摘　要

随着大模型进一步掀起人工智能领域的革新浪潮，业界对于算力的需求也变得越来越大，在这种形式下，由于在性能能耗比上的优势，基于 FPGA 开发方便的适配小参数模型的边缘计算设备，仍然有着不错的前景。本文选择对边缘部署有大量需求的目标检测领域的经典算法 YOLOv4-Tiny 作为着手点，为其定制一套轻量化的软硬件加速方案。

本文首先根据 YOLOV4 所引入的新的训练方法，并采用 LeakyRelu 作为激活函数且参数选择为 0.125 方便硬件进行移位操作，通过数据增强和基于 CIOU 的 Loss 训练得到了原始网络，之后进行了层融合与 8bit 定点量化。

本文基于 Vitis HLS 高层次综合工具，基于卷积循环优化，流水线优化，二级缓存，乒乓操作等多种方法设计了专用加速器，并开发了与之匹配的驱动函数库。最终部署在 ZYNQ7020 平台，该系统在 VOC 测试集下系统性能达到了 1.395GOPS，是 ARM(CortexA9) 的 93 倍。性能功耗比达到了 0.78GOPS/W，达到了 ARM(CortexA9)的 78 倍，达到了低功耗高性能的目的。

关键词：卷积神经网络，目标检测，软硬件协同设计，硬件加速

ABSTRACT

With the large model further set off the wave of innovation in the field of artificial intelligence, the industry's demand for computing power has become more and more large, in this form, due to the advantages in the performance and energy consumption ratio, based on FPGA development of convenient edge computing equipment suitable for small parameter models, still has a good prospect. In this thesis, we choose YOLOv4-Tiny, a classical algorithm in the field of object detection, which has a large demand for edge deployment, as the starting point, and customize a set of lightweight hardware and software acceleration scheme for it.

This thesis first adopts the new training method introduced by YOLOV4, adopts LeakyRelu as the activation function and the parameter is set to 0.125 to facilitate the hardware shift operation, obtains the original network through data enhancement and CiOU-based Loss training, and then carries out layer fusion and 8bit fixed-point quantization.

In this thesis, a special accelerator is designed based on Vitis HLS high-level synthesis tool, convolutional cycle optimization, pipeline optimization, second-level cache, ping-pong operation and other methods, and a matching driver library is developed. Finally deployed on the ZYNQ7020 platform, the system achieved 1.395GOPS on the VOC test set, 93 times the performance of ARM(CortexA9). The performance power consumption ratio reaches 0.78GOPS/W, which is 78 times that of ARM(CortexA9), and achieves the purpose of low power consumption and high performance.

Keywords: deep learning, target detection, hardware accelerator

目 录

摘　要 I
ABSTRACT II
第一章 绪论 1
1.1 研究背景及意义 1
1.2 研究现状 2
1.3 研究内容及设计指标 2
1.3.1 研究内容 2
1.3.2 设计指标 2
1.4 论文组织结构 3
第二章 YOLOV4-TINY 算法基础 4
2.1 YOLOv4-Tiny 算法 4
2.1.1 卷积神经网络 4
2.1.1.1 输入 4
2.1.1.2 卷积层 4
2.1.1.3 池化层 7
2.1.1.4 激活函数（非线性层） 8
2.1.1.5 上采样层 8
2.1.2 YOLOv4-Tiny 算法 8
2.1.2.1 目标检测网络基本结构 8
2.1.2.2 YOLOv4-Tiny 网络结构 9
2.1.2.3 Mosaic 数据增强 10
2.1.2.4 损失函数与 NMS（非极大抑制） 11
2.2 网络优化 13
2.2.1 层融合 13
2.2.2 线性量化 14
2.2.2.1 卷积层量化 14
2.2.2.2 LeakyRelu 层量化 15
2.2.2.3 整体量化 15
2.3 本章小节 17
第三章 软硬件协同架构设计 18
3.1 ZYNQ-7000 平台 18
3.1.1 AXI 总线 (Advanced eXtensible Interface)与接口 18
3.1.2 中断系统 20
3.2 软硬件任务划分 20

3.3 软硬件协同设计系统.....21
3.4 PS 端模块设计.....22
3.4.1 图像预处理.....22
3.4.2 数据控制.....23
3.4.3 YOLO Head 解码.....24
3.4.4 计时统计.....24
3.5 本章小节.....24
第四章 硬件加速建模与设计.....25
4.1 硬件加速需求分析.....25
4.2 卷积运算通路建模.....25
4.2.1 乘加器阵列建模.....27
4.2.2 片上缓存建模.....28
4.2.3 数据排布方式建模.....29
4.3 卷积运算通路设计.....31
4.4 池化运算通路设计.....31
4.5 辅助运算通路建模与设计.....32
4.6 访存模块设计.....33
4.7 本章小节.....33
第五章 系统验证与分析.....34
5.1 ZYNQ 系统下 CPU 验证.....34
5.2 硬件加速器仿真测试.....35
5.3 目标检测 SoC 验证.....36
5.4 本章小节.....39
第六章 总结与展望.....40
6.1 总结.....40
6.2 展望.....40
致 谢.....41
参考文献.....42
外文资料原文.....44
外文资料译文.....45

第一章 绪论

1.1 研究背景及意义

最近十几年是神经网络研究出现飞速进展的时代。从 2010 年左右，一些大型的公共数据集开始对公众开放。一个名为 ImageNet 的大量图像集合包含大约 1400 万张带注释的图像，催生了 ImageNet 大规模视觉识别挑战赛。2012 年，卷积神经网络首次用于图像分类，这导致分类错误显着下降（从近 30% 下降到 16.4%）。2015 年，Microsoft Research 的 ResNet 架构达到了人类水平的精度。

从那时起，神经网络在许多任务中都表现出非常成功的行为：

表 1-1 神经网络进展

时间	实现达到人类水平
2015	图像分类
2016	会话语音识别
2018	自动机器翻译
2020	图片说明

在过去的这几年里，我们有幸见证了大型语言模型的巨大成功，如 BERT 和 GPT-3。这主要是因为有大量可用的通用文本数据，使我们能够训练模型来捕获文本的结构和含义，在通用文本集合上预先训练它们，然后将这些模型专门用于更具体的任务。与此同时，视觉领域也在大模型方向进行了各种尝试，诸如 CLIP 这样的预训练的视觉语言模型在不同的下游视觉任务上展现了强大的零样本泛化性能；通过视觉输入提示能够执行与类别无关分割的 SAM 等等。

随着神经网络领域在大数据量的方向上越走越远，与之匹配的算力资源的单一和稀缺问题也变得越来越明显。尤其当我们把目光放到边缘计算这种有低功耗需求的领域，GPU 体积大，能效比较差的问题已经变得无法忽略。尽管基于 ASIC 设计实现专用加速器可以完美的解决这一问题，但 ASIC 设计研发周期长，灵活性也较差。基于 FPGA 的加速器设计刚好可以避免前面所提到的这些问题，FPGA 计算资源丰富、设计灵活且有着较低的功耗，基于 FPGA 实现新的特殊模型的边缘、端点和数据中心的计算需求变得越来越受欢迎。

因此，本课题根据卷积神经网络的特点，以目标检测与嵌入式领域的经典轻量化模型 Yolo4 tiny 为着手点，对基于 FPGA 的 CNN 优化实现技术进行了研究。

1.2 研究现状

由于目前大部分目标检测网络都依托于 GPU 进行训练，而 GPU 平台提供了丰富的浮点运算资源，因此现有的大部分网络都是基于浮点数进行训练和部署的。但为了减少功耗，进一步配合现有的 FPGA 计算资源 DSP，我们需要对于网络进行量化处理。学术界已经对各种线性或非线性的训练时量化进行了大量的探索，论文[7]通过高压缩比和通过采用高效的位操作显著减少了吞吐量时间在大量研究中脱颖而出。然而量化模型和全精度的模型之间仍然存在着不可忽略的精度差距，包括量化的离散型及其有限的表征能力[14]，梯度逼近不可微量化函数的难度[15]，以及阻碍优化的量化振荡[16]。

卷积神经网络内大部分运算操作都是通过卷积来实现的，因此，加速方案硬干专注于并行计算的管理以及跨多级存储器（如片外动态随机存取存储器（DRAM）、片上存储器和本地寄存器）的数据存储和访问的组织。在 CNN 中，卷积包含四个层次的循环执行，这些循环沿着内核和特征的映射进行滑动，这为并行化设计、计算排序和存储管理提供了一个很大的优化空间。一方面，学术界早已对卷积中涉及到的循环优化策略[13]进行了充分的讨论，例如循环展开、平铺和交换；另一方面，也有研究[1]对于采用这些技术优化硬件设计过程中的性能进行了细致的分析。

1.3 研究内容及设计指标

1.3.1 研究内容

在对网络进行层融合之后，本文首先基于 Google 团队发表，TF-Lite 所采用的训练后线性量化方法对网络进行了量化。之后本文基于论文[10]对卷积循环进行融合和再展开操作，使得加速器中的并行计算阵列对于 Yolo4 tiny 网络更加适配且可以得到充分的利用，并采用论文[12]对存储结构的设计，更加适配本文所使用的硬件平台和网络结构。

最终，我们基于 ZYNQ-7000 芯片配置 SoC 系统，设计了对应的驱动和软件，通过运行测试程序来验证硬件加速器和整个加速系统的功能，并对硬件测试的结果进行深入分析。通过将 ARM 处理器与可配置的硬件加速器相结合，我们可以为面向 IOT 的终端设备降低设计成本，同时实现更高的工作能效比。

1.3.2 设计指标

硬件加速器采用可调整的参数化设计，以满足对于其他网络的移植灵活性。

我们使用 Xilinx ZYNQ-7000 FPGA 开发板来实现和验证这个加速系统。在工作频率为 100MHz 的情况下，加速器系统的运行算力为 100GOPS，系统的功耗低于 5.0W，能效比达到 20GOPS/W。

1.4 论文组织结构

本文主要的工作重点为卷积神经网络硬件加速器和基于 ZYNQ 的异构加速系统设计，最终将利用 YOLOv4-Tiny 网络模型对所设计的硬件加速系统在仿真环境和 FPGA 测试环境进行验证，论文的主要分为以下五个章节：

第一章为绪论，首先简要介绍了近期神经网络这一研究领域的进展，阐明了开发低能耗高效率的算力平台的意义，然后对于神经网络量化和硬件加速的研究现状进行了介绍，之后明确了本文的主要研究内容和最终实现的设计指标，最后介绍本文的整体结构安排。

第二章为算法层面的分析与设计，本章首先介绍了卷积神经网络的基本原理和组件，之后对于本文采用的 YOLOv4-Tiny 算法的结构和训练策略进行了讲解。此外，本文还详细介绍了层融合和训练后线性量化等网络优化操作，以满足后续部署与设计硬件加速器的需求。

第三章为本文的 SoC 系统搭建思路。首先简单介绍了硬件平台的资源情况、总线接口以及中断系统，之后对于目标检测系统进行了详细的功能分析与软硬件任务划分论证，对本次软硬件协同设计的流程进行了阐述，最后详细介绍了 PS 端的各部分功能块的设计方案。

第四章为本文的加速器硬件设计思路和细节。首先我们对于 YOLOV4-Tiny 算法各个部分对于并行化计算和处理的需求进行了分析，优化了卷积循环算法，并设计了对应的卷积运算通路模型，同时也设计了上采样、LeakyRelu 层和池化运算层，并进行了相应的性能分析。

第五章为本文软硬件设计验证的性能记录。我们首先对网络移植到 ZYNQ 系统下 CPU 进行部署并对各个组件的延迟进行了测量，之后基于 Vitis HLS 平台对于各个加速器部件进行测试调优，最终进行目标检测系统的整体验证。

第二章 YOLOv4-Tiny 算法基础

本章首先介绍了卷积神经网络的基本原理和组件，之后对于本文采用的YOLOv4-Tiny 算法的结构和训练策略进行了讲解。此外，本文还详细介绍了层融合和训练后线性量化等网络优化操作，以满足后续部署与设计硬件加速器的需求。

2.1 YOLOv4-Tiny 算法

本节将从卷积神经网络的基本原理讲起，逐个介绍在 YOLOv4-Tiny 中所涉及到的基本网络结构，以及基本的基于反向传播的训练原理。

2.1.1 卷积神经网络

CNN 是相对来说最简单也最基础的计算机视觉深度学习网络，通过大量的可训练卷积核进行卷积运算可以在训练后对目标特征进行识别和分类。CNN 的结构受视觉皮层的组织和功能的影响。它被设计成类似于人脑神经元之间的连接。[3]

2.1.1.1 输入

在数字图像中，根据图像存储的协议和分辨率不同，具体存储排布也有一定区别。对于一个常见的 RGB 图片来说，范围从 0-255 的连续像素以三维矩阵的排列方式进行排列。如图 2-1 所示，它的像素值指定每个像素的亮度和色调。

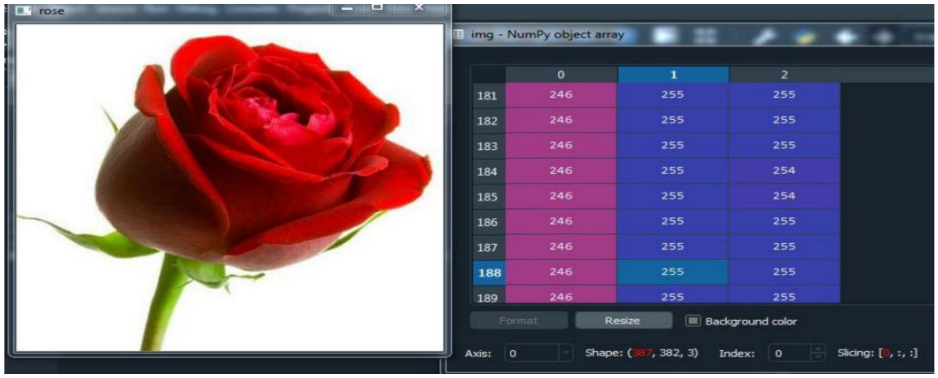

图 2-1 图片在计算机中的存储

2.1.1.2 卷积层

卷积层是 CNN 体系结构中非常重要的一层。它将图像作为输入，并通过使用不同大小的可训练的卷积核，对数据进行处理，如图 2-2 所示。

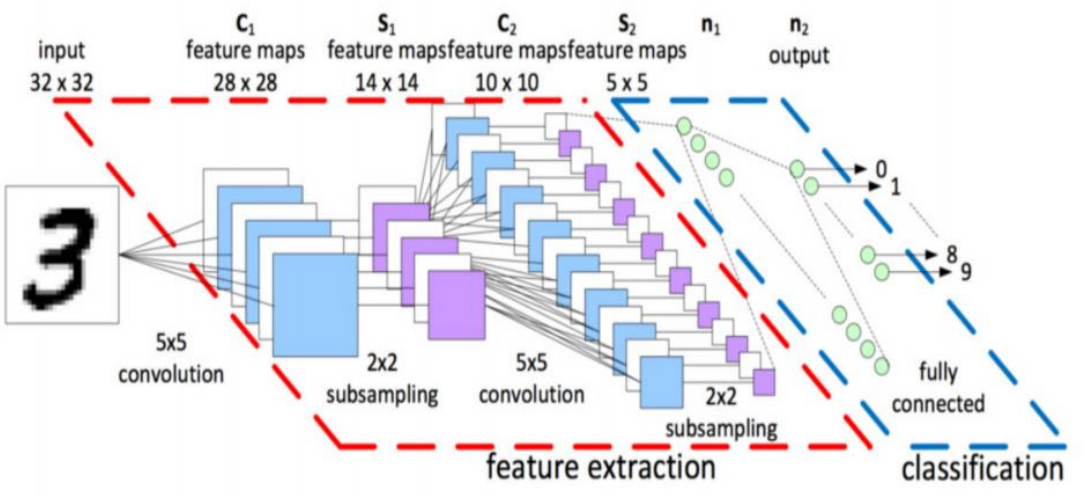

图 2-2 多个卷积层对 MNIST 数据集的特征提取

在图 2-3 和图 2-4 中，3×3 大小的卷积核在输入图像上从左上角开始滑动，卷积核与图像的重叠值一一相乘，然后求出的总和即为对应的输出特征的第一个值，之后重复执行这一操作，直到遍历整个输入图像。

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

1	0	1
0	1	0
1	0	1

图 2-3 卷积示例：输入特征（左），卷积核（右）

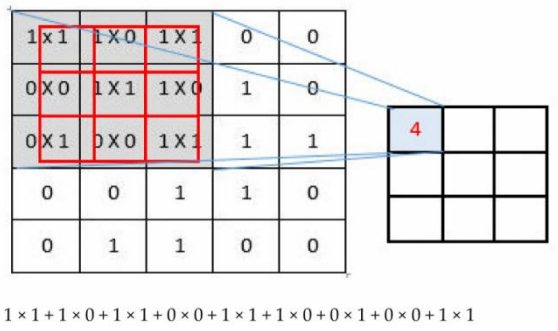

图 2-4 卷积计算示意

当图像有许多通道时，内核具有与输入图像相同的深度，例如 RGB(红、绿、

蓝)。如图 2-6 所示，卷积核拥有和输入特征匹配的深度，每一个输入通道下依次执行卷积操作后，不同输入通道的结构相加，最后加上偏置，得到输出。于此同时，卷积核还存在输出通道，不同的输出通道往往对于特征的提取目标是各异的。也就是说，输出矩阵对应的每个神经元其实都存在着重叠的接受野。第一个卷积层的目标通常是包括梯度方向、边缘、颜色等低级特征的提取。最终我们通过添加多个卷积层来适应高级特征，为复杂目标的实现提供可能。图 2-5 展示了输入通道为 3，一个输出通道下卷积层的运算步骤。

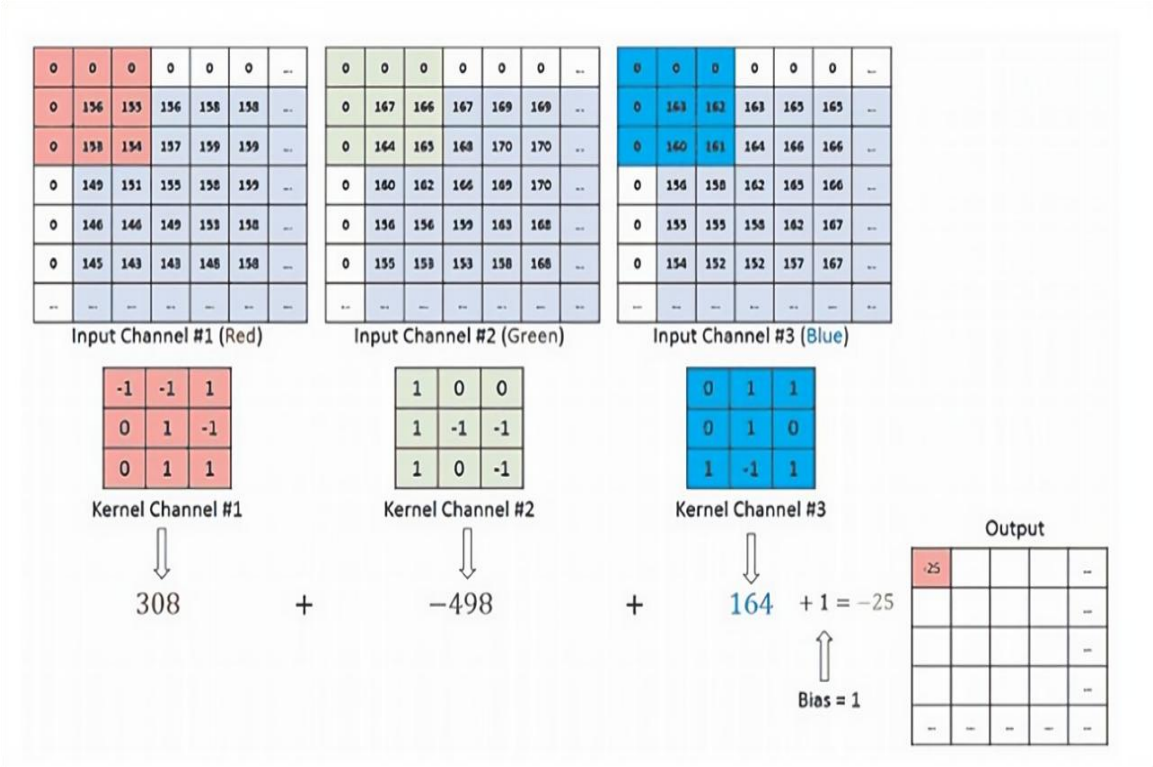

图 2-5 RGB 图像的矩阵运算

CNN 以其自动提取特征的能力而闻名。该操作产生两种结果:

1. 一种与输入相比较，卷积特征的维数减少的类型;

2. 维数没有减少而是增强或保持的类型。填充是用来完成这个任务的。

在 CNN 中经常使用填充，以防止每层特征图的大小缩小。如图 2-5 中所示的输入特征的边缘都填充了 0 元素。例如，将 5 × 5 × 1 的图像增强为 7 × 7 × 1 的图像，然后将其应用于 3 × 3 × 1 的核，可以观察到复矩阵的维数为 5 × 5 × 1，如图 2-6 所示。它表示输出图像与输入图像具有相同的尺寸(相同的填充)。如果在没有填充的情况下执行相同的过程，则可以在输出中接收到尺寸减小的图像。因此，一个 5 × 5 × 1 的图像将变成一个 3 × 3 × 1 的图像。

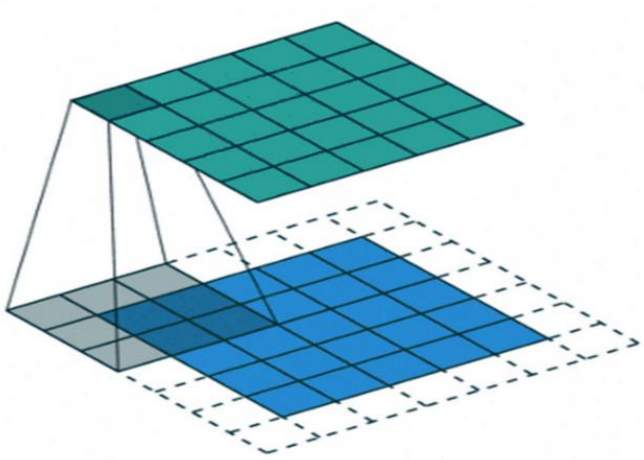

图 2-6 图像 Padding 后卷积

内核在转发过程中遍历图片的宽度和高度。它会产生有关的接收区域的视觉表征。它生成一个激活图，一个图像的二维表示，显示内核在图像的每个空间位置的响应。步长是内核滑动时的大小。假设输入图像的大小为 W × W × D。如果空间维数为 F，步幅为 S，填充量为 P 的核数未知，则输出体积可由下式计算:这将产生大小为 Wout × Wout × Dout 的输出。

2.1.1.3 池化层

随着网络层数的增多，为了防止对于特征的过拟合，有时候我们需要通过池化操作实现网络特征分辨率的降低，即缩小卷积特征的空间大小。同时，维数降低后导致后续参数量也大幅降低，这有效的降低了计算机的计算功耗和前向推理的延迟。也有研究指出，池化也有利于提取位置和旋转不变量的特征。常见的池化有两种形式：最大值池化和平均池化。池化层的输入是特征矩阵。通过 N ×N(在上述示例中为 2 × 2)的核在矩阵中移动，并根据不同的计算策略来依次计算对应位置的输出元素。

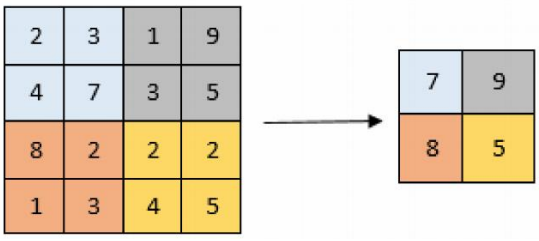

图 2-7 最大池化

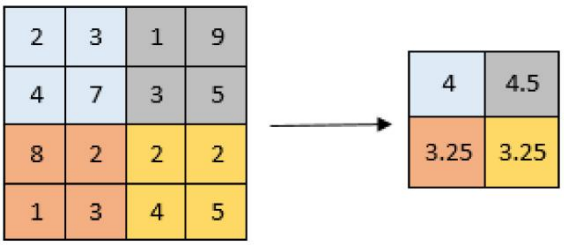

图 2-8 平均池化

2.1.1.4 激活函数（非线性层）

激活函数是为整个网络提供强大拟合能力的最关键的组件。我们很容易验证，无论神经网络中采用了多少的卷积层，输出和输入都是线性关系，与没有中间的隐藏层所能达成的效果是一致的，即最原始的感知机，此时整个网络的拟合能力就非常有限。引入非线性运算层后，实现对各种复杂函数的逼近相对来说会变得更加容易。

对于激活函数来说，当输入趋于最大值和最小值时的导数都趋于 0，则称为饱和激活函数。随着层数增多，小于 1 的导数不断相乘进而会导致梯度消失，这使得基于梯度的学习会变得困难。饱和激活函数主要是 Sigmoid 函数和 tanh 函数，由于以 Relu 及其变体为代表的一系列非饱和激活函数，相比于前者在避免梯度消失具有极大的优势，如今基本很少在网络中使用饱和激活函数。

2.1.1.5 上采样层

上采样是池化的逆运算，起到放大特征，提高图像的分辨率的作用。比较常见和基础的是通过线性插值来实现上采样。其中尤属双线性插值使用最为广泛，尽管它的计算上相较于其他插值策略更为复杂，但相较于卷积层的计算资源消耗量便可忽略不计了，除此之外的其他插值方式还有最近邻插值、三线性插值等。我们后续在训练的 YOLOv4-tiny 中使用的就是最近邻插值，这种插值方法比较容易交给硬件实现。

2.1.2 YOLOv4-Tiny 算法

本节将首先介绍目标检测的基本结构，之后具体介绍 YOLOv4-Tiny 算法的结构、训练策略以及对应的解码方法

2.1.2.1 目标检测网络基本结构

目标检测是视觉神经网络的一个分支领域，神经网络的输出目标是检测每一

帧图片中出现的目标类别物体，并画框圈出位置。在这一领域中，我们把搭建的整个网络称为检测器，结果近几年的不断深入研究，近年来发展起来的目标检测器通常由三部分组成，在输入图像张量上进行预训练的主干和用于预测对象类别和边界框的头部，以及两者之间用于收集不同阶段的特征图的颈部。

在目前的业界中，目标探测器的各个组件的常用模型如下表所示：

表 2-1 目标检测网络组件的常用模型

网络组件	常用模型	
Input	Image, Patches, Image Pyramid	
Backbones	VGG16, ResNet-50, SpineNet, EfficientNet-B0/B7 , CSPResNeXt50 ,CSPDarknet53	
Neck	Additional blocks	SPP , ASPP, RFB, SAM
	Path-aggregation blocks	FPN,PAN ,NAS-FPN , Fully-connected FPN, BiFPN, ASFF, SFAM
Heads	Dense Prediction (one-stage)	RPN , SSD , YOLO, RetinaNet(anchor based) CornerNet , CenterNet , MatrixNet, FCOS (anchor free)
	Sparse Prediction (two-stage)	Faster R-CNN , R-FCN , Mask RCNN (anchor based) RepPoints (anchor free)

2.1.2.2 YOLOv4-Tiny 网络结构

YoloV4-Tiny 作为 YoloV4 的简化版，为了提高模型部署时的能耗比，对网络结构进行了高效的精简，将原本 YoloV4 约 6000 万的参数量直接精简到了 600 万。精简后的 YoloV4-Tiny 仅通过两个探测器头部，就实现了分类和回归预测的功能。

对于主干网络部分来说，精简后的 YoloV4-Tiny 使用 CSPdarknet53_tiny 作为主干特征提取网络。相较于原来的 CSPdarknet53，将激活函数重新修改为更方便硬件通过移位进行计算的 LeakyReLU。

CSPdarknet53_tiny 模型的主干结构通过将原始残差块拆分为左右两部分来简化原始残差块堆栈。主干部分继续经过中间层进行前向推理，而另一部分作为剩余以类似残差边的形式，直接连接到末端。值得注意的是主干网络的通道分割，该模型在主干进行 3x3 卷积后，将通道分成两部分。利用该特征提取网络，获得两种形状的有效特征层，并将其传递到增强的特征提取网络中，用于 FPN 的构建。

在网络中反复使用到的 BasicConv 组件是卷积+ BN + LeakyReLU 的简单结构，而 Resblock_body 则更复杂，由四个 BasicConv 块和一个 MaxPooling 块组成。输入数据在 conv1 之后被分成两部分，从而在 conv2 之后计算 route1。Route1 通过

conv3 进一步处理，与结果连接，并输入到 maxpool 以生成最终输出。

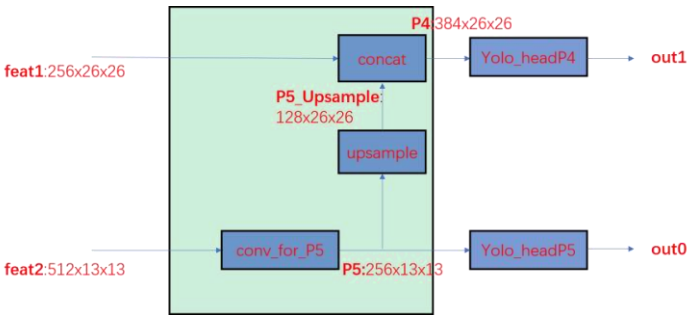

图 2-9 FPN 结构

FPN 主要结合最初获得的两个层的特征。将 feat2 卷积后得到的有效特征层进行上采样，然后与 feat2 进行叠加后再卷积得到 out1。上采样前的特征直接经过卷积得到 out0 。

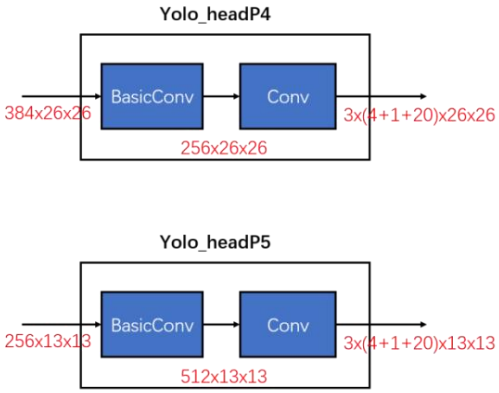

图 2-10 YOLO head 结构

在特征利用部分，我们前面颈部输出得到的 out0 和 out1 将分别经过两个头进行解码，总共提取出两个特征层。其中输出层的维度为(13,13,75)和(26,26,75)。因为模型使用了 voc 数据集，其标注总共有 20 个类，于此同时由于 YoloV4-Tiny 每个特征层只使用 3 个先验框，所以最后一个维度中的 75 包含了 3×(4+1+20)，括号内的数字分别代表 x_offset、y_offset、h 和 w、置信度、分类结果。

2.1.2.3 Mosaic 数据增强

Yolov4 中使用的 Mosaic 数据增强的灵感来自于 2019 年底提出的 CutMix 数据

增强方法。CutMix 涉及两幅图像的合并，而 Mosaic 数据增强则通过采用随机缩放、随机裁剪和随机排列来集成四幅图像。在常规的项目训练中，小目标的平均精度(AP)一般要比大中型目标低得多。但对于 Coco 数据集来说，难点还在于小目标的不均匀分布和较大的占比。

表 2-2 目标类型定义

目标类型	最小矩形区域	最大矩形区域
小型目标	0*0	32*32
中等目标	32*32	96*96
大型目标	96*96	oo*oo

表 2-3 COCO 数据集中不同大小数据集占比

COCO 数据集	小型目标	中等目标	大型目标
方框面积总占比(%)	41.4	34.3	24.3
图片包含目标的概率(%)	52.3	70.7	83

如上表所示，Coco 数据集中小型目标的比例达到 41.4%，超过了中型和大型对象的数量。然而，只有 52.3%的训练集图像包含小型目标，而大中型目标的分布相对更加均匀。

为了应对这种情况，Yolov4 的作者采用了 Mosaic 数据增强技术。基本步骤包括:1;一次读取四个图像。2. 对四个图像应用转换，如翻转、缩放和颜色空间变化，将它们排列在四个不同的方向上。3. 结合图像及其相应的边界框。

该方法通过随机利用四幅图像丰富数据集，随机缩放，然后随机组合，从而显著增强了检测数据集。特别是随机缩放引入了大量的小对象，提高了网络的鲁棒性。此外，这种方法有助于减少对大量 GPU 资源的需求。虽然有些人可能会认为随机缩放和传统的数据增强可以达到类似的结果，但作者认为许多人可能只能访问一个 GPU。因此，通过在训练过程中使用马赛克增强，可以直接从四个图像中计算数据，从而消除了对大型 Mini-batch 大小的需要，并且只需一个 GPU 就可以获得有效的结果。

2.1.2.4 损失函数与 NMS（非极大抑制）

YOLO V4-tiny 的损失函数主要分为三部分：

（1）bounding box regression 损失：这部分采用 CIOU[2]（Complete IoU Loss），具体计算如下公式：

$$\mathcal{L}_{CIoU}=1-IoU+\frac{\rho^2\left(\mathbf{b},\mathbf{b}^{gt}\right)}{c^2}+\alpha v \tag{1}$$

其中有：

$$IoU=\frac{\left|B\cap B^{gt}\right|}{\left|B\cup B^{gt}\right|} \tag{2}$$

$$\alpha=\frac{v}{(1-IoU)+v} \tag{3}$$

$$v=\frac{4}{\pi^2}\left(\arctan\frac{w^{gt}}{h^{gt}}-\arctan\frac{w}{h}\right)^2 \tag{4}$$

Bgt = (xgt;，ygt， wgt， hgt) 是真实框，B = (x，y，w，h)是预测框。b 和 bgt 表示 b 和 bgt 的中心点， ρ 是欧几里得距离，c 是覆盖两个方框的最小方框的对角线长度。如果真实框和预测框的宽度和高度相似，那么惩罚项将没有任何影响，因为 v 将为 0。因此，从直观上看，这个惩罚项的目的是控制预测框的宽度和高度，使其尽可能快速接近地面真值框的宽度和高度。

（2）置信度损失

$$lobj=\lambda_{\text{noobj}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{\text{noobj}}\left(c_i-\widehat{c_i}\right)^2+\lambda_{obj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{obj}\left(c_i-\widehat{c_i}\right)^2 \tag{5}$$

（3）分类损失

$$lcls=\lambda_{\text{class}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{obj}\sum_{c\in\text{ classes}}p_i(c)\log\left(\widehat{p_i}(c)\right) \tag{6}$$

其中有 S: 代表 grid size， S^2 代表 13x13,26x26 即 Yolohead 输出的方框大小；B: box；$1^{obj}_{i,j}$: 如果在 i,j 处的 box 有目标，其值为 1，否则为 0；$1^{nobj}_{i,j}$: 如果在 i,j 处的 box 没有目标，其值为 1，否则为 0。

本文最终基于 pytorch 中的 BCEWithLogitsLoss 实现了上述置信度损失和分类损失的计算。最终整体 Loss 的训练效果如下图所示：

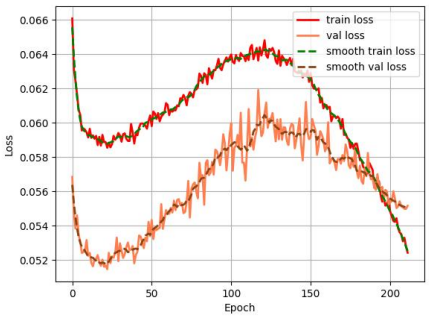

图 2-11 网络训练 Loss

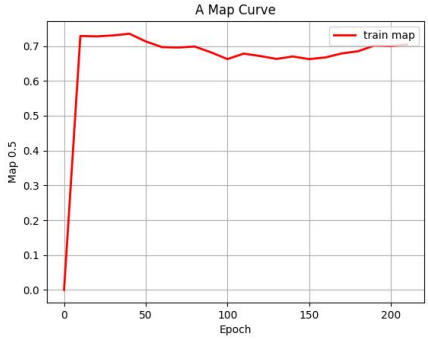

图 2-12 网络训练 maP

在最初的 NMS 中，IoU 度量是用来抑制冗余检测盒的，其中重叠区域是唯一的因素，对于有遮挡的情况往往会产生错误的抑制。YOLO V4 采用了 DIoU 来优化了这部分计算。其中 DIOU 和我们前面的 CIOU Loss 属于同一种策略。

此时，最终解码得到的评分可以由下式得到：

$$s_i = \begin{cases} s_i, IoU - \mathcal{R}_{DIoU}\left(\mathcal{M}, B_i\right) < \varepsilon \\ 0, IoU - \mathcal{R}_{DIoU}\left(\mathcal{M}, B_i\right) \geq \varepsilon \end{cases} \tag{7}$$

$$\mathcal{R}_{\mathrm{DIoU}} = \frac{\rho^2\left(\mathbf{b}, \mathbf{b}^{gt}\right)}{c^2} \tag{8}$$

2.2 网络优化

为了提升整个部署后网络的性能，我们需要对基于 Pytorch 训练的网络进行优化，一方面契合我们之后设计的硬件加速器的基本数据类型，即定点数；另一方面，我们需要把仅对训练有用的 BN 层参数融合到卷积层里，以便提升速度的同时，更方便对网络全方面的设计加速器。

为了节约一定的时间成本，我们首先决定直接对于训练后的网络进行量化。我们首先将模型基于 numpy 将网络的前向推理重新编写了一遍，以便基于自己的量化策略灵活优化网络。在此基础上，对其进行了层融合与量化。

2.2.1 层融合

在我们要部署的 yolotiny-4 中每一层基本的卷积层都需要经过 BN 层再给到激活函数，但 BN 层在网络中存在的作用更多的体现在对于训练时的帮助，即借由优化数据分布提高训练的速度和稳定性，而如果将其直接部署则会平白无故增加很大的计算量。

在 pytorch 中，训练完毕的 BN 层参数包括有运行均值，运行方差，gamma，

beta，其中的运行均值和运行方差是大批量数据训练时得到的参数，在部署时则不进行计算，直接调用，由此我们可以得到 BN 层部署时的计算逻辑：

$$W_{\text{fused}} = \frac{W_{BN}}{\sqrt{\text{Var}+\text{Eps}}} \cdot W_{conv} \tag{9}$$

$$Out_{BN} = \frac{W_{BN}}{\sqrt{\text{Var}\ +\text{Eps}}} \cdot x + Bias_{BN} - \frac{W_{BN} \cdot \text{mean}}{\sqrt{\text{var}\ +\ \text{eps}}} \tag{10}$$

$$Bias_{\text{fused}} = Bias_{BN} - \frac{W_{BN} \cdot \text{mean}}{\sqrt{\text{var}\ +\ \text{eps}}} \tag{11}$$

显然，层融合是和原网络完全等价，不引入误差的。

2.2.2 线性量化

对于本文采用的整个 YOLO 网络来说，在层融合后需要量化的仅有卷积层和激活函数层，其他的层如上采样和池化等，都是仅存在拓扑关系和关系运算的层，与数据的精度无关。下面我将分别介绍对于卷积层和 LeakyRelu 层的量化，基础方案参考了工业界广泛使用的 Google TFLite 量化方法。

2.2.2.1 卷积层量化

r 表示浮点实数， q 表示量化后的定点整数，两者的基本换算公式为：

$$\mathbf{r} = \mathbf{S} \cdot (q - Z) \tag{12}$$

$$q = \text{round}\left(\frac{r}{S} + Z\right) \tag{13}$$

若要保证最大化利用位宽，[1]则其中 S（scale），Z（zero point）可由下式得到：

$$S = \text{round}\left(\frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}\right) \tag{14}$$

$$Z = \text{round}\left(q_{\max} - \frac{r_{\max}}{S}\right) \tag{15}$$

假设卷积的 weight 为 w，bias 为 b，输入为，输出的值为 a。由于卷积本质上就是矩阵运算，因此可以表示成：

$$a = \sum_{i}^{N} w_i x_i + b \tag{16}$$

为了方便，我们直接将 bias 的零点直接设为 0,之后我们确定权重与输入的量化后精度为 8bit，bias 的精度为 32bit，Sb 等于 SwSx，将量化转换式代入化简可得：

$$q_a = \sum_{i}^{N} M\left(q_w - Z_w\right) q_x + Z_x \cdot M \sum_{i}^{N} \left(Z_w - q_w\right) + M \cdot q_b + Z_a \tag{17}$$

其中：

$$M=\frac{S_w S_x}{S_a}=2^{-n}\cdot M_0 \tag{18}$$

则对于量化后的卷基层而言，有：

$$W=M\cdot\left(q_w-Z_w\right) \tag{19}$$

$$Bias=Z_x\cdot M\sum_i^N\left(Z_w-q_w\right)+M\cdot q_b+Z_a \tag{20}$$

由于其中 M 也经由量化处理，则新的参数都转化为了定点运算。

2.2.2.2 LeakyRelu 层量化

LeakyRelu 层所作的处理如下式：

$$LeakyReLU(x)=\begin{cases}x & x>=0\\ \alpha x & x<0\end{cases} \tag{21}$$

与前面的推导类似，我们最终可得：（其中 q2、q1 分别指输出与输入的定点数）

$$q_2=\begin{cases}\frac{S_1}{S_2}\left(q_1-Z_1\right)+Z_2 & q_1>=Z_1\\ \frac{S_1}{S_2}\left(\alpha_q-Z_1\right)\left(q_1-Z_1\right)+Z_2 & q_1<Z_1\end{cases} \tag{22}$$

$$S_{Relu}=\frac{S_1}{S_2}\cdot 2^{-n} \tag{23}$$

之后我们对于量化后产生的乘数因子 S_{Relu} 也进行了类似于卷积中 M 的处理，即对这个参数单独定点化。

2.2.2.3 整体量化

表 2-4 量化参数表（其中卷积量化为 8bit，M0 量化为 16bit，S_relu 量化为 15bit。）

网络层	M0	out_scale	S_relu
backbone/conv1	72	0.413527517	24424
backbone/conv2	319	0.490048886	31566
resblock_body1/conv1	243	0.427794964	26815
resblock_body1/conv2	149	0.163296858	50555
resblock_body1/conv3	183	0.153397687	21794
resblock_body1/conv4	253	0.237925586	20616
resblock_body2/conv1	193	0.139571948	35075
resblock_body2/conv2	222	0.125092397	16956
resblock_body2/conv3	114	0.143402302	16384
resblock_body2/conv4	212	0.189659769	16384
resblock_body3/conv1	138	0.155288185	25394
resblock_body3/conv2	206	0.160647439	17661

resblock_body3/conv3	116	0.180454503	16384
resblock_body3/conv4	232	0.211566533	17519
backbone/conv3	140	0.123423579	37322
conv_for_P5	216	0.139703795	27624
yolo_headP4/conv1	49	0.130539054	23452
yolo_headP4/conv2	99	0.134265063	
yolo_headP5/conv1	75	0.18341629	17319
yolo_headP5/conv2	70	0.171383158	
upsample/conv1	347	0.11809825	22267

对于整个网络都进行量化的话，我们首先需要严谨的确保每个相连网络的量化参数的选取策略是一致的，或者直接对相连部分的参数进行传递。对于整个网络的具体量化参数如上下表。

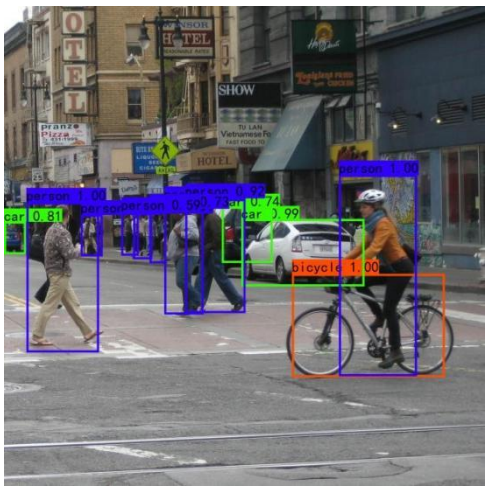

图 2-13 层融合后前向推理效果

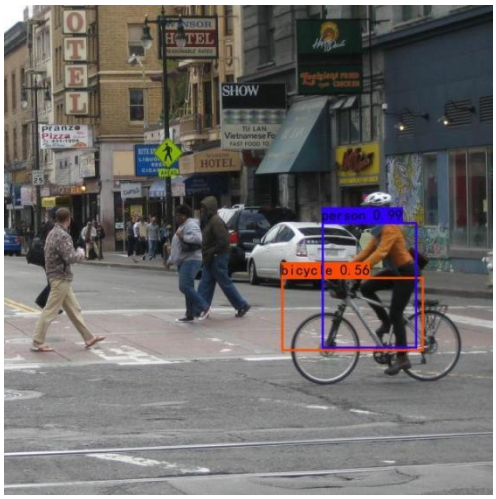

图 2-14 8bit 量化后前向推理效果

2.3 本章小节

本章首先从卷积神经网络的基本原理引入了视觉神经网络领域的一些基础概念，之后对于 YOLOv4-Tiny 算法中涉及到的基本网络结构进行了详细且通俗易懂的介绍。从最基础的图片所代表的数字张量作为输入，到一个基本的卷积层实现特征的提取，再到之后的池化对特征进行模糊或者上采样进行超分辨率特征提取，以及神经网络中的非线性层（激活函数），我们逐步对于本文采用算法的神经网络的基本组件在最低的维度得到了了解。

之后我们从目标检测网络出发，对于本文采用的 YOLOv4-Tiny 算法的结构和训练策略进行了更深层次的讲解和分析。我们首先介绍了通常情况下，目标检测网络的基本实现形式，之后我们具体介绍了 YOLOv4-Tiny 算法的拓扑结构，并在基本运算的基础之上，对于网络的拓扑关系进行了分析和讲解。本文还介绍了该算法训练时采用的一些策略细节：一方面在训练时对于数据进行了拼接的预处理，即 Mosaic 数据增强；另一方面在损失函数方面借助 Pytorch BCE 实现了置信度和目标的 Loss 计算，同时基于 CIoU 实现了对于 bounding box 的 Loss 的计算。最后本文还简单介绍了基于 DIoU 的 NMS 计算方法。我们最终通过 200 epoch（每 epoch 训练 xxx 次）得到了 maP 为 70 的模型可训练参数。

此外，本文还详细介绍了本文最终所采用的一系列网络优化操作，以满足后续部署与设计硬件加速器的需求。我们首先将网络中的 BN 层和原来的不含有 Bias 的 Conv 层融合为一个含有 Bias 的 Conv 层，之后对与涉及到计算的 conv 和 leakyRelu 层进行了统一的线性量化，在对于不同的量化参数进行了分析后，我们最终决定采用整体 8bit 量化，卷积乘数因子量化和激活函数乘数因子量化到 16bit，偏置量化到 32bit。最终量化后的网络经测试仍能很好的完成目标检测的任务。

第三章 软硬件协同架构设计

本文的设计目标是一个基于 ZYNQ 的目标检测软硬件协同的加速系统。本文以 SD 卡作为图片和权重的存储媒介，并通过 DDR 实现了对于 PL 和 PS 端的数据交互。通过搭建定时器中断和外部中断实现了对于 CPU 资源的高效利用和精准的延迟测量。本章首先简单介绍了硬件平台的资源情况、总线接口以及中断系统，之后对于目标检测系统进行了详细的功能分析与软硬件任务划分论证，对本次软硬件协同设计的流程进行了阐述，最后详细介绍了 PS 端的各部分功能块的设计方案。

3.1 ZYNQ-7000 平台

本研究使用的硬件平台是基于赛灵思 Zynq-7000 系列的 ZYNQ7020，如图所示。ZYNQ 系列芯片主要由双核 ARM Cortex-A9 处理器(Processing System, PS)和 FPGA 可编程逻辑(programmable logic, PL)组成，PS 和 PL 间通过高速 AXI 总线和一系列接口相互连接。通常情况下，研究人员在 PL 端为需要并行化或专有化加速的部件设计硬件逻辑，然后由 PS 端的 ARM 处理器控制数据流移动。本研究在 PL 端加速了卷积神经网络中大量的乘加操作以及池化、上采样、激活函数等运算，而 PS 端主要负责控制数据传输。

3.1.1 AXI 总线 (Advanced eXtensible Interface)与接口

在 ZYNQ 平台下，PL 端和 PS 端都有可以直接访问的 DMA 控制器。一方面，在处理系统(PS)中集成有硬核 DMA，我们可以利用官方提供的驱动进行配置和使用；在可编程逻辑(PL)中有可以配置和使用的软核 AXI DMA IP。各种接口方式的比较如下表所示：

表 3-1 ZYNQ 系统中不同接口比较

方式	优点	缺点	估计吞吐率
CPU 控制的 IO	软件简单 最少的逻辑资源 逻辑接口简单	吞吐率最低	<25MB/s

表 3-1 （续）

方式	优点	缺点	估计吞吐率
PS 的 DMAC	最少的逻辑资源 吞吐率中等 多个通道 逻辑接口简单	DMAC 配置有 一定难度	600 MB/s
PL 的 DMA 和 AXI_HP	吞吐率最高 多个接口 有 FIFO 缓存	只能访问 OCM 和 DDR 逻辑设计复杂	1200 MB/s(每个接口)
PL 的 DMA 和 AXI_ACP	吞吐率最高 延时最低 可选的 Cache 一致性	大块数据传输引起 Cache 问题 共享了 CPU 的互联带宽 更复杂的逻辑设计	1200 MB/s
PL 的 DMA 和 AXI_GP	吞吐率中等	更复杂的逻辑设计	600 MB/s

显然，通过 PL 的 DMA 和 AXI_HP 接口的传输可以支持大块数据的高性能传输，带宽高。即适用于本文的卷积硬件加速场景下的数据移动。该种传输在硬件层面的具体移动路径如图中标红所示：

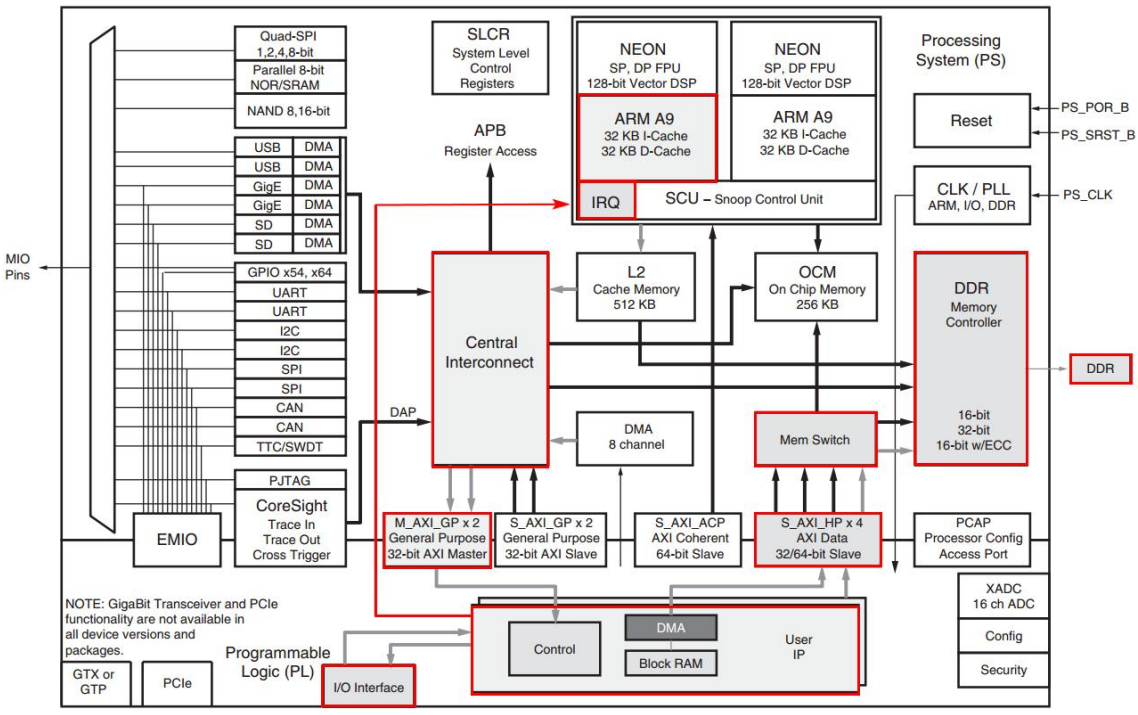

图 3-1 ZYNQ-7000 系统中 DMA 框架

本文使用 HLS 配置的硬件 IP AXI 总线上的数据传输便通过 HP 接口实现与 DDR 进行的交互。ZYNQ-7000 共有 4 个 HP 接口，每个 HP 接口都有控制和数据

FIFO，这些 FIFO 用于缓冲大数据量的突发传输，使 HP 接口成为高速数据传输接口的理想选择。

3.1.2 中断系统

在 ZYNQ 嵌入式系统中，有大量可用的定时器资源。两个 Cortex-A9 处理器各自有独立的 32 位私有计时器和 32 位看门狗计时器，并共享一个 64 位全局定时器(GT)。除此之外，PS 中还有两个三重定时器计数器(TTC)和一个 24 位的系统看门狗定时器(SWDT)。TTC 用于计算来自 MIO 引脚或 EMIO 引脚的信号的脉宽，每个 TTC 有三个独立的计数器。

定时器的系统框图如图所示：

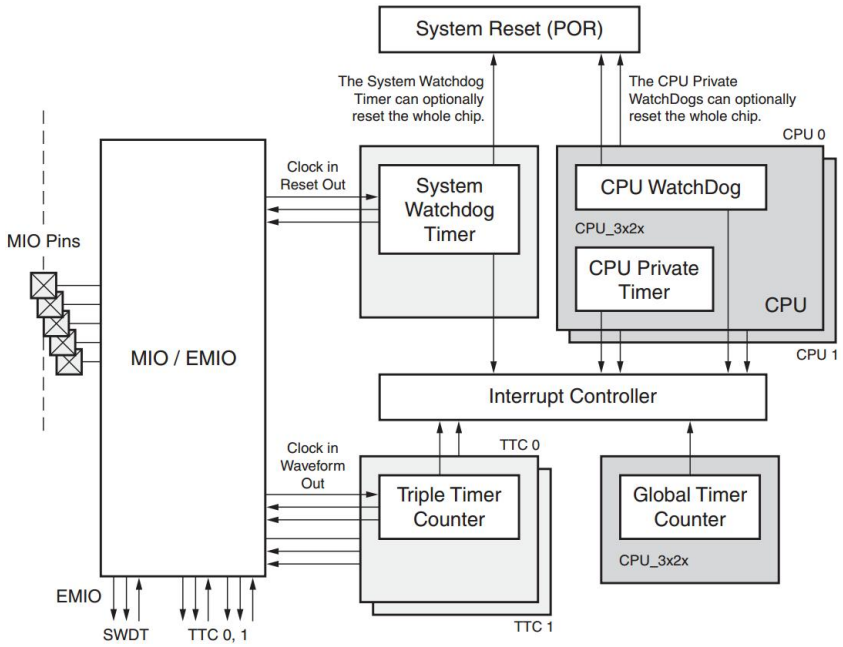

图 3-2 ZYNQ-7000 系统中定时器资源

本文基于其中的私有定时器来满足系统中的计时需求，其时钟频率为 CPU 时钟频率的一半，为 333.333Mhz。私有定时器的特性如下：

1、32 位计数器，递减计数，计到 0 产生中断；

2、有 8 位预分频；

3、可配置单次定时与自动重载模式；

3.2 软硬件任务划分

通过对软硬件任务进行合理的划分，我们可以发挥在软硬件各自的优势的同

时，快速开发出高性能的嵌入式设备。在 YOLOV4-Tiny 架构中，存在大量的卷积层，且涉及到了两种步进尺寸和卷积核尺寸，并且涉及多通道的特征图与权重数据，而且卷积层中包含了复杂的乘加运算。在进行卷积运算时，对于 CPU 来说，软件只能顺序执行每一步计算，而使用 FPGA 对数字电路进行设计时，可以利用其高并行性特点，同时读取多个像素值，再同时对其进行乘加运算，计算时间将大大减少。与此同时，YOLOV4-Tiny 架构中大部分卷积层都是直接连接激活函数层的，因此我们把激活函数在硬件中通过 MUX 直连卷积的输出。除此之外的池化层和上采样层，我们也做出了对应的硬件设计。

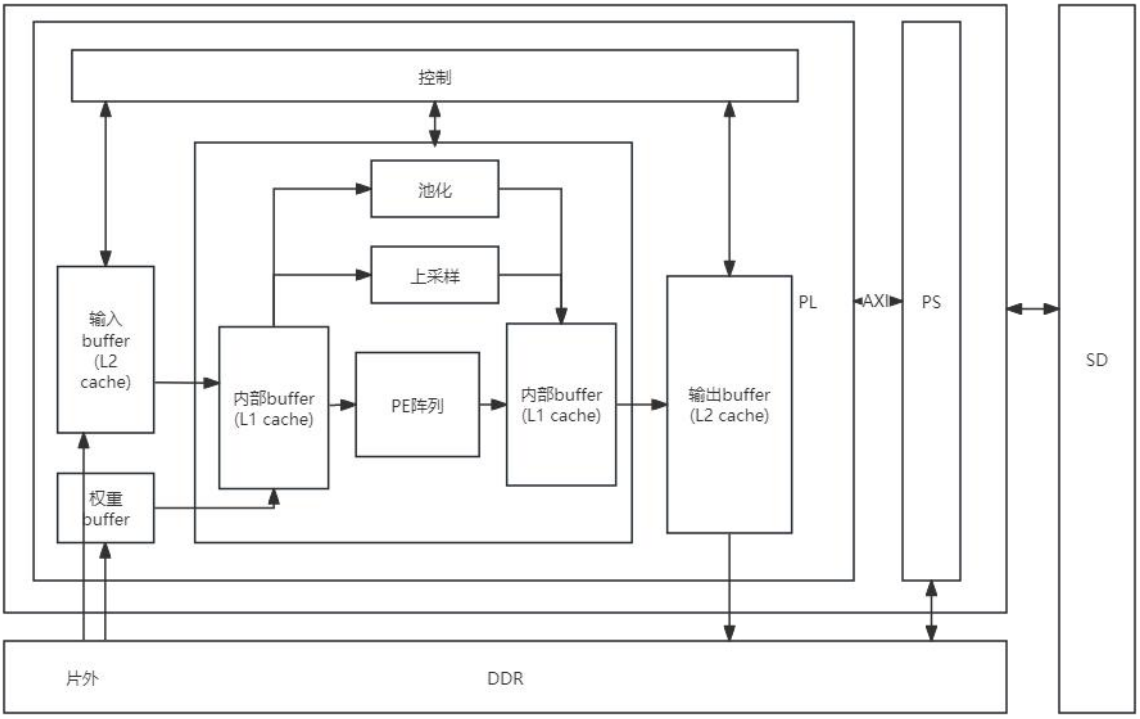

图 3-3 软硬件框架图

3.3 软硬件协同设计系统

针对 YOLOV4-Tiny 算法实现软硬件加速部署，协同设计的流程图可表示为下图，在对 YOLOV4-Tiny 算法的分析，人工软硬件功能划分后，分别建模设计并实现软硬件各自的模块以及二者之间交互的 AXI 接口，然后再对软硬件系统进行整体调试。

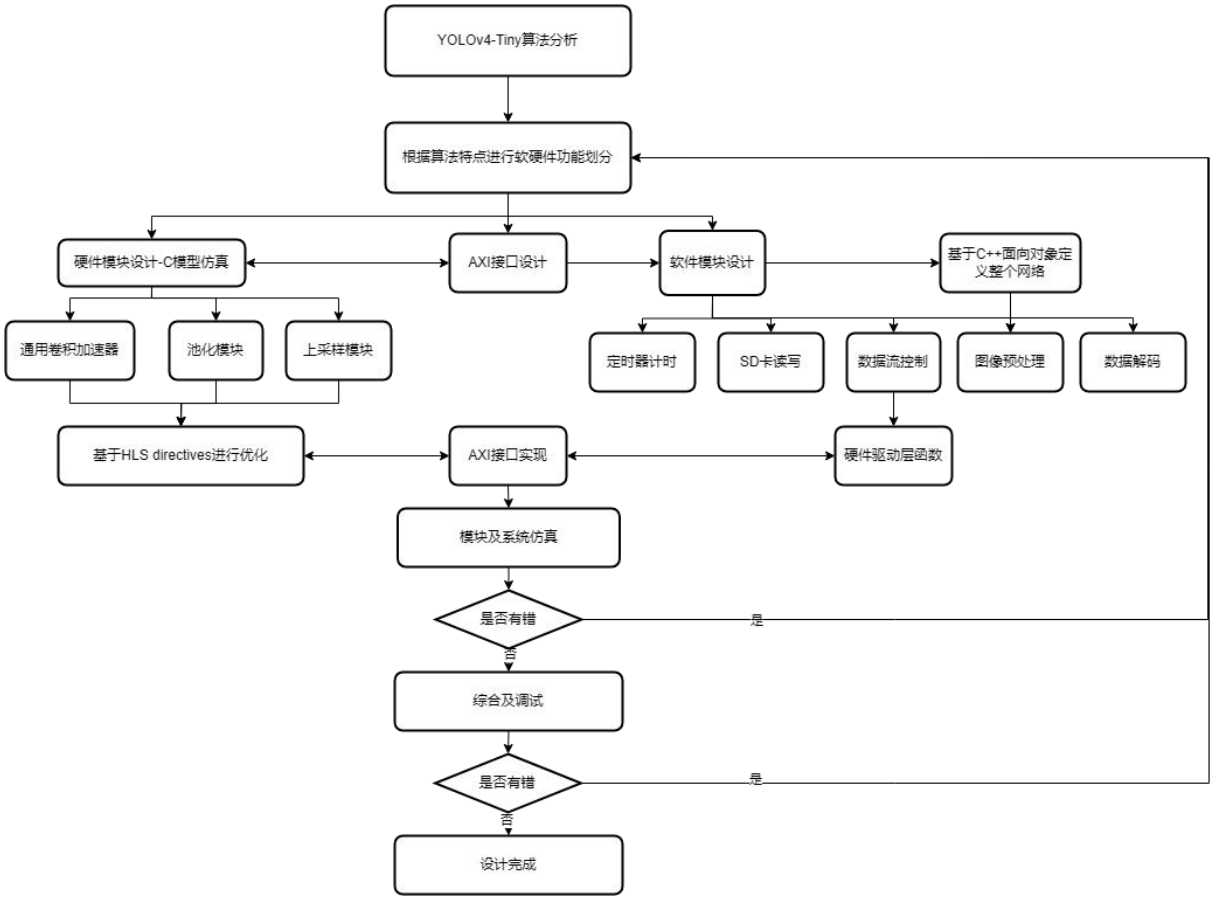

图 3-4 软硬件协同设计流程图

3.4 PS 端模块设计

PS 端主要负责控制 SD 卡，DDR3 之间的内存管理，以及控制整个算法的数据流。数据首先需要通过 PS 端进行一定的处理：从 SD 卡导入的图片需要进行尺寸剪裁的预处理；各个层的参数与权重图在从 SD 卡导入后需要 PS 端 CPU 动态分配其在 DDR 的存储空间，进而传输给 PL 端。当 PL 端完成全部的运算，将对应数据存到 DDR 后，PL 端的结束信号便作为外部中断触发，CPU 进而对上一轮的数据进行处理并准备进行新一轮的计算。最终进行完全部网络的前向推理后，再将最终得到的结果经过 NMS 处理后，解码得到边框坐标，最终绘制出带有预测框的图片，并存储到 SD 卡中。

3.4.1 图像预处理

由于在 SD 卡存储的图片的大小与网络的输入尺寸 3*416*416 并不一致，因此需要对输入的图片进行缩放，采用双线性插值进行处理。

图 3-5 缩放前图片

图 3-6 缩放后图片

3.4.2 数据控制

数据控制模块基于 Vitis 自动生成的 HLS AXI IP driver，对本文设计的控制寄存器进行写入，实现控制信号的传输。同时利用对 M_AXI Slave 接口的初始地址进行改写，实现了 PS 端动态内存管理与硬件 PL 端数据交互的无缝衔接。其中的输入输出 AXI 接口的最大突发长度均设为 128，经过验证，在 HLS 生成电路时会生成类似于 DMA 的结构。

本文设计的加速器参数如下

表 3-2 AXI 控制寄存器

控制寄存器	功能描述
M0	卷积层量化产生的乘数因子
K_LOOP_MAX	K 循环边界
P_LOOP_MAX	P 循环边界
C_LOOP_MAX	C 循环边界
OUT_MAP_HEIGHT	卷积层输出张量的高度
OUT_MAP_WIDTH	卷积层输出张量的宽度
OUT_MAP_CHO	卷积层输出张量的通道数
IN_MAP_CHI	卷积层输入张量的通道数
IN_MAP_HEIGHT	卷积层输入张量的高度
IN_MAP_WIDTH	卷积层输入张量的宽度
ACTIVATION_EN	使能 leakyrelu 运算
S_RELU	leakyrelu 量化时产生的乘数因子

3.4.3 YOLO Head 解码

本文实现了在 PS 端对最后两个 YOLO head 的输出结果进行解码。解码的主要目的是进一步处理从主网络提取的特征，将特征解码成可以理解的对象类别、置信度、边界框等信息。解码首先需要利用 Sigmoid 函数对数据进行归一化处理，对每个“锚点”的置信度进行迭代，并通过与阈值评分的比较过滤掉可能包含可检测对象的“锚点”，最后经过非最大抑制(NMS)算法计算得到评分。NMS 算法所涉及到的具体公式与计算步骤在前文均已经详细提及，本节不再过多赘述。在计算结束后，将对两组数据解码得到的置信度累加，便可识别不同大小的对象。

3.4.4 计时统计

由于本文的核心目标是对于算法进行加速，因此在 PS 端对整个过程的每一个步骤进行精确的计时是非常必要的。我们使用到了 PS 端的私有定时器，其时钟频率为 333.333Mhz，我们通过设置计数器的重装载值，每隔 10ms 进入一次中断函数，以此实现对于时间的记录。

3.5 本章小节

本章首先简单介绍了硬件平台的资源情况、总线接口以及中断系统，在此基础上对于目标检测系统进行了详细的功能分析与软硬件任务划分论证。详细介绍了本文最终实现的 SoC 系统中 PS 端和 PL 端的分工，以及这个过程中的数据流形式。最后还详细介绍了 PS 端的各部分功能块的设计思路和方案。

第四章 硬件加速建模与设计

在前面的内容中，我们已经对算法进行了详细的分析与介绍，并对软硬件的工作进行了划分，还介绍了软件端的功能模块设计思路。本章我们将继续对目标检测系统的硬件进行建模分析与设计。

4.1 硬件加速需求分析

经过上一个章节对于 YOLOV4-Tiny 算法的网络结构的分析之后，我们可以知道该算法共包含三种基本的结构块，在这之下，经过层融合后，基本的层与运算主要有：

BasicConv 层：卷积运算+LeakyRelu，在整个网络中占比最大，而且消耗资源最多的部分，这将是我们进行硬件加速的重点。

张量剪切与拼接：在 FPN 和 Backbone 的 Resblock_body 中出现，我们可以通过 CPU 对地址偏移进行操作轻松实现。

池化与上采样：在 FPN 和 Backbone 的 Resblock_body 中出现，在网络中的占比较小，我们可以对数据流简单的建模之后进行简单的硬件设计，进行硬件资源与优化后对于延迟的减小进行权衡，来确定是否需要花费一部分资源进行对于这两种计算进行加速。

Yolo head 解码：遍历算法和 NMS 算法通过硬件实现相对麻烦，我们选择将该功能放到 PS 端去实现。

综上所述，我们将首先重点对于卷积运算的行为进行建模与分析，由于网络中有个别的层卷积运算后没有经过 LeakyRelu 运算，所以我们将卷积输出的结果经过一级 MUX 选择是否继续进行 LeakyRelu 层，除此之外的池化和上采样，则是与卷积相对并行的硬件模块。卷积层本身又包含了偏置运算、卷积乘运算以及量化因子乘法与移位运算，我们将偏置、量化因子乘与移位、LeakyRelu 运算共同作为辅助运算，在卷积乘运算完之后进行，这参考了 NVDLA 的架构，可以十分有效的减少硬件访存的次数。最终，辅助运算与卷积乘运算通路紧耦合，池化和上采样模块与卷积乘松耦合，整体共同构成最终的硬件加速器通路。

4.2 卷积运算通路建模

如前所述，常规的卷积操作可以通过嵌套循环实现，如图 4-1 所示，嵌套循环涵盖了输出通道、输出行和输出列、输入通道、卷积核高度和卷积核宽度的循环。

论文[4]对于优化卷积循环结构来设计对应的硬件进行了详细的讨论，由于循环变量的独立性，我们可以灵活地重新排列循环的顺序，分解循环以提高硬件执行效率。我们合并输出行和输出列的循环变量，进而分解为 p 循环和 pp 循环;同理，将输入通道的循环变量分解为 c 和 cc 的乘积，将输出通道的循环变量分解为 k 和 kk 的乘积。最终，我们将图 4-1 所示的传统的卷积算法表示的 6 个嵌套循环分解为 8 个嵌套循环，如图 4-2 伪代码所示。转换后的算法使不同大小的卷积操作的最内层循环的循环大小变成了固定的大小，这就为不同层不同尺寸的卷积都能充分的利用并行化的基本运算硬件单元提供了条件。

Algorithm 1 Traditional convolution

Input: W[Pof][Pif][K][K],In[Pif][S*Pc+K-S][S*Pc+K-S],Bias[Pof]
Output: Out[Pof][Pr][Pc]

```
Out[Pof][Pr][Pc] ← 0
for kr = 0 → K do
    for kc = 0 → K do
        for r = 0 → Pr do
            for c = 0 → Pc do
                for out = 0 → Pof do
                    for in = 0 → Pif do
                        Out[out][r][c]+=In[in][S*r+kr-1][S*c+kc-1]*W[out][in][kr][kc]
                        if (kr==K-1) and (kc==K-1) and (in==Pif-1) then
                            Out[out][r][c]+=Bias[out]
                        end if
                    end for
                end for
            end for
        end for
    end for
end for
return Out
```

图 4-1 传统卷积伪代码

Algorithm 2 Loop-8 convolution

Input: W[Pof][Pif][K][K],In[Pif][S*Pc+K-S][S*Pc+K-S],Bias[Pof]
Output: Out[Pof][Pr][Pc]

```
Out[Pof][Pr][Pc] ← 0
for k = 0 → ceil(Pof/Tp) do
    for p = 0 → ceil(Pr * Pc/Tp) do
        sum[Tp][Tk] ← 0
        for c = 0 → ceil(Pif/Tc) do
            for ky = 0 → K do
                for kx = 0 → K do
                    for pp = 0 → Tp do
                        if p*Tp¡Pr*Pc then
                            h ← (p * Tp + pp)/Pr
                            w ← ((p * Tp + pp))Mod(Pr)
                            for kk = 0 → Tk do
                                for cc = 0 → Tc do
                                    if (c*Tc+cc)¡Pif then
                                        sum[pp][kk]+=In[c*Tc+cc][h*S+ky][w*S+kx]*W[k*Tk+kk][c*Tc+cc][ky][kx]
                                    end if
                                end for
                            end for
                        end if
                    end for
                    for pp = 0 → Tp do
                        h ← (p * Tp + pp)/Pr
                        w ← ((p * Tp + pp))Mod(Pr)
                        for kk = 0 → Tk do
                            Out[k*Tk+kk][h][w]=sum[pp][kk]+Bias[k*Tk+kk]
                        end for
                    end for
                end for
            end for
        end for
    end for
end for
return Out
```

图 4-2 改进后的 8 层循环卷积

传统的卷积运算路径，是由外部存储器+DMA+乘累加阵列+累加器结构构成的，但这样简单的一级存储结构完全的浪费了数据重用的潜力，大大增加了加速器与外部存储器之间的数据读写操作次数。根据 Roofline 模型，这会影响乘法-累积阵列的效率，从而降低加速器的性能。

为了解决这个问题，我们在论文的指导下，设计了一个五阶段的流水线卷积操作通道架构，如图 4-3 所示。在这个架构中，我们添加了两级的片上缓存，L1 cache 存储在计算过程中需要重复使用的权重数据以及数据量较小的偏置矩阵和量化参数，L2 cache 存储切片后的原始输入特征和权重卷积核。原子计算阵列只需要从 L1 cache 加载数据即可，我们将 L1 cache 设计成高度展开的存储阵列，这将大大提升计算流水线的效率，于此同时，DMA1 将数据缓存到 L2 cache 实现了数据重用，大幅降低了加速器访问外部内存的频率，进而解决带宽瓶颈问题，同时更方便总线实施 Burst 读取操作，提升 AXI 总线的使用效能。

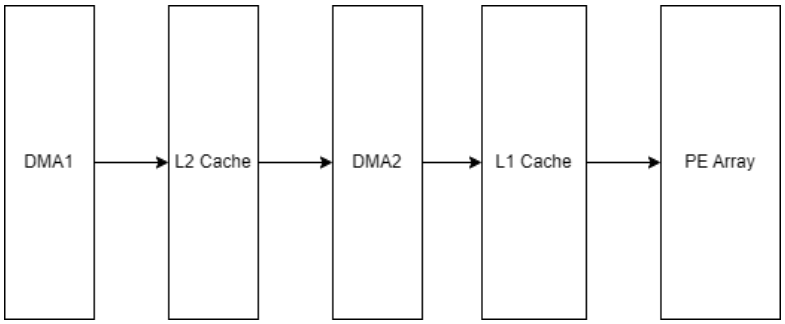

图 4-3 卷积运算通路

4.2.1 乘加器阵列建模

如图 4-2 算法所示，乘加阵列的规模由改进后算法的最内层循环，即原子运算所决定。为了将原子运算整个更方便的融入流水线，提升吞吐量，并进行并行化平铺处理，我们首先需要将权重数据映射到展开的 Tk×Tc 大小的乘加器阵列上，即准备好对应的 L1 Cache 缓存，这部分对应的运算可通过图 4-2 直观的分析。本文将图 4-4 整体计算称为内循环，内循环持续 Tp 次，在这个过程中权重和特征数据保持不变，并且一直存储在 L1 Cache 中，实现了数据的复用。

由于原子运算中的权重数据量是输入特征的 Tk 倍，即并行单元所需要的两种输入的缓冲长度有较大的差距，因此，如果直接简单地进行流水线加载数据，由于相同带宽的接口加载权重所消耗的周期数是加载输入特征的 Tk 倍，加速器的硬件在时间上的利用率会被更慢的缓存加载所拖垮。为了解决这个问题，本文设计的加速器将原子计算所涉及到的整个权重矩阵使用的 L1 Cache 完全展开，由于本

文所使用的片上 DSP 资源有限所导致的 Tc 与 Tk 实际上是较小的，因此完全展开所消耗的 LUT 和寄存器资源相对来说是完全可以接受的。

但是对于 DMA2 来说，数据的读取与存储和计算由于数据依赖所导致的流水线效率浪费仍然是不可忽视的，我们针对此还采用了乒乓结构。即对 L1 cache 使用了双 Buffer 机制，交替使用两块 L1 的两块 Buffer，使得下一次数据的存储交互和此次计算阵列的流水线可以并行进行。使得整体流水线的效率进一步提高。

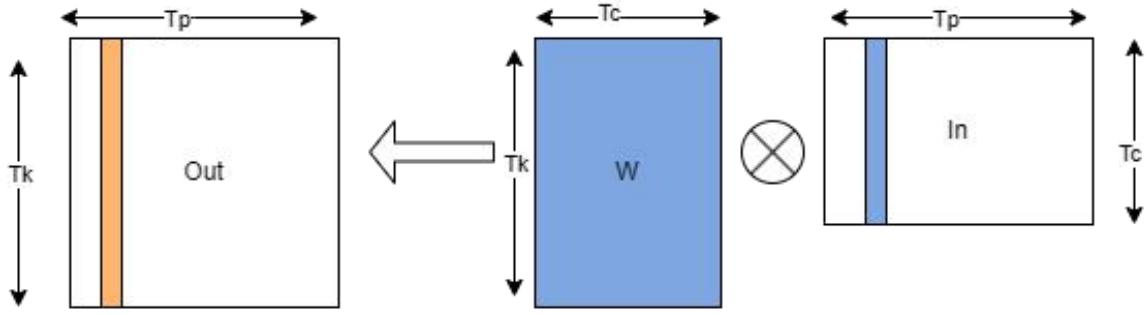

图 4-4 PE 原子计算示意图（阴影表示原子计算，大矩阵单元代表 L1 Cache）

4.2.2 片上缓存建模

在最理想的情况下，在卷积层计算开始之前，该层所需的所有特征和权重就可以全部被预先加载到片上缓存上，之后整个层内的所有计算就不用再去访问外部存储器。但我们可以看到，Yolov4-tiny 网络各层的权重和特征数据量如表 4-1 所示，尽管本文采用了 8bit 量化，但仍然无法做到对每一层都全部缓存到 L2 cache 中（本文所使用的硬件平台的 Block RAM 资源是不够全部缓存的）。因此，如图 4-3 所示，本文的设计将较大的数据分批写入片上缓存 L2 cache 中。经过对网络数据尺寸的分析，并考虑到片上 BRAM 的资源，本文最终将 L2 cache 中输入特征的尺寸设为 512×13×13，输出的 L2 cache 为 16×13×13，权重的尺寸为 512×3×3×16，不难发现，大部分卷积层都可以转化为这些尺寸的整数倍，同时，我们通过这样的分割策略对于算法 4-2 中的卷积循环是没有任何影响的。我们唯一需要注意的是补 0 操作，一方面是当卷积核大小为 3 时需要的 padding 操作，另一方面，当缓存中的某些尺寸有溢出时，对多余的存储空间我们需要置 0 以保证乘加阵列不受影响。

表 4-1 YOLOv4-Tiny 各层数据

网络层	输入	权重	偏置	输出
backbone/conv1	(1, 3, 416, 416)	(3, 3, 3, 32)	(32,)	(1, 32, 208, 208)
backbone/conv2	(1, 32, 208, 208)	(32, 3, 3, 64)	(64,)	(1, 64, 104, 104)

表 4-1 （续）

网络层	输入	权重	偏置	输出
backbone/conv3	(1, 512, 13, 13)	(512, 3, 3, 512)	(512,)	(1, 512, 13, 13)
resblock_body1/conv1	(1, 64, 104, 104)	(64, 3, 3, 64)	(64,)	(1, 64, 104, 104)
resblock_body1/conv2	(1, 32, 104, 104)	(32, 3, 3, 32)	(32,)	(1, 32, 104, 104)
resblock_body1/conv3	(1, 32, 104, 104)	(32, 3, 3, 32)	(32,)	(1, 32, 104, 104)
resblock_body1/conv4	(1, 64, 104, 104)	(64, 1, 1, 64)	(64,)	(1, 64, 104, 104)
resblock_body2/conv1	(1, 128, 52, 52)	(128, 3, 3, 128)	(128,)	(1, 128, 52, 52)
resblock_body2/conv2	(1, 64, 52, 52)	(64, 3, 3, 64)	(64,)	(1, 64, 52, 52)
resblock_body2/conv3	(1, 64, 52, 52)	(64, 3, 3, 64)	(64,)	(1, 64, 52, 52)
resblock_body2/conv4	(1, 128, 52, 52)	(128, 1, 1, 128)	(128,)	(1, 128, 52, 52)
resblock_body3/conv1	(1, 256, 26, 26)	(256, 3, 3, 256)	(256,)	(1, 256, 26, 26)
resblock_body3/conv2	(1, 128, 52, 52)	(128, 3, 3, 128)	(128,)	(1, 128, 26, 26)
resblock_body3/conv3	(1, 128, 26, 26)	(128, 3, 3, 128)	(128,)	(1, 128, 26, 26)
resblock_body3/conv4	(1, 256, 26, 26)	(256, 1, 1, 256)	(256,)	(1, 256, 26, 26)
conv_for_P5	(1, 512, 13, 13)	(512, 1, 1, 256)	(256,)	(1, 256, 13, 13)
yolo_headP4/conv1	(1, 384, 26, 26)	(384, 3, 3, 256)	(256,)	(1, 256, 26, 26)
yolo_headP4/conv2	(1, 256, 26, 26)	(256, 1, 1, 75)	(75,)	(1, 75, 26, 26)
yolo_headP5/conv1	(1, 256, 13, 13)	(256, 3, 3, 512)	(512,)	(1, 512, 13, 13)
yolo_headP5/conv2	(1, 512, 13, 13)	(512, 1, 1, 75)	(75,)	(1, 75, 13, 13)
upsample/conv1	(1, 256, 13, 13)	(256, 1, 1, 128)	(128,)	(1, 128, 13, 13)

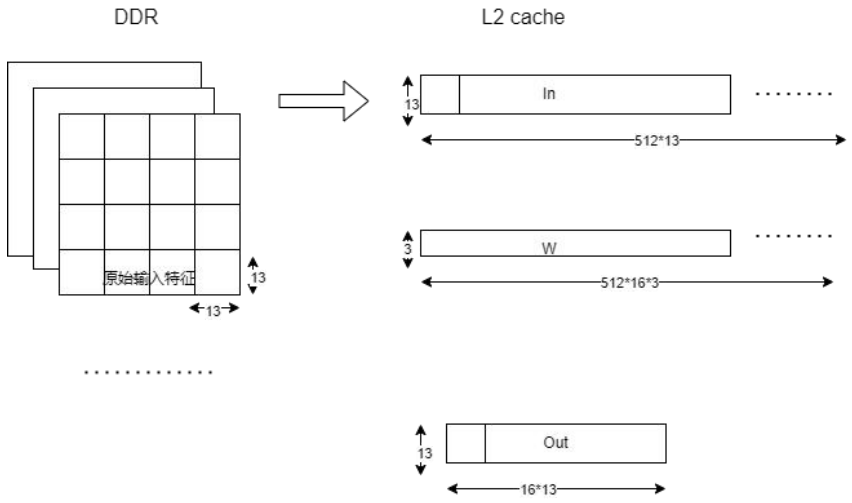

图 4-5 分批次计算输出特征示意图（从输入特征的宽和高，以及输出通道进行分解）

4.2.3 数据排布方式建模

如图 4-3 所示，本文所设计的加速器中权重和特征的存储结构与排布方式直接由两个 DMA 模块控制。从图 4-5 可以看出，由于原子计算中乘加阵列的固定性，权重和特征数据在 L1 Cache 的排布也是固定的。相对的，数据在 L2 Cache 和 DDR 的排布方式则具有灵活性，这关系到 DMA1 和 DMA2 模块的设计细节。

首先，由于权重数据和输入特征移动的单向性，为了简化 DMA1 的设计逻辑，本文保证了数据在外部存储器和片上二级缓存 L2 Cache 的排布方式一致性，即输入使用 W→H→C 的顺序进行排列，权重以 K→W→H→C 的顺序排列。

其次，在确定好 L2 Cache 的排布顺序以后，由于 L1 Cache 在之前对乘加器阵列建模时就已经确定好了数据的排布方式，我们对于 DMA2 的设计剩下所需要分析的主要在于如何将 DMA2 的流水效率提高。本文所采用的并行单元 Tk×Tc 为 64，相对来说并不算庞大，因此可以将权重矩阵进行完全展开，即权重的 L1 Cache 将全部由最快速的寄存器构成。同样的，由于原子计算一次只需要用到 1×Tc 的输入特征，因此我们只对于特征的 L1 Cache 的第一个维度进行展开，对于其第二个维度进行流水缓冲。对于输出矩阵的 L1 Cache 来说，由于每一次进入内层循环都需要刷新，而且其并行度直接决定了后续辅助运算的效率，本文对于输出的 L1 Cache 也进行了完全的展开。

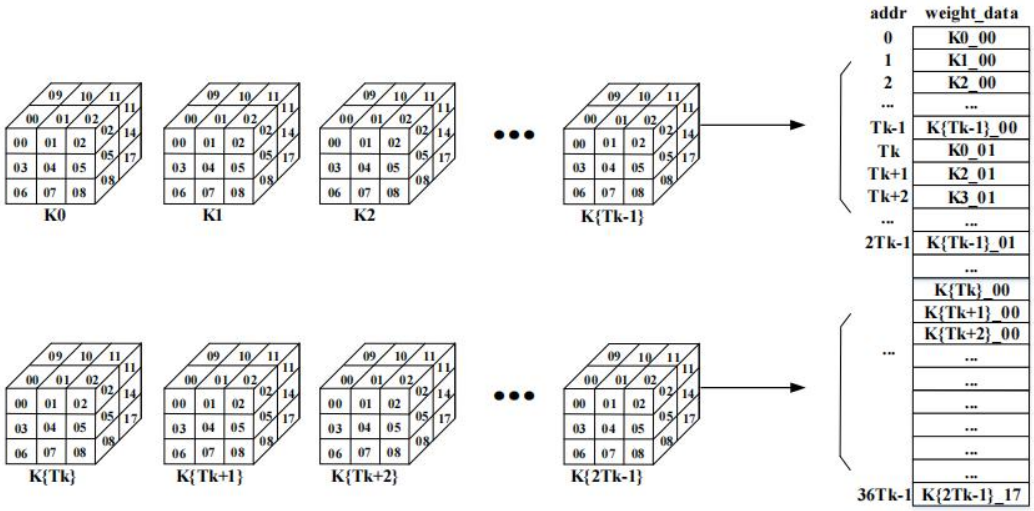

图 4-6 权重在片上缓存中的排布方式

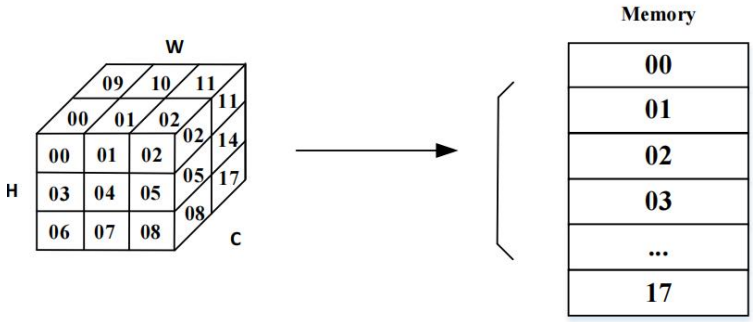

图 4-7 特征数据的排布方式

4.3 卷积运算通路设计

如图 4-3 所示，我们设计的五级流水线卷积通道模块由 DMA1、L2 Cache、DMA2、L1 Cache 和乘加阵列组成。

DMA1 模块负责将包括输入特征和权重的数据从外部存储器传输到片上缓存 L2 Cache 中，它将分割后的数据进行缓存。当该批次所有输出计算完毕，才进行下一轮 DMA1 的读取与存取操作。DMA1 使用连续地址对 AXI 总线进行访问，以便提高速率。与此同时，当运算结束，也由 DMA1 将输出的 L2 Cache 中的数据写入外部存储中。

DMA2 模块的职责是将数据从 L2 Cache 传输到 L1 Cache 中，同时，在这个传输过程中对需要的地方进行补 0 判断，以便为乘加器提供数据。

我们在两级 Cache 中都引入了应答机制，例如当乘加阵列需要数据时，首先由 DMA2 向 L1 Cache 发送请求信号，L1 Cache 在确认其存储的是还未使用过的 DMA1 新写入的数据后，就会响应请求，启动数据的传输。

乘加器阵列按照流水线执行图 4-4 的操作，阵列总共中有 Tc×Tk 个乘加器并行单元。在乘加器阵列内部的 2×Tc×Tk×8bit 大小的 L1 Cache 临时存储着权重数据，通过乒乓操作与计算单元进行交互。与流水线紧密结合的累加器模块负责将乘加器阵列的输出进行累加，将输出数据暂存到输出的 L1 Cache 中。辅助运算通路与 L1 Cache 紧密耦合，当使能打开 MUX 时，可以直接将卷积结果送到辅助通路中继续进行运算。最终，输出特征数据通过 DMA1 输出到 L2 Cache 中。

4.4 池化运算通路设计

Yolov4-tiny 网络中使用了核尺寸为 2 的最大池化运算。对于 AXI 总线来说，只有地址连续的读取才能进行 Burst 模式高速读写，因此如果不对输入进行缓存，直接对总线读取到的数据进行流水线操作的话是很难高效实现池化功能的。而又由于池化操作非常简单，单独进行全部缓存是非常浪费时间的，因此本文采用分步计算，仅缓存临时中间矩阵的方式，通过进行两个方向上的池化得到最终的输出。

如图 4-8 所示，下图中的 OP 运算即表示最大池化运算，输出为特征图的第 i 行第 j 列的数值。我们先进行横向运算，然后进行纵向运算。经过对网络中几个池化层尺寸的统计，本文将中间缓存大小设为 52×104，输入特征按照输入通道被切割成子块，按照通道方向输入到横向运算模块。横向运算的输出连到固定大小的片上缓存中，最后经过纵向运算模块输出特征。

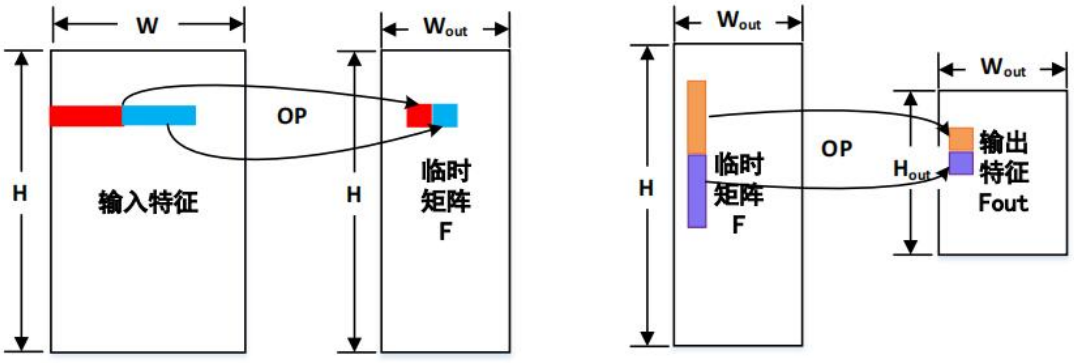

图 4-8 池化运算建模

需要注意的是，本文在池化运算中并行化的优化主要在于对中间缓存的间隔为 2 的展开，实际上并行的优化是有限的。总的来说，池化模块的硬件主要由比较器、选择器和片上缓存组成。性能的提升相比于 CPU 主要体现在硬件的流水线化和高效的 AXI Burst 读写上。

4.5 辅助运算通路建模与设计

在本文中，我们设计了一个辅助运算路径，该路径包含了偏置运算、Leaky Relu 运算以及上采样运算。偏置运算中的偏置是由于我们进行层融合所产生的偏置，尽管仅有 YOLO head 层的 conv2 没有偏置，我们仍把偏置的使能专门设计出来，因为它比较好的符合了辅助运算通路的逻辑，即操作数为输出元素个数一一对应。Leaky Relu 运算，如前文的理论分析所提及的，经过我们的优化以后，这个运算操作变成了过比较器后乘一个 16bit 数并进行移位，其置于偏置运算之后并且共享使能，因为所有需要 Leaky Relu 的卷积层在最初都含有 BN 层进行归一化，也即偏置存在时一定也存在 Leaky Relu 运算。上采样是网络中比较特殊的一个运算，由于其出现次数很少，而且仅出现在 FPN 中的一次卷积之后，具有固定的输入特征大小，因此我们将其固化为了硬件，当处在那个特殊层的时候，进行完偏置和非线性运算再继续进行上采样。由第二章的分析可知，采用临近值采样策略的上采样其实算是池化层的逆运算，因此它的硬件化设计也面临到了 AXI 总线利用的问题。因此，我们采用了类似与池化设计的策略，先横向再纵向计算，首先对数据进行横向插值，然后进行纵向插值。

总的来说，如图 4-9 所示，本文所设计的辅助运算通路的结构包括偏置运算模块、非线性模块和上采样模块。由辅助运算通路使能进行控制，选择是否将卷积模块的 L2 Cache 送入辅助运算模块。

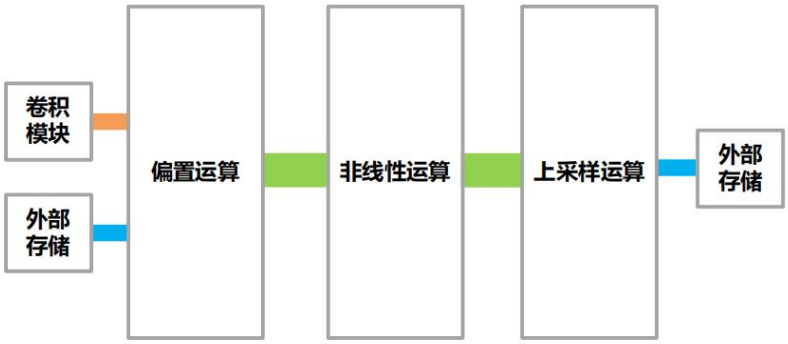

图 4-9 辅助运算通路硬件结构

4.6 访存模块设计

根据前述几个通路的构建，实际上我们的硬件加速器总共有五个读取通道：分别是包括外循环边界和量化因子的参数，输入特征图，输入权重矩阵，输入偏置矩阵，池化输入；以及两个写回通道：卷积输出通道和池化输出通道。本文使用的是 HLS 硬件开发环境，采用的是 m_axi slave 对这些通道变量进行设置。这样可以相对方便的通过 PS 端设置输入矩阵的初始地址。

4.7 本章小节

本章我们系统论述了本文的硬件设计思路和细节。首先我们对于 YOLOV4-Tiny 算法各个部分对于并行化计算和处理的需求进行了详细的分析，明确了硬件设计的重点在于对于卷积层的高效架构设计。本文通过对六层卷积循环进行合并后重新分解得到了八层卷积循环，为之后不同尺寸的卷积层都能高效的利用原子计算单元阵列提供了算法基础。

我们详细论证了乘加器阵列、片上缓存以及数据排列方式的卷积运算通路模型，并搭建了相应的 C 模型进行了仿真验证。并在此基础上，设计了上采样和 LeakyRelu 层的硬件，作为辅助运算通路，直接连接在卷积层的输出之后。我们还设计了池化运算层，并单独进行了相应的性能分析。

经过对 DMA1 的 AXI 接口读写进行优化，对 DMA2 准备的 L1 Cache 片上缓存进行数组优化，并对多个循环进行流水线优化处理，我们硬件的 HLS 设计性能得到了大幅度的提升。

第五章 系统验证与分析

本章主要介绍将整个硬件与软件系统烧录到 FPGA 开发板上进行板上测试的具体方法与流程，将网络移植到 ZYNQ 系统下 CPU 执行的验证，基于 Vitis HLS 平台对于各个加速器部件的前仿测试，以及最终整个目标检测系统的整体验证。

5.1 ZYNQ 系统下 CPU 验证

本文使用黑金 7020 开发板作为开发平台，主要使用到了板上的 Micro SD 卡接口，串口，以及 JTAG。通过 SD 卡给输入的图片并存储最终输出的图片。

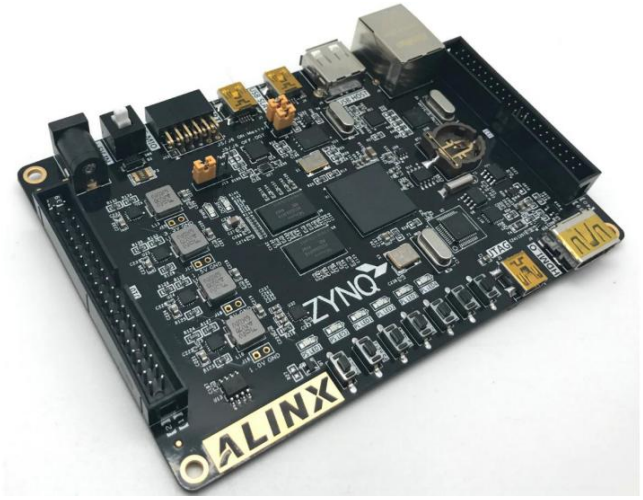

图 5-1 XILINX ZYNQ7000 开发平台

我们首先在 CPU 下将整个 8bit 量化后的网络进行了纯 CPU 的部署，部署后跑一张图片的具体延迟如下表所示，可以看到是非常慢的。

表 5-1 ARM 下的网络延迟总览

网络层	延迟（/10ms）
backbone/conv1	787+16
backbone/conv2	3806+8
resblock_body1/conv1	7612+8
resblock_body1/conv2	1893+4
resblock_body1/conv3	1893+4
resblock_body1/conv4	586+9
resblock_body1/maxpool	8
resblock_body2/conv1	7663+4
resblock_body2/conv2	1885+2

表 5-1 （续表）

网络层	延迟（/10ms）
resblock_body2/conv3	1886+2
resblock_body2/conv4	580+4
resblock_body2/maxpool	4
resblock_body3/conv1	7547+2
resblock_body3/conv2	1878+1
resblock_body3/conv3	1878+1
resblock_body3/conv4	580+2
resblock_body3/maxpool	2
backbone/conv3	7405+2
conv_for_P5	290+0
yolo_headP4/conv1	11402+2
yolo_headP4/conv2	169
yolo_headP5/conv1	1816+0
yolo_headP5/conv2	42
upsample/conv1	72+0
upsample/upsample	1

5.2 硬件加速器仿真测试

本文首先基于加速器的架构进行了 C 模型的构建，在这之后利用 HLS 环境添加了优化的 directive 后，对于硬件的资源占用、延迟和流水间隔进行了大量的优化，最终的优化结果如下：

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
basic_conv				-	46466138	4.650E8	-	46466139	-	no	1	17	9515	13408
basic_conv_Pipeline_1				-	514	5.140E3	-	514	-	no	0	0	12	42
Loop 1				-	512	5.120E3	1	1	512	yes	-	-	-	-
basic_conv_Pipeline_2				-	34	340.000	-	34	-	no	0	0	8	36
Loop 1				-	32	320.000	1	1	32	yes	-	-	-	-
basic_conv_Pipeline_3				-	18	180.000	-	18	-	no	0	0	7	37
Loop 1				-	16	160.000	1	1	16	yes	-	-	-	-
Bias_load	II Violation	Memory Dependency	1	-	39	390.000	-	32	-	yes	0	0	962	488
VITIS_LOOP_187_1				-	46465552	4.650E8	5808194	-	8	no	-	-	-	-
VITIS_LOOP_188_2				-	5808192	5.808E7	1074	-	5408	no	-	-	-	-
Loop_2				-	1007	1.007E4	-	1007	-	no	0	17	5944	9887
Loop_2_Pipeline_VITIS_LOOP_107_5_VITIS_LOOP_112_6				-	60	600.000	-	60	-	no	0	4	1217	1265
VITIS_LOOP_107_5_VITIS_LOOP_112_6				-	58	580.000	28	1	32	yes	-	-	-	-
VITIS_LOOP_96_2_VITIS_LOOP_97_3				-	945	9.450E3	105	-	9	no	-	-	-	-
Loop_2_Pipeline_VITIS_LOOP_140_1	II Violation			-	54	540.000	-	54	-	no	0	1	1710	1938
VITIS_LOOP_140_1				-	52	520.000	32	3	8	yes	-	-	-	-
Loop_2_Pipeline_VITIS_LOOP_101_4				-	23	230.000	-	23	-	no	0	12	509	1496
VITIS_LOOP_101_4				-	21	210.000	8	2	8	yes	-	-	-	-
PE_Top				-	6	60.000	-	2	-	yes	0	12	491	1140
DMA_2	II Violation			-	2	20.000	-	2	-	yes	0	0	23	74
VITIS_LOOP_188_2.1				-	64	640.000	2	-	32	no	-	-	-	-

图 5-2 卷积加速器资源占用

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
base_maxpool				-	-	-	-	-	-	no	14	32	4198	4867
Maxpool_deal				-	-	-	-	-	-	no	14	32	3046	3105
Maxpool_deal_Pipeline_1				-	10818	1.080E5	-	10818	-	no	0	0	16	47
Loop 1				-	10816	1.080E5	1	1	10816	yes	-	-	-	-
Maxpool_deal_Pipeline_2				-	5410	5.410E4	-	5410	-	no	0	0	15	45
Loop 1				-	5408	5.408E4	1	1	5408	yes	-	-	-	-
Maxpool_deal_Pipeline_3				-	2706	2.706E4	-	2706	-	no	0	0	14	44
Loop 1				-	2704	2.704E4	1	1	2704	yes	-	-	-	-
VITIS_LOOP_48_1				-	-	-	-	-	-	no	-	-	-	-
Maxpool_deal_Pipeline_VITIS_LOOP_59_1_VITIS_LOOP_60_2				-	-	-	-	-	-	no	0	3	658	678
VITIS_LOOP_59_1_VITIS_LOOP_60_2				-	-	-	12	1	-	yes	-	-	-	-
Maxpool_deal_Pipeline_VITIS_LOOP_69_1				-	-	-	-	-	-	no	0	0	52	89
VITIS_LOOP_69_1				-	-	-	3	1	-	yes	-	-	-	-
Maxpool_deal_Pipeline_VITIS_LOOP_84_1_VITIS_LOOP_86_2				-	-	-	-	-1	-	no	0	1	313	402
VITIS_LOOP_84_1_VITIS_LOOP_86_2				-	?	?	7	1	-	yes	-	-	-	-
Maxpool_deal_Pipeline_VITIS_LOOP_101_1_VITIS_LOOP_102_2				-	-	-	-	-1	-	no	0	3	472	628
VITIS_LOOP_101_1_VITIS_LOOP_102_2				-	?	?	10	1	-	yes	-	-	-	-

图 5-3 池化模块资源占用

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
base_upsample				-	176948	1.769E6	-	176949	-	no	3	3	1340	3009
Upsample_deal				-	176946	1.769E6	-	176946	-	no	3	3	292	1255
Upsample_deal_Pipeline_1				-	171	1.710E3	-	171	-	no	0	0	10	39
Loop 1				-	169	1.690E3	1	1	169	yes	-	-	-	-
Upsample_deal_Pipeline_2				-	340	3.400E3	-	340	-	no	0	0	11	40
Loop 1				-	338	3.380E3	1	1	338	yes	-	-	-	-
Upsample_deal_Pipeline_3				-	678	6.780E3	-	678	-	no	0	0	12	42
Loop 1				-	676	6.760E3	1	1	676	yes	-	-	-	-
VITIS_LOOP_39_1				-	176256	1.763E6	1377	-	128	no	-	-	-	-
Upsample_deal_Pipeline_VITIS_LOOP_50_1_VITIS_LOOP_51_2				-	173	1.730E3	-	173	-	no	0	1	39	118
VITIS_LOOP_50_1_VITIS_LOOP_51_2				-	171	1.710E3	4	1	169	yes	-	-	-	-
Upsample_deal_Pipeline_VITIS_LOOP_59_1				-	171	1.710E3	-	171	-	no	0	0	19	59
VITIS_LOOP_59_1				-	169	1.690E3	2	1	169	yes	-	-	-	-
Upsample_deal_Pipeline_VITIS_LOOP_68_1_VITIS_LOOP_70_2				-	343	3.430E3	-	343	-	no	0	1	117	194
VITIS_LOOP_68_1_VITIS_LOOP_70_2				-	341	3.410E3	5	1	338	yes	-	-	-	-
Upsample_deal_Pipeline_VITIS_LOOP_80_1_VITIS_LOOP_81_2				-	682	6.820E3	-	682	-	no	0	1	49	125
VITIS_LOOP_80_1_VITIS_LOOP_81_2				-	680	6.800E3	6	1	676	yes	-	-	-	-

图 5-4 上采样模块资源占用

5.3 目标检测 SoC 验证

在之前的硬件和 CPU 工程基础上，我们基于 Vitis 自动生成的对于 HLS IP 上的 AXI 总线进行读写的驱动函数库，编写了中间层调度函数，构建了新的使用加速器计算的整个 SoC 系统，系统的延迟如下表所示：

表 5-2 使用加速器后的网络延迟总览

网络层	延迟（/1ms）
backbone/conv1	67
backbone/conv2	230
resblock_body1/conv1	532
resblock_body1/conv2	152
resblock_body1/conv3	154
resblock_body1/conv4	46
resblock_body1/maxpool	3
resblock_body2/conv1	655
resblock_body2/conv2	134
resblock_body2/conv3	134
resblock_body2/conv4	43
resblock_body2/maxpool	2
resblock_body3/conv1	679

表 5-2 （续表）

网络层	延迟（/1ms）
resblock_body3/conv2	146
resblock_body3/conv3	146
resblock_body3/conv4	49
resblock_body3/maxpool	1
backbone/conv3	609
conv_for_P5	24
yolo_headP4/conv1	952
yolo_headP4/conv2	13
yolo_headP5/conv1	163
yolo_headP5/conv2	3
upsample/conv1	7
upsample/upsample	1

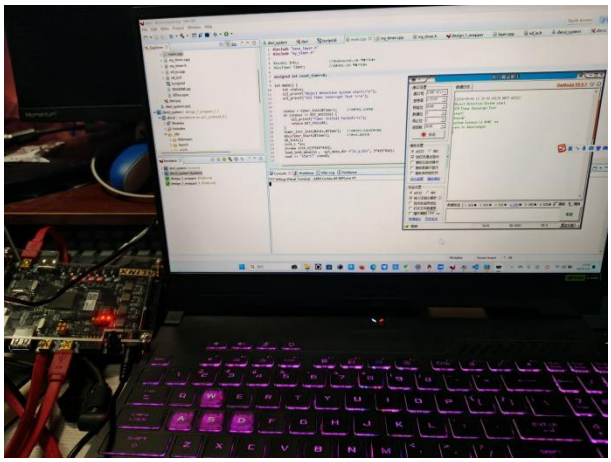

图 5-5 SoC 系统最终测试

整个系统平台如图5-5所示，下面展示的左侧是纯ARM CPU工程跑出的结果，右侧是使用加速器跑出的结果，可以看到是完全一致的，并且也精准的识别到了我们要求识别的对象。

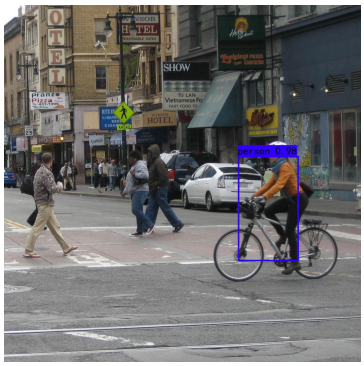

图 5-6 纯 ARM 推理输出结果

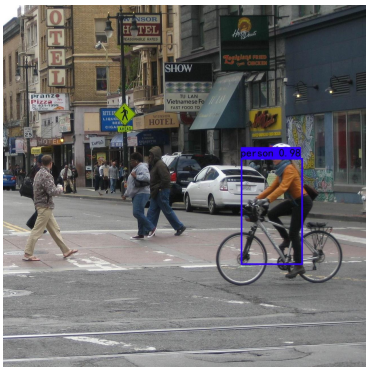

图 5-7 使用加速器计算结果

系统的资源占用情况如图 5-8 所示，DSP 总共消耗了 89 个，占比为 36%，BRAM 消耗了 51 个，占比为 28%，FF 共消耗了 29457 个，占比为 28%，LUT 消耗了 23440 个，占比为 44%。如图 5-9 所示，系统总功耗为 1.87W。

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	23440	29457	298	9994	22617	823	51	89	130	1
design_1_i (design_1)	23440	29457	298	9994	22617	823	51	89	0	1
axi_mem_intercon (d	175	265	0	92	165	10	0	0	0	0
base_top_0 (design_	22885	28733	298	9776	22132	753	51	89	0	0
processing_system7_	0	0	0	0	0	0	0	0	0	1
ps7_0_axi_periph (de	364	426	0	173	305	59	0	0	0	0
rst_ps7_0_50M (desig	17	33	0	8	16	1	0	0	0	0

图 5-8 系统资源占用

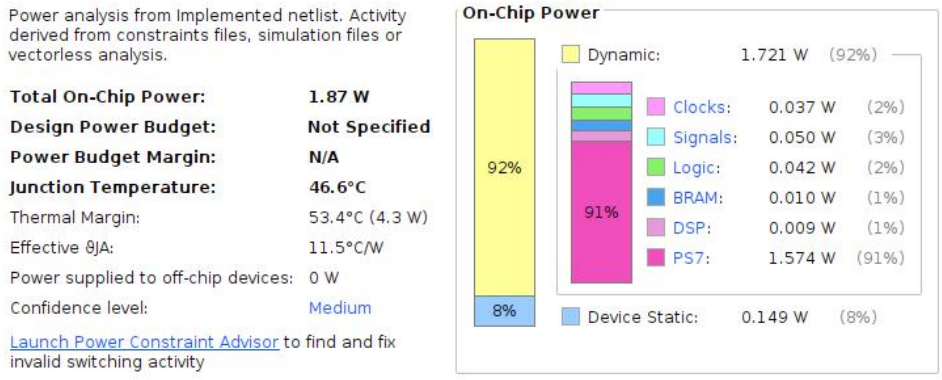

图 5-9 系统能耗

YOLOV3-Tiny 的网络计算量 BFLOPs 大小为 6.9，对应的 GOPS 计算方法如公式所示，其中 Ts 为前向推理时间。

$$GOPS = \frac{1}{T_s} \times 6.9$$

本文设定的 FPGA 时钟频率为 100MHz，PS 端的 ARM 时钟频率为 667MHz，系统总功耗功耗报告得出，在 ZYNQ 系统下纯 CPU 与使用本文所设计的加速器后的性能对比如表 5-3 所示，与其他文献所设计的加速器的性能对比如表 5-4 所示。

表 5-3 本文算法 ZYNQ 系统下使用加速器前后性能对比

平台	ARM（CortexA9）	ZYNQ7020
工作频率	667M	100M+667M
数据精度	8bit 定点	8bit 定点
处理耗时	451790ms	4945ms
性能	0.015	1.395
性能能耗比	0.01	0.78

表 5-4 基于 FPGA 的神经网络加速系统的性能对比

	文献[12]	文献[11]	文献[9]	本文
FPGA 型号	ZYNQ XC7Z020	ZYNQ XC7Z100	ZYNQ XC7Z035	ZYNQ XC7Z020
频率（MHz）	100	150	100	100
数据精度	定点 6 位	定点 8 位	定点 16 位	定点 8 位
DSP	181	536（26.5%）	157	89（36%）
FF	*	215K（39%）	*	29K（28%）
LUT	*	211K（76%）	*	23K（44%）
算力（Gops）	24.32	129.6（峰值）	27.52	1.395
功耗（W）	1.38	4.8	2.15	1.87
能效比（Gops/W）	24.32	27.1	12.8	0.78

如表所示，本文最终搭建的系统相较于纯 ARM CPU 部署的网络性能得到了极大的提升，但相较于其他相关研究所开发的加速器效能仍有一定差距，这主要是由于本文的加速器硬件 DMA 对 AXI 接口的利用率并不高所导致的。

5.4 本章小节

本章主要介绍将整个硬件系统映射到 FPGA 上进行板上测试的具体方法与流程，将网络移植到 ZYNQ 系统下 CPU 执行的验证，基于 Vitis HLS 平台对于各个加速器部件的前仿测试，以及最终整个目标检测系统的整体验

第六章 总结与展望

6.1 总结

本文的工作重点为基于 ZYNQ 的异构加速系统和神经网络硬件加速器设计，最终将利用 YOLOv4-Tiny 网络模型对所设计的硬件加速系统在仿真环境和 FPGA 测试环境进行验证。本文首先根据 YOLOV4 所引入的新的训练方法，并采用 LeakyRelu 作为激活函数且参数选择为 0.125 方便硬件进行移位操作，通过数据增强和基于 CIOU 的 Loss 训练得到了原始网络，之后进行了层融合与 8bit 定点量化。

本文基于 Vitis HLS 高层次综合工具，基于卷积循环优化，流水线优化，二级缓存，乒乓操作等多种方法设计了专用加速器，并开发了与之匹配的驱动函数库。在 C++环境下系统开发了完整的 YOLOv4-Tiny 算法量化后的前向推理，利用定时器中断的计时模块，数据预处理模块，以及数据解码模块。本文的系统最终部署在 ZYNQ7020 平台，在 VOC 测试集下系统性能达到了 1.395GOPS，是 ARM(CortexA9) 的 93 倍。性能功耗比达到了 0.78GOPS/W，达到了 ARM(CortexA9) 的 78 倍，达到了低功耗高性能的目的。

6.2 展望

本文研究并实现了基于 FPGA 的深度学习软硬件协同设计，但由于个人能力与研究时间有限，在本文已有的研究工作基础上，仍有以下方向值得进一步研究。

(1) 优化中间变量位宽：本文所设计的加速器主要在架构的设计上进行了大量的优化，但是由于使用的是 HLS 环境，还需要对数据类型进行精简，本文的设计大部分采用了标准位宽，因此存在大量的位宽浪费。

(2) 优化 AXI 接口效率：在搭建 SoC 系统时对于 HP 高速接口的利用并不充分，可以将 AXI 接口配置成匹配 HP 接口最高带宽，以进一步提高系统速度。同时，由于时间仓促，本文的设计对于 Burst 模式并没有充分仿真验证，根据综合仿真结果估计仍存在一部分时间浪费，仍需要在 HLS 环境下进一步优化改进。

(3) 改进模型：首先对原始模型的量化，本文采用的是训练后量化，这也导致了大量的准确率损失，尽管 16bit 与 8bit 量化的精确率差距很大，为了减少资源的占用，最后还是选择了 8bit 量化。后续可以进行训练时的动态量化以提高准确率，同时我们可以进行模型的效率分析，实施剪枝、蒸馏等操作进一步精简模型。

致 谢

2020 年我踏入了电子科技大学的校门，成为了一名大学生，四年时间转眼间就过去了，我在这里收获了一批志同道合的伙伴和一颗永远怀揣热爱渴求知识的心。值此毕业之际，我由衷的感谢那些在我本科学习期间给予我帮助的老师、家人和同窗好友。

首先，我要感谢我的指导老师阮爱武老师和同门师兄王枭，没有老师的悉心指导与师兄的耐心解惑，我是很难坚持下来并最终完成如此工作量的毕业设计的。尽管最后设计的加速器性能一般，但在设计过程中遇到各种困难，一路解决下来所积累的经验，我想一定会让我终生受益的。祝导师事业进步，家庭幸福；祝师兄学业有成，文章丰收。

另外，我要感谢我的家人，感谢他们一直以来对我无微不至的关心和照顾，无条件的支持我的学习与生活。愿他们永远健康平安。

最后，我要感谢这几年一路陪伴我学习、进步、成长的伙伴们。我们一路走来，相互感染，相互扶持，相互激励，不知不觉已收获良多，成长许多。希望我们永远如来时一样无畏，永远怀揣着希望，永远热血，未来一路顺风。

参考文献

[1] Y. Ma, Y. Cao, S. Vrudhula and J. -s. Seo, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 7, pp. 1354-1367, July 2018, doi: 10.1109/TVLSI.2018.2815603.

[2] Zheng, Z., Wang, P., Liu, W., Li, J., Ye, R., & Ren, D. (2020). Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression. Proceedings of the AAAI Conference on Artificial Intelligence, 34(07), 12993-13000. https://doi.org/10.1609/aaai.v34i07.6999.

[3] Bhatt, D; Patel, C; Talsania, H; Patel, J; Vaghela, R; Pandya, S; Modi, K; Ghayvat, H. CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope. Electronics 2021, 10, 2470. https://doi.org/10.3390/electronics10202470.

[4] T. Yuan, W. Liu, J. Han and F. Lombardi, "High Performance CNN Accelerators Based on Hardware and Algorithm Co-Optimization," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 1, pp. 250-263, Jan. 2021, doi: 10.1109/TCSI.2020.3030663.

[5] Bochkovskiy, Alexey et al. “YOLOv4: Optimal Speed and Accuracy of Object Detection.” ArXiv abs/2004.10934 (2020): n. pag.

[6] C. -Y. Wang, A. Bochkovskiy and H. -Y. M. Liao, "Scaled-YOLOv4: Scaling Cross Stage Partial Network," 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Nashville, TN, USA, 2021, pp. 13024-13033, doi: 10.1109/CVPR46437.2021.01283.

[7] B. Jacob et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 2704-2713, doi: 10.1109/CVPR.2018.00286.

[8] D. Nguyen, D. Kim and J. Lee, "Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, Switzerland, 2017, pp. 890-893, doi: 10.23919/DATE.2017.7927113.

[9] 周围. 基于 FPGA 的深度学习软硬件协同设计的仿真实现[D]. 电子科技大学,2023.DOI:10.27005/d.cnki.gdzku.2022.001140.

[10] 张洪博. 基于 RISC-V CPU 的目标检测硬件加速 SoC 设计[D]. 吉林大学,2022.DOI:10.27162/d.cnki.gjlin.2022.004868.

[11] 贾贤飞. 基于开源 RISC-Ⅴ内核的卷积神经网络加速系统设计[D]. 东南大学,2022.DOI:10.27014/d.cnki.gdnau.2021.001836.

[12] 赵嘉宇. 面向轻量级 YOLOv4-tiny 算法的软硬件加速研究与实现[D]. 黑龙江大学,2024.DOI:10.27123/d.cnki.ghlju.2023.000445.

[13] Bacon, David F. et al. “Compiler transformations for high-performance computing.” ACM Comput. Surv. 26 (1994): 345-420.

[14] Miyashita, Daisuke et al. “Convolutional Neural Networks using Logarithmic Data Representation.” ArXiv abs/1603.01025 (2016): n. pag.

[15] Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., Yu, F., and Yan, J. Differentiable soft quantization: Bridging fullprecision and low-bit neural networks. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 4852–4861, 2019

[16] Liu, Shi et al. “Oscillation-free Quantization for Low-bit Vision Transformers.” International Conference on Machine Learning (2023)

外文资料原文

Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA

Yufei Ma, *Student Member, IEEE*, Yu Cao, *Fellow, IEEE*, Sarma Vrudhula, *Fellow, IEEE*, and Jae-sun Seo, *Senior Member, IEEE*

***Abstract*— As convolution contributes most operations in convolutional neural network (CNN), the convolution acceleration scheme significantly affects the efficiency and performance of a hardware CNN accelerator. Convolution involves multiply and accumulate operations with four levels of loops, which results in a large design space. Prior works either employ limited loop optimization techniques, e.g., loop unrolling, tiling, and interchange, or only tune some of the design variables after the accelerator architecture and dataflow are already fixed. Without fully studying the convolution loop optimization before the hardware design phase, the resulting accelerator can hardly exploit the data reuse and manage data movement efficiently. This paper overcomes these barriers by quantitatively analyzing and optimizing the design objectives (e.g., memory access) of the CNN accelerator based on multiple design variables. Then, we propose a specific dataflow of hardware CNN acceleration to minimize the data communication while maximizing the resource utilization to achieve high performance. The proposed CNN acceleration scheme and architecture are demonstrated by implementing end-to-end CNNs including NiN, VGG-16, and ResNet-50/ResNet-152 for inference. For VGG-16 CNN, the overall throughputs achieve 348 GOPS and 715 GOPS on Intel Stratix V and Arria 10 FPGAs, respectively.**

***Index Terms*— Accelerator architectures, convolutional neural networks (CNNs), field-programmable gate array (FPGA), neural network hardware.**

I. Introduction

THE field-programmable gate arrays (FPGA) are fast becoming the platform of choice for accelerating the inference phase of deep convolutional neural networks (CNNs). In addition to their conventional advantages of reconfigurability and shorter design time over application-specific integrated circuits (ASICs) [20], [21] to catch up with the rapid evolving of CNNs, FPGA can realize low latency inference with competitive energy efficiency (~10–50 GOP/s/W) when compared to software implementations on multicore processors with GPUs [10], [12], [13], [17]. This is due to the fact that modern FPGAs allow customization of the architecture and can exploit the availability of hundreds to thousands of on-chip DSP blocks. However, significant challenges remain in mapping CNNs onto FPGAs. The state-of-the-art CNNs require a large number (>1 billion) of computationally intensive task (e.g., matrix multiplications on large numbers), involving a very large number of weights (>50 million) [4], [5]. Deep CNN algorithms have tens to hundreds of layers, with significant differences between layers in terms of sizes and configurations. The limited computational resources and storage capacity on FPGA make the task of optimal mapping of CNNs (e.g., minimizing latency subject to energy constraints or vice versa) a complex and multidimensional optimization problem. The high cost of off-chip communication is another major impediment to achieving higher performance and lower energy. In fact, the energy cost associated with the large amount of data movements and memory accesses often exceeds the energy consumption of the computations [8], [20]. For these reasons, energy-efficient hardware acceleration of CNNs on a FPGA requires simultaneous maximization of resource utilization and data reuse, and minimization of data communication.

More than 90% of the operations in a CNN involve convolutions [2]–[4]. Therefore, it stands to reason that acceleration schemes should focus on the management of parallel computations and the organization of data storage and access across multiple levels of memories, e.g., off-chip dynamic random access memory (DRAM), on-chip memory, and local registers. In CNNs, convolutions are performed by four levels of loops that slide along both kernel and feature maps as shown in Fig. 1. This gives rise to a large design space consisting of various choices for implementing parallelism, sequencing of computations, and partitioning the large data set into smaller chunks to fit into on-chip memory. These problems can be handled by the existing loop optimization techniques [6], [9], such as loop unrolling, tiling, and interchange. Although some CNN accelerators have adopted these techniques [9], [11], [13], [19], the impact of these techniques on design efficiency and performance has not been systematically and sufficiently studied. Without fully studying the loop operations of convolutions, it is difficult to efficiently customize the dataflow and architecture for high-throughput CNN implementations. This paper aims to address

Manuscript received October 27, 2017; revised February 3, 2018; accepted March 6, 2018. Date of publication April 3, 2018; date of current version June 26, 2018. This work was supported in part by the NSF I/UCRC Center for Embedded Systems through NSF under Grant 1230401, Grant 1237856, Grant 1701241, Grant 1361926, Grant 1535669, Grant 1652866, and Grant 1715443; and in part by the Intel Labs, and in part by the Samsung Advanced Institute of Technology. *(Corresponding author: Yufei Ma.)*

Y. Ma, Y. Cao, and J.-s. Seo are with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: yufeima@asu.edu; yu.cao@asu.edu; jaesun.seo@asu.edu).

S. Vrudhula is with the School of Computing, Informatics, Decision Systems Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: vrudhula@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2815603

外文资料译文

优化卷积运算以加速 FPGA 上的深度神经网络

卷积循环的加速

卷积神经网络(CNN)的大部分运算都是由卷积完成的，因此卷积加速方案对卷积神经网络硬件加速器的效率和性能影响很大。卷积涉及四层循环的乘法和累加运算，导致设计空间很大。之前的工作要么采用有限循环优化技术，例如循环展开、平铺和交换，要么在加速器架构和数据流已经固定之后只调整一些设计变量。

在硬件设计阶段之前，如果不充分研究卷积循环优化，所得到的加速器很难有效地利用数据重用和管理数据移动。本文通过基于多个设计变量对 CNN 加速器的设计目标(如内存访问)进行定量分析和优化，克服了这些障碍。然后，我们提出了一种特定的硬件 CNN 加速数据流，以最大限度地减少数据通信，同时最大限度地提高资源利用率，从而实现高性能。通过实现包括 NiN、VGG-16 和 ResNet-50/ResNet152 在内的端到端 CNN 进行推理，验证了所提出的 CNN 加速方案和架构。对于 VGG-16 CNN，在 Intel Stratix V 和 Arria 10 fpga 上的总吞吐量分别达到 348 GOPS 和 715 GOPS。

A.通用 CNN 加速系统

最近报道的 CNN 算法涉及大量数据和权重。对它们来说，片上存储器不足以存储所有数据，需要千兆字节的外部存储器。因此，典型的 CNN 加速器由三层存储结构组成:1)外部存储器;2)片上缓冲器;3)与处理相关的寄存器(PE)。基本流程是从外部存储器获取数据到片上缓冲区，然后将它们输入寄存器和 PE。PE 计算完成后，结果被传送回片上缓冲器，必要时也传送到外部存储器，作为下一层的输入。

B.卷积循环

卷积是 CNN 算法中的主要操作，涉及到输入特征映射和核权值的三维乘法累加(MAC)操作。卷积由四层循环实现，为了有效地映射和执行卷积循环，采用了循环展开、循环平铺和循环交换三种循环优化技术，对加速器的计算和通信模式进行了三层内存层次的定制。

C.回路优化和设计变量

对于给定的 CNN，使用多个维度来描述每个卷积层的特征和核映射的大小。循环展开和循环平铺的硬件设计变量将决定加速因子和硬件占用空间。一个内核(或过滤器)窗口的宽度和高度由(Nkx, Nky)描述。(Nix, Niy)和(Nox, Noy)分别定义一个输入和输出特征映射(或通道)的宽度和高度。Nif 和 not 分别表示输入和输出特征映射的个数。循环展开设计变量是(Pkx, Pky)、Pif、(Pox, Poy)和 Pof，它们表示并行计算的数量。循环平铺设计变量是(Tkx, Tky)， Tif， (Tox, Toy)和 Tof,表示存储在片上的四个循环的数据缓存。这些维数和变量的约束由 1≤P* ≤T* ≤N* 给出，其中 N* 、T* 和 P* 分别表示前缀为大写 N、T 和 P 的任何维数或变量。例如:1≤Pkx≤Tkx≤Nkx。默认情况下，P* ，T* 和 N* 应用于所有卷积层。

输入输出变量的关系受式(1)-式(3)的约束，其中 S 为滑动窗口的步长，零填充大小包含在 Nix、Niy、Tix 和 Tiy 中。

1)循环展开:展开不同的卷积循环导致不同的计算并行化，这影响了数据重用机会和内存访问模式方面的最佳 PE 架构。

Loop-1 展开:在这种情况下，每循环计算同一特征和核映射中不同(x, y)位置的 Pkx × Pky 像素(或激活)和权重的内积。这个内积需要一个带有 Pkx × Pky 扇入的加法器树来对 Pkx × Pky 的并行乘法结果求和，以及一个累加器来将加法器树的输出与前面的部分和相加。

Loop-2 展开:在每个循环中，Pif 的像素数/权重从 Pif 不同的特征/内核映射到计算内积需要相同的(x, y)位置。内积操作产生与展开 Loop-1 相同的计算结构，但使用不同的加法器树扇入 Pif。

Loop-3 展开:在每个循环中，将同一特征图中不同(x, y)位置的像素个数乘以相同的权值。因此，这个权重可以被重用 Pix × Piy 次。因为 Pix × Piy 并行乘法有助于独立的 Pix x Piy 输出像素，Pix x Piy 累加器用于连续累加乘法器输出，不需要加法器树。

Loop-4 展开:在每个循环中，一个像素在相同(x, y)位置乘以 Pof 个权重，但来自 Pof 个不同的内核映射，并且该像素被重用 Pof 次。计算结构与使用 Pof 乘法器和累加器展开 Loop-3 相同，没有加法器树。四个卷积循环的展开变量值共同确定并行 MAC 操作的总数以及所需乘法器(Pm)的数量。

2)循环平铺(Loop Tiling): fpga 的片上存储器并不总是大到足以存储深度 CNN 算法的全部数据。

因此，使用密度更大的外部 dram 来存储各层的权重和中间像素结果是合理的。循环平铺用于将整个数据划分为多个块，这些块可以容纳在片上缓冲区中。通过

正确分配循环平铺大小，可以增加数据的局部性以减少 DRAM 访问的数量，这会导致长延迟和高功耗。循环平铺设置所需片上缓冲区大小的下限。输入像素缓冲区所需的大小为 Tix × Tiy × Tif × (pixel_datawth)。权重缓冲区的大小为 Tkx × Tky × Tif × Tof × (weight_datawth)。输出像素缓冲区的大小为 Tox × Toy × Tof × (pixel_datawth)。

3)循环互换:循环互换决定了四个卷积循环的顺序计算顺序。环路交换有两种形式，即段内环路和段间环路。块内循环顺序决定了数据从片上缓冲区到 pe 的移动模式。层间循环顺序决定了数据从外部存储器到片上缓冲区的移动。