南京工業大學

2021 届毕业设计（论文）

题　　目:	深度神经网络硬件加速系统设计与优化
专　　业:	电子信息工程
学　　号:	1402170120
姓　　名:	王磊
指导老师:	朱艾春
起讫日期:	2020.12.01 ~ 2021.06.07

2021 年 6 月

深度神经网络硬件加速系统设计与优化

摘要

近年来，深度神经网络已经被证明在包括图像分类、目标检测和自然语言处理等任务上能够取得相当不错的效果。现如今，大量的应用程序都配备了与之相关的深度学习算法，但是对于手机、无人机等资源有限的嵌入式设备、或者是时常需要处理深度神经网络作业的服务器上，仅以软件优化作为加速深度神经网络运行的手段已经不能满足日益增长的速度和能效要求，如何设计领域专用的深度神经网络加速器已经成为学术领域的研究热点。

在硬件设计层面，本设计主要基于 ZYNQ 7000 器件将英伟达开源的 NVDLA 框架在 FPGA 侧进行了实现与评估，并将其通过 AXI4 总线与双核的 ARM A9 处理器进行互联。软件设计层面，在服务器端，使用 Caffe 框架针对 MNIST、CIFAR10、IMAGENET 训练了三个有精度的分类模型，并结合 TensorRT 进行了 INT8 量化；在主机端，编译神经网络编译器接受训练好的模型生成序列化数据流文件；在硬件端，本设计为 ARM A9 处理器移植了 Ubuntu 操作系统，将硬件加速器的驱动程序挂载到 Linux 内核，通过运行时调度硬件加速器进行推理，分析了该过程中的精度损失情况、与 CPU 相比的运行速度差异，并分析了原因。

关键词：深度神经网络　FPGA　NVDLA　硬件加速

Hardware Acceleration System Design and Optimization of Deep Neural Network

Abstract

In recent years, deep neural networks have been proven to achieve quite good results in tasks including image classification, target detection, and natural language processing. Nowadays, a large number of applications are equipped with related deep learning algorithms. However, for embedded devices with limited resources such as mobile phones and drones, or servers that often need to process deep neural network operations, only software optimization is required. As a means of accelerating deep neural networks, it can no longer meet the increasing speed and energy efficiency requirements. How to design field-specific deep neural network accelerators has become a research hotspot in the academic field.

At the hardware design level, this design is mainly based on the ZYNQ 7000 device. The Nvidia open source NVDLA framework is implemented and evaluated on the FPGA side, and it is interconnected with the dual-core ARM A9 processor through the AXI4 bus. At the software design level, on the server side, the Caffe framework is used to train three accurate classification models for MNIST, CIFAR10, and IMAGENET, and combined with TensorRT for INT8 quantization; on the host side, the neural network compiler is compiled to receive the trained model generation Serialized data stream files; on the hardware side, this design transplants the Ubuntu operating system to the ARM A9 processor, mounts the driver of the hardware accelerator to the Linux kernel, and schedules the hardware accelerator at runtime for reasoning, and analyzes the process of The accuracy loss, the difference in operating speed compared with the CPU, and the reasons are analyzed.

Keywords: Deep neural network; FPGA; NVDLA; Hardware speedup

目　录

摘要 …… I
Abstract …… III
第一章　绪论 …… 1
1.1 研究背景及国内外研究现状 …… 1
1.1.1 神经网络计算机与芯片 …… 1
1.1.2 深度学习处理器 …… 2
1.2 研究意义及前景 …… 2
1.3 研究内容及结构安排 …… 2
第二章　相关技术介绍 …… 5
2.1 深度神经网络概述 …… 5
2.1.1 感知机模型 …… 5
2.1.2 卷积神经网络概述 …… 8
2.1.3 LeNet5 介绍 …… 8
2.1.4 Resnet 18 网络结构 …… 10
2.2 FPGA 与 ZYNQ 器件 …… 10
2.3 神经网络硬件加速平台 …… 11
2.3.1 众核处理器 …… 11
2.3.2 GPU …… 12
2.3.3 Google TPU …… 12
2.3.4 DianNao 系列 …… 12
2.3.5 NVIDIA Deep Learning Accelerator …… 13
第三章　系统总体设计 …… 15
3.1 系统设计框图 …… 15
3.2 硬件设计架构与互联 …… 16
3.3 验证平台 …… 17
3.4 用户友好的输入页面设计方案 …… 18

第四章 硬件设计实现 …… 19
4.1 Xilinx FPGA 设计套件 …… 19
4.2 NVDLA 硬件设计概述 …… 19
4.2.1 NVDLA 硬件结构分析 …… 19
4.2.2 NVDLA 自定义配置 …… 21
4.3 NVDLA IP 生成描述 …… 21
4.3.1 基于 *make* 的 RTL 代码生成 …… 21
4.3.2 RAM 优化 …… 22
4.3.3 控制总线协议优化与封装 …… 23
4.4 Block Design 设计 …… 24
4.4.1 APB to AXI Bridge …… 24
4.4.2 AXI Smart Connect …… 24
4.4.3 ZYNQ 7000+ AP SOC …… 25
第五章 软件设计实现 …… 27
5.1 NVDLA 软件工具链概述 …… 27
5.2 Caffe 与模型训练 …… 29
5.3 TensorRT 与模型量化 …… 30
5.3.1 TensorRT 量化原理 …… 30
5.3.2 量化步骤与精度损失 …… 31
5.3.3 calibtabel 生成 …… 31
5.4 Ubuntu 16.04 嵌入式操作系统移植 …… 32
5.4.1 Petalinux 工具介绍 …… 32
5.4.2 针对硬件设计的启动文件制作 …… 32
5.4.3 替换根文件系统 …… 33
5.5 KMD 内核程序挂载 …… 33
5.5.1 Linux 设备树新增节点 …… 34
5.5.2 Petalinux 内核添加应用 …… 34
5.5.3 驱动程序加载 …… 34
5.6 UMD 应用程序编译 …… 35

5.6.1 LIBJPEG 链接库编译 …… 35
5.6.2 UMD 构建 …… 35
5.6.3 Runtime 测试 …… 35
第六章 测试与分析 …… 37
6.1 资源利用率 …… 37
6.2 最大工作频率分析 …… 38
6.3 基于 NVDLA 的硬件加速系统性能评估 …… 40
6.3.1 实验配置 …… 40
6.3.2 精度对比分析 …… 40
6.3.3 推理速度分析 …… 41
第七章 总结与展望 …… 43
参考文献 …… 45
致谢 …… 47
附录 A …… 1
A.1 *small.spec* …… 1
A.2 *Nv_nvdla_wrapper.v* …… 2
A.3 *Lenet5* 量化关键代码 …… 4
A.4 *Cache to Json* …… 5
A.5 *system-user.dtsi* …… 6
A.6 *Makefile* …… 6
A.7 *opendla.bb* …… 7

第一章　绪论

1.1　研究背景及国内外研究现状

总的来说，深度神经网络的硬件加速系统设计的研究可分为两个阶段，一是上世纪 50 年代第一次人工智能热潮开始到第二次人工智能热潮结束期间，针对感知机和多层感知机等浅层神经网络研发的神经网络计算机与芯片；二是针对第三次人工智能热潮中的深度神经网络设计的深度学习处理器。

1.1.1　神经网络计算机与芯片

在第一次人工智能热潮中，D. Hebb 提出了 Hebb 学习法则，在这之后不久的 1951 年，M. Minsky 研制出了国际上首台神经网络模拟器 SNARC；1957 年，F. Rosenblatt 提出了感知机模型，随后国际上首台基于感知机的神经网络计算机 Mark-I 就在 1960 年被研制出来，它可以连接到照相机上使用单层感知机完成简单的图像处理任务。

在第二次人工智能热潮中，著名的反向传播算法被提出使得神经网络的研究取得了一些重要突破。20 世纪 80 年代和 90 年代初，很多大公司、创业公司和研究机构都参与到了神经网络计算机/芯片的研制，包括 Intel 公司研发的 ETANN[1]、1990 年发布的 CNAPS[2]、基于脉动阵列结构的 MANTRA I[3]，以及 1997 年由中国科学院半导体研究所研发的预言神[4] 等。

然而，从当今深度学习技术发展的角度来看，这些早期的神经网络计算机/芯片有诸多缺陷，由于其只能处理很小规模的浅层神经网络算法，所以没有在工业实践中获得广泛应用。这一方面是因为浅层神经网络的应用比较局限，缺乏像当下的诸如目标检测、自然语言处理等领域的核心应用；另一方面，当时的主流集成电路工艺还是 1 微米工艺（今天的主流集成电路工艺已经达到 7 纳米），在一个芯片上只能放数量相当少的运算器，Intel 的 ETANN 芯片中仅能集成 64 个硬件神经元；最后，受限于当时的计算机体系结构技术还没有发展成熟，没有办法将大规模的算法神经元映射到少数的硬件神经元上。

而第二次人工智能热潮也随着日本的五代机计划失败而结束。基于上述原因，从 20 世纪 90 年代开始，神经网络计算机/芯片的创业公司纷纷破产，大公司也裁减掉了相关的研究部门，各个国家暂停了这方面的科研资助，但这些瓶颈在处于第三次人工智能热潮的今天都得到了解决和改善，于是深度神经网络的硬件加速系统设计进入了第二个阶段。

1.1.2 深度学习处理器

2006 年，深度学习技术由 G. Hinton、Y. Bengio 和 Y. LeCun 等人的推动而兴起，于是有了第三次人工智能热潮。而深度学习处理器也在这种环境下重新焕发了生机，在 2008 年，中国科学院计算技术研究所的陈云霁、陈天石等人开始了人工智能和芯片设计的交叉研究，之后来自法国 Inria 的 O. Temam 也参与到可项目中。在这些人的推动下，国际上第一个深度学习处理器架构 DianNao 于 2013 年被设计出来。和第一阶段设计的神经网络计算机/芯片不同，DianNao 架构不会受到网络规模的限制，可以灵活、高效地处理上百层的深度学习神经网络，并且显得更对于通用的 CPU，DianNao[5] 可以取得两个以上数量级的能效优势。2014 年，陈云霁等人在 DianNao 架构上改进，设计出了国际上首个多核的深度学习处理器架构 DaDianNao[6] ，以及机器学习处理器架构 PuDianNao[7] 。进一步，中国科学院计算技术研究所提出了国际上首个深度学习指令集 Cambricon[8] 。在这之后，首款深度学习处理器芯片“寒武纪 1 号”问世，目前由寒武纪研发的深度学习处理器已经应用于超过一亿台嵌入式设备中。

1.2 研究意义及前景

深度神经网络已经被证明在包括图像分类、目标检测和自然语言处理等任务上能够取得相当不错的效果。而随着其层数和神经元数量以及突触的不断增长，CPU 和 GPU 等传统体系已经很难满足神经网络增长的速度和需求。例如 2016 年，Google 公司研发的 AlphaGo 与李世石进行围棋对弈时使用了 1202 个 CPU 以及 176 个 GPU[9] ，在这场比赛中 AlphaGo 每盘棋需要消耗上千美元的电费，而作为人类的李世石一盘棋的功耗仅需要 20 瓦，从此可以看出传统芯片的速度和能效难以满足大规模深度学习应用的需求。如今，大量的应用程序都配备了与之相关的深度学习算法，但是仅以软件优化作为加速深度神经网络的手段已经不能满足日益增长的速度和能耗要求，基于深度神经网络的智能应用需要广泛普及的趋势使高性能、低功耗的深度学习处理器研发呼之欲出，如何利用硬件设计加速器已经成为学术领域的研究热点。

1.3 研究内容及结构安排

本设计首先参考 CNNIOT 使用 HLS 构建了一种通用的卷积神经网络硬件加速器，实现了 Lenet5 网络的加速，并使用 Bootstrap 设计了用户友好的交互页面。而在分析了 HLS 设计的硬件加速器的缺点之后，本设计重点将英伟达开源的 NVDLA 框架映射到 FPGA 开

发板卡上。

硬件设计层面，本文主要在 FPGA 平台上设计实现了 NVDLA 深度学习加速器，并将其接口封装为 AXI4 总线协议与双核 ARM A9 处理器互联。软件设计层面，在前端，使用神经网络编译器接受预训练的神经网络模型，并做硬件无关优化。在后端，将驱动程序挂载到了 Ubuntu 操作系统上，由 Runtime 自动调度推理。最后，分析了所设计的硬件加速器在 FPGA 上的最高工作频率与精度损失，与 ARM A9 处理器的推理速度进行了对比。

本文共设有七个章节，各章节的内容安排如下：

第一章：主要介绍了课题的研究背景，结合国内外研究现状，提出本设计的研究意义与研究方向，并对本文的内容做基本介绍。

第二章：主要对本文涉及到的相关技术和知识背景进行介绍，首先简要介绍了深度神经网络算法的发展过程，其次对 FPGA 与 ZYNQ 架构进行了简单的概述，最后对现有的神经网络硬件加速平台及其应用进行了介绍。

第三章：根据系统设计框图介绍了系统的设计思路，包括用户友好的设计页面的演示与硬件设计架构与互联方式，对比了 CNNIOT 与 NVDLA 两个架构设计，最后简单分析了验证平台的选型。

第四章：对系统的硬件设计实现进行了详细介绍，包括了 NVDLA IP 的生成与 Block-Design 设计两个部分，简单讲述了一下静态时序分析方法。

第五章：对系统的软件设计实现进行了详细介绍，包括结合 TensorRT 进行的量化方案，为板卡移植 Ubuntu 的根文件系统，以及 Runtime 的编译。

第六章：对设计完成的 IP 进行测试与分析，主要包括基于静态时序分析得出 NVDLA 工作的最大频率，针对预训练的三个网络模型的精度损失情况以及推理速度分析。

第七章：总结研究内容，分析了本次设计的不足，以及其中可以进一步研究与改进的方向进行讨论。

第二章　相关技术介绍

由于本硬件加速系统设包含了软件与硬件的协同设计，所以本章将对涉及到的关键技术，包括深度神经网络算法、ZYNQ 器件做基本的介绍。如第一章节所述，本章还会对不同类型的硬件加速体系结构进行基本概述，介绍对应的应用案例。

2.1　深度神经网络概述

2.1.1　感知机模型

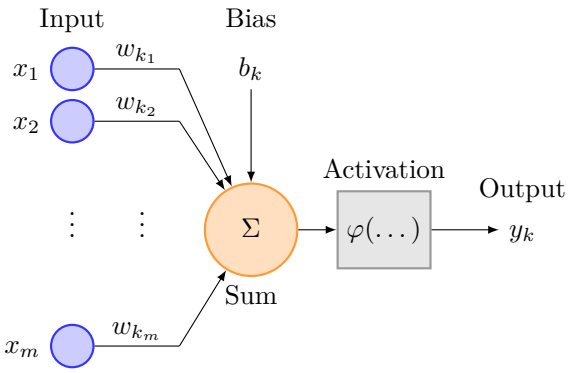

图 2-1 神经元模型

一个神经元的模型如图 2-1所示，该模型也被称为单层感知机，由输入信号、权值、偏置、加法器和激活函数共同构成，其中 w_{kj} 下标的含义为：k 表示第 k 个神经元；j 表示第 j 个输入。因此，w_{kj} 表示第 k 个神经元的第 j 个输入对应的权值。单个神经元的数学公式如公式 (2-1) 所示：

$$\begin{cases} u_k = \sum_{j=1}^{m} w_{kj} x_j \\ v_k = u_k + b_k \\ y_k = \varphi(v_k) \end{cases} \tag{2-1}$$

其中 x_j 表示第 k 个神经元的第 j 个输入，w_{kj} 表示第 k 个神经元的第 j 个输入对应的权值，b_k 表示第 k 个神经元的偏置，$\varphi(\dots)$ 为激活函数，y_k 为该神经元的输出。

感知器模型于 1958 年，由美国心理学家 Frank Rosenblatt 提出[10]，其中激活函数采用的一般是符号函数，其如公式 (2-2)所示：

$$o = sgn(x_1 w_1 + x_2 * w_2 + b) \tag{2-2}$$

进一步，为了简化问题分析本质，我们假设神经元只有两个输入: x_1 和 x_2 ，则模型的公式进一步简化为:

$$\begin{cases} 1, & x_1 w_1 + x_2 * w_2 + b >= 0 \\ -1, & x_1 w_1 + x_2 * w_2 + b < 0 \end{cases} \tag{2-3}$$

对该公式进行分析，我们可以把 o 当作因变量、x_1 和 x_2 当作自变量，对于分界

$$x_1 w_1 + x_2 w_2 + b = 0 \tag{2-4}$$

可以抽象成三维空间里的一个分割面，能够对该面上下方的点进行分类，则该感知机能完成的任务是用简单的线性分类任务。因为在该公式中，y 的结果只有 1 和-1 两个值，可以分别使用实心点和空心点来表征，这样就可以在二维平面上将问题可视化，表 2-1所示为使用感知机模型实现逻辑“与”和逻辑“或”。

表 2-1 逻辑真值表

逻辑与			逻辑或			逻辑异或		
x_1	x_2	o	x_1	x_2	o	x_1	x_2	o
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

但是对于非线性问题，如异或问题，如图 2-2所示，单层感知机没有办法找到一条直线能够完成分类任务。

为了弥补单层感知机的不足，多层感知机模型在之后被提了出来，其模型如图 2-3所示，在原有的输入层和输出层中间增加了一层或多层的隐藏层，隐藏层与隐藏层之间的神经元、隐藏层和输入层的神经元、隐藏层和输出层的神经元都是完全连接的，以这种方式进行连接的神经网络层也叫全连接层。

但是，即使增加再多的隐藏层，其能够达成的效果依然与仅有一层隐藏层的感知机等价，问题的根源在于多个仿射变换的叠加仍然是一个仿射变换，对于该问题的解决方法是引入非线性变换，及将激活函数置换成非线性函数[11]。这个非线性函数被称为激活函数（activation function），常用的激活函数包括了 Sigmoid、Relu、Tanh 等。

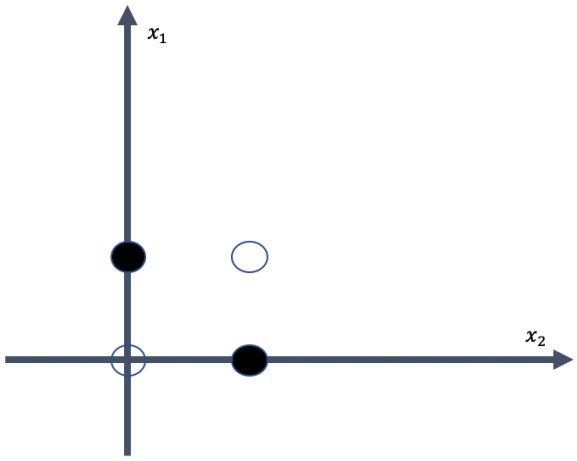

图 2-2 异或平面

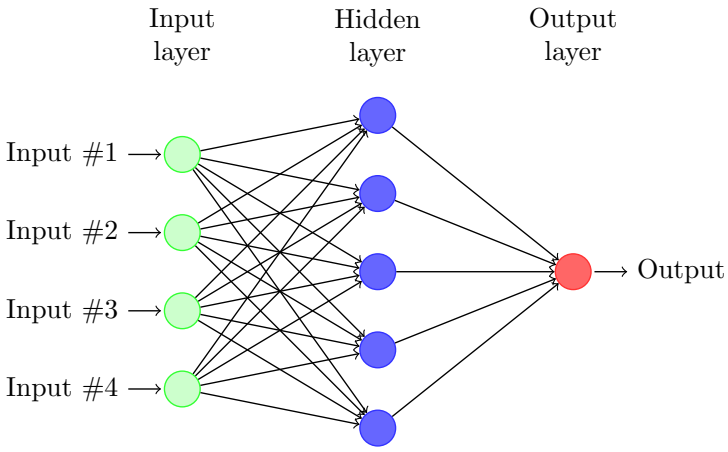

图 2-3 前向神经网络结构

理论上，深度神经网络，及多层感知机已经可以拟合任意的函数[12]。

2.1.2　卷积神经网络概述

卷积神经网络（Convolutional Neural Network, CNN）是目前最常用的深度神经网络，对于大型图像处理有出色表现，卷积神经网络一般由卷积层、非线性激活函数、池化层组成。

1. 卷积层（Convolution Layer）的本质是输入图像与权重矩阵的乘积累加运算 (Multiply Accumulate, MAC)，如图 2-4所示卷积核在输入特征图像上滑动，用来匹配局部的细节。一般来说，随着卷积的层数逐渐加深，卷积核的数目也会随之增加，所表征的细节也会更加抽象。

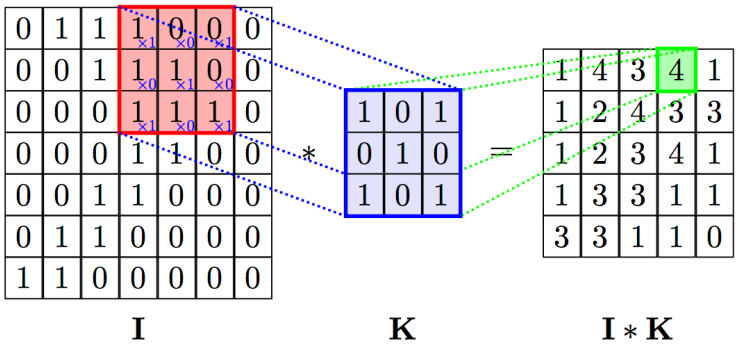

图 2-4 卷积运算

2. 激活函数（Activation Function）是非线性单元，它的存在给深度神经网络体系增加了非线性元素，是深度神经网络能够拟合任意函数的基础。比较重要且常用的激活函数有 ReLU、Sigmoid 和 Tanh 。此外，其余常用的激活函数还有 PReLU, ELU 等，他们的函数图像与取值范围如图 2-5所示：

3. 池化层（Pooling Layer）的作用是对特征图进行下采样，从而可以达到增加感受野、增加卷积神经网络运算的平移不变性、降低优化难度，减少神经网络参数的目的。典型的池化层 Max Pooling 如图 2-6所示，他会选择被池化矩阵选内最大的值作为输出得到新的特征图。Average Pooling 选取池化矩阵内的平均值，是另一种常用的池化层。

2.1.3　LeNet5 介绍

手写字体识别模型 LeNet5[13] 诞生于 1994 年，是最早的卷积神经网络之一。

利用卷积、参数共享、池化等操作提取特征，LeNet5 避免了大量的计算成本，最后再使用全连接神经网络进行分类识别，该网络也是最近大量神经网络架构的起点，LeNet5 的网络结构如图 2-7所示。

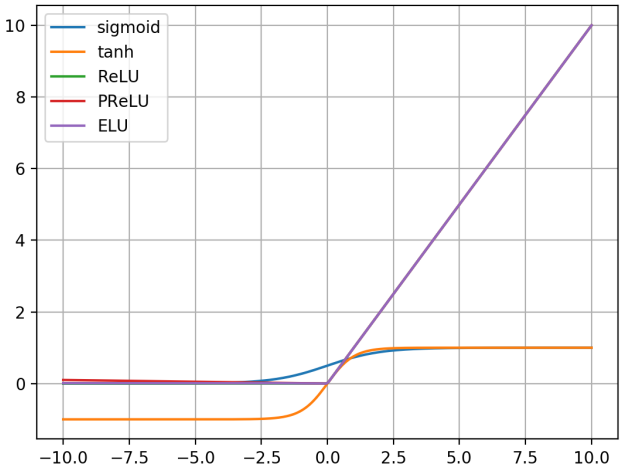

图 2-5 常见的激活函数图像

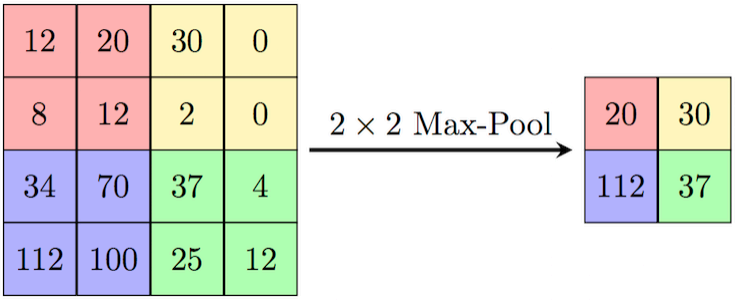

图 2-6 Max Pooling

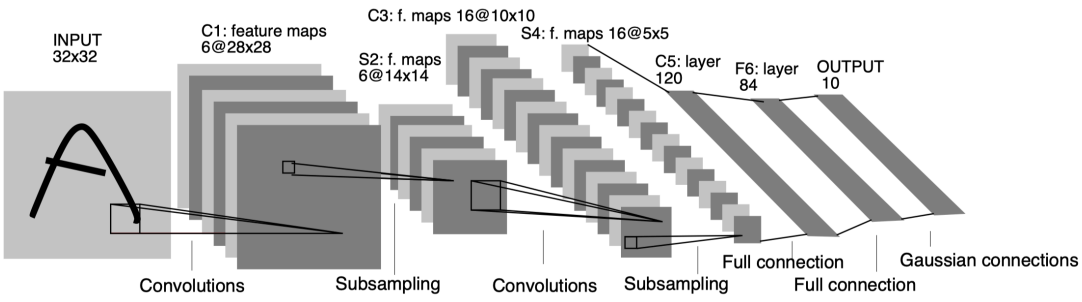

图 2-7 LeNet-5 网络结构

2.1.4　Resnet 18 网络结构

纵观神经网络的发展脉络，我们可以看到研究者一直在不断尝试各种技术以加深神经网络的层数。从 8 层的 AlexNet，到十九层的 VGG，再到 22 层的 GoogLeNet，可以说，技术一直在进步。但是在突破神经网络深度的问题上真正最具有颠覆性的技术还是来自 ResNet[14]。

为了避免随着神经网络的深度加深从而带来的退化问题，ResNet 采用了一种不同于常规卷积神经网络的基础结构，其基本单元如图 2-8所示，增加了从输入到输出的直连通道。其卷积拟合的是输出与输入的差，及残差，所以这种结构又被称为残差结构。残差网络的响应小于常规网络。残差网络的优点是对数据波动更加灵敏，更容易求的最优解，能够改善深层网络的训练。

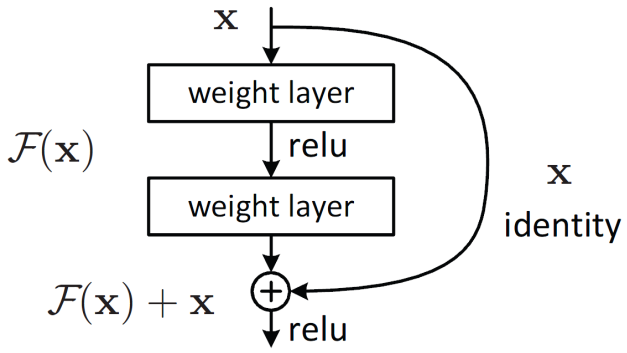

图 2-8 Resnet 残差结构

2.2　FPGA 与 ZYNQ 器件

现场可编程逻辑门阵列（Field Programmable Gate Array, FPGA）, 是一种半定制的集成电路（Integrated Circuit, IC）芯片。根据数字电路所学，查找表（Look Up Table，LUT）理论上可以实现任意的逻辑，FPGA 利用这种思路从而可以实现不同逻辑特性，通过重复烧写改变 PLB 的连接方式实现了可重构计算。因此理论上 FPGA 可以实现任意的数字逻辑，该特性为短周期内设计可使用的硬件加速系统提供了可能性。

ZYNQ 器件是 Xilinx 公司研发的以 ARM 为主导，FPGA 为辅的嵌入式系统结构，其拥有集成度高的开发环境，支持高层次综合、Vivado 硬件逻辑设计、Simulink 协同设计、SDK 软件设计。对于一个不熟悉 FPGA 的软件工程师来说，也完全可以把 ZYNQ 器件当前简单的双核 ARM 来使用，利用 SDK 编写软件程序，如果软件调试过程中发现某些算法的速度太慢，这时候就可以使用 Verilog 硬件描述语言，或者使用高层次综合为这部分算法设计硬件加速器，ARM 处理器与加速器之间通过 AXI 标准总线协议进行通信，Xilinx

提供了若干免费的加速 IP 供用户使用，由于有了 ARM 处理器，我们甚至可以在 ZYNQ 器件上搭建 SOC 降低开发的难度。

2.3 神经网络硬件加速平台

在过去，深度神经网络算法往往被运行在通用体系结构里，如 CPU、众核处理器与 GPU 设备，但是这些设备的特点是运行功耗大，并且会带来极高的内存使用率。当谈及深度神经网络，往往最让我们头疼的是训练过程，这个步骤一般在使用 GPU 完成加速，然后将预训练好的模型部署在目标平台，当平台对功耗和运算能力有要求的时候，传统的通用处理器的体系结构就不适合了，于是领域专用的，针对深度神经网络的硬件加速系统设计呼之欲出。本小节首先介绍两种典型的通用硬件加速体系结构，然后介绍三种面向深度神经网络运算的领域专用运算。当然，除了针对深度神经网络的专用体系结构设计，上层的软件设计也有对应的加速方案，如 NNPACK、Intel MKL、CuDNN 等针对通用体系结构来优化神经网络算子的解决方案，他们与硬件无关的部分在某些程度上对专用体系结构也是适用的。

2.3.1 众核处理器

众核处理器（Manycore Processor）由成百上千的简单分立的处理器核心组成。与传统的多核处理器所不同的是，其设计牺牲了单核运行的性能来换取多核处理时的吞吐量，减少多核处理时的能量损耗。典型的使用众核处理器设计是 Kalray MPPA-256 架构。如图 2-9所示，使用众核处理器进行深度神经网络运算时，每个处理器核心都可以贡献出一部分性能进行神经网络的运算，之后通过 Router 决定在物理总线上数据传输的方向。

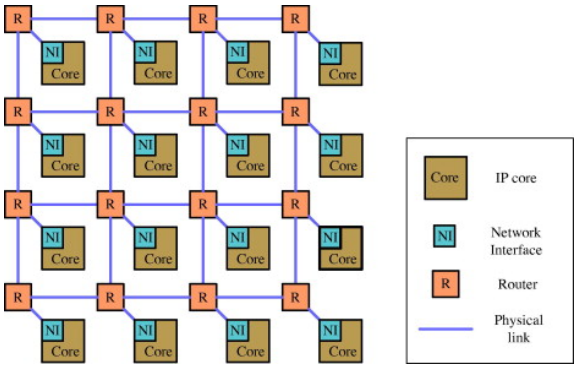

图 2-9 Net work on Manycore Processor

众核处理器是一种通用处理器，这意味着他具有一定的灵活性，除了可以运用在神经网络运算上，还可以用在各种各样的领域，尤其是那些有高并行度要求的场景，例如 HPC。

但如果仅针对深度神经网络的推理，通用处理器本身便会存在一些多余、不合适的设计。

2.3.2　GPU

GPU 也可以看作是一种特殊的，针对向量运算的众核处理器。起初 GPU 被设计出来的目的是为了提高计算机对图像与视频的处理能力，但是当下，GPU 设备被广泛应用在深度学习应用的训练和推理加速中。并且对于 GPU 的调用，NVIDIA 还提供了软件支持，即 CUDA，方便了用户手动便携并行算子。

VOLTA 架构是 NVIDIA 推出的新一代的 GPU 设计架构，基于该架构的显卡被广泛运用在深度学习领域。其中，TESLA V100 是第一款被设计出来专门针对深度神经网络训练和推理的 GPU 设备（如图 2-10）。

图 2-10 TESLA V100

2.3.3　Google TPU

张量处理单元（Tensor Processing Unit）是 Google 牵头设计的一款为深度神经网络运算定制的 ASIC 器件，事实上其也取得了非常瞩目的性能，2015 年发布的 29nm 工艺的 TPUv1 版本，能够运行在 700Mhz 的频率，但功率只有 28W ~40W[16]。

2.3.4　DianNao 系列

为了能够实现大规模深度学习网络的专用硬件执行。2014 年，中国科学院计算技术研究所研究团队提出了 DianNao 架构。DianNao 这篇文章是率先探索机器学习加速设计的先驱文章之一，开创了专用处理器实现深度学习的先河。这篇文章在 65nm 工艺，0.98Ghz 的频率，面积为 3.02mm^2 的 ASIC 芯片上针对机器学习算法（DNN，CNN）实现了一个高性能的 Diannao 处理器架构，相比于 128bit 2GHz 的 4 发射 SIMD 处理器，达到了 117.87x 的加速比，21.08x 能耗比[5]。

2.3.5 NVIDIA Deep Learning Accelerator

NVDLA 是 NVIDIA 公司开源出来的深度学习加速器框架，其结构与 DianNao 类似。DianNao 系列芯片每个周期最多能够计算 16 个神经元，NVDLA 能够计算 64 个，而 TPU 则能够计算 256 个。

NVIDIA 在 2019 年 12 月发布了 Jetson Xavier NX 套件，如图 2-11所示，该套件上集成了已经流片的 NVDLA 为深度神经网络推理提供硬件加速支持。

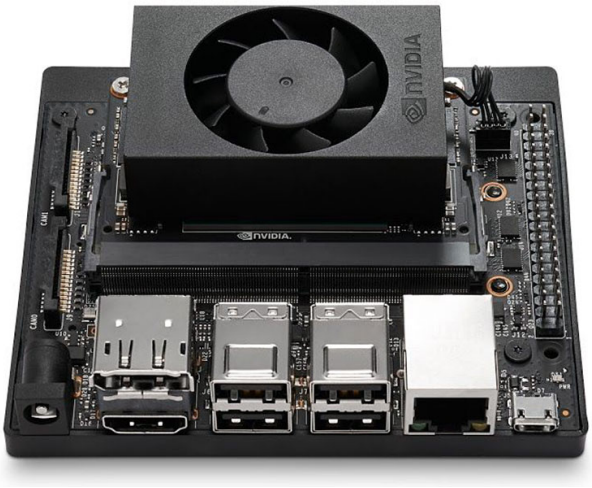

图 2-11 Jetson Xavier NX

与前两个领域专用的硬件加速体系结构不同，NVDLA 完全开源，其不仅提供了 Verilog 硬件描述、CMOD 硬件仿真程序，还为硬件加速器设计了神经网络编译器与运行时，自上而下打通了硬件栈与软件栈，基本能够实现端到端的推理，非常规范。虽然 NVDLA 自 2018 年以来已经没有人维护，但其系统设计思路仍然具有学习意义和指导意义。除了英伟达官方将 NVDLA 进行流片并嵌入到 Jetson 内以外，还有不少研究机构、商业公司也将 NVDLA 研究并成功流片，但是在 FPGA 端进行验证设计官方没有公开的项目做指导，本设计将原本面向 ASIC 设计的 NVDLA 映射到 FPGA 器件进行验证与设计，而有关 NVDLA 的内容将在之后的章节具体介绍。

第三章　系统总体设计

在进行系统的设计之前需要了解整个设计的流程，所以本章首先介绍系统设计的框图；除此之外，还将介绍本设计的硬件架构设计方案，并在该小节比较了基于 HLS 设计的硬件加速结构 CNNIOT 相对于 NVDLA 的缺陷；在本文的任务中，还要考虑到资源使用的情况，所以本章还介绍了硬件验证平台的选择；最后，为了方便演示，介绍了本设计的演示页面。

3.1　系统设计框图

本设计的系统的总体设计结构如图 3-1所示分为四个阶段：

1. 在服务器端，基于 GPU 使用 Caffe 框架训练模型，又因为本设计的硬件加速系统仅支持 INT8 模型的推理，还结合 TensorRT 完成了模型的量化。

2. 在主机端，使用深度神经网络编译器将已经量化好的模型进行编译与优化，将模型的参数、数据等信息进行序列化压缩，方便传输。

3. 在 ARM 处理器上，使用深度神经网络运行时将序列化文件进行反序列化，通过 AXI4 总线与硬件加速器 NVDLA 进行通信。

4. 在 FPGA 上，实现了 NVDLA 的硬件加速器系统，其通过 AXI4 总线协议与外部通信，其内部包含五种引擎（NVDLA 有六个引擎，但因为本设计没有使能第二高速存储，所以 BDMA 引擎并未在本设计中被实现）。

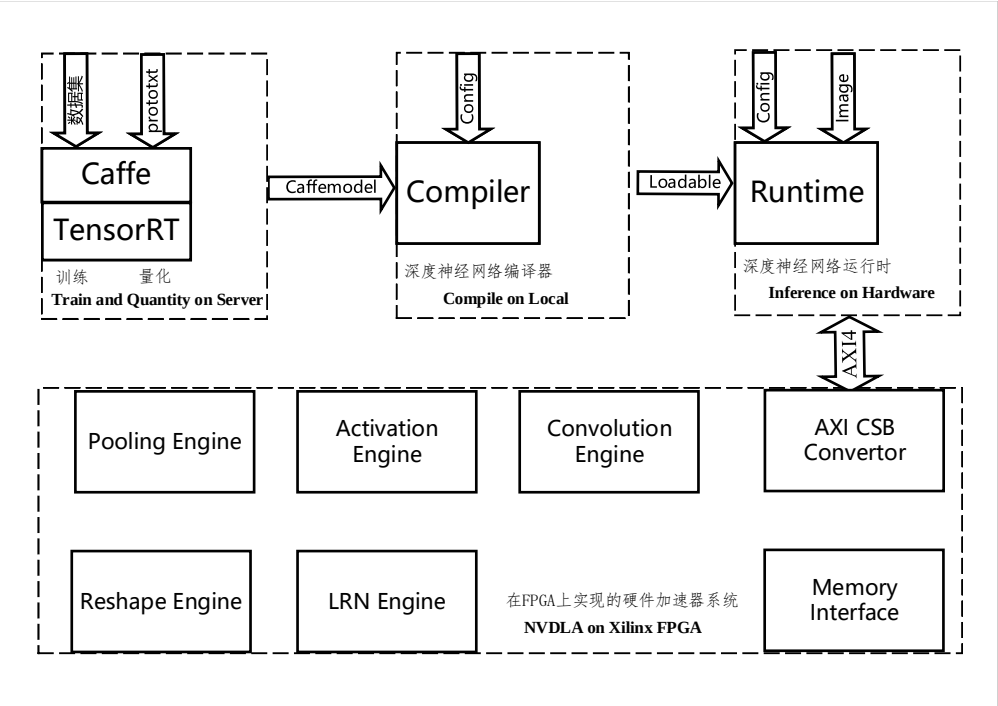

图 3-1 硬件总体设计架构

3.2　硬件设计架构与互联

本设计的硬件总体架构设计如图 3-2所示，双核的 32 位 ARM A9 是本设计的主控制器，通过 AXI4 总线与加速器进行通信。在系统的外部，通过一块 SD Card 来存放 Linux 启动文件与文件系统；通过以太网和外部进行通信；

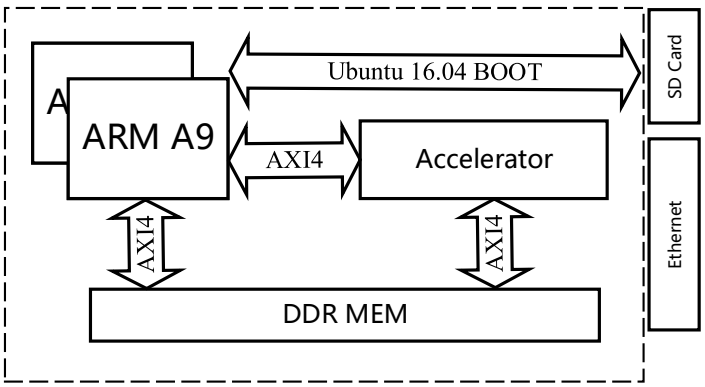

图 3-2 硬件总体设计架构

在设计的初期，加速器（Accelerator，ACC）参考了 CNNIOT[17] 设计了通用的卷积神经网络加速器，其支持配置可以调节的卷积、池化、激活函数、全连接等基本卷积神经网络算子。基于该加速器，设计实现了手写数字识别网络 LeNet5，平均处理一张图像需要 168ms，而 Arm A9 处理一张图像需要 1204ms。为了实现整个网络，需要：

1. 使用 Caffe 预训练模型得到 Caffemodel 文件。

2. 使用 Netron 将 Caffemodel 文件内部的权重参数导出为二进制文件。

3. 根据加速器在软件端配置网络、载入数据，并进行推理。

针对仅有三层卷积层，两层全连接层的 Lenet5 网络来说，配置的过程已经十分复杂，更不用说设计包含残差结构的 Resnet 网络。由此可以看出，硬件加速器被设计出来只是第一步，还需要构建完整的软件栈体系。

在经过一番调研之后，加速器设计选择了英伟达公司开源的 NVDLA 框架，经过英伟达团队几年的努力，NVDLA 自上而下打通了软件栈与硬件栈，可以实现端到端的推理，是一个适合深入学习的框架。

在 Xilinx 的设计工具中提供的 IP 核大多使用 AXI4 总线来实现各级之间的逻辑控制和数据传输。前文中提到，NVDLA 亦是使用 AXI4 总线访问存储，无疑极大方便了我们的设计，但是 NVDLA 的控制总线协议不是 AXI4 协议，在本设计中，其通过 csb2apb 电路将 CSB 协议转换为 APB 总线协议，在 Vivado 设计中，我们使用 Xilinx 官方提供给的 APB2AXI Bridge IP 将 APB 总线再转换为 AXI4 总线挂载到 ARM 处理器上。

3.3 验证平台

NVDLA 的驱动程序需要挂载到 Linux 内核上，所以本设计需要一款通用处理器作为主控，并且为其移植 Linux 操作系统。并且，考虑到本设计采用的 small 配置需要 7 万以上的查找表资源。则可供选型的器件型号有：

1. ZYNQ 7000 系列，该芯片集成了一块双核 32 位 ARM A9 处理器作为主控制器，而 FPGA 侧的资源随芯片的型号不同而不同，想要实现 NVDLA 至少需要使用到 ZYNQ 7045 器件。

2. ZYNQ MPSoc 系列，该芯片集成了一块多核的 64 位的 ARM A53 处理器作为主控制器，性能相较于 ZYNQ 7000 器件强，官方开发板卡的价格在 2 万左右。

3. 纯 FPGA 逻辑器件，如官方板卡 VCU118，有足够大的 LUT 资源，可以将 RISC-V 处理器与 NVDLA 一起实现，并在 RISC-V 处理器上移植 riscv-linux，开发周期较长。

综合以上，本设计选用的器件型号为 XC7Z045-2FFG900I，开发板卡为如图 3-3的第三方板卡，价格为三千多元人民币。

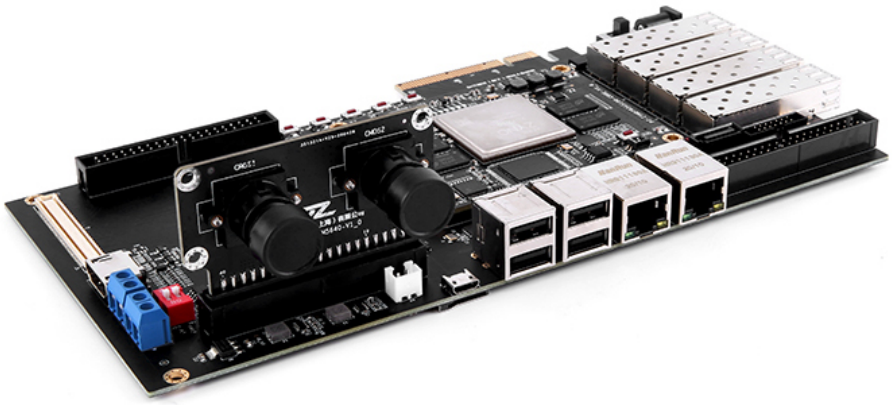

图 3-3 ZYNQ 7045 开发板

3.4　用户友好的输入页面设计方案

除此之外，本设计还基于 Bootstrap 与 Flask 等 Web 应用技术开发了用户友好的输入页面与数据显示应用服务。

Bootstrap 是由 Twitter 推出的用于前端开发的开源工具包，其简洁灵活，功能强大，本次设计初始阶段使用 Bootstrap 设计了一个手写数字页面，用户可以通过鼠标在 Canvas 画布上调节画笔大小，绘制数字，通过按键来决定采用 ARM 或者 FPGA 进行深度神经网络的推理。当用户按下按键之后，Web 前端会使用 Ajax 发送 HTTP 请求，通过以太网与开发板通信，获取处理结果。在板卡上，本设计使用 Flask 部署了 Web 应用的后端，负责接受 HTTP 请求，决策推理平台，将推理结果以及相关信息打包发送 HTTP 响应。

第四章　硬件设计实现

本章节将介绍基于 Xilinx Vivado 设计套件的系统的硬件设计实现，Xilinx Vivado 设计套件的开发流程包括四个步骤：设计、综合、实现、生成。在设计阶段，本章详细阐述了 NVDLA 的 RTL 代码生成方法，将 NVDLA 进行了部分优化与高度的封装，以便能够在 Vivado 中进行 BlockDesign 设计。

4.1　Xilinx FPGA 设计套件

在上一章节中提到，本设计选用的芯片信号为 XC7Z045-2FFG900I，该芯片由 Xilinx 公司供应。Xilinx 提供了非常完整的可编程逻辑开发软件工具链，及 Vivado 设计套件，使用该套件可以将我们的 Verilog 文件封装成知识产权 IP，进行硬件电路的仿真，生成比特流与硬件描述文件等功能。使用 ZYNQ 器件，也需要我们在 Vivado 的 BlockDesign 页面中分配 ARM 处理器使用到的资源。除此之外，Vivado 设计套件还包括了开发 ARM 裸机的 Vivado SDK、能够基于 LLVM 将 C\C++ 程序转换为底层逻辑电路的 Vivado HLS 等应用。

本章节将说明如何使用 Vivado 设计套件完成 NVDLA IP 的打包，Block Design 设计，查看资源利用、时序、功耗等报告，最后生成比特流文件与硬件描述文件。

4.2　NVDLA 硬件设计概述

NVDLA 由 NVIDIA 公司在 2017 年年末公布并且开源，遗憾的是该项目自公布完成一年后便草草停止了维护。NVDLA 是一款模块化、可配置的神经网络推理加速器框架。通过核心 MAC 阵列、与查找表逻辑，NVDLA 可以支持深度神经网络算子，已经支持的有：

- 卷积运算
- 激活函数
- 池化
- 归一化

实际上，由于 NVDLA 是完全开源的，完全可以自己手动添加相关算子的硬件逻辑。

4.2.1　NVDLA 硬件结构分析

前文中提到，NVDLA 具有模块化的属性，这很有助于我们理解加速器的工作流程，如图 4-1所示，NVDLA 的各个模块可以独立、同时工作。

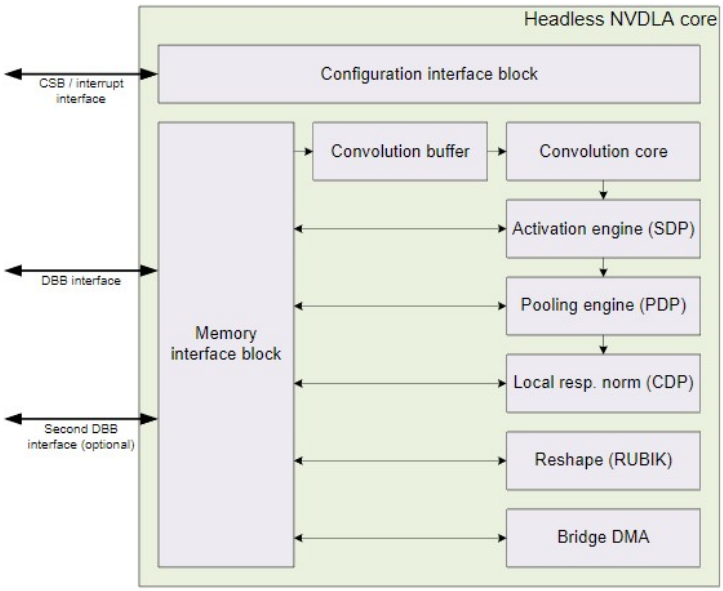

图 4-1 NVDLA 结构

- 卷积引擎配置了卷积 Buffer，在使能卷积运算的时候，需要先通过寄存器配置给出 Input Image 与 Weights 在内存上的地址，然后将数据缓存到 Buffer 中来进行运算，减少访存的开销。NVDLA 支持两种卷积模式：
  - 直接卷积，及标准的卷积，可以通过 MAC 阵列并行加速。
  - Winograd 快速卷积，从算法层面通过共用权重来减少运算量。

卷积引擎的核心是 MAC 阵列，该阵列的大小是可配置的，本文使用阵列大小为 8x8 。卷积的结果不可以直接写回内存，必须经过 SDP，及激活函数层，其他引擎则没有这个限制。

- 除去卷积引擎之外，NVDLA 一共还有五个引擎，他们是负责完成激活函数的 SDP 引擎，负责完成池化操作的 PDP 引擎，负责完成 LRN 操作的 CDP 引擎，做图形 Reshape 的 RUBIK 引擎，最后，NVDLA 设计还提供了一个 BDMA 引擎，用来在 DRAM 和高速存储之间搬移数据。

NVDLA 在外侧暴露出四个接口，它们分别是 CSB 总线，主存储接口、高速存储接口、中断接口。

- 寄存器配置总线（Configuration Space Bus Interface，CSB），用来读写 NVDLA 的寄存器，对于 CSB 总线，我们不甚讲解，其在读写地址的时候需要做移位来压缩指令大小，NVDLA 提供了 csb2apb 转换电路，在读写的时候使用该电路包裹即可利用 APB 总线协议读写 NVDLA 的寄存器。
- 主存储接口（Data Backbone interface，DBB），用来访存，该接口使用的协议为 AXI4 总线协议，可以挂载在 AXI4 接口的存储控制器上，在本文中，我们将其挂载到片上的

DDR 存储，与 ARM 处理器共享内存。

- 高速存储接口（high-bandwidth interface），可以外接第二块 SRAM 作为辅存储，存储中间数据，降低访存瓶颈。
- 中断接口（Interrupt interface），NVDLA 不仅支持通过寄存器查询的方式查询任务是否完成，也支持产生硬件中断，方便我们编写驱动程序。

4.2.2 NVDLA 自定义配置

在本设计中，我们使用官方提供的 small 配置，即最精简的 NVDLA 配置，该配置有如下特点：

1. 最小化 MAC 阵列，本设计将 NVDLA 的 MAC 阵列大小配置为 8 * 8。
2. 关闭高速存储接口，只使用主存储接口。
3. 仅支持 INT8 格式的 MAC 运算。
4. 不支持权重压缩。
5. 不支持 WINOGRAD 快速卷积。
6. 不支持 Reshape 特征图。

4.3 NVDLA IP 生成描述

这一小节将详细介绍如何修改 NVDLA 的配置文件，以及根据配置文件生成 NVDLA 的 RTL 代码，并将 NVDLA 的接口进行封装，打包成 IP。

总的来说，生成 NVDLA 的 RTL 代码有两个路径，其一是英伟达官方提供的 hw 项目，可以根据预先定义好的 spec 文件生成，但是这个工作流程需要预先构建好官方提供的 *make* 工具，该工具又依赖 GCC、Java、Perl、Verilator、Python 等，稍显复杂。第二条路径是使用一名由伯克利大学的研究生使用 Chisel 语言编写的 NVDLA 项目，其也是可以生成 RTL 代码的，但是由 Chisel 生成的 RTL 代码会被压缩在一个文件之内，NVDLA 的代码输出高达数万行，不利于阅读与分析。

综合以上，本文使用 *make* 进行 RTL 生成，并且使用 Docker 容器技术分离环境解决 *make* 软件依赖过复杂的问题。

4.3.1 基于 *make* 的 RTL 代码生成

为了能够在不污染主机环境的情况下构建 *make* 工具，本设计使用 Docker 容器技术搭建软件环境，基于 Ubuntu：16.04 容器，安装好如下环境：

- GCC 4.8.5

- OpenJDK 1.8.0
- SystemC 2.3.0
- Python 2.7.12
- VCS 2016.06
- Verilator 3.912
- Clang 3.8.0

安装好环境之后，我们可以通过新建一个 spec 文件来修改 NVDLA 的配置。例如通过定义宏 FEATURE_DATA_TYPE_INT8 可以指定 Feature Map 的数据格式为 INT8；通过定义宏 WINOGRAD_DISABLE 来关闭 WINOGRAD 卷积电路。本设计采用的是官方提供的最小配置，及使用 small.spec 文件。

有关 small.spec 的详细内容详见附录，在使用 *tmake* 之前，我们还需要在根目录通过 *make* 命令选择 small.spec 为配置文件，并输入安装的软件生态的可执行文件路径，该 *make* 命令最终会生成一个 tree.make 的文件，*tmake* 工具的本质是根据与现实先定义好的宏，将模板文件夹 vmod 里的 Verilog 代码进行文本预处理，将不需要的文本删除输出。

```
1 ./tools/bin/tmake -build vmod
```

执行完该命令之后，可以被综合的 Verilog 代码将会被存放在 out 文件夹下，此时已经可以通过 Vivado 导入文件夹解析依赖关系，进行分析。

4.3.2　RAM 优化

由于 NVDLA 是面向 ASIC 设计，内部 RAM 的 Verilog 代码是结构级描述，这意味着在例化 RAM 的时候会消耗大量 FPGA 片上珍贵的 LUT 资源，此时对其综合会导致浪费极多的 LUT 资源，所以在导入之前本设计将所有的 RAM 替换成 FPGA 内部的 Block RAM 以减小 LUT 的开销，提高运行速度。

替换 RAM 的方法也有两种，其一是例化 Vivado 设计免费提供的 BRAM Controller IP，但是 NVDLA 使用到的 RAM IP 过于繁杂，替换工作量巨大，第二是使用官方提供的行为级描述的 Verilog RAM 模块代码，本设计使用第二种。

具体步骤为，将原本的 *vmod/rams/synth* 文件夹删除，在 Vivado 导入文件夹的时候，使用 *vmod/rams/fpga* 文件夹。

由于 NVDLA 是面向 ASIC 设计，内部的 RAM 在例化的时候默认有 Clock Gating 电路用来降低功耗，但是 FPGA 的时钟树是设计好的，不需要该电路，否则可能会因为 Clock

Buf 资源不够导致布线过不去，在 Vivado 的全局变量里，添加如下几个 Global Define，关闭不必要的电路：

- VLIB_BYPASS_POWER_CG
- NV_FPGA_FIFOGEN
- FIFOGEN_MASTER_CLK_GATING_DISABLED
- FPGA
- SYNTHESIS

4.3.3 控制总线协议优化与封装

从 Vivado 导入 vmod 文件夹之后，NVDLA 的顶层模块文件为 *NV_nvdla.v* ，但是本设计还做了另一层封装，使用英伟达官方提供的 csb2apb 电路，将原本的 CSB 总线转化为 APB 总线，这样能够方便我们在 Vivado 设计中使用内存映射读写 NVDLA 的寄存器。

在 Vivado 设计中，新建一个 wrapper 文件，命名为 *Nv_nvdla_wrapper.v* ，部分内容见附录。在其中例化了 *NV_nvdla* 与 *NV_NVDLA_apb2csb* 两个电路，并且补全了 AXI 总线与 APB 总线缺失的信号线，方便在后期 IP Package 阶段自动推导封装总线接口。

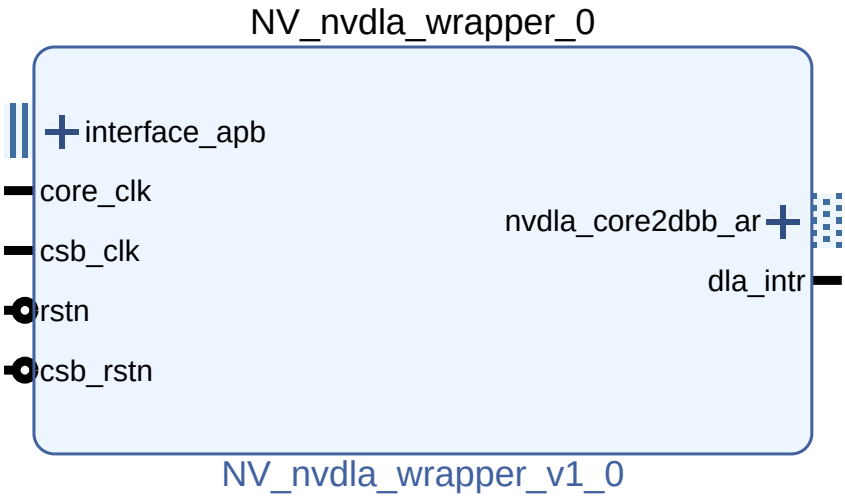

图 4-2 NVDLA Wrapper IP

为了方便之后 BlockDesign 中连线，也为了给 APB 总线挂上 Address Block，在 IP Package 阶段，我们需要 AXI 和 APB 总线进行包装，由于先前已经补全的信号线，这里我们让 Vivado 自动推导，包装完成后的 IP 核如图 4-2所示。

在 IP Package 阶段，AXI 的 Memory Block 会自行分配，这样在进行 Block Design 的时候 Vivado 会自动分配地址完成内存映射。但是 APB 总线的 Memory Block 需要自行添加，

在本设计中，为 APB 总线分配了 4KB 大小的寻址空间。

4.4　Block Design 设计

在我们的主工程中导入已经打包好的 NVDLA IP，并引入 APB to AXI Bridge、AXI Smart Connect、ZYNQ 7000+ 等其他知识产权 IP，他们的连线关系如图 4-3所示。

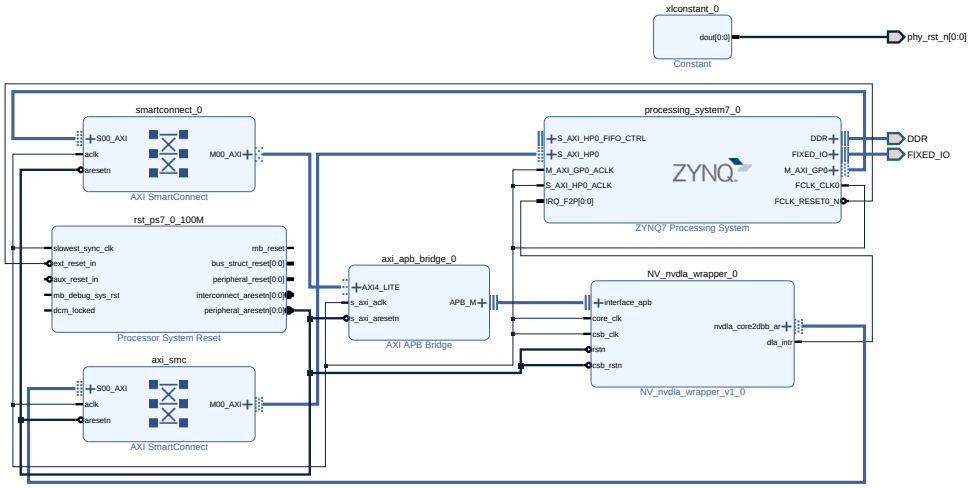

图 4-3 Block Design Connect

NVDLA 有两个工作时钟，csb_clk 与 core_clk，分别用于读写寄存器与加速器工作，这会使 NVDLA 变为一个异步电路，难以进行静态时序分析。根据分析，时序问题主要产生在 core_clk 之中，所以本设计将 csb_clk 与 core_clk 短接，使之变为同步电路，方便进行时序分析。

4.4.1　APB to AXI Bridge

使用 APB to AXI Bridge 可以把 APB 总线协议转化为 AXI 总线协议，这样方便我们使用 Vivado 内部的 Connect IP 自动做内存映射。

4.4.2　AXI Smart Connect

Axi Smart Connect 的作用是自动配置 AXI 设备的内存映射，与 Axi InterConnect 的作用是一样的，但是 Smart Connect 更紧密的嵌入到了 Vivado 内部，不需要用户太多的干涉。

在本设计中用到了两个 Smart Connect ，其中一个是将 ZYNQ 的 AXI Master 接入了 NVDLA 的控制总线，以便通过内存映射机制读写 NVDLA 的寄存器；另一个将 NVDLA

的主内存接口接入了 ZYNQ 的 AXI Slave，以便 NVDLA 访问挂在在 ARM 侧的 DDR 存储，与处理器共用内存，处理器可以通过硬件 DMA 搬移数据，加快访存速度。

4.4.3 ZYNQ 7000+ AP SOC

ZYNQ 7000+ 的 IP 如图 4-4所示，本设计中使用到了如下资源：

- 以太网接口（Ethernet0），在软件设计章节，本文将在 SOC 上构建 Linux 操作系统，使能以太网接口可以使板卡通过以太网线访问互联网，方便开发与调试。
- SD 卡接口（SD0），用来存放 Linux 操作系统的 BOOT 与 IMAGE 文件。
- 串口（UART0），用来实现串口终端，方便调试。
- 快速中断请求，用来接受由 NVDLA 产生的处理完成中断信号。
- FCLK_CLK0，输出时钟，该时钟可以作为可编程逻辑端的输入时钟，这样不需要额外对片外的时钟输入做约束，在本设计中该时钟取 100Mhz，有关 100Mhz 的取值，详见第六章测试与分析。

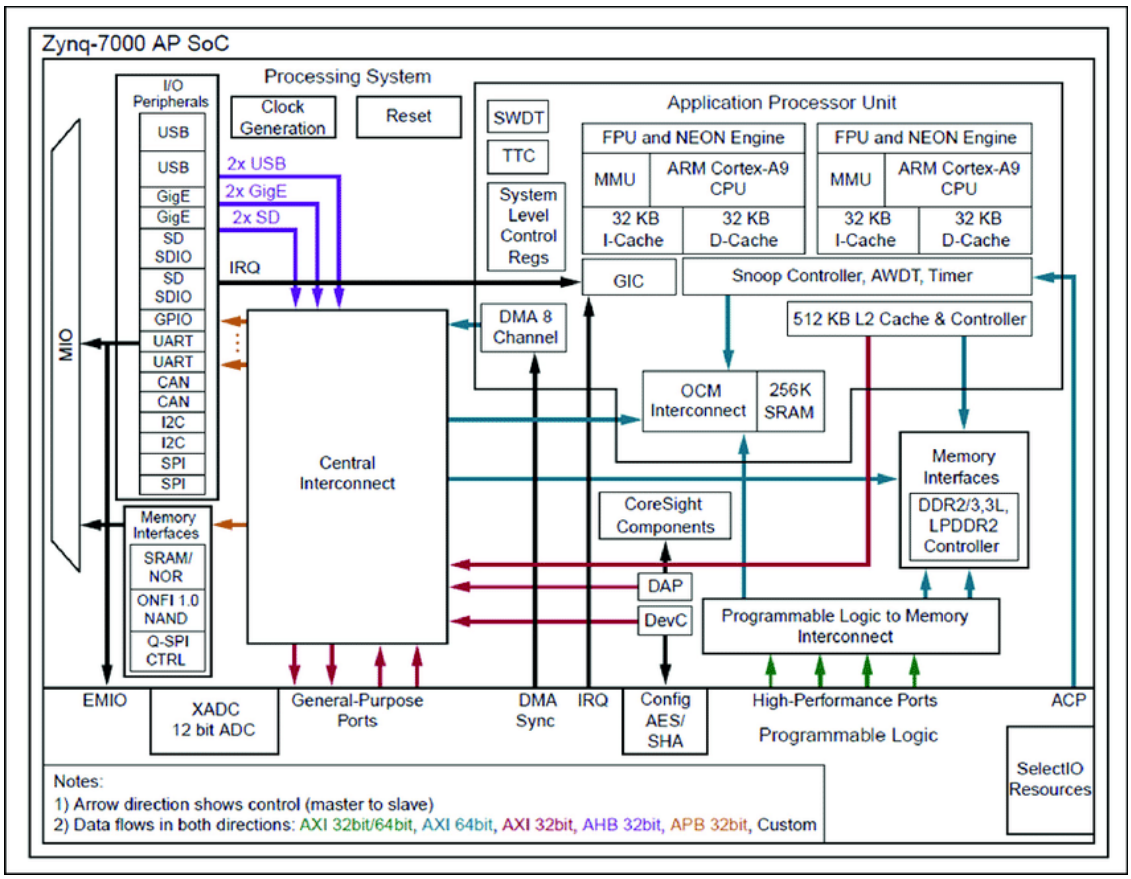

图 4-4 ZYNQ 7000+ block diagram

第五章　软件设计实现

硬件设计不是唯一的难点，本章节将详细概述软件设计实现。流程如图 5-1所示，首先在服务端，本设计使用 Caffe 框架针对 MNIST、CIFAR10 和 IMAGENET2012 三个数据集训练了三个分类模型，其次由于 NVDLA 的 small 配置仅支持 INT8 格式的数据，本章还将介绍结合 TensorRT 的模型量化方法；经过量化的模型将在主机端通过深度神经网络编译器进行硬件无关的优化与压缩，主机端还需要使用 Xilinx Petalinux 工具为 ARM 处理器移植 Linux 操作系统，完成驱动程序的编译，为深度神经网络运行时准备环境；在 ARM 处理器上，由深度神经网络运行时进行反序列化与加速器的调度。

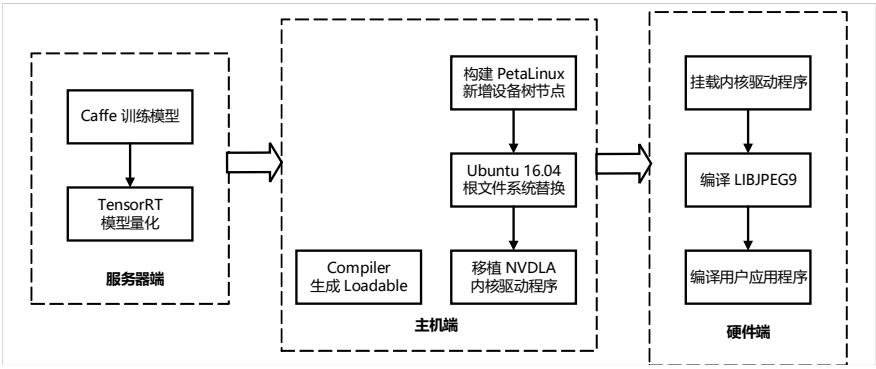

图 5-1 软件设计流程

5.1　NVDLA 软件工具链概述

如图 5-2，NVDLA 的软件工具链分为 Compiler 和 Runtime 两个部分。

Compiler, 即深度神经网络编译器，与硬件无关，可以在主机运行。它可以接受其他深度神经网络训练框架训练完成的模型，完成一些硬件无关的优化，例如算子融合、数据重用等，并且使用 FlatBuffers 将处理结果经过序列化，生成 Loadable 数据流文件。

Runtime，即深度神经网络运行时，与硬件紧密结合，其负责接受 Compiler 生成的 Loadable 文件，进行反序列化，调度加速器程序，自动分配内存，自动配置寄存器，自动处理中断。而 Runtime 内部又被划分为两个部分，UMD 和 KMD 吗，分别对应了 Linux 的用户态驱动与内核驱动。

- 用户态驱动程序（USER MODE DRIVER，UMD）由 C++ 编写，负责解析 Loadable 文件，分配内存，并将上下文封装成一个 Task，最后发送给底层的 KMD 程序执行。

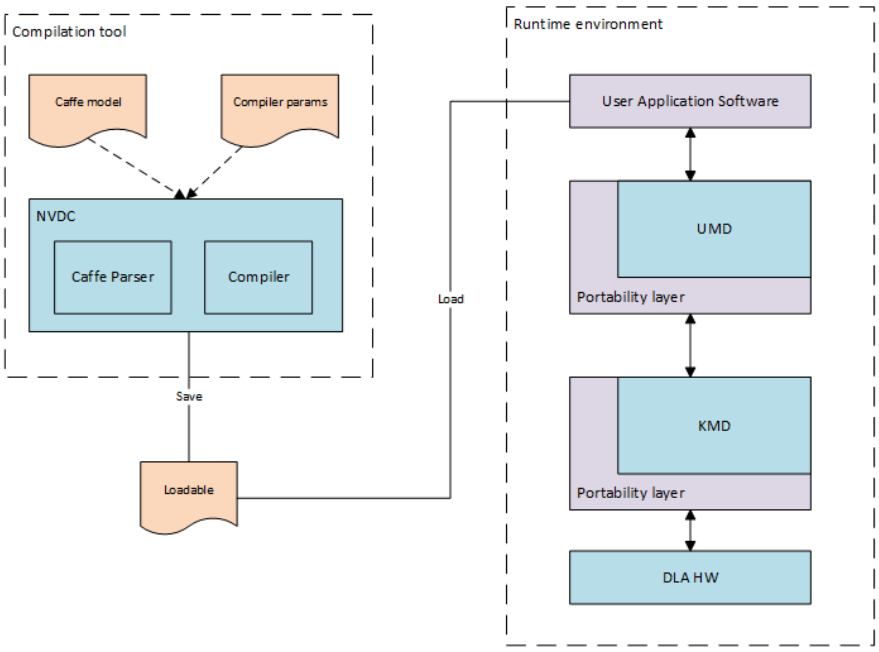

图 5-2 NVDLA Software

- 内核态驱动程序（KERNEL MODE DRIVER）由 C 语言编写，是 NVDLA 固件的驱动程序，他是真正与 NVDLA 进行交互的部分。

在这一小节，我们将详细的介绍 NVDLA 的软件工具链。如图 5-2，英伟达官方提供了完整的软件生态。

Compiler 是软件工具链的前端，与硬件无关，Compiler 能够接受的参数如下：

```
./nvdla_compiler -h
Usage: ./nvdla_compiler [-options] --prototxt <prototxt_file> --caffemodel <
    caffemodel_file>
where options include:
    -h                                                print this help message
    -o <outputpath>
    --profile <basic|default|performance|fast-math>   computation profile
    --cprecision <fp16|int8>                          compute precision
    --configtarget <nv_full|nv_large|nv_small>        target platform
    --calibtable <int8 calib file>
    --quantizationMode <per-kernel|per-filter>
```

- profile 指定了优化方案，在 Compiler 内部提供了四种优化方案，能够支持一些硬件无关的网络优化，例如算子融合、内存重用等，本设计使用默认的 fast-math，启用全部的优化选项。

- cprecision 指定了精度，在这里我们需要选择 int8，并且可以结合 TensorRT 生成

calibtabel 文件，如果不给出 calibtabel 参数，Compiler 内部会使用简单的量化方案，但在实际测试情况下，效果并不理想。

- configtarget 本设计选择 nv_small，匹配本设计的 small 配置。
- calibtabel 需由 TensorRT 生成，将在下一小节详细介绍。
- quantizationMode 有两个选项，per-kernel 选项是对每一层卷积使用相同的量化参数，per-filter 则是对每一层卷积使用不同的量化参数，这需要结合 calibtabel 文件选择，在本设计中使用 per-kernel。

最终，Compiler 生成 Loadable 文件，交给 Runtime 进行加速器的调度。

Loadable 文件是 Compiler 与 Runtime 之间通信的媒介，其由 Google 开源的 FlatBuffers 序列化协议所组织，能够将对象与数据进行压缩，以便在网络中进行传输。简单来讲，我们需要在流中传输一个对象，比如网络流。一般我们需要把这个对象序列化之后才能在流中传输（例如，可以把对象直接转化为字符串），然后在接收端进行反序列化（例如把字符串解析成对象）。但是显然把对象转成字符串传输的方法效率十分低下，于是有了各种流的转换协议，FlatBuffers 也是其中一种。

Runtime 与硬件紧密贴合，其又分为 UMD 和 KMD 两个部分：UMD 是用户应用，需要我们在 Linux 上编译运行，其接受 Loadable 文件并解析，最后递交一个推理任务到 KMD；KMD 是内核驱动，需要我们在构建 Linux 的时候编译，运行 Linux 的时候挂载，接受推理任务之后负责调度网络，配置寄存器，处理中断等任务。Runtime 能够接受的参数如下：

```
./nvdla_runtime -h
Usage: ./nvdla_runtime [-options] --loadable <loadable_file>
where options include:
    -h                  print this help message
    -s                  launch test in server mode
    --image <file>      input jpg/pgm file
    --normalize <value> normalize value for input image
    --mean <value>      comma separated mean value for input image
    --rawdump           dump raw dimg data
```

5.2 Caffe 与模型训练

本设计使用 Caffe 在服务器端训练了三个深度神经网络，其验证精度如表 5-1所示。

表 5-1 Caffe 模型验证精度

Network	Valiadation Accuracy %
Lenet5-MNIST	99.7
Resnet18-CIFAR10	90.2
Resnet18-IMAGENET2012	88.1(Top5)

5.3 TensorRT 与模型量化

本设计使用的 small 配置，仅支持 INT8 的推理，而 Caffe 框架仅支持 Float32 类型的训练，所以我们必须进行模型的量化。前文提到，在 Compiler 中量化需要结合英伟达公司的 TensorRT 框架。

5.3.1 TensorRT 量化原理

将高精度的浮点型转化为八比特的定点类型的量化方法，都遵循如下的公式:

$$T_F = SF * T_I + B$$

其中 T_F 指 Float 类型的张量、SF 指 Scale Factor、B 指偏置 Bias，通过实验测得，Bias 去掉对精度的影响不大，最终该公式变为:

$$T_F = SF * T_I$$

综合以上，进行神经网络的量化，本质上是求得每个参数的 Scale Factor 。最简单的量化方案是 max-max 映射:

$$SF = \frac{|W|_{max}}{128}$$

将权重数据根据绝对值的最大值作为阈值，归一化到 -128 到 127 之间，该方法针对分布均匀的权重数据是很有效果的。但是很明显，如果权重数据分布不均匀，该方法带来的精度损失很大。

TensorRT 选择的量化方法是 KL-divergence，即转化为最小化相对熵的问题[15]。相对熵表述的就是两个分布的差异程度，在量化问题上表述的是量化前后两个分布的差异程度，而我们的问题自然而然就代表着差异最小。关于求解的算法本设计不阐述，实际上英伟达也只是公开了解决方案，但是软件算法实现并没有开源，在 TensorRT 中我们也是只能调用其封装好的链接库。

5.3.2 量化步骤与精度损失

使用 TensorRT 进行量化[18] 过程如下：

1. 准备一个校准数据集;
2. 每一层进行 Float32 的推理;
3. 输入采样图片，收集激活值的直方图;
4. 基于不同的阈值产生不同的量化分布;
5. 计算每个分布与原分布的相对熵，选择熵最小的参数;

校准是核心部分，校准数据集可以从训练集或者验证集中采样，本设计选择从验证集中进行采样方便对比精度损失情况，关于采样多少张图片，英伟达官方建议采样 500 到 1000 张图片。

本文设计并且量化了三个网络：

1. 针对 MNIST 数据集的 Lenet5
2. 针对 CIFAR10 数据集的 Resnet18
3. 针对 IMAGENET2012 数据集的 Resnet18

这些网络的结构与量化样例代码略复杂，在附录里给出了 Lenet5 网络的量化关键代码，除此之外仅给出量化前后的精度损失情况，如表 6-8。

表 5-2 TensorRT 量化精度损失

Network	Valiadation Accuracy %	Calibration Accuracy %
Lenet5-MNIST	99.7	99.5
Resnet18-CIFAR10	90.2	86.7
Resnet18-IMAGENET2012	88.1(Top5)	77.0(Top5)

5.3.3 calibtabel 生成

使用 TensorRT 量化会生成 cache 文件，内部每一行代表每一个 layer 的 Scale Factor，但是 NVDLA 的 Compiler 在进行量化时需要接受 Json 格式的文件，如下所示：

```
{"first_conv":
    {"scale": 1.0007381439208984,
     "max": 0,
     "offset": 0,
     "min": 0
}    },
```

其中 first_conv 是 Layer 的 name，本设计使用 Python3 结合 PyCaffe 自行编写了 Cache 文件到 Json 文件转化的脚本，该脚本的内容详见附录。

5.4　Ubuntu 16.04 嵌入式操作系统移植

在主机端，使用深度神经网络编译器接受训练好的 Caffemodel 与本章节生成的 Json 文件，即可生成适用于 small 版本的 NVDLA 的 Loadable 文件。为了解析 Loadable 文件，即进行反序列化，需要在 ARM 处理器上编译出深度神经网络运行时，而在这之前又需要为 ARM 处理器移植操作系统。

为此，本小节将介绍基于 Xilinx Petalinux 的 Linux 操作系统移植方法，而 Petalinux 不具备包管理工具，为了更方便开发与调试，本小节还将介绍 Ubuntu 16.04 的根文件系统替换方法。

5.4.1　Petalinux 工具介绍

Xilinx Petalinux 是一个定制版的 Yocto 工具，包括了 Linux Kernel、u-boot、device-tree、rootfs 等源码，可以很方便的生成、配置、编译及自定义 Linux。对于 ZYNQ 7000+，基于 Petalinux 能够极大缩短移植 Linux 的周期，本设计使用 Petalinux 的命令行工具完成了硬件配置的解析，将 Vivado 中添加的 NVDLA 设计新增到了设备树中并分配内存，生成了基于 Linux Kernel 4.19 版本的 Linux 镜像文件，编译了 NVDLA 的驱动程序。

5.4.2　针对硬件设计的启动文件制作

由于本设计需要将根文件系统替换为 Ubuntu 16.04，首先需要做的是将根文件系统与 Boot 文件分离，在 Petalinux 项目构建时，需要配置从 SD 卡中读取根文件系统。

使用 *petalinux-config* 命令, 指定上一章节生成的硬件描述文件的路径，在 *Image Packaging Configuration*|*Root Filesystem Type*，选中 SD card，然后进行编译。

```
petalinux-config --get-hw-description=./PathOfhdf
```

修改此处后，linux 根文件系统 rootfs 将配置到 SD 中，而非默认的 raminitfs，后者是将根目录系统镜像在 boot 阶段加载到内存中，一旦裁剪的 kernel 较大（大概超过 120M），那么系统无法 boot。

由于我们已经将 rootfs 配置到 SD 中，那么就要取消掉 kernel 的 RAM intial，否则在 boot 阶段，kernel 在内存中找不到 rootfs 的符号镜像。

```
petalinux-config -c kernel
```

使用该命令配置取消 *Initial RAM filesystem and RAM disk support* 。

裁剪之后，进行编译即可生成适配板卡的 BOOT 和 Image 文件，以及 rootfs 文件系统，使用以下命令进行编译与生成：

```
petalinux-build
petalinux-package --boot --fsbl images/linux/zynq_fsbl.elf --fpga --u-boot --force
```

BOOT.BIN 与 IMAGE.ub 文件以及 ROOTFS 将会在目录下生成。

5.4.3 替换根文件系统

本设计使用了一张 32 GB 大小的 SD 卡，因为前文配置了 rootfs 从 SD 卡启动，所以我们需要对 SD 卡进行分区，分区情况如表 5-3所示。

表 5-3 SD 卡分区

Volume Name	Type	Device	Space (GB)
BOOT	FAT32	/dev/sdc1	4
ROOTFS	EXT4	/dev/sdc2	20
WorkSpace	EXT4	/dev/sdc3	8

其中，BOOT 分区格式为 FAT32，用来存储 BOOT.BIN 和 IMAGE.ub 文件，ROOTFS 需要的空间较大，用来存储根文件系统，WorkSpace 可有可无，我们使用他作为一个外置的存储，来存储一些数据。根据 Petalinux 的用户手册，BOOT 和 ROOTFS 需要严格在第一分区与第二分区。

本设计不采用 Petalinux 自己生成的根文件系统，使用 *ubuntu-16.04.2-minimal-armhf-2017-06-18* 作为根文件系统，覆盖由 Petalinux 产生的原生文件系统。把文件存入 SD 卡对应的分区，插入开发板则 Linux 操作系统移植过程结束。

5.5 KMD 内核程序挂载

在上一小节，本文讲解了如何利用 Petalinux 为 ZYNQ 器件移植 Ubuntu 的操作系统，在这一小节，我们将为 Ubuntu 增加 NVDLA 的内核驱动程序。由于本设计基于 Petalinux 2019.1（Linux Kernel 版本为 4.19）完成 KMD 环境的移植与 UMD 应用的编译，而 NVDLA

官方仓库的代码针对 Linux Kernel 4.13 与 64 位架构处理器，其有一些函数已经在本内核版本中已经被废弃，还有一些操作在本设计采用的 32 位处理器上不被允许，这些问题都将在本章得到解决。

5.5.1　Linux 设备树新增节点

使用 Petalinux 构建操作系统的时候会自动为我们生成设备树文件，但是他只包括了一些基本的硬件属性。本设计考虑到 NVDLA 的内核驱动程序，对设备树文件做以下更改：

1. 为 NVDLA 保留 256 MB 内存，因为在内核驱动程序中使用 DMA 搬移数据加快访存速度。

2. 将原本 PL 侧的 NVDLA 的设备树的 compatible 属性修改以适应内核驱动程序。

更改后的设备树文件详见附录 *system-user.dtsi*。

5.5.2　Petalinux 内核添加应用

使用 Petalinux 内建的工具即可新增内核应用，本设计创建了一个名为 opendla 的子模块：

```
petalinux-create -t modules -n opendla --enable
```

将 NVDLA 官方的 KMD 内核应用程序进行移植，首先需要对源代码做如下更改。

1. 在 *nvdla_gem.c* 里面，将 *dma_declare_coherent_memory* 函数所分配的内存大小更改为设备树中为 NVDLA 保留的内存。

2. *DMA_MEMORY_MAP* 标志已经在 Linux kernel 4.19 版本中被废弃，删除即可。

3. 在 *nvdla_core_callbacks.c* 中，求解运算时间使用到了 64 位除法，这在 32 位处理器的内核程序上是不被允许的，需要做一些改动。

对 KMD 的源代码改动完毕之后，还需要重新组织文件结构，使其适应 Petalinux 的模块构建方案，此外还需要重新编写 opendla 模块目录下的 *Makefile* 与 *opendla.bb* 文件，这两个文件的详细内容见附录。

修改完成后，重新进行编译，生成的内核驱动文件会在 Linux 根文件系统的 */lib/modules* 目录下，该文件会生成在 Petalinux 自己的根文件系统中，我们需要将其解压到 Ubuntu 的根文件系统的相同目录下。

5.5.3　驱动程序加载

启动系统之后，即可在 Ubuntu 中进行驱动程序的挂载：

```
insmod /lib/modules/4.19.0-xilinx-v2019.1/extra/opendla.ko
```

加载成功之后，系统设备信息中会多出 NVDLA 的中断信号和驱动。

5.6 UMD 应用程序编译

内核驱动挂载成功之后，UMD 程序才可以正确运行，而 UMD 也可分为两个主要部分：

- 第一部分负责解析提前编译好的 Loadable 文件，将权重、输入图像等数据导入内存，并指定输出图像的地址。
- 第二部分负责把任务发送给 KMD 执行。

在本小节将介绍在开发板卡上进行的 UMD 应用程序的改动与编译。

5.6.1 LIBJPEG 链接库编译

NVDLA 的 Runtime 可以接受 jpeg 格式文件的图片，但是需要提前编译好 LIBJPEG 的链接库。

官方提供的 64 位处理器的 LIBJPEG 链接库无法在 32 位处理器上链接，所以需要自行编译，本设计使用最新发行的 libjpeg9b，还需要更改 *jconfig.h* 中的 *JPEG_LIB_VERSION* 的值为 90。

5.6.2 UMD 构建

将自行编译的 libjpeg.a 文件拷贝到项目中来，则可以进行 UMD 应用的编译，将源代码都拷贝到 Ubuntu 操作系统上，执行以下命令即可完成编译：

```
cd ~/umd
export TOP=${PWD}
make runtime TOOLCHAIN_PREFIX=/usr/bin/
```

完成编译后，会生成两个目录文件，分别为 runtime 的链接库与 runtime 的可执行文件。将 runtime 的链接库拷贝到 runtime 可执行文件路径下，runtime 可执行文件才能被正常运行。

5.6.3 Runtime 测试

使用 Resnet18-CIFAR10 推理一张小猫的照片进行测试，其输出信息如下：

```
./nvdla_runtime --loadable ~/resnet18-cifar10-caffe/loadables/fast-math.nvdla --image
    ~/resnet18-cifar10-caffe/Image/cat_32.jpg --rawdump
creating new runtime context...
Emulator starting
dlaimg height: 32 x 32 x 3: LS: 256 SS: 0 Size: 8192
submitting tasks...
Work Found!
Work Done
execution time = 295854.000000 us
Shutdown signal received, exiting
Test pass
cat output.dimg
0 0 0 99 26 0 0 0 0 0
```

类别猫在 CIFAR10 数据集中的序号是 3，分类正确，耗时 295 毫秒，这代表着软件设计能够正常工作。

第六章　测试与分析

硬件设计完成之后，首先需要经过综合，综合阶段主要查看资源利用率情况确保资源足够。此外，Vivado 还会构建出理想的时钟树，给出理想的 Timing 和功耗报告。但综合阶段的数字电路设计没有映射到实际的物理器件上，所以这部分将基于实现阶段的 Timing 与功耗进行分析。最终，本章节将会对本设计的硬件加速系统性能进行评估。

6.1　资源利用率

在硬件设计实现章节，本设计将 NVDLA 内部存储优化为了 FPGA 片上的 BRAM 资源，在替换之前，其资源利用率情况如表 6-1所示。

表 6-1 优化 RAM 之前的资源利用率

RESOURCE	UTILIZATION	AVAILABLE	Utilization %
LUT	421698	218600	192.91
LUTRAM	8	70400	0.01
FF	92622	437200	22.33
DSP	33	900	3.67
BUFG	12	32	37.5

此时发现，LUT 资源的使用严重超过可使用数量，而 FPGA 内部的 BRAM 资源几乎未被使用，经过优化后，NVDLA 在 FPGA 上的资源利用率信息最终如表 6-2，资源的使用情况都在合理范围之内。

表 6-2 NVDLA 的资源使用情况

RESOURCE	UTILIZATION	AVAILABLE	Utilization %
LUT	77248	218600	35.34
LUTRAM	372	70400	0.53
FF	87999	437200	20.13
BRAM	95	545	17.43
DSP	33	900	3.67
BUFG	2	32	6.25

表 6-3给出了在最终总体的硬件设计中各个模块的资源使用率情况，包括了两个智能的总线连接器、一个同步复位单元、一个 AXI 转 APB 的桥接器、少量由 PL 端消耗的逻辑资源以及一个 NVDLA 硬件加速器。从表格中可以看出，NVDLA 占用了绝大多数的逻辑资源，占比 93.9%，表 6-3中的 In Total 项表明了本次设计的硬件加速系统所有的硬件资源开销。

表 6-3 各模块的资源使用情况

Module	Number	LUT	FF	BRAM	DSP
SmartConnect	2	2397	3313	0	0
rst_ps7_0_100M	1	19	40	0	0
processing_system7_0	1	112	0	0	0
axi_apb_bridge_0	1	89	144	0	0
NV_nvdla_wrapper_0	1	76888	87943	95	42
In Total	6	81902(37.47%)	94753(21.67%)	95(17.43%)	42(4.67%)

6.2　最大工作频率分析

静态时序分析（Static Timing Analysis，STA）的基本功能是对设计电路的时序进行检查。如图 6-1所示，时序检查是指建立时间与保持时间检查：

- 建立时间即 t_{setup}，检查该时间是为了保证数据可以在给定时钟周期内到达触发器，如果建立时间无法满足，则触发器会在数据到达之前采集到错误的数据。
- 保持时间即 t_{hold}，检查该时间是为了保证数据在被触发器采样后还能保持一定时间，如果保持时间无法满足，则触发器会采集到滞后的数据。

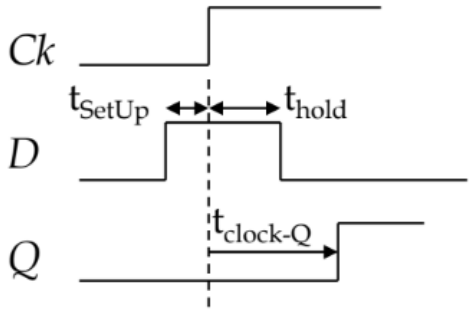

图 6-1 Setup 和 Hold 示例

一个数字电路中只要存在环路边界，就会存在理论上的最大工作频率。时序检查的目的都是为了保证触发器可以发送并且采样到正确的数据，确保电路输入的时钟频率在理论的最大频率之下。

首先将 10Mhz 的时钟接入 NVDLA，实现之后的 Timing 报告如表 6-4所示，其中 Slack 指的是建立时间与到达时间的差值，如果整个设计的 Slack 存在负值，则就存在部分寄存器无法获取到数据，电路无法正常工作。从该表中可以看出，对于 Setup Slack，设计的关键路径保有 88.126 ns 的余量，而关键路径的 Hold Slack 只有 0.030 ns 的余量，不存在负数，电路可以正常工作。

本设计在分析 NVDLA 最大工作时钟时，选取了几个典型的时钟值，分别是 25Mhz、50Mhz、75Mhz、100Mhz，他们的关键路径的时钟情况如表 6-5所示，可以发现当时钟频率

表 6-4 10 Mhz 情况下的关键路径时序情况

Frequency	Worst Setup Slack	Worst Hold Slack	Total Negative Slack
10Mhz	88.126 ns	0.030 ns	0.000 ns

不断增加的时候，关键路径的 Slack 值不断在变小，影响 NVDLA 工作的主要问题是 Setup 时间不足，在 100Mhz 的时候已经不足 1 ns。

表 6-5 不同时钟下的关键路径时序情况

Frequency	Worst Setup Slack	Worst Hold Slack	Total Negative Slack
25Mhz	29.048 ns	0.021 ns	0.000 ns
50Mhz	9.959 ns	0.007 ns	0.000 ns
75Mhz	3.832 ns	0.028 ns	0.000 ns
100Mhz	0.751 ns	0.030 ns	0.000 ns

如表 6-6所示，将时钟再提升至 150Mhz，则时钟已经出现了违例，具体表现为最差路径的 Slack 为负值，电路无法正常工作。此时强行上板会导致时序混乱，造成读写相关的寄存器失败。

由以上分析，在本设计中给 NVDLA 的输入时钟为 100Mhz，但是在 FPGA 上工作的最大时钟不代表进行 ASIC 设计能够工作的最大时钟，例如 Jetson Xavier NX 板卡上的 NVDLA 的工作时钟为 600 Mhz，根据中国科学院信息工程研究所的流片经验，其工作时钟为 800Mhz。

表 6-6 150 Mhz 情况下的关键路径时序情况

Frequency	Worst Setup Slack	Worst Hold Slack	Total Negative Slack
150Mhz	-0.324 ns	0.033 ns	-9.728 ns

如表 6-7，本文还给出了 10Mhz、25Mhz、50Mhz、75Mhz、100Mhz 时钟下的功耗情况，当时钟频率增加的时候，系统的功耗不断增加，其中静态功耗占比较小，约为 10%，动态功耗的占比较大，约为 90%。

表 6-7 不同时钟下的功耗情况

Frequency	Total Power	Static Power	Dynamic Power
10Mhz	1.805W	0.220W(12.2%)	1.585W(87.8%)
25 Mhz	1.921W	0.221W(11.5%)	1.700W(88.5%)
50 Mhz	2.129W	0.223W(10.5%)	1.906W(89.5%)
75 Mhz	2.325W	0.224W(9.6%)	2.101W(90.4%)
100 Mhz	2.528W	0.226W(8.9%)	2.302W(91.9%)

6.3　基于 NVDLA 的硬件加速系统性能评估

6.3.1　实验配置

本设计基于 Caffe 框架自行训练了三个网络，其网络结构限于篇幅，全部的 prototxt 文件与 loadables 文件已开源在 Github[20]。此外，为了对比 CPU 的运行时间，本设计在 ARM A9 处理器上进行了 Caffe 框架的移植与编译，对预训练的 Caffemodel 进行 CPU 侧的推理，由于 CPU 的访存特性，Caffe 在进行推理的时候速度会由慢到快，最终趋于稳定，所以本设计取迭代五十次后取推理时间的平均值。如图 6-2所示，本节还在主机端基于 Bootstrap 与 JQuery 等 Web 应用技术开发了的网页服务，并通过 Ajax 向后端发送推理请求，运行在 ZYNQ 7000+ 芯片上的 Ubuntu 操作系统里使用 Flask 框架接受请求并且调度。但是在测试过程中进行数据集级别的推理以及与 Caffe 框架进行的比较部分均不在该网页服务里完成。在测试环境中：NVDLA 和 Caffe 的输入 Batch 均为 1；NVDLA 进行推理的数据精度是 8 位的定点整型，Caffe 进行推理的数据精度为 32 位的浮点型；NVDLA 的运行时间从 UMD 向 KMD 发送任务开始计算，KMD 接收到中断完成任务停止，Caffe 的运行时间由第一层的推理开始计算，到最后一层推理结束停止，仅统计正向传播所用的时间。

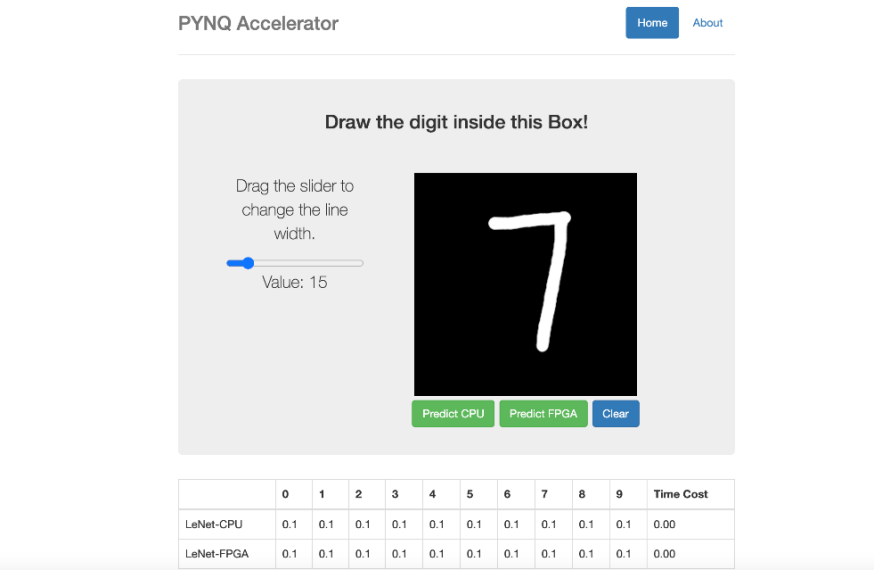

图 6-2 基于 Bootstrap 开发的网页服务

6.3.2　精度对比分析

本节将 TensorRT 为了量化而采样的一千张图像再次使用 NVDLA 进行推理并验证精度是否存在损失。其中，Resnet18-IMAGENET2012 网络参数较多，而本设计搭建的硬件平台仅有 500MB 左右可用的片上存储，使得该网络无法在开发板卡上正常推理，精度的得出是使用的 QEMU 虚拟环境，经过 HPC 模拟硬件加速器调度得出。

表 6-8 精度对比

Network	Valiadation Accuracy %	Calibration Accuracy %	NVDLA Accuracy %
Lenet5-MNIST	99.7	99.5	97.5
Resnet18-CIFAR10	90.2	86.7	80.7
Resnet18-IMAGENET2012	88.1(Top5)	77.0(Top5)	77.0(Top5)

实验结论：理论上，NVDLA 的精度应该与 TensorRT 量化之后的精度一致，但是实际测试中，NVDLA 的量化精度稍微低一些。经过分析，是因为 Caffe 训练的模型内部使用 OpenCV 进行图像读取操作，而 OpenCV 读取的图像为 BGR 格式，在 NVDLA 的 Runtime 阶段，由于 BGR 转 RGB 的操作会引发一系列错误，所以本设计将该操作去除，影响了部分精度，而 Resnet18-IMAGENET2012 未进行该处理，精度保持一致。

6.3.3 推理速度分析

本节针对设计的三个模型统计 NVDLA 的推理速度，其中 Resnet18-IMAGENET2012 因为需要分配的内存过大，而无法在板卡上运行，所以本设计没有对运行速度进行评估，其余两个模型，分别推理了一千张图像，在 ARM 端，本小节使用 Caffe 对同一模型迭代 50 次得到平均推理时间。表 6-9记录了推理的平均耗时情况，图 6-3记录了前五十次推理 NVDLA 相对 Caffe 的加速比。

表 6-9 运行速度

Network	1000 Images Execution Time	Time Per Image	FPS	Time ARM A9 (666 Mhz)
Lenet5-MNIST	11976 ms	11.20 ms	89.2	26.89 ms
Resnet18-CIFAR10	36529 ms	36.53 ms	27.4	307.69 ms
Resnet18-IMAGENET2012	\	\	\	6558.88 ms

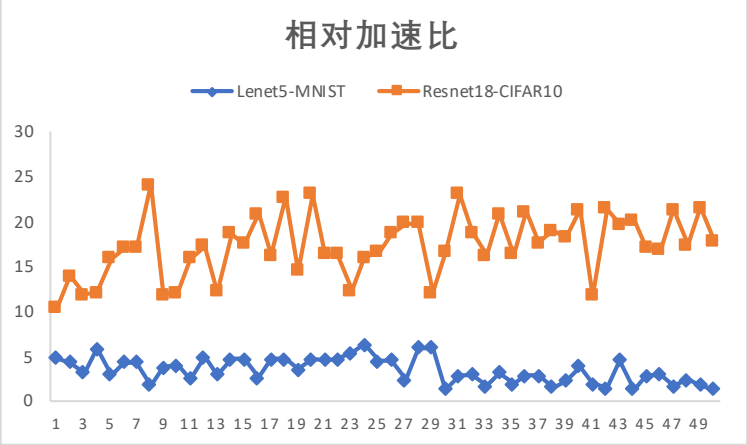

图 6-3 相对加速比

实验结论：如表 6-9所示，对于 Lenet5，ARM A9 的推理需要 20ms，而 NVDLA 仅需

要 10ms 左右的时间，加速比为 2.4；对于 Resnet18-CIFAR10，在 ARM 侧推理的速度是 Lenet5 的 14.5 倍，但是在 NVDLA 侧推理速度仅为 Lenet5 的 3 倍，加速比为 8.42；在两个网络上，NVDLA 相比于 ARM 侧都能实现不错的加速效果。实验中的 NVDLA 是在 FPGA 上实现的，由于 FPGA 的芯片性能以及硬件资源的限制，使得 NVDLA 的性能得到了限制，其工作频率仅有 100 Mhz，如果使用 ASIC 流程进行 NVDLA 实现，则性能的提升会更加明显，根据中国科学院自动化研究所的研究，在 600 Mhz 的工作频率下，使用 Lenet5 完成一幅图像的推理仅需要 0.2 ms[19]。

第七章　总结与展望

总的来说，本设计的重点是将原本面向 ASIC 设计的 NVDLA 移植到 FPGA 上进行验证，学习了其自上而下且规范的设计方法。对于已经停止维护的，最新版本的 NVDLA 到 FPGA 的映射过程，在本设计之前并没有检索到相关的工作。NVDLA 于 2017 年开源发布，实际的研发时间应该在 2015 至 2016 年开始，所以其整体的结构于 DianNao、DaDianNao 类似，但相比于当下的深度学习加速器体系结构来说，还是有些落后的。例如在 DaDianNao 之后，中科院计算技术研究所又提出了深度神经网络专用的指令集[8]。

根据设计的结果来看，在 FPGA 上的实现的 NVDLA 的速度可观，在打通软件栈之后其能够推理任意复杂，具备支持算子的网络模型，但由于时间有限，还有许多可以进一步研究的地方：

1. NVDLA 内部运算的核心是 MAC 阵列，但是由于其是面向 ASIC 设计，MAC 阵列在 FPGA 上映射到了 LUT 查找表资源，这会导致运行效率的低下，而根据资源利用情况来看，可以将 MAC 阵列消耗转化为片上的 DSP 资源以进一步提高效率。

2. 基于现有的研究[21]，优化 NVDLA 的 MAC 运算。

3. 针对不支持的算子，例如目标检测网络中的 RPN 算子、反卷积算子，NVDLA 不会工作，可以自行增加软件栈的特性，使这些不被硬件加速器支持的算子由软件实现。

4. 官方开源的 NVDLA 软件栈仅支持 Caffemodel，而现有的其他支持 NVDLA 作为后端的深度神经网络编译器，如 ONNC 可以支持 ONNX 生成 Loadable 文件，极大增加了灵活性。

参考文献

[1] HOLLER, TAM, CASTRO, et al. An electrically trainable artificial neural network (etann) with 10240 'floating gate' synapses[C/OL]//International 1989 Joint Conference on Neural Networks. 1989: 191-196 vol.2. DOI: 10.1109/IJCNN.1989.118698.

[2] HAMMERSTROM D. A vlsi architecture for high-performance, low-cost, on-chip learning[C/OL]//1990 IJCNN International Joint Conference on Neural Networks. 1990: 537-544 vol.2. DOI: 10.1109/IJCNN.1990.137621.

[3] VIREDAZ M, IENNE P. Mantra i: a systolic neuro-computer[C/OL]//Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan): volume 3. 1993: 3054-3057 vol.3. DOI: 10.1109/IJCNN.1993.714364.

[4] 王守觉陈向东 曾玉娟. 人工神经网络硬件化途径与神经计算机研究[J]. 深圳大学学报, 1997, 1(6): 8-13.

[5] CHEN T, DU Z, SUN N, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning[J/OL]. SIGARCH Comput. Archit. News, 2014, 42(1):269–284. https://doi.org/10.1145/2654822.2541967.

[6] CHEN Y, LUO T, LIU S, et al. Dadiannao: A machine-learning supercomputer[C/OL]//2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. 2014: 609-622. DOI: 10.1109/MICRO.2014.58.

[7] LIU D, CHEN T, LIU S, et al. Pudiannao: A polyvalent machine learning accelerator[J/OL]. SIGARCH Comput. Archit. News, 2015, 43(1):369–381. https://doi.org/10.1145/2786763.2694358.

[8] LIU S, DU Z, TAO J, et al. Cambricon: An instruction set architecture for neural networks[C/OL]//2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). 2016: 393-405. DOI: 10.1109/ISCA.2016.42.

[9] 薛永红, 王洪鹏. 机器下棋的历史与启示——从"深蓝"到 AlphaZero[J]. 科技导报, 2019, 37(19):87-96.

[10] KANAL L N. Perceptron[M]. GBR: John Wiley and Sons Ltd., 2003: 1383–1385.

[11] 王斌. 基于深度图像和深度学习的机器人抓取检测算法研究[D]. 浙江大学, 2019.

[12] HORNIK K. Approximation capabilities of multilayer feedforward networks[J/OL]. Neural Networks, 1991, 4(2):251-257. https://www.sciencedirect.com/science/article/pii/089360809190009T. DOI: https://doi.org/10.1016/0893-6080(91)90009-T.

[13] LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-based learning applied to document recognition[J/OL]. Proceedings of the IEEE, 1998, 86(11):2278-2324. DOI: 10.1109/5.726791.

[14] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[J/OL]. CoRR, 2015, abs/1512.03385. http://arxiv.org/abs/1512.03385.

[15] SHEN H, GONG J, LIU X, et al. HIGHLY EFFICIENT 8-BIT LOW PRECISION INFERENCE OF CONVOLUTIONAL NEURAL NETWORKS[EB/OL]. 2019. https://openreview.net/forum?id=SklzIjActX.

[16] JOUPPI N P, YOUNG C, PATIL N, et al. In-datacenter performance analysis of a tensor processing unit [J/OL]. CoRR, 2017, abs/1704.04760. http://arxiv.org/abs/1704.04760.

[17] MFARHADI. Cnniot[EB/OL]. 2018. https://github.com/mfarhadi/CNNIOT/.

[18] 周阳. 神经网络参数压缩和推断加速方法的研究[D]. 中国科学院大学 (中国科学院深圳先进技术研究院), 2020.

[19] FENG S, WU J, ZHOU S, et al. The implementation of lenet-5 with nvdla on risc-v soc[C/OL]//2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS). 2019: 39-42. DOI: 10.1109/ICSESS47205.2019.9040769.

[20] LEIWANG1999. Nvdla loadables[EB/OL]. 2021. https://github.com/LeiWang1999/nvdla_loadables/.

[21] 祁琛. 应用于神经网络的高效能计算单元的研究与实现[D]. 东南大学, 2018.

致　谢

时光易逝，随着毕业论文的定稿四年的本科求知时光也就要结束了，行文至此，心中满是感恩。

细细想来，这一路的打怪升级要始于三年前的夏天。我有幸加入了学院的硬件部，跟随在已经推免至东南大学的周玉乾学长后面学习，在 FPGA 学习上，学长给了我莫大的帮助；当时一同在硬件部学习的王镇、吴佳昱两位同学也成了我参加很多比赛的队友，一起在创新工作室学习的葛志来同学也带给我一些欢乐，他们分别推免至清华大学、南京大学、复旦大学，对他们都有了好的去处我作为朋友很是开心；在创新工作室的前辈们，尤其是陈宇明学长，跟随在陈宇明学长身后运维南工在线的时间使我掌握了很多前沿的技术，他们在我的毕业设计中被灵活运用。

说起本次毕业设计，已经退休的包亚萍教授利用自己的科研经费为本毕业设计采购了开发板卡，包老师严谨的治学态度、精益求精的工作作风深刻地影响着我的本科生活；朱艾春老师是本次毕业设计的指导老师，其认真负责，指导了很多本论文的写作；中国科学院信息工程研究所的王兴宾博士曾经参与过 NVDLA 流片工作，与他交流解决了我的一些困惑；在计算所的许浩博师兄、肖航师兄、卢美璇师姐以及韩银和老师，在本次毕业设计他们提供了包括但不限于研究方向、采购开发板卡、GPU 服务器、论文写作等支持与指导。

本科不是终点而是起点，山鸟与鱼不同路。回忆起四年前一只脚迈进大学之门的时候，感到的更多的是不确定和迷茫，而现在即将迈出下一步的时候，未来依然是如此的不确定，但是期待却又更多了一些。感谢所有给予过我帮助的老师和同学，使我拥有了无比丰富精彩的大学时光！

附录 A

A.1 *small.spec*

```
#define FEATURE_DATA_TYPE_INT8
#define WEIGHT_DATA_TYPE_INT8
#define WEIGHT_COMPRESSION_DISABLE
#define WINOGRAD_DISABLE
#define BATCH_DISABLE
#define SECONDARY_MEMIF_DISABLE
#define SDP_LUT_DISABLE
#define SDP_BS_ENABLE
#define SDP_BN_ENABLE
#define SDP_EW_DISABLE
#define BDMA_DISABLE
#define RUBIK_DISABLE
#define RUBIK_CONTRACT_DISABLE
#define RUBIK_RESHAPE_DISABLE
#define PDP_ENABLE
#define CDP_ENABLE
#define RETIMING_DISABLE
#define MAC_ATOMIC_C_SIZE_8
#define MAC_ATOMIC_K_SIZE_8
#define MEMORY_ATOMIC_SIZE_8
#define MAX_BATCH_SIZE_x
#define CBUF_BANK_NUMBER_32
#define CBUF_BANK_WIDTH_8
#define CBUF_BANK_DEPTH_512
#define SDP_BS_THROUGHPUT_1
#define SDP_BN_THROUGHPUT_1
#define SDP_EW_THROUGHPUT_x
#define PDP_THROUGHPUT_1
#define CDP_THROUGHPUT_1
#define PRIMARY_MEMIF_LATENCY_64
#define SECONDARY_MEMIF_LATENCY_x
#define PRIMARY_MEMIF_MAX_BURST_LENGTH_1
#define PRIMARY_MEMIF_WIDTH_64
#define SECONDARY_MEMIF_MAX_BURST_LENGTH_x
```

```
#define SECONDARY_MEMIF_WIDTH_x
#define MEM_ADDRESS_WIDTH_32
#define NUM_DMA_READ_CLIENTS_7
#define NUM_DMA_WRITE_CLIENTS_3

#include "projects.spec"
```

A.2 *Nv_nvdla_wrapper.v*

```
    NV_NVDLA_apb2csb apb2csb (
        .pclk                   (csb_clk)
        ,.prstn                  (csb_rstn)
        ,.csb2nvdla_ready        (m_csb2nvdla_ready)
        ,.nvdla2csb_data         (m_nvdla2csb_data)
        ,.nvdla2csb_valid        (m_nvdla2csb_valid)
        ,.paddr                  (paddr)
        ,.penable                (penable)
        ,.psel                   (psel)
        ,.pwdata                 (pwdata)
        ,.pwrite                 (pwrite)
        ,.csb2nvdla_addr         (m_csb2nvdla_addr)
        ,.csb2nvdla_nposted      (m_csb2nvdla_nposted)
        ,.csb2nvdla_valid        (m_csb2nvdla_valid)
        ,.csb2nvdla_wdat         (m_csb2nvdla_wdat)
        ,.csb2nvdla_write        (m_csb2nvdla_write)
        ,.prdata                 (prdata)
        ,.pready                 (pready)
    );
    NV_nvdla nvdla_top (
        .dla_core_clk                   (core_clk)
        ,.dla_csb_clk                    (csb_clk)
        ,.global_clk_ovr_on              (1'b0)
        ,.tmc2slcg_disable_clock_gating  (1'b0)
        ,.dla_reset_rstn                 (rstn)
        ,.direct_reset_                  (1'b1)
        ,.test_mode                      (1'b0)
        ,.csb2nvdla_valid                (m_csb2nvdla_valid)
        ,.csb2nvdla_ready                (m_csb2nvdla_ready)
        ,.csb2nvdla_addr                 (m_csb2nvdla_addr)
```

```
        ,.csb2nvdla_wdat                (m_csb2nvdla_wdat)
        ,.csb2nvdla_write               (m_csb2nvdla_write)
        ,.csb2nvdla_nposted             (m_csb2nvdla_nposted)
        ,.nvdla2csb_valid               (m_nvdla2csb_valid)
        ,.nvdla2csb_data                (m_nvdla2csb_data)
        ,.nvdla2csb_wr_complete         () //FIXME: no such port in apb2csb
        ,.nvdla_core2dbb_aw_awvalid     (nvdla_core2dbb_aw_awvalid)
        ,.nvdla_core2dbb_aw_awready     (nvdla_core2dbb_aw_awready)
        ,.nvdla_core2dbb_aw_awaddr      (nvdla_core2dbb_aw_awaddr)
        ,.nvdla_core2dbb_aw_awid        (nvdla_core2dbb_aw_awid)
        ,.nvdla_core2dbb_aw_awlen       (nvdla_core2dbb_aw_awlen)
        ,.nvdla_core2dbb_w_wvalid       (nvdla_core2dbb_w_wvalid)
        ,.nvdla_core2dbb_w_wready       (nvdla_core2dbb_w_wready)
        ,.nvdla_core2dbb_w_wdata        (nvdla_core2dbb_w_wdata)
        ,.nvdla_core2dbb_w_wstrb        (nvdla_core2dbb_w_wstrb)
        ,.nvdla_core2dbb_w_wlast        (nvdla_core2dbb_w_wlast)
        ,.nvdla_core2dbb_b_bvalid       (nvdla_core2dbb_b_bvalid)
        ,.nvdla_core2dbb_b_bready       (nvdla_core2dbb_b_bready)
        ,.nvdla_core2dbb_b_bid          (nvdla_core2dbb_b_bid)
        ,.nvdla_core2dbb_ar_arvalid     (nvdla_core2dbb_ar_arvalid)
        ,.nvdla_core2dbb_ar_arready     (nvdla_core2dbb_ar_arready)
        ,.nvdla_core2dbb_ar_araddr      (nvdla_core2dbb_ar_araddr)
        ,.nvdla_core2dbb_ar_arid        (nvdla_core2dbb_ar_arid)
        ,.nvdla_core2dbb_ar_arlen       (nvdla_core2dbb_ar_arlen)
        ,.nvdla_core2dbb_r_rvalid       (nvdla_core2dbb_r_rvalid)
        ,.nvdla_core2dbb_r_rready       (nvdla_core2dbb_r_rready)
        ,.nvdla_core2dbb_r_rid          (nvdla_core2dbb_r_rid)
        ,.nvdla_core2dbb_r_rlast        (nvdla_core2dbb_r_rlast)
        ,.nvdla_core2dbb_r_rdata        (nvdla_core2dbb_r_rdata)
        ,.dla_intr                      (dla_intr)
        ,.nvdla_pwrbus_ram_c_pd         (32'b0)
        ,.nvdla_pwrbus_ram_ma_pd        (32'b0)
        ,.nvdla_pwrbus_ram_mb_pd        (32'b0)
        ,.nvdla_pwrbus_ram_p_pd         (32'b0)
        ,.nvdla_pwrbus_ram_o_pd         (32'b0)
        ,.nvdla_pwrbus_ram_a_pd         (32'b0)
    ); // nvdla_top
assign nvdla_core2dbb_aw_awsize = 3'b011;
assign nvdla_core2dbb_ar_arsize = 3'b011;
assign m_axi_awburst = 2'b01;
assign m_axi_awlock  = 1'b0;
```

```
assign m_axi_awcache = 4'b0010;
assign m_axi_awprot  = 3'h0;
assign m_axi_awqos   = 4'h0;
assign m_axi_awuser  = 'b1;
assign m_axi_wuser   = 'b0;
assign m_axi_arburst = 2'b01;
assign m_axi_arlock  = 1'b0;
assign m_axi_arcache = 4'b0010;
assign m_axi_arprot  = 3'h0;
assign m_axi_arqos   = 4'h0;
assign m_axi_aruser  = 'b1;
assign pslverr = 1'b0;
endmodule
```

A.3 *Lenet5* 量化关键代码

```
# Returns a numpy buffer of shape (num_images, 1, 28, 28)
def load_data(filepath):
    test_imgs = []
    global fileList
    fileList = os.listdir(filepath)
    mean = np.ones([3, 224, 224], dtype=np.float)
    mean[0,:,:] = 104
    mean[1,:,:] = 117
    mean[2,:,:] = 123
    for img_path in fileList:
        img_path = filepath +'/'+ img_path
        img = cv.imread(img_path)
        img = crop_img(img, [224, 224])
        img = img.transpose((2, 0, 1))
        img = img - mean
        test_imgs.append(img)
    # Need to scale all values to the range of [0, 1]
    return np.ascontiguousarray(test_imgs).astype(np.float32)

# Returns a numpy buffer of shape (num_images)
def load_labels(filepath):
    global fileList
    test_labels = []
```

```
    labels_mapping = {}
    with open(filepath, 'r') as f:
        lines = f.readlines()
        for each in lines:
            imageName, labels = each.strip('\n').split(' ')
            labels_mapping[imageName] = int(labels)
    for each in fileList:
        test_labels.append(labels_mapping[each])
    return np.ascontiguousarray(test_labels)
```

A.4 *Cache to Json*

```
import json
from collections import OrderedDict
from google.protobuf import text_format
import caffe.proto.caffe_pb2 as caffe_pb2       # 载入caffe.proto编译生成的caffe_pb2文
    件


caffeprototxt_path = "./deploy.prototxt"
calibletable_json_path = "./resnet18-cifar10-int8.json"
# load deploy.prototxt
net = caffe_pb2.NetParameter()
text_format.Merge(open(caffeprototxt_path).read(), net)
# load jsonfile
with open(calibletable_json_path, "r") as f:
    calible = json.load(f, object_pairs_hook=OrderedDict)
_scales = []
_mins = []
_maxs = []
_offsets = []
_new = OrderedDict()
items = calible.items()
for key, value in items:
    _scales.append(value['scale'])
    _mins.append(value['min'])
    _maxs.append(value['max'])
    _offsets.append(value['offset'])
for idx, _layer in enumerate(net.layer):
```

```
    _tempDict =  OrderedDict({
        "scale": _scales[idx],
        "min": _mins[idx],
        "max": _maxs[idx],
        "offset": _offsets[idx],
    })
    _new[_layer.name] =_tempDict
with open('resnet18-cifar10-int8-fixed.json', 'w') as f:
    json.dump(_new, f)
```

A.5 *system-user.dtsi*

```
/include/ "system-conf.dtsi"
/ {
    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        nvdla_reserved: buffer@0x30000000 {
            compatible = "shared-dma-pool";
            no-map;
            reg = <0x30000000 0x10000000>;
        };
    };
};

&NV_nvdla_wrapper_0{
    compatible = "nvidia,nv_small";
    memory-region = <&nvdla_reserved>;
};
```

A.6 *Makefile*

```
obj-m := opendla.o

###append all of sources###
```

```
opendla-objs := nvdla_core_callbacks.o nvdla_gem.o scheduler.o engine.o bdma.o conv.o
    sdp.o cdp.o pdp.o rubik.o cache.o common.o engine_data.o engine_isr.o engine_debug
    .o
############################


SRC := $(shell pwd)

all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
    rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
    rm -f Module.markers Module.symvers modules.order
    rm -rf .tmp_versions Modules.symvers
```

A.7 *opendla.bb*

```
SUMMARY = "Recipe for  build an external opendla Linux kernel module"
SECTION = "PETALINUX/modules"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

SRC_URI = "file://Makefile \
           file://nvdla_core_callbacks.c \
           file://nvdla_gem.c \
           file://scheduler.c \
           file://engine.c \
           file://bdma.c \
           file://conv.c \
           file://sdp.c \
           file://cdp.c \
           file://pdp.c \
           file://rubik.c \
```

```
19              file://cache.c \
                file://common.c \
21              file://engine_data.c \
                file://engine_isr.c \
23              file://engine_debug.c \
                file://common.h \
25              file://dla_debug.h \
                file://dla_fw_version.h \
27              file://dla_engine.h \
                file://dla_engine_internal.h \
29              file://dla_err.h \
                file://dla_interface.h \
31              file://dla_sched.h \
                file://engine_debug.h \
33              file://nvdla_interface.h \
                file://nvdla_linux.h \
35              file://opendla.h \
                file://opendla_initial.h \
37              file://opendla_small.h \
                file://nvdla_ioctl.h \
39              file://COPYING \
        "
41  S = "${WORKDIR}"

43  # The inherit of module.bbclass will automatically name module packages with
    # "kernel-module-" prefix as required by the oe-core build environment.
```