过程中，这样的读写方式仍然比较麻烦，具体体现在二进制文件中具体某张图片的定位不够灵活，选择更换图片时需要对 FPGA 进行重新烧写。为了能够改进这一状况，我们在 SD 卡扇区读写的基础上实现了文件读写。可以自动适配 FAT16/FAT32 文件系统，根据文件名在扇区中搜索定位读取文件。这样一来，我们只需对 SD 卡中文件进行修改就能够实现图片的切换。具体的，我们设计在 SD 卡中存在 10 张图片，文件名称分别为 0.txt 到 9.txt（当然可以更多），每个文件的内容和上面 test_batch.bin 中单张图片对应内容是相同的。

3.2.6 FAT16/FAT32 文件系统

FAT(File Allocation Table) 是“文件分配表”，用来记录文件所在位置的表格，它对于硬盘的使用是非常重要的，假若丢失文件分配表，那么硬盘上的数据就会因无法定位而不能使用了。微软在 Dos 和 Windows 系列操作系统前后曾使用了 6 种不同的文件系统，FAT12、FAT16、FAT32、NTFS、NTFS5.0 和 WINFS，现在的 CF 卡以及 SD 卡等闪存卡大部分都可以支持 FAT16 和 FAT32。本次试验中我们实现了对 FAT16 和 FAT32 的支持。

上面已经提到，一个扇区一般为 512 个字节，一个簇则是由若干个扇区组成用来存取数据的最小单位。如果簇大小为 16K，文件大小为 1 字节，那也要用一个簇来存，而且该簇不用再拿来他用。FAT16 使用了 16bit 来描述一个簇，故称之为 FAT16；FAT32 使用了 32bit 来描述一个簇，故称之为 FAT32。FAT16 由于受到最大支持容量和簇大小关系的限制，因此每超过一定容量的分区之后，它所使用的簇 (Cluster) 大小就必须扩增，以适应更大的磁盘空间。这也导致了 AT16 文件系统有两个最大的缺点：

(1) 磁盘分区最大只能到 2GB。FAT16 文件系统已不能适应当前这种大容量的硬盘，必须被迫分区成几十甚至几百个磁盘空间。而分区磁盘的大小又牵扯出簇的问题来，可谓影响颇大。

(2) 使用簇的大小不恰当。试想，如果一个只有 1KB 大小的文件放置在一个 1000MB 的磁盘分区中，它所占的空间并不是 1KB，而是 16KB，足足浪费了 15KB。

以上这两个问题常常使得用户在“分多大的分区，才能节省空间，同时又可使硬盘的使用更加方便有效”的抉择中徘徊不定。为了解决 FAT16 存在的问题，开发出 FAT32 系统。FAT32 使用了 32bit 来表示每个簇。利用 FAT32 所能使用的单个分区，最大可达到 2TB(2048GB)，而且各种大小的分区所能用到的簇的大小，也是恰如其分，上述两大优点，造就了硬盘使用上更有效率。具体的，两者的对比如下：

分区大小	FAT16 簇大小	FAT32 簇大小
16MB-32MB	2 KB	不支持
32MB-127MB	2 KB	512bytes
128MB-255MB	4 KB	512bytes
256MB-259MB	8 KB	512bytes
260MB-511MB	8 KB	4 KB
512MB-1023MB	16 KB	4 KB
1024MB – 2047MB	32 KB	4 KB
2048MB – 8 GB	不支持	4 KB
8GB-16GB	不支持	8 KB
16GB-32GB	不支持	16 KB
32GB 以上	不支持	32 KB

由于 FAT32 是 FAT116 的改进版本，两者的整体结构设计几乎相同，限于篇幅，这里只简单介绍 FAT32 的磁盘结构，见图6。

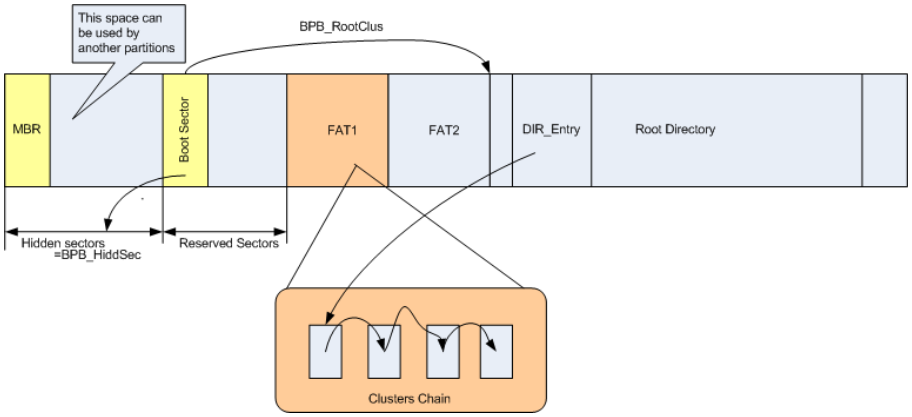

图 6: FAT32 磁盘结构

对于使用 FAT32 文件系统的每个逻辑盘内部空间又可划分为三部分，依次是引导区（BOOT 区）、文件分配表区（FAT 区）、数据区（DATA 区）。引导区和文件分配表区又合称为系统区，占据整个逻辑盘前端很小的空间，存放有关管理信息。数据区才是逻辑盘用来存放文件内容的区域，该区域以簇为分配单位来使用。

FAT32 文件系统的第一个扇区就是引导扇区，其内存放了一个文件系统的很多具体信息，例如 FAT 表个数、每个 FAT 表的大小（扇区数目）、每扇区内的字节数目、每簇中所包含的扇区数目、被保留的扇区数目、文件系统大小（扇区数目）、根目录的起始簇号及一些其它的附加信息。引导区（BOOT 区）从第一扇区（逻辑扇区号 0）开始使用了三个扇区，保存了该逻辑盘每扇区字节数，每簇对应的扇区数等等重要参数和引导记录。之后还留有若干保留扇区，其中有一至三扇区的备份。

文件分配表区（FAT 区）是 FAT 文件系统管理磁盘空间和文件的最重要区域，它保存逻辑盘数据区各簇使用情况信息，采用位示图法来表示，文件所占用的存储空间及空闲空间的管理都是通过 FAT 实现的。FAT 区共保存了两个相同的文件分配表，便于第一个损坏时，还有第二个可用。FAT 表的大小由该逻辑盘数据区共有多少簇所决定，取整数个扇区。数据区中每簇的使用情况通过查找其在 FAT 表中相应位置的填充值可知晓。FAT32 表中每簇占用四个字节（32 位）表示，开头的 8 个字节（0H-07H 字节）用来存放该盘介质类型编号了，因此有效簇号从 02H 开始使用。02H 簇的使用情况由 08H-0BH 字节组成的 32 位二进制数指示出来，03H 簇的使用情况由 0CH 一 0FH 字节组成的 32 位二进制数指示出来，依此类推。未被分配使用和已回收的簇相应位置写零，坏簇相应位置填入特定值 0FFFFFF7H 标识，已分配的簇相应位置填入非零值，具体为：如果该簇是文件的最后一簇，填入的值为 0FFFFFFFH，如果该簇不是文件的最后一簇，填入的值为该文件占用的下一个簇的簇号，这样，正好将文件占用的各簇构成一个簇链，保存在 FAT32 表中。

数据区是被用来存放用户数据的，位于 FAT2 后，同样被划分成簇，从 2 开始编号，即 2 号簇起始位置即是数据区的真正起始位置。

(1) 根目录

通常情况下根目录位于 2 号簇，但是原则上 FAT32 文件系统中的根目录可以位于数据区的任意位置。根目录区保存根目录下的各文件的目录项，每个目录项占用 32 字节。FAT32 文件系统中，根目录作为数据区的一部分，采用与子目录相似的管理方式，这一点与 FAT12、FAT16 明显不同，如 FAT16 文件系统的根目录区（ROOT 区）是固定区域、固定大小的，占用从 FAT 区之后紧接着的 32 个扇区，最多保存 512 个目录项（其根目录保存的文件数受限的原因在此），作为系统区的一部分。

(2) 子目录

FAT32 文件系统中，除了根目录外，全部子目录均在被使用过程中，根据具体的需要而建立。若在根目录之下创建了一个新的子目录，则称该子目录是根目录的子目录，并且称根目录为该子目录的父目录。子目录被新建时，在为其父目录分配的簇中建立目录项，目录项中描述了目录的起始簇号，并且为其在空闲的空间中分配一个簇并清零，将该簇的簇号记录在其目录项之中。为子目录创建目录项的同时，为子目录分配的簇中通过用前两个目录项来记录其与对应父目录的关系。

(3) 目录项

FAT32 文件系统由于结构的不同，一般将目录项分成四种：卷标目录项、“.” 目录项和 “..” 目录项、短文件名目录项、长文件名目录项。短文件名目录项中存放子目录或文件的短文件名、属性、起始簇号、时间值、内容大小等基本的信息。

3.3 RAM 设计

对于输入层，我们设计从 SD 卡读取并储存到 SRAM 中，然后向 CNN 模块发送数据并通过 VGA 展示输入图片。

对于输出层，和输入层的设计类似，我们从 CNN 模块读取处理后像素矩阵，完成读取后通过 VGA 展示输出层。

- Input-ram：3x32x32x8bit；
  vaild-in 拉高传输 8x8bit，共 384 次；传给 vga 时打包成 3x8bit。
- Layer-ram：14x14x8bit；
  valid-out 为高接收 14x8bit，共 14 次。

值得注意的是，一定要确定好数据是否全部接受完成后再发送。两个 ram 的状态机都较为简单，这里不再赘述。

3.4 FPGA CNN

3.4.1 CNN 各层接口设计

CNN 各层接口包括输入层接口，隐藏层（卷积层 1）接口，输出层接口。下面分别就这些接口的设计进行介绍。

(a) 输入层接口

接口设计思路 输入神经网络的是 3 通道，长度为 32 ，宽度为 32 的特征图。也就是一个 $3 \times 32 \times 32$ 的定点数矩阵。定点数矩阵的每一个元素是一个 peak 为 11 位，resolution 为 20 位的定点数（符号位占 1 位），其值是经过归一化（归一化在输入层内完成）后的颜色值。因此外界需要传入的仅是 RGB 颜色值，不需要转换为定点数。归一化公式如下：

$$fixed = \left(\frac{color}{255} - 0.5\right) \div 0.5 \times 2^{20} \tag{1}$$

其中，$fixed$ 是归一化后的定点数，$color$ 指原始颜色值。

接口介绍 在接口中，我们设计了 8 个 8 位的变量接收原始颜色值。此外，我们还设计了 valid 用于指示信号是否有效。发送方要传输特征图时，只需要先拉高 valid ，每个时钟周期按行优先，红绿蓝的顺序传 8 个 32 位的定点数信号。不传数据时拉低 valid 即可。

(b) 隐藏层接口

接口设计思路 卷积层 1 的输出为 4 通道，长度为 14，宽度为 14 的特征图。也就是一个 $4 \times 14 \times 14$ 的定点数矩阵。定点数矩阵的每一位是一个定点数，格式与上文相同。特别地，这些定点数会在 CNN 模块内转换为 RGB 颜色值。转换公式就是公式 (1) 的逆过程，即从 $fixed$ 到 $color$ 的过程。因此外界接收到的是 RGB 颜色值，不会接触到定点数。

接口介绍 在隐藏层接口中，我们设计了 14 个 8 位的变量输出这些 RGB 颜色值。此外，我们还设计了 valid 用于指示信号是否有效。接收方只需在 valid 为高时每个时钟周期接收 14 个信号即可。

(c) 输出层接口

接口设计思路 输出层是最后一层，负责输出神经网络的结果，即类别编号（0 - 9 中的一类）。

接口介绍 在接口中，我们设计了 1 个 4 位的变量输出一个整数值，这个整数值正是分类结果的类别编号。此外，我们还设计了 valid 用于指示信号是否有效。接收方只需在 valid 为高时接收 1 个信号即可。

(d) 其他

我们还设置了一个 9 位的变量，用于输出 CNN 模块一共接收了多少个颜色值。该接口的作用是用于辅助联合调试，没有其他实际的作用。

3.4.2 CNN 实现概述

由于 CNN 的逻辑实现较为复杂，如果仍然采用传统的 Verilog 实现可能需要花费较大的编程精力，时序方面的处理也会比较麻烦。在助教和余泰来同学的推荐下，我们积极尝试了使用 SpinalHDL 进行编写 CNN。在编写 CNN 的过程中，我们采用的是自顶向下的设计思路，顶层模块是 ClassificationNet，该模块封装了整个神经网络。底下的子模块有 ConvolutionBlock 等，分别表示卷积层等结构。模块各自时序独立，有 SpinalHDL 的时钟域进行管理。模块间的时序有 Control 模块和顶层模块负责调度。以下是 SpinalHDL 源文件的简要介绍

- MyTopLevel.scala

  该源文件是 SpinalHDL 的顶层模块，其作用是用于生成 Verilog 代码。

- MyTopLevelSim.scala

  该源文件是用于仿真的模块，其作用是配合 Verilator 将图像测试集导入神经网络进行仿真。

- Net.scala

  该源文件是定义神经网络的模块，该模块确定了神经网络的结构并且调用底层模块（如卷积层模块等）实现了神经网络的功能。并且定义了 FIFO 和必要的数据转换代码完成了对外接口的适配。

- GlobalFunction.scala

  该源文件定义了一些需要用的全局函数。主要有：1、读入图像测试集的函数，目的是为了仿真测试使用；2、读入神经网络权重的函数；3、计数器，用于将 RAM 中的待计算数据和权重调整成适当的顺序给神经网络进行运算。这是实现流水线必备的关键函数，该函数会返回下一个应该读入的地址；4、SearchMax 函数，该函数的目的是从神经网络的 10 个输出中选择一个数值最大的通道作为分类结果。

- LayterCtrl.scala

  该源文件负责对卷积层或池化层的运算时序进行控制，包括控制读入待计算数据和权重的顺序等。

- FeatureMap.scala

  该源文件定义了存储张量的数据结构——特征图。特征图在神经网络中以流的形式传输，一个时钟周期传输一行，并且配有 valid 信号表示数据是否有效。

- WeightMem.scala

  该源文件定义了存储权重的 RAM。

- Relu.scala

该源文件定义了 ReLu 激活层。

- AvgPoolBlock.scala AvgPoolLayer.scala
  该源文件定义了平均池化层的控制逻辑和运算逻辑。
- ConvolutionBlock.scala ConvolutionLayer.scala
  该源文件定义了卷积层的控制逻辑和运算逻辑。

3.4.3 CNN 数据结构

定点数：与 CPU 实现的 CNN 不同，基于 FPGA 实现的 CNN 不采用浮点数而是采用定点数来表示数值。其原因主要是 FPGA 处理浮点数的运算逻辑较为复杂，涉及指数等运算，实现起来不仅代码逻辑复杂而且要消耗大量的逻辑资源。而定点数的计算就如同整数一样简单，天然地被 FPGA 支持。本项目采用的是 1 位符号位，11 位整数位，20 位小数位的定点数。

特征图（FeatureMap）：本项目采用特征图存储 CNN 中的张量，即中间计算结果。对于一个大小为 $H \times W \times C$ 的张量，本项目采用一个长度为 W 的 Flow 存储，也就是每个时钟周期传一行，W 个数，一共传 $H \times C$ 个时钟周期可以传完。

权重 RAM：CNN 运算所需的权重实现写入 RAM 中，当需要计算的时候按一定的顺序从 RAM 中取出。

3.4.4 CNN 数据流

CNN 中的数据流示意图如下，其中每条箭头边上的 x/y 表示每个时钟周期传 x 个数，一共需要传 y 个数。这里的“一共需要传”指的是对于一次图像分类计算这一层需要传的总数据量。对于长方形内的内容，“FindMax”指的是从卷积层输出的 10 个通道中选出数值最大的通道作为分类结果。

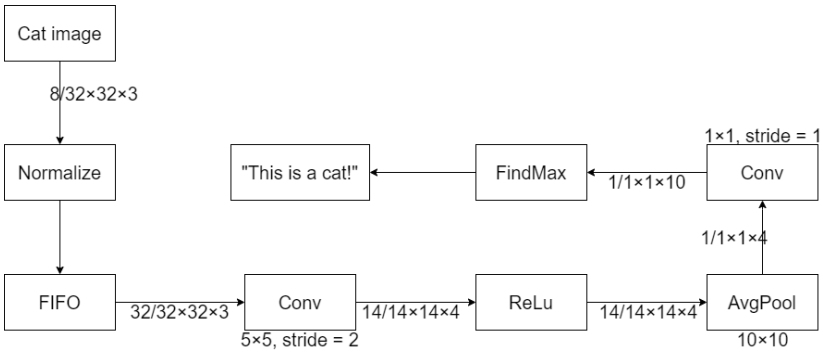

图 7: CNN 数据流示意图

3.4.5 ReLu 层实现介绍

ReLu 层的实现相对比较简单，只需对每一传过来的定点数与 0 比较大小即可。如果比 0 小就赋值为 0，否则保持原来的数不变。

3.4.6 Convolution 层实现介绍 & 流水线介绍

由于卷积层（Convolution）的计算量比较大，并且实时演示对运算速度有一定的要求，因此卷积层采用的是流水线的设计模式。这种方式能最大限度地提高神经网络的吞吐量，加快运算速度。如下是流水线的示意图：

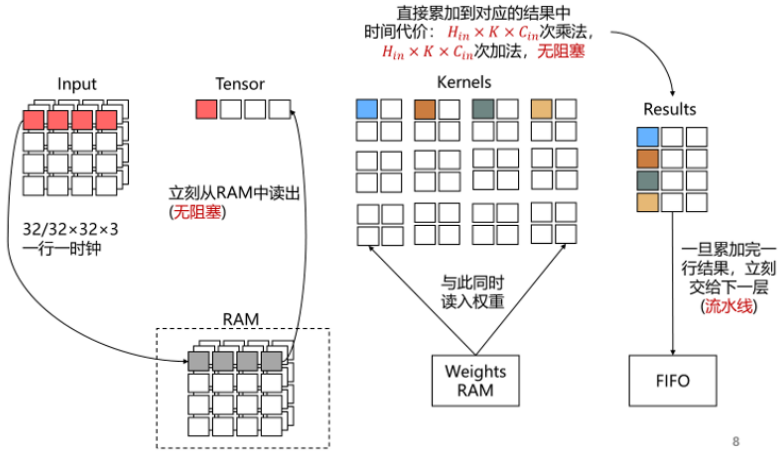

图 8: 卷积层流水线示意图

下面对流水线的设计思路做一个比较详细的介绍。首先是符号的有关约定：

符号约定：假设输入的特征图大小为 $H_{in} \times W_{in} \times C_{in}$，输出的特征图大小为 $H_{out} \times W_{out} \times C_{out}$。假设卷积核大小为 $K \times K$，padding 为 Pad，步长为 $Stride$。其中，C_{in} 由上游的输出决定（这也是卷积层的参数，但是该参数必须与上游的输出通道数匹配），C_{out} 为卷积层的参数，即使用多少个参数独立的卷积核。H_{out}, W_{out} 与 H_{in}, W_{in} 之间有如下关系：

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times Pad - (K-1) - 1}{Stride} + 1 \right\rfloor \tag{2}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times Pad - (K-1) - 1}{Stride} + 1 \right\rfloor \tag{3}$$

输入与输出之间有如下关系：

$$out(C_{out_j}) = \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(k) \tag{4}$$

若写成张量形式，$\boldsymbol{out}(i_0, j_0, c_0)$ 的值是 $\{\boldsymbol{in}(i, j, c) | \forall i \in [Stride \times i_0, Stride \times i_0 + K - 1], \forall j \in [Stride \times j_0, Stride \times j_0 + K - 1], \forall c \in [0, C_{in} - 1]\}$ 与第 c_0 个卷积核做卷积运算的结果。注意，这里的表达式索引是考虑了 padding 之后的索引。

下面介绍流水线的设计思路：

根据数据结构部分的介绍可知，上游传给卷积层的数据是按行优先一行一行（每时钟周期）传过来的，也就是依次传入 W_{in} 个数。当传入这些数后，我们立刻将其存到 RAM 中。等待上一次卷积计算完成后，便从 RAM 中将这行读出进行卷积计算。与此同时从存放有权重的 RAM 中读出对应的权重进行计算。

由公式 (4) 可知，输入 $\boldsymbol{in}(i_1, j_1, c_1)$ 是输出 $\{\boldsymbol{out}(i, j, c) | \forall i \in [i_1 - K + 1, i_1], \forall j \in [j_1 - K + 1, j_1], \forall c \in [0, C_{out} - 1]\}$ 的计算结果的一**部分**。注意，为简化解释起见，这里假设步长 $Stride$ 为

1。由于卷积预算是线性运算，因此我们可以把这部分结果算出来，累加到之前算出的结果中。这样一来，只要每来一行都累加一次，待到累加完成了就可以把结果输出给下一层。

这就是流水线设计的核心思路，只要每来一行输入，对于这一行的每一个数，我们把它与 $K \times K$ 个权重全部相乘得到多个结果，并且分别将其累加到对应的输出结果中（意思是这个数参与了这个输出结果的卷积计算公式的一部分）。

这是最理想的流水线设计，但实际上这样操作也会有问题。主要问题是如果我们想把这些运算在一个时钟周期内完成，则需要相当多的逻辑计算资源。这在我们实际实践中发现是不可行的。因此我们的设计将理想流水线进行了拆分，分为多个周期完成。具体来说，我们不是一次对所有数进行操作，而是一次对 W_{out} 个数进行操作，即每行第 $i, i+K, i+2K, \ldots$ 个数进行操作。那么下一次就是对第 $i+1, i+1+K, i+1+2k, \ldots$ 个数进行操作。尽管进行了这一权衡，但实际上核心设计思路还是一样的。

此外，还有一个值得注意的点。既然我们每一次都计算的是中间结果，那么是否说明需要在这一层的输出加很多缓存呢？实际上并不需要，经过观察不难发现，只要合理的控制输入和权重的顺序，可以使得每一次累加的结果都属于最终结果的同一行（指分属于 C_{out} 个通道的同一行），因此只需要一行的缓存即可。通过计数器可以很好地调整顺序，并且在最终结果累加完成后及时把 valid 拉高。这里的**计数器**就是通过 GlobalFunction.scala 中的计数器类实现的。

3.4.7 AvgPool 层实现介绍

平均池化层（AvgPool）的实现思路与卷积层（Convolution）几乎是完全一致。事实上，平均池化层可以看作是卷积层的一个特例。对于一个 $K \times K$ 的卷积层，卷积核内含有 K^2 个数，如果将每个数对应的权重都设为 $1/K^2$，并且不允许在训练过程中改变，那么卷积层就退化为了均值池化层。

4 遇到的困难及解决方案

4.1 CNN 算术误差大

由于在 PC 上训练 CNN 时采用的是 pytorch 内置的浮点数，而在 FPGA 上实现却用的是定点数，二者之间存在一定的算术误差，所以在实现中 CNN 遇到了算术误差较大的问题。特别地，通过使用 Verilator 仿真输出中间结果发现，随着神经网络深度加大，误差会越来越大，这严重影响了神经网络的预测准确率。我们的测试表明，对于五层以上网络，其预测精度跟随机选择相差无几。既然深度是算术误差的重要因素，因此我们的解决方案便是限制神经网络的深度，将神经网路的深度限制在三层以内。

4.2 FPGA 片内逻辑资源不足

由于神经网络（尤其是卷积层）的运算量较大，对逻辑单元的消耗较多，因此也存在 FPGA 片内逻辑资源不足的困难。我们通过控制变量法对各个超参数进行反复测试，最终发现隐藏层通道数是逻辑资源消耗大的最敏感因素。我们将隐藏层通道数从原来的 10 个通道削减为 4 个，

之后再跟一个 1×1 的卷积层升维得到结果。我们通过这个操作成功将逻辑单元消耗从 145% 降到了 60%。

4.3 计算量太大

神经网络的计算量较大，不仅 CPU 上如此，在 FPGA 上也是如此。如果直接使用串行多周期进行运算，将会极大地降低神经网络的处理速度，进而影响到实验演示。为此，我们在神经网络模块中采用了并行流水线的设计思路，成功地提高了运算速度，使得在实际演示中几乎是立刻可以算出结果。

4.4 SD 卡调试

SD 卡的调试充满艰辛，一开始是无法识别我们的闪迪 SD 卡的型号，在更换金士顿的 SD 卡后问题解决。后来 SD 卡数据读取一直显示完成，由于内部信号非常多，需要一一耐心排查。我们按照 SD 卡的步骤一步步检查对应的信号，最后尝试在文件结尾尝试增加一个 bit，发现问题解决。并由此追根溯源发现了代码中的一个错误。

5 实验心得体会

5.1 刘松铭

在本次实验中，我感受最深的有三点。第一，硬件设计的思路和软件设计完全不一样，需要考虑很多诸如时序这样既复杂又容易出错的问题。并且硬件受到的干扰因素更多，就像我们在做 SD 卡的时候做了半天，最后发现是 SD 卡有问题，后面换了一个 SD 卡才解决。此外，硬件 Debug 也相对而言比较困难，不像软件那样只需要设置断点就可以轻松解决。因此对于硬件而言，仿真就显得尤为重要。能在仿真期间解决的 Bug 一定要在仿真期间解决，等到上板之后就会变得很麻烦了。说到上板，SignalTap 一定是有用的利器，帮助我们发现了不少隐秘的 Bug。第二，了解新的编程语言是非常有帮助的。在本次项目中，我用 SpinalHDL 编写 CNN，体验非常良好。不仅帮我节约了很多时间，（主要体现在丰富的语言特性，精简的语法和良好的时序管理），还帮我避免了很多 Bug 的产生。并且 SpinalHDL 的仿真功能也很强大，结合 Verilator 可以很容易地 print 出任意的中间变量，方便 Debug，极大地提升了我的生活品质。感谢助教和余泰来同学对 SpinalHDL 的安利！

5.2 刘程华

本次实验给我最大的感受三点。一是硬件测试中一定要用好 SignalTap，按照逻辑逐步选择要观察的变量，不能盲目凭感觉任意抓取要观察的变量，这样往往会一头雾水。二是多和队友沟通进度和方案，一开始的很多想法是不够成熟的，和队友的积极沟通往往会出现一方轻微的改进使得另一方任务轻松许多的情况。三是在工作中，一定要先构思框架，做好设计，理清思路，不能急于写代码，这样往往欲速则不达。