电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER THESIS

论文题目 基于 FPGA 的神经网络加速器

学科专业	**计算机科学与技术**
学　　号	**202021081001**
作者姓名	**张渤钰**
指导教师	**吴跃　教授**
学　　院	**计算机科学与工程学院 (网络空间安全学院)**

分类号 TN791 密级 公开

UDC[注1] 004.27

学 位 论 文

基于 FPGA 的神经网络加速器

（题名和副题名）

张渤钰

（作者姓名）

指导教师 **吴跃 教授**

电子科技大学 成都

（姓名、职称、单位名称）

申请学位级别 **硕士** 学科专业 **计算机科学与技术**

提交论文日期 **2023 年 3 月 28 日** 论文答辩日期 **2023 年 5 月 26 日**

学位授予单位和日期 **电子科技大学 2023 年 6 月**

答辩委员会主席

评阅人

注 1：注明《国际十进分类法 UDC》的类号。

A Neural Network Accelerator Based on FPGA

A Master Thesis Submitted to
University of Electronic Science and Technology of China

Discipline:	**Computer Science and Technology**
Student ID:	**202021081001**
Author:	**Zhang Boyu**
Supervisor:	**Prof. Wu Yue**
School:	**School of Computer Science and Engineering(School of Cyberspace Security)**

摘 要

近年来，神经网络已在我们日常生活的各个领域得到了广泛的应用。卷积神经网络和长短期记忆神经网络在各自的应用领域表现出优异的性能，其中前者具有出色的特征提取能力，后者能够很好地处理序列数据。随着神经网络的发展，将其部署到移动端的需求越来越高，传统平台上使用的中央处理器和图形化处理器都存在着高功耗的弊端，无法满足大多数离线嵌入式平台的需求。现场可编程门阵列则在兼具低功耗的优势下，同样具有适合并行计算的特点，因此成为了部署神经网络，对其进行硬件加速的选择。本文的具体工作如下：

1. 对神经网络计算可加速的空间进行了探索和分析，分析了本文所采用的加速平台以及技术路线的优势，介绍了本文设计的加速器所具有的灵活和高可移植性的特点。
2. 针对平台片上存储有限的情况，本文首先对网络进行了轻量化处理，降低输入数据占用的存储空间；所有输入数据存储在外存，通过总线将运算所需数据传输到可编程逻辑部分，在该部分封装了相应的接口并将接收到的数据暂存在片上存储中；片上存储采用了乒乓缓冲的结构减少读写的冲突；设计了相应的数据分块方案，以减少相同数据在总线上重复传输的次数。
3. 针对片上计算资源有限以及神经网络计算密集的特点。本文利用硬件描述语言可以更精细化地进行优化的特点，结合计算中的并行性，充分挖掘了计算资源的利用效率，通过循环展开、数据重排和循环并行的方式实现对神经网络的加速。
4. 基于 Zynq UltraScale+ 系列的 ZCU102 开发板完成了神经网络的部署并进行了相应的测试。结果表明本文设计的硬件加速系统可在较高的时钟频率，即 250MHz 的工作频率下运行，同时功耗为 4.457W。每秒千兆操作数/数字信号处理器的运算量分别为 0.454 和 0.265，表明本文的设计对提高计算资源利用效率的优化是有效的。

关键词：硬件加速，长短时记忆网络，卷积神经网络，现场可编程门阵列

ABSTRACT

In recent years, neural networks have led to wide applications in various fields of our daily life. Convolutional Neural Networks and Long Short-Term Memory Neural Networks have shown excellent performance in their own application fields. The former has excellent feature extraction ability, and the latter can process sequence data well. With the development of neural network, the demand for deploying it to the mobile devices is increasing. The central processing unit and graphic processing unit have the disadvantage of high power consumption, which cannot meet the needs of most offline embedded platforms. Field programmable gate array has the advantages of low power consumption and is also suitable for parallel computing, so it becomes the choice of deploying neural network and hardware acceleration. The specific work of this thesis is as follows:

1. This thesis explores and analyzes the accelerating space of neural network computing, analyzes the advantages of the accelerating platform and technical route adopted in this thesis, and introduces the characteristics of flexibility and high portability of the accelerator designed in this thesis.
2. Because of the limited on-chip storage of the platform, this thesis first reduces the storage space through quantization; The data is stored in the off-chip cache, and the required data is transmitted to the programmable logic part through the bus, and the corresponding interface is encapsulated in this part to temporarily store the data in on-chip storage; On-chip storage adopts ping-pong buffer structure to reduce the conflict between reading and writing; The corresponding data tiling scheme is designed to decrease the number of repeated transmission through the bus.
3. According to the characteristics of limited on-chip computing resources and intensive neural network computing, this thesis makes use of the feature that the hardware description language can optimize in a finer level to fully exploit the efficiency of computing resources. In addition, this thesis realizes the acceleration of neural network through loop unrolling, data tiling and loop parallel.
4. Based on Zynq UltraScale+series ZCU102, this thesis has completed the deployment of neural network and processed corresponding tests. The results show that the hardware system designed in this thesis can run at the working frequency of 250*MHz*, and the overall resource consumption is at a low level. At the same time,

the on-chip power is 4.457*W*. The Giga Operations per second/digital signal processor is 0.454 and 0.265 respectively, indicating that the design in this thesis is effective in improving the efficiency of computing resources utilization.

Keywords: LSTM, CNN, FPGA, Accelerator

目 录

第一章 绪 论

1.1 研究工作的背景与意义

随着神经网络的快速发展，神经网络支撑起的各类人工智能技术被广泛应用于现实生活中，影响甚至改变了各行各业中相关人员的工作方式和生活方式。由于具有强大的特征提取能力，卷积神经网络（Convolutional Neural Network，CNN）在图像识别[1]、视频分析[2]等方面取得了巨大成功；同时，以长短期记忆神经网络（Long Short-Term Memory，LSTM）为代表的递归连接的递归神经网络（Recurrent Neural Network，RNN），由于具有能够保存时间信息的特点被广泛用于处理序列数据，例如语言处理[3]、文本翻译[4]等领域。目前的趋势来看，具有不同特点的神经网络模块构成的深度神经网络正承担起愈来愈大的作用。

近几十年来，人类在计算机领域的发展无不和硬件技术的发展息息相关，从摩尔定律下的半导体技术发展推动的计算机小型化桌面化的历程，到体系架构的不断革新推动的性能提升。人工智能技术的广泛应用，除了算法的不断迭代带来的性能精度、泛化能力等方面的不断提升，同样也依赖于硬件的发展。而随着神经网络的发展，在实际的应用中，将其部署到离线场景中的应用需求也随之增多。这些在离线平台上的部署对功耗、资源消耗等各项硬件资源提出了更严苛的要求。同时，随着神经网络在解决复杂问题中的不断应用，网络的深度越来越深，模型的拓扑结构也越来越复杂。因此，不断增长的计算增加了对计算资源的需求，而不断扩展的数据规模则提出了对内存资源更大的要求。这些都是当前神经网络硬件加速所需要面临的挑战[5]。

当前主流的神经网络部署平台，采用的方法主要都是异构计算平台：有利用并行计算能力实现高时钟频率和高带宽的图形处理器（Graphics Processing Units，GPUs），其主要思路是将模型部署在服务器集群上，利用图形处理器承担核心运算，这样的部署适于在线平台上对于大批量数据的处理，通过反向传播进行训练。但是离线平台的部署主要是进行前向推理，且图形处理器受制于高能耗、低时延的原因，并不适用于在体积受限的离线平台上进行部署；低功耗占地面积小的专用集成电路（Application Specific Integrated Circuit，ASIC）和现场可编程门阵列（Field Programmable Gate Array，FPGA）。ASIC 具有功耗低、高可靠性的特点，但是具有开发周期长、灵活性较低的劣势。相较而言，同样具有低功耗特点的 FPGA 平台，具有着开发周期短、成本低并且灵活性很高的优势。对于使用者来说，可以在节省成本的情况下进行研发，更快速地获得优化后的结果。同时，由于 FPGA

上的硬件架构是可扩展的，它能够定制并行度，配置流水线，实现精细的内存优化和通信优化，减少功耗和延迟以提高效率。因此，FPGA 适合作为网络推理的嵌入式平台。

当前神经网络的发展趋势决定了其推理过程中涉及到的计算量、参数量随之日益剧增。作为嵌入式平台的 FPGA，所需计算的参数必然无法通过片上存储来进行存放，将大量的参数存储在片外存储上将带来巨大的访存压力。因此加速器设计面临的挑战是在资源受限的情况下，尽可能提高计算资源的利用效率，同时尽可能减少访存带来的 IO 延迟。Zynq UltraScale+ 系列的开发板是 Xilinx 推出的一款系统级芯片（System on Chip，SoC）系列产品，它将 Xilinx FPGA 与多核 ARM Cortex-A53 处理器集成在一起，因此可以更好地支持软硬件协同工作，同时支持了高清多媒体接口（High Definition Multimedia Interface，HDMI）、安全数字（Secure Digital，SD）卡和 GPIO 等外设接口，便于进行通信和交互。在 Zynq UltraScale+ 系列嵌入式异构平台上进行神经网络硬件加速的部署，可以利用该系列开发板的高性能 FPGA 可编程逻辑、多核 ARM 处理器、丰富的高速接口和外设以及完整的开发工具链，实现可配置的神经网络加速计算，并加速神经网络推理过程，提高系统性能和效率。将非计算密集型的工作量放在处理器上进行，将计算密集型的任务交给计算单元进行处理，这是当前各异构计算平台对于神经网络硬件加速的一个基本方向。因此，本文选定 Zynq UltraScale+ 系列的开发板作为硬件平台部署的目标平台，这对于研究嵌入式平台下的神经网络硬件加速的解决方案具有较大的意义。

综上所述，神经网络硬件加速的主要挑战主要是如何平衡有限的计算资源、存储资源以及数据传输带来的延迟。本文旨在根据卷积神经网络和长短期记忆神经网络的运算特点，基于 FPGA 的嵌入式异构平台，设计实现可以根据不同的网络结构特点进行相应并行度和数据流配置的神经网络加速器。其中，通过 ARM 核进行数据流的调度，利用 FPGA 处理不规则并行和自定义数据类型的能力实现自定义的神经网络计算单元，从而提高神经网络的计算性能。本文探索了基于 FPGA 的异构硬件平台对多种网络结构进行硬件加速的思路和解决方案，同时为图像分类、声纹识别在内的多种神经网络在嵌入式平台上的发展和应用起到了一定的作用。

1.2 国内外研究现状

1.2.1 卷积神经网络研究现状

卷积神经网络的概念最早出现在上世纪六十年代，由加拿大神经科学学家提出了神经元中感受野的概念[6]，随着感受野的发现和证实，标志着视觉系统中的神经网络结构第一次被提出。

卷积神经网络发展的一个重要节点始于上世纪 80 年代，日本科学家福岛邦彦在前人工作的基础上提出了一个多层的神经网络结构[7]。该结构尝试模拟生物视觉系统，并以此网络进行手写字体和一些模式化的识别工作。在这个网络结构中最重要的两个组成部分，称作“S 细胞”和“C 细胞”，其思路是通过两者的叠加实现特征提取和抽象泛化能力。这实际上便是卷积神经网络中卷积层和池化层的最初雏形。

随后在 1998 年时，LeCun 等人在前人的基础上提出了一个五层的卷积神经网络 LeNet-5 用于手写数字识别[8]，其中通过卷积层进行特征提取，通过池化层进行空间平均的方式得到子样本，通过梯度下降算法的反向传播进行模型的监督学习。LeNet-5 取得了很好的识别效果，并且得到了一定规模的实际应用，这标志着卷积神经网络发展到了一个新的时代，促进了之后卷积神经网络的快速发展。网络中使用的卷积层、池化层和全连接层这几种网络结构也奠定了后续卷积神经网络的基本结构。

卷积神经网络的下一个里程碑出现在 2012 年，Krizhevsky 等人提出的 AlexNet 在 ImageNet 数据集的分类比赛中凭借远超其他模型的精度斩获第一名[9]。这是第一个在 ImageNet 数据集上取得显著成果的卷积神经网络模型，也就此收获了巨大的关注度，继续推动了卷积神经网络领域的热度和研究。相较之前的 LeNet-5 来说 AlexNet 网络结构更深，由八层网络结构构成，采用了多层卷积和池化操作，并使用 ReLU 激活函数来加速收敛速度。通过 AlexNet 网络带来的启发，2014 年 Karen Simonyan 和 Andrew Zisserman 提出了 VGGNet[9]，该模型将卷积神经网络的结构带到了更深的层次：采用了 16-19 个卷积层，并且提出了通过多个小的感受野去替代一个大的感受野的思路，在网络结构中只使用了 3*3 大小的卷积核来替代原先网络中采用的 5*5 和 7*7 大小的卷积核，并在每个卷积层后添加了 ReLU 激活函数和最大池化操作。VGGNet 在 ImageNet 上取得了更好的性能，其思想也成为后来许多卷积神经网络的基础。同年出现的 GoogLeNet[10]，由 Google 公司提出，在网络中采用了一种称为 Inception 的模块化设计，该设计旨在通过使用多个卷积核大小和不同的滑动窗口大小，以最小的计算成本来提高模型性能。该网络模型同样在 ImageNet 图像分类竞赛中斩获佳绩。

在 2015 年，由微软公司提出的 ResNet[11] 是一种深度可达到 1000 层的深度卷积神经网络模型。ResNet 采用了残差块结构，旨在解决深度神经网络中的梯度消失和梯度爆炸问题，即导致网络层数增加时，准确率反而下降的主要原因。残差块是一种包含多个卷积层和激活函数的基本单元，其中下一层的输入是经由本层输入信号通过卷积层和激活函数之后输出的临时结果，再与本层的输入信号直接相加而求得的输出结果。这种设计使得网络可以更轻松地学习到输入和输出之间的残差，从而更容易训练非常深的网络。此外，为了进一步加快训练速度，ResNet 还引入了批量归一化技术，从而可以使得每层的输入数据分布更加稳定，有利于大规模的深度网络训练和收敛。该网络模型同样在 ImageNet 图像识别等计算机视觉比赛中取得了突出的成绩。ResNet 的出现解决了深度网络训练过程中的退化问题，使得神经网络的后续发展方向朝着深度增加，复杂度变高的方向不断迈进，从而实现了对更复杂数据的表征，更准确地完成更复杂的任务。该网络模型使用到的很多结构与思路也成为了后来许多深度学习任务，比如图像识别、物体检测等任务进行应用的基础。

2017 年，通过引入注意力机制，Google 提出了基于自注意力机制的神经网络模型 Transformer[12]，用于处理序列到序列的任务，例如机器翻译、语音识别等。Transformer 模型由编码器和解码器两部分组成，每个部分都包含多个层。在每个层中，Transformer 模型使用自注意力机制来计算输入序列中各个位置之间的关系，以便更好地对序列进行编码和解码。具体来说，自注意力机制会根据每个位置的输入向量计算出该位置与其他位置之间的相似度，然后根据这些相似度来加权求和其他位置的输入向量，以获得每个位置的上下文表示。该模型中的自注意力机制可以并行计算，因此可以更好地利用现代硬件和分布式计算平台。此外，可以同时访问所有输入位置的信息，因此可以更好地处理长序列数据，避免了梯度消失和梯度爆炸问题。由于其出色的性能，获得了广泛的应用。

EfficientNet 由 Google 于 2019 年提出[13]，是一种新的卷积神经网络结构。它的特点是在同等计算量下，比其他已有的网络具有更好的性能。该网络的主要贡献是采用了复合系数的方法，在深度、宽度和分辨率方面进行优化，以获得更高的准确率。在传统的网络结构中，通常只调整网络深度或宽度来提高性能，而分辨率通常被固定为某个预定义的值。然而，EfficientNet 发现，通过同时调整深度、宽度和分辨率，可以在同等计算量下提高网络的性能。实验表明，EfficientNet 在 ImageNet 数据集上表现出色，相对于 ResNet-50，该网络在 Top-1 准确率提高明显的同时保证了参数量只增加了不到 1.5 倍。

1.2.2 循环神经网络研究现状

循环神经网络的发展历史最早可以追溯到上世纪 80 年代末和 90 年代初提出的简单循环神经网络，它是一种全连接的循环神经网络，用于进行如语音识别、手写体识别、自然语言处理等任务，即对于时间序列数据进行建模和预测。这些最初的基本循环神经网络结构很简单，通常由一个隐藏层构成。在每个时刻，输入会经过一个全连接的线性层，然后与上一个时刻的隐藏状态一起送入隐藏层。隐藏层的输出经过一个激活函数（如 tanh）后作为该时刻的输出，并且这个输出会作为下一个时刻的隐藏状态。由于信息中包含了和时刻相关的概念，隐藏状态可以看作是对历史信息的一种记忆。其中提出的网络模型以 Elman 网络[14] 和 Jordan 网络[15] 为代表。Jeff Elman 提出了 Elman 网络，它是一种基于前馈神经网络的循环神经网络，通过在前馈神经网络的隐层和输出层之间添加一个循环连接来引入历史信息。具体来说，Elman 网络将前一个时刻的隐层输出作为当前时刻的输入，以此来处理序列数据；Jordan 网络同样也是一种基于前馈神经网络的循环神经网络，由 Michael Jordan 于 1997 年提出。Jordan 网络与 Elman 网络的区别在于，Jordan 网络将前一个时刻的输出作为当前时刻的输入，也可以用于序列数据的建模和预测。但是最初的循环神经网络具有一定的缺陷，即在处理长序列数据时，由于信息在循环层中的反复传递，容易造成梯度消失或梯度爆炸的问题，这极大地限制了其的应用范围。

为了解决这一问题，Hochreiter 和 Schmidhuber 等人在 1997 年提出了一种新的循环神经网络结构：LSTM[16]。LSTM 的主要思想是通过引入记忆单元来捕捉长期依赖关系，并以此解决梯度消失和梯度爆炸问题。具体来说，网络中的记忆单元类似于一个传送带，可以在上面添加或删除信息，而其中的门控单元则可以控制信息的流动。LSTM 中的门控单元包括遗忘门、输入门和输出门。遗忘门控制上一时刻的记忆单元中的信息是否被遗忘；输入门则控制当前输入信息对记忆单元的影响；输出门则控制记忆单元中的信息对当前时刻的输出是否有贡献。通过这些门控单元的灵活组合和控制，长短时记忆网络可以有效地捕捉长期依赖关系，从而以此缓解梯度消失和梯度爆炸问题。长短时记忆网络模型的出现，对于语音识别、自然语言处理等诸多设计序列处理的领域来说具有里程碑式的意义，为后续的循环神经网络模型的发展奠定了重要基础。

对于 LSTM 的优化同样一直在进行，在标准的 LSTM 中，遗忘门和输入门没有考虑记忆单元的状态，只考虑了上一时刻的隐藏状态和当前输入。针对这一缺陷，Gers 和 Schmidhuber 于 2000 年提出了针对传统长短时记忆网络的改进：Peephole[17]。其基本思想是在长短时记忆网络中引入额外的连接，让遗忘门和输

入门可以看到记忆单元中的状态信息，将记忆单元的状态也加入到门控单元的计算中，从而更好地控制信息的流动，以此提高模型的性能。Peephole 的引入改善了长短时记忆网络网络在一些特定任务上的性能，如语音识别、图像识别等。同时，Peephole 的计算代价相对较小，不会显著增加网络的复杂度。因此，Peephole 已经成为了许多进一步针对长短时记忆网络改进的网络模型的基础。

为了优化长短时记忆网络模型中的参数量，Greff 等人提出了 Coupled LSTM[18]。它的基本思想是一个双 LSTM 模型的结构，其中两个 LSTM 单元在时间轴上交错运行。在每个时间戳，两个 LSTM 单元会分别计算两个门控，但是这两个门控在计算时使用相同的输入和输出。这种方法可以减少参数量和计算时间，同时提高模型的鲁棒性。标准的 LSTM 网络中包含输入门、遗忘门和输出门三个门控单元，它们分别用来控制新的输入、上一时刻的记忆保留和输出的流动。但是，这些门控单元之间可能存在竞争，导致计算效率较低。而 Coupled LSTM 中，输入门和遗忘门的作用被合并成一个单一的更新门，它同时控制了输入和遗忘，避免了门控单元之间的竞争，从而提高了计算效率。

随着注意力机制的出现和发展，基于注意力机制的 LSTM 由 Dzmitry Bahdanau 等人提出[19]。该网络模型引入了一种称为“注意力权重”的机制，通过动态地对输入的不同位置进行加权处理，使网络能够更加准确地关注序列中的重要信息。其在每个时刻通过一个额外的神经网络来计算注意力权重，这个神经网络的输入包括当前时刻的输入和前一时刻的隐藏状态，最后输出一个和输入序列长度相同的向量，向量中的每个元素表示对应位置的注意力权重。这些权重通过和输入序列的加权求和，再得出当前时刻的输入。基于注意力机制的 LSTM 可以更好地处理序列中长距离的依赖关系，同时也更加灵活地对不同位置的信息进行处理，因此在语音识别、自然语言处理等任务中取得了较好的效果。

残差网络的提出同样推动了 LSTM 的优化与发展，2017 年 Kim 和 El-Khamy 等人提出了基于残差连接的 LSTM[20]。在基于残差连接的 LSTM 中，每个 LSTM 单元都有一个跨时间步的残差连接。也就是说，对于每个 LSTM 单元，其输入通过一个全连接层得到一个中间状态，然后中间状态再经过一个全连接层得到输出。在这个过程中，中间状态会和输入通过残差块再传入下一个 LSTM 单元。基于残差连接的 LSTM 网络可以减少梯度消失的情况，提高了网络的深度和性能，因此具有更好的训练效果和更快的收敛速度。

1.2.3 神经网络硬件加速研究现状

1.2.3.1 卷积神经网络的硬件加速研究现状

近年来，嵌入式硬件平台在 CNN 加速领域取得了很大的进展。其中加速的思路可以分为以下的几种：

很多研究致力于通过网络结构优化来提高 CNN 的性能，主要的方法有 CNN 网络模型的量化[21,22]，CNN 模型的剪枝等[23,24]，其主要思路是通过减少模型推理时所需的参数量，通过在可接受范围内降低准确率的方式，换取在模型前向推理时的时延、吞吐量上优化；类似的方法还有搜索技术[25]和知识蒸馏[26]，搜索技术的目标是在候选模型空间中找到神经网络模型的最佳结构和超参数配置，以此实现对模型结构的搜索和优化，获得更高的准确率和更好的泛化能力，其实现主要是通过自动化搜索算法来进行寻找；知识蒸馏的思路是用一个已经训练好的复杂模型去指导一个较为简单的目标模型的训练，即利用已有模型的知识来辅助训练新模型。在训练目标模型的过程中，尽可能让其同时拟合训练数据和已有模型的预测结果。以此提高目标模型的泛化能力和准确率，并且减小模型的体积和计算复杂度。这一方法的主要思路便是针对网络模型结构进行优化和压缩，从而满足嵌入式离线部署的需求。

同样有很多研究者的思路主要是通过降低卷积计算量，提高计算速度和效率。主要的方法有深度可分离卷积[27]和 Winograd 变换[28–30]等。深度可分离卷积是将标准卷积分解为深度卷积和逐点卷积两部分的卷积操作。其中，深度卷积是在每个输入通道上分别执行卷积操作，产生一组输出特征图；而逐点卷积是在深度卷积所得结果的通道上对特征图进行卷积操作，最终混合获得输出特征图。与标准卷积相比，深度可分离卷积中的深度卷积仅需要计算输入数据单通道的二维平面卷积，因此计算量大大降低；Winograd 变换是通过线性变换将卷积操作转化为转换为更小的矩阵，然后将输入通过一定的线性变换得到一个更小的张量，最后使用矩阵乘法计算输出张量矩阵乘法操作，变换后的矩阵乘法的计算量更小从而减少卷积计算的次数和复杂度，以此加速卷积操作。这种方法存在的缺点是会带来一些额外的计算开销。因此，在选择优化方法时需要综合考虑模型的结构、硬件平台等多个因素。

针对硬件上的优化思路主要是挖掘卷积计算中存在的并行性，提高计算效率，减少存储资源的占用，其中以脉动矩阵[31–33]和分块计算[34–36]等方法为代表。脉动矩阵方法是将输入数据和卷积核的运算拆分成两个部分，即首先将卷积核进行变换，将其变为一个脉动矩阵，然后在进行卷积计算时，将输入数据划分为多个小块，每个小块与脉动矩阵进行卷积计算；分块计算是将大的卷积核或大的输入

数据同时按照空间位置和时间轴方向分成若干个小块，每个小块都和卷积核做卷积操作，然后将它们合并起来得到最终的结果。

1.2.3.2 长短时记忆网络的硬件加速研究现状

随着 LSTM 在序列任务方向上的高效表现，对其进行基于离线平台加速的研究也获得了很大的进展，其中加速的思路同样有以下几种：

很多研究的思路主要是针对网络模型进行优化，从而提高计算效率，主要的方法有 LSTM 模型的剪枝[37] 和可变精度的量化[38] 等。可变精度的量化是通过低精度的数来进行网络的前向推理，以牺牲一定精度为代价，显著降低计算量。主要的思路是在不同的时间戳上的 LSTM 细胞单元内，动态地采用高精度或低精度的数值来表示细胞的状态。当细胞的状态值以缓慢的速度变化时，采用低精度表示状态值的误差非常的小。而在细胞状态变化很快时，高精度和低精度之间的差异非常明显，会导致精度明显的损失。LSTM 模型的剪枝主要采用的是稀疏化剪枝，将一定数量的权重进行赋零的操作，从而可以可以避免对隐藏状态的零值元素进行计算。为了平衡效率和精度，主要的思路是将矩阵进行分块，对每一个子块中的数值进行粗粒度的剪枝操作，块大小作为其元素的平均值，粗粒度修剪可以保留一些较小的权重，并删除一些较大的权重，因为它以块的粒度进行修剪。当稀疏度较大时，再进行细粒度剪枝，从而保留那些对精度影响较大的大权重。

还有很多加速方案的思路主要是针对 LSTM 各个门及其计算中存在的并行性[39–41]，在硬件上进行改进，从而来实现运算加速。主要的方法有批处理和调度同步技术等。批处理可以解决剪枝后的网络模型在边缘端设备上内存带宽有限的情况下带来的运算模块利用率不足和吞吐量降低的问题[42]。通过使用更多的存储元件来存储每个批次的临时结果，从而来进行批处理操作。批处理的挑战在于，在任何周期中，只有当所有批处理的所有输入元素都为零时，才能跳过计算。同时，批处理还可以并行的计算 LSTM 中出现的相似子图[43]，当输入数据到达时，其计算被分解为多个子图，如果相同拓扑的两个子图的参数权重相同，并且它们的输入具有相同的形状和计数，则称其为相同类型。如果同一类型的子图没有相互依赖性，则可以完成它们的批处理，此时可以选择一组公共子图进行批处理，从而实现并行运算，提高效率。调度同步同样是借助了批处理的思路来进行优化的，由于在网络模型中不同时间戳之间的数据具有串行依赖性，需要显式同步。并且，还需要在每个时间戳时从内存中加载递归权重矩阵。为了优化这一操作，他们使用“多批量同步并行”的方法，将递归权重矩阵进行分块，每一块在一个计算模块上进行计算，这样尽可能平衡模块间通信时延和单个模块承担全部串行计算的时延，从而用批处理并行的思想进行了计算的加速。

1.3 本文的主要贡献与创新

本论文提出了一种支持 CNN 和 LSTM 定点量化后进行前向推理的硬件架构。基于寄存器传输级别（Register Transfer Level，RTL）在更精细的层次上优化了计算模块的架构提高了并行性，提高了计算资源的利用率。同时，分析了 LSTM 的计算过程，通过数据分块和重组来减少数据依赖性，从而减少了流水线中的空等时间以提高效率。

该系统通过与片上 ARM 核合作，使得所有输入数据都存储在片外存储器中，数据读写经由片外 DDR 来执行。这种设计可以支持大参数量规模的网络模型，满足当前神经网络发展的趋势，同时简化了寄存器传输级别部分的硬件控制逻辑。本工作主要的创新点与贡献如下：

1) 提出一种提高数字信号处理器（Digital Signal Processor，DSP）计算效率的硬件架构设计。设计当前的硬件架构时通过原语的方式充分探究 DSP 内部的结构，继而选择出 DSP 的模式，构建系统的计算模块。针对定点量化后的输入数据，在单个 DSP 内实现了输入特征图一个维度上的并行，这样可以最大限度地提高模块中计算资源，即 DSP 的使用效率，系统最终运行 CNN 和 LSTM 时 GOPS/DSP 的值分别为 0.454 和 0.265，高于当前主流加速器的成果，对于计算资源的利用效率有较高的提升。
2) 提出了一种能够支持更多维度循环展开和并行加速的硬件加速器设计。该设计支持定点量化后的 CNN 和 LSTM。基于硬件描述语言（Hardware Description Language，HDL）优化了硬件加速器的架构，较大限度地挖掘出运算过程中的并行性。同时减少了量化后冗余的计算过程，优化了计算资源利用效率。
3) 提出了一种针对网络加速的数据分块和数据流重排方案。优化了 LSTM 计算中出现的数据依赖，从而减少了流水线中出现的空转时间。各模块之间形成流水线，并且结合循环展开、循环并行和数据分块的方式尽可能利用片上存储中的数据实现流水的高效运算，使得本设计可以在较高的时钟频率下运行，最终可以在 250*MHz* 的工作频率下正确运行。
4) 采用了具有高可移植性的设计。硬件加速器模块设计未使用硬件平台提供的 IP 核，通过 HDL 和原语等方式构建了所需的各个基础模块，通过模块中的参数实现对于各项资源的精细化分配，不会受到平台和 IP 核版本的限制，有很强的可移植性。
5) 提出了一套对 FPGA 的命令控制方案。通过 AXI-Lite 的控制逻辑，实现了基于处理器和 FPGA 协同的硬件加速器。基于 Zynq UltraScale+ 系列的 ZCU102

开发板进行神经网络加速器的部署，通过 ARM 核承担大部分对于数据通路的控制，简化 FPGA 部分逻辑控制的工作。FPGA 部分的硬件架构支持通过参数动态配置，由 ARM 核通过 AXI-Lite 读写 32 位的寄存器来实现不同的指令控制，从而调配加速器的并行度和数据通路等各项运行模式。

1.4 本论文的结构安排

本文的章节结构安排如下：

第一章为绪论部分。介绍了本文所进行研究的背景与研究意义，分别梳理了当前 CNN 和 LSTM 的发展历程，同时回顾了当前主流硬件加速方法的研究现状，最后总结了本系统的贡献与创新。

第二章为理论基础和方法论的相关研究。首先介绍了本文加速器目标网络模型及其网络结构的一些基本构成和原理；最后，介绍了本文采用的定点量化和基于 HDL 的加速方案技术路线。

第三章主要介绍了本硬件加速系统在处理系统（Processing System，PS）端的工作。首先总体介绍了加速器的整体架构，解释了工作负载是如何在可编程逻辑（Programmable Logic，PL）和 PS 端上进行分配的；之后主要介绍了数据如何从外部存储中进行读取；然后介绍了 PS 端与 PL 端之间进行通信所采用的传输协议，并介绍了 PL 端接收数据所封装的接口。

第四章主要介绍了本硬件加速系统在 PL 端的工作。从对神经网络进行加速的几个方面介绍了 PL 端做的设计，首先是循环展开与循环并行对循环计算的加速策略；其次是数据分块策略，介绍如何将数据进行分块便于存储在片上存储当中；最后介绍了 PL 端流水线中的其他几个重要模块。

第五章主要是介绍了如何对设计好的硬件加速系统进行相应的测试。首先分别介绍了 PL 端和 PS 端测试前所需的准备工作，之后对于测试得到的数据进行了相应的分析，证明了当前硬件加速系统可以在一个较高的时钟频率下运行。根据资源利用效率的数据也证明了本文中针对计算资源的优化是有效的。

第六章是总结与展望。总结了本文所做工作的特点，并对工作中的不足进行了分析，展望了自己未来的研究方向。

第二章 神经网络相关理论基础与加速方法的相关研究

经过数十年的探索与发展，目前深度神经网络已然成为了当前人工智能领域的重要基石，而人工智能领域的各项技术也早已深入到人们的日常生活当中，从人脸识别到语音识别等等。神经网络由各模块构成，其中基础的卷积神经网络具有强大的特征提取能力，可以提取出需要的高层次信息，配合上具有广泛应用能力适应不同输入数据的循环神经网络，可以实现更多的功能。鉴于本加速系统的目标是支持卷积神经网络和长短时记忆网络的加速，因此本章将首先介绍卷积神经网络的基础知识，继而介绍循环神经网络并且着重介绍其中的长短时记忆网络，并分别分析两种网络的加速空间以及其中的共性。由于嵌入式离线推理平台对于计算资源和存储资源的限制，如何针对模型进行合理的轻量化和提高资源利用率的设计十分重要，因此本章最后对我采用的加速技术路线进行介绍。

2.1 深度神经网络相关算法原理

2.1.1 卷积神经网络

卷积神经网络通常由卷积层、激活函数层池化层和全连接层构成，其中所具有的特征提取能力经由多层卷积层和池化层来实现，这些特征可以表示输入数据中的局部结构信息、空间关系和高层次语义信息，从而实现对数据的自动特征提取；所具有的泛化能力则是经由反向传播算法更新网络的权重参数，在训练过程中可以自动学习特征提取器，从而对输入数据中的不同特征具有良好的适应性和鲁棒性。

卷积神经模型中的卷积层采用局部连接和权值共享的方式，通过卷积操作提取局部特征。激活函数层通常在卷积层之后，以 Sigmoid、ReLU、LeakyReLU 等为代表主要作用是对卷积层的输出特征图进行非线性变换，以增加模型的非线性表达能力。其计算过程可由如下伪代码描述:

算法 2-1 CNN 中卷积层的计算过程

Input: 输入数据 X，卷积核 W，步长 s，填充 p

Output: 输出数据 Y

初始化: Y 为大小为 (N_c, H_o, W_o) 的全零矩阵

```
for i = 1 to N_c do
    for j = 1 to H_o do
        for k = 1 to W_o do
            // 计算在当前位置的输出值
            y_ijk = b_i
            for u = 1 to K do
                for v = 1 to K do
                    for c = 1 to N_{c-1} do
                        // 在当前位置应用卷积核，累加到输出值
                        y_ijk += w_{u,v,c,i} × x_{(j-1)s+p+u,(k-1)s+p+v,c}
                    end
                end
            end
            // 应用激活函数
            y_ijk = activation(y_ijk)
        end
    end
end
```

该算法中，输入数据 X 是一个大小为 (N_c, H_i, W_i) 的矩阵，其中 N_c 为输入数据的通道数，H_i 和 W_i 为输入数据的高度和宽度。卷积核 W 是一个大小为 (K, K, N_{c-1}, N_c) 的矩阵，其中 K 为卷积核的大小，N_{c-1} 和 N_c 分别为输入数据的通道数和输出数据的通道数。步长 s 表示在每个方向上卷积核的移动距离，填充 p 表示在输入数据的边缘上添加 0 的数量。输出数据 Y 是一个大小为 (N_c, H_o, W_o) 的矩阵，其中 H_o 和 W_o 为输出数据的高度和宽度。算法中的 b_i 表示卷积层的偏置项，$w_{u,v,c,i}$ 表示卷积核的权重。

池化层一般以激活函数层的输出作为输入，通过降采样的方式将输入数据的一个局部区域转化为一个值，从而减小特征图的大小，同时保留重要的特征信息，提高模型的鲁棒性和计算效率。其计算过程可描述为:

算法 2-2 池化层计算过程

Input: 输入特征图 X，池化窗口大小 K，池化步长 S

Output: 池化层的输出特征图 $Y \in \mathbb{R}^{H' \times W' \times C}$

for $c = 0$ **to do**
  for $i = 0$ **to** $H' - 1$ **do**
    for $j = 0$ **to** $W' - 1$ **do**
      $h_0 = i \times S$，$w_0 = j \times S$
      $h_1 = \min(h_0 + K, H_i)$，$w_1 = \min(w_0 + K, W_i)$
      //对输入特征图 X 在区域 $[h_0, h_1) \times [w_0, w_1)$ 进行池化操作
      $Y_{i,j,c} = \underset{h \in [h_0, h_1), w \in [w_0, w_1)}{pooling} (X_{h,w,c})$
    end
  end
end
return Y

该算法中，输入数据 X 是一个大小为 (C, H_i, W_i) 的矩阵，其中 C 为输入数据的通道数，H_i 和 W_i 为输入数据的高度和宽度。步长 s 表示在每个方向上卷积核的移动距离，输出数据 Y 是一个大小为 (C, H', W') 的矩阵，其中 H' 和 W' 为输出数据的高度和宽度。

全连接层是将前一层的所有神经元都与后一层的每一个神经元相连构成的计算层，其作用是将卷积层和池化层得到的高维特征进行降维，提取出更高层次的特征，为分类器提供更丰富的特征信息。在全连接层中，每一个输入样本都会得到一个输出向量，该向量的长度等于全连接层的神经元个数。全连接层的运算实际上是一种特殊的卷积运算，其计算过程可简单描述为:

算法 2-3 全连接层的计算过程

Input: 输入特征图 X，权重矩阵 W，偏置向量 b

Output: 输出特征图 Y

将输入特征图 X 展开为一维向量 x
$y = Wx + b$//计算全连接层的加权和
$Y = f(y)$//对加权和进行非线性变换
return Y

2.1.2 长短时记忆网络

RNN 是一种基于时间序列的神经网络结构，与传统的前馈神经网络不同，RNN 在每个时间步骤中都会保留一定的状态，即“记忆”，并将其传递到下一个时间步骤中，从而使得模型能够“记忆”到之前的信息并利用这些信息进行后续的预测。RNN 一经提出得到了广泛的应用，不过其存在的梯度爆炸或梯度消失问题则是由引入记忆细胞的 LSTM 来进行了优化和改进。LSTM 是 RNN 的一种变体，具有复杂的拓扑结构并被广泛使用。LSTM 的核心概念是细胞状态和门结构，这使得它能够避免长期依赖问题并传递过去的信息。单元中有三个门：输入门、遗忘门和输出门。

算法 2-4 LSTM 中输入门的计算过程

Input: 当前时刻的输入 x_t，前一时刻的隐藏状态 h_{t-1}，当前时刻的记忆细胞状态 c_{t-1}，权重矩阵 W_i、U_i，偏置向量 b_i

Output: 当前时刻的输入门状态 i_t

//计算输入门的加权和：

$a_i = W_i x_t + U_i h_{t-1} + b_i$

//对加权和进行非线性变换：

$i_t = \sigma(a_i)$

return i_t

输入门用于控制当前时刻的输入对记忆细胞状态的影响。输入门的计算过程如上所示。其中，通过一个 sigmoid 函数 $\sigma(\cdot)$ 对加权和进行非线性变换得到当前时刻的输入门状态 i_t。最终，将输入门状态 i_t 作为输出返回。

算法 2-5 LSTM 中遗忘门的计算过程

Input: 当前时刻的输入 x_t，前一时刻的隐藏状态 h_{t-1}，当前时刻的记忆细胞状态 c_{t-1}，权重矩阵 W_f、U_f，偏置向量 b_f

Output: 当前时刻的遗忘门状态 f_t

//计算遗忘门的加权和：

$a_f = W_f x_t + U_f h_{t-1} + b_f$

//对加权和进行非线性变换：

$f_t = \sigma(a_f)$

return f_t

遗忘门用于控制前一时刻的记忆细胞状态对当前时刻的记忆细胞状态

的影响。遗忘门的计算过程如上所示。其中 W_f 和 U_f 分别是输入和前一时刻隐藏状态的权重矩阵，然后通过一个 sigmoid 函数 $\sigma(\cdot)$ 对加权和进行非线性变换得到当前时刻的遗忘门状态 f_t。最终将遗忘门状态 f_t 作为输出返回。

算法 2-6 LSTM 中输出门的计算过程

Input: 输入特征 x_t，上一时刻的隐藏状态 h_{t-1}，上一时刻的细胞状态 C_{t-1}，输出门的权重矩阵 W_o，偏置项 b_o

Output: 输出值 y_t，当前时刻的细胞状态 C_t，当前时刻的隐藏状态 h_t

将 x_t 和 h_{t-1} 拼接为一个向量 v_t

//计算输出门的输入：

$z_t = W_o \cdot v_t + b_o$

//计算输出门的输出：

$o_t = \sigma(z_t)$

//计算当前时刻的细胞状态：

$C_t = f_t \odot C_{t-1} + i_t \odot g_t$

//计算当前时刻的隐藏状态：

$h_t = o_t \odot \tanh(C_t)$

//计算输出值：

$y_t = h_t$

return y_t, C_t, h_t

输出门控制着 LSTM 细胞的输出，决定着有哪些信息要输出。在该算法中，σ 表示 Sigmoid 函数，tanh 表示双曲正切函数，$\odot$ 表示逐元素相乘。最终，本文的设计中将当前时刻的隐藏状态 h_t 作为输出值 y_t 返回，同时返回当前时刻的细胞状态 C_t 和隐藏状态 h_t。

2.2 加速技术路线

2.2.1 定点量化

由于 FPGA 的计算资源和存储空间有限，对于大规模的神经网络模型，需要对其进行量化，以便在 FPGA 上运行。首先在工作时钟频率方面，通过浮点数进行推理时，在综合实现中设置时钟频率过高时会提示浮点 IP 核要求时钟至少为 100ns，即最多为 10MHz[44]，限制了时钟频率，这也极大地影响了推理的速度；其次，在存储方面，对于输入特征图和权重，无论是采用 8 位定点量化还是 16 位定点量化所用到的存储均减少。这种减少对于片上存储这种稀缺资源来说无疑是宝

贵的。此外，在计算资源消耗方面，对于 32 位浮点推理而言，每一个乘法器都会消耗两个 DSP。而对于定点乘法，一个 DSP 可以完成两组 8 位定点乘法或一组 16 位乘法。这样对于计算资源的占用也会大大减小。最后，在运算速度方面，在使用流水线情况下 DSP 支持的巅峰运算频率是浮点运算的六倍[45]。因此设计中对网络的输入特征图、权重等数据采用量化推理的方式进行定点数计算，这能够极大的降低参数所占存储，减小计算的复杂度。量化的基本思想如式 (2-1)：

$$f = s(q - z) \tag{2-1}$$

在上面的定义中，f 表示浮点数，s 表示线性变化的伸缩因子，q 为量化后的定点数，z 为零点即浮点中的 0 映射到对应的量化后的定点数。

在卷积神经网络的量化推理过程中[46]，每一个输出结果中的一个值的卷积计算过程可以表示为式 (2-2)

$$f_{out} = \sum_{c=0}^{IC}\sum_{x=0}^{KW}\sum_{y=0}^{KH} f_{in}(c, x+ow+y+oh) f_w(c,x,y) \tag{2-2}$$

将原式替换为定点数，可以得到式 (2-3)，其中仍然存在一个得到浮点数结果的计算 $\frac{s_{in}s_w}{s_{out}}$，可以通过转换为定点数 M 加移位 n 的方式进行，如式2-4所示:

$$\begin{aligned} q_{out} &= z_{out} + \frac{s_{in}s_w}{s_{out}} * \\ &\sum_{c=0}^{IC}\sum_{x=0}^{KW}\sum_{y=0}^{KH}(q_{in}(c, x+ow+y+oh) - z_{in})(q_w(c,x,y) - z_w) \end{aligned} \tag{2-3}$$

$$\frac{s_{in}s_w}{s_{out}} = 2^{-n}M \tag{2-4}$$

通常在卷积运算结束后，会连接一个 ReLU 激活函数，而在量化后，执行的 ReLU 操作实际如式 (2-5)

$$q_{out} = \begin{cases} z_{out} + X & z_{out} + X \geq z_{out} \\ z_{out} & z_{out} + X < z_{out} \end{cases} \tag{2-5}$$

对于 LSTM 采用的量化方式则在此基础上有所修改[47,48]，其中遗忘门、记忆门、输出门中涉及到的计算，输入采用相同的 8 位定点模式。由于它们的输出都需要经过非线性的激活函数，因此为了保证精度，输出向量中的一个元素的计算过程

如式 (2-6) 可得

$$f_{out} = s_{(x,h)} s_w * \sum_{i=0}^{cols} (q_{(x,h)}(i,0) - z_{in})(q_w(row,i) - z_w) \tag{2-6}$$

经过激活函数后得到的输出如式 (2-7) 所示：

$$f_{gate} = activation_function(f_{out} >> shift\_bits) \tag{2-7}$$

LSTM 网络中用到的激活函数为 sigmoid 函数和 tanh 函数两种非线性函数，而 tanh 可以由 Sigmoid 函数表示，如式 (2-8) 所示：

$$\tanh(x) = \sigma(2x) - 1 \tag{2-8}$$

在该式中，σ 表示 Sigmoid 函数，tanh 表示双曲正切函数。因此，之后讨论定点量化的数据如何选取时，只考虑 Sigmoid 函数，tanh 可以相应的由式 (2-8) 得出。由于 Sigmoid 函数在 [-6,6] 的定义域区间之外值趋于饱和，在这个区间外的值可以分别视作 1 和 0，此时并不会带来许多误差。因此将 Sigmoid 函数截取如式 (2-9) 所示：

$$\sigma(x) = \begin{cases} 0 & x < -6 \\ \sigma(x) & -6 \le x \le 6 \\ 1 & x > 6 \end{cases} \tag{2-9}$$

由于 Sigmoid 函数的输出有着对于小数表示的需求，为了保证计算的精度，减少误差，所以不能采用整数截断的方式。出于精度和简化工作量上的考量，并且为了使硬件计算模块能够兼容卷积神经网络采用的 8 位定点整数量化的乘法运算。本文设计的硬件加速系统中采用的浮点数表示方法不能使用 IEEE754 标准的浮点数，而是选择通过另一种传统的 FPGA 上对于浮点数的表示方式，即采用定点数 $Q_{m,n}$ 的表示方法。$Q_{m,n}$ 中有 1 位为符号位，m 位为整数位，n 位为小数位。$Q_{3,12}$ 可以表示 [-8,8) 范围内的值，满足本文在式 (2-9) 中的表示需求，因此对于输出的 f_{out} 采用 $Q_{3,12}$(3 位整数位、12 位小数位) 的 16 位定点数进行表示，而超过 $Q_{3,12}$ 可表示范围的值则采用截断的方式进行保存。这样的表示方法用最少的整数位来表示式 (2-9) 中未截断的部分，并且保留了尽可能多的小数位，从而保证激活函数输出时的精度。同时，16 位定点数的激活函数考虑进行硬件部署或程序实现时，通过查找表的方式来实现激活函数从存储的角度和复杂度的角度来看都是可行的。对于激活函数的输出，tanh 激活函数的输出值域为 [-1,1]，Sigmoid 激活函数的输

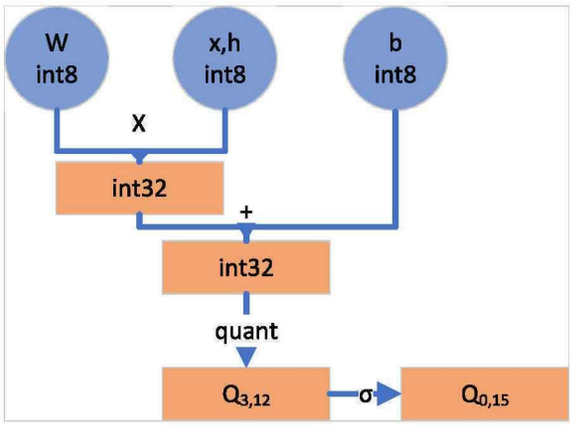


图 2-1 普通门的输出

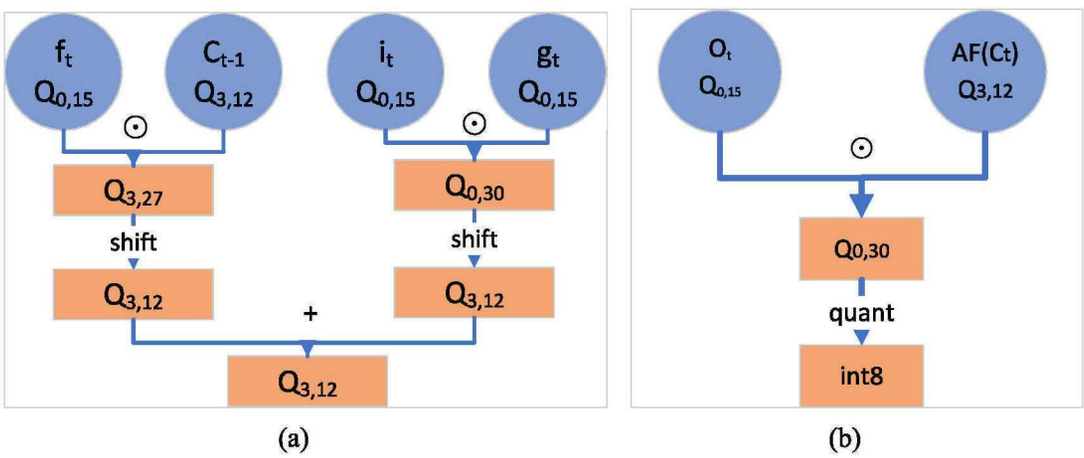


图 2-2 (a) 细胞状态；(b) 隐藏状态。

出值域为 [0,1]。由于 $Q_{0,15}$ 可以表示 $[-1, 1-2^{-15}]$ 的值，因此可以继续沿用 16 位定点数的表示方法，即采用 $Q_{0,15}$(0 位整数位、15 位小数位) 的方式来进行表示，如上图2-1所示。

最终 C_t 的值采用 $Q_{3,12}$ 的格式进行保存，如图2-2(a)所示，而 h_t 则经过定点数乘法量化后重新作为 8 位定点整数作为下一次的输入继续使用，如图2-2(b)所示。如上展示的量化基本思路所示，通过量化操作，浮点数计算都变成了定点数计算，极大地减小了数据量的存储占用以及计算的复杂度，从而利于网络在资源受限的平台上进行部署。

2.2.2 加速平台

FPGA 是一种可编程逻辑器件，能够根据用户的需求重新编程，以完成特定的任务。FPGA 由大量可编程的模块构成：逻辑模块是 FPGA 最基本的模块之一，它由逻辑门和触发器等基本逻辑元件组成，可以实现各种基本逻辑功能，例如与、

或、非等；存储模块也是 FPGA 中比较重要的模块之一，它由寄存器、FIFO(First Input First Output)、RAM(Random Access Memory) 等组成，用于存储数据和状态信息。存储模块通常用于实现 FPGA 中的状态机、缓存、存储器等功能; 时钟管理模块用于产生和管理时钟信号，保证 FPGA 中各个模块的时序一致性和稳定性。通常包括时钟发生器、分频器、锁相环等功能，用于产生、分频和同步时钟信号; 数字信号处理模块是 FPGA 中比较常用的模块之一，它由各种 DSP 组成，用于实现各种数字信号处理功能，例如乘法、加法、滤波、FFT 等。数字信号处理模块通常用于实现数字信号处理器和通信系统等应用；片上系统模块是 FPGA 中的高级模块之一，它包括各种处理器核、总线接口、存储控制器等功能，可以实现各种高级应用，例如嵌入式系统、智能控制器等。目前的 FPGA 已经具备了很高的计算能力和较大的存储容量，在人工智能领域，FPGA 常用于加速卷积神经网络、循环神经网络等深度学习模型的推理。FPGA 的并行计算能力和低功耗特性，使得它在大规模神经网络加速方面有着很大的优势。当前市场上使用的主流 FPGA 主要来自 Xilinx、Intel 和 Lattice 三家公司，也分别对应三种 EDA 开发工具，即 Vivado, Quartus Prime 和 Diamond。

本系统使用的是 Xilinx 公司的 Zynq UltraScale+ 系列开发板，配备了多核 ARM Cortex-A53/Cortex-R5 处理器，可以运行 Linux 操作系统和其他应用程序。这使得 Zynq UltraScale+ 系列开发板成为一种强大的嵌入式开发平台，可以满足各种不同的应用需求。同时，还包括 Xilinx FPGA 可编程逻辑，可以支持各种不同的硬件设计和加速功能。这使得开发者可以根据具体需求自定义硬件逻辑，从而实现更高效的嵌入式系统设计。开发的工具则使用配套的 Xilinx Vivado 设计套件和 Xilinx Vitis 套件形成的工具链，包括设计、仿真、综合、实现和调试等功能，同时也支持高级综合和嵌入式开发。

使用 Xilinx 工具链开发的流程主要是先将编写的 HDL 源文件添加到工程中。可以直接在 Vivado 中编写，也可以通过导入文件的方式添加；其次是约束文件的编写，以此定义 FPGA 的引脚和时序等信息，Vivado 中可以在生成详细设计后进行可视化的分配，也可以直接添加约束文件编写；然后是在 Vivado 中进行综合，将 HDL 代码转化为 FPGA 中的逻辑门级别的结构，并生成 RTL 级别的网表文件。之后是在 Vivado 中进行实现，将 RTL 级别的网表文件进行优化、布局和布线，并生成比特流文件，通过 vitis 进行处理器部分逻辑的编写。最后则是通过 JTAG，将生成的比特流文件下载到 FPGA 中进行验证和测试。

2.2.3 技术路线

在进行硬件加速器的设计时，虽然目前采用高层次综合 (High Level Synthesis，HLS) 技术路线进行设计的方式受到了广泛的关注与应用，但是其存在的问题是产生冗余代码，不能很好地控制电路的逻辑结构和资源分配等，因此产生的硬件电路的性能和资源利用效率，甚至精度都会较差；与所用平台相关性强，可移植性较差，导致很多应用场景无法应用。因此对于规模较大的，复杂度较高的神经网络加速设计来说，并不适宜采用此技术路线。

本系统选择采用 HDL 技术路线进行设计，其优点主要在于可以灵活地描述数字电路的结构和行为，可以精确地控制硬件资源的使用，满足不同神经网络模型的需求；使用 HDL 进行硬件设计可以获得高性能的硬件加速器，无需像软件运行在 CPU 上那样进行复杂的指令解码等操作，而是硬件是直接执行，可以极大地提高运行效率；HDL 可以通过原语等方式对硬件进行精细化的设计，实现对时序、面积、功耗等方面进行优化，可以在保证功能正确的前提下，实现更高效的硬件加速器；设计的硬件模块可以被复用，可以快速构建更复杂的硬件系统，本系统中未使用 IP 核，可以在几乎所有 FPGA 硬件平台上进行部署，实现了很好的可移植性。

2.3 本章小结

本章主要介绍了 FPGA 加速神经网络的一些基础算法原理和技术路线。首先，介绍了 CNN 和 LSTM 的网络架构并重点着墨于其中的计算过程。此外介绍了所使用的加速平台，数据预处理的基本思路。最后介绍了加速平台，通过与 HLS 进行比较，证明了采用 HDL 进行高可移植性、精细化的加速器设计的可行性。

第三章 片上芯片部分架构设计

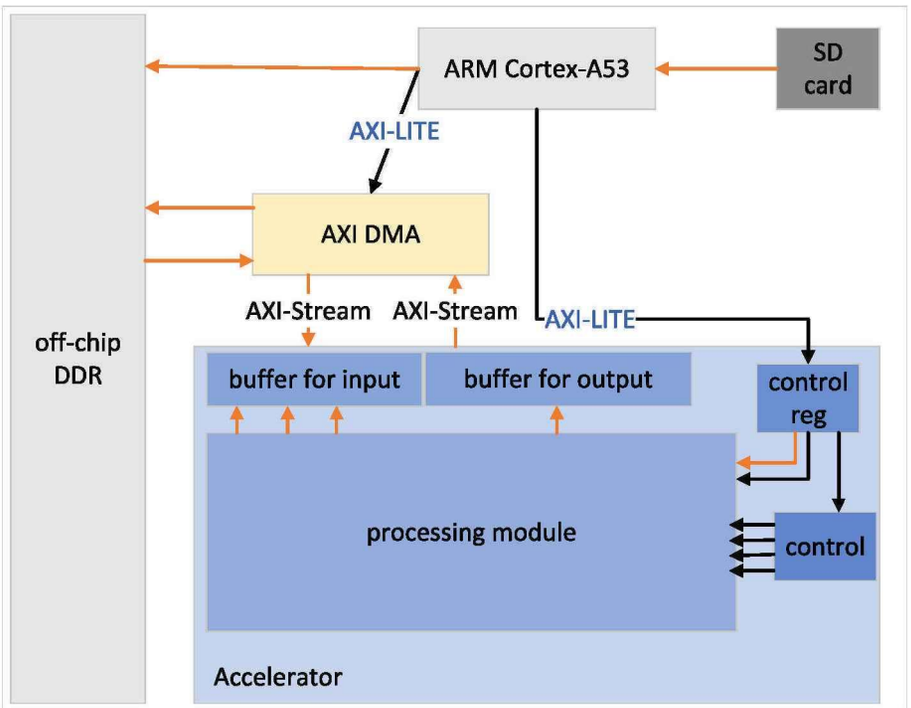


图 3-1 系统整体架构

如前述章节介绍的那样，在当前的发展趋势下，愈发需要将神经网络部署到嵌入式的离线平台上。而同时，神经网络为了提高泛化能力和精度也不断向着深度神经网络的方向在演化。对于资源受到限制的嵌入式平台，当前的大多数大规模的神经网络模型其所需的参数都无法完全放置在片上存储中，因此本文的设计中将网络推理所需要的输入数据和相关参数放置在 SD 卡上。

系统的整体架构如上图3-1所示，在本硬件加速系统中，只使用了一个 ARM Cortex-A53 的核。大多数的运算由基于 verilog 编写的 PL 端的 FPGA 部分进行，即图中的 Accelerator 部分，这部分的工作即对应第四章所介绍的部分。在 PS 端部分，因为 PS 端的 ARM 可以直接对 PS 端的 DDR 进行读写，因此通过 PS 端的 ARM 将 SD 卡上的数据读取到 PS 端的 DDR 上。ARM 核通过 AXI-Lite 配置 AXI DMA 从而将数据通过 PL 端实现的 AXI-Stream 接口进行传输，将每次需要计算的数据经由 AXI DMA 搬运到 FPGA 实现的加速器。此外，ARM Cortex-A53 还通过 AXI-Lite 读写 PL 端的寄存器，从而实现命令控制的功能。

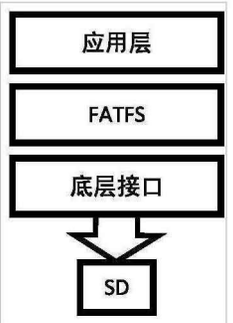


图 3-2 FATFS 层次结构

3.1 数据的存储

神经网络模型通常包含大量的参数，这些参数需要被加载到硬件加速器中进行计算，这些参数通常存储在外部存储设备中，通过数据接口加载到硬件加速器的内存中。同时，神经网络推理所需的输入数据和中间数据也需要存储。特别是对于大规模的神经网络和大数据集，其输入数据和中间结果往往非常庞大，无法完全容纳在硬件加速器的内存中。为了满足对更大规模的神经网络计算的需求，本设计需要将数据存储在外部存储中。

由于输入数据经过量化通常只有 8 位和 16 位的大小，但是为了提高传输的效率，所采用的传输方案中一般一个时钟周期可以传输 64 位和 128 位的数据。因此，需要考虑数据的排布方式，一次传输多组数据。本文设计中采用的是先按照输入通道维度再按照输入窗口通道维度来进行数据的排列，也就是首先遍历读取到的的数据是输入通道当中的数据，之后是输入窗口内列和行上的数据。

在本文部署的开发板上常用的外部存储设备主要有 SD 卡和 Flash，Flash 一般用于存储引导镜像和文件系统，而 SD 卡还会更多的用于存储数据文件和配置文件。因此，本文选择将所需外部数据存储在 SD 上。FATFS 是一个 FAT 文件系统模块，通常用于小型的嵌入式系统，层次结构如图3-2所示。Xilinx Vitis 的 standalone 已经移植好了 FATFS 文件系统，因此在 Vitis 中添加 xilffs 库后，只需要在应用层调用相应的函数编写 C 代码即可实现对 SD 卡的读写。因为本文系统中用到的数据都通过文件的形式存储在 SD 卡上，读写只需要对 SD 卡上对应的文件进行操作即可。算法如3-1所示，首先需要初始化 SD 卡设备，这个过程包括配置 SD 卡的引脚、时钟等信息。接着需要初始化 FATFS 文件系统，这个过程会创建一个虚拟的文件系统。在 SD 卡设备可用的情况下，由于 ARM 核可以直接通过地址访问的方式对 DDR 进行读写，因此可以创建一个缓冲区从而通过 FATFS 提供的 API 函数来实现对 SD 卡上的数据到 DDR 数据之间的搬运。最后需要卸载 FATFS 文件系统，释放相应资源，并且关闭 SD 卡设备。

算法 3-1 基于 FATFS 对 SD 读写

```
Input: SD 卡设备
Output: 读写后的文件
初始化并挂载 SD 卡;
重新挂载 SD 卡;
if 挂载失败 then
    格式化 SD 卡，并结束;
else
    while SD 卡设备可用 do
        打开文件，指定读模式或写模式;
        if(读模式){
            移动读指针;
            读文件到缓冲区;
            将缓冲区中的数据写入 DDR 对应的地址;}
        else{
            从 DDR 对应的地址将数据写入缓冲区;
            移动写指针;
            将缓冲区中的数据写入文件中;}
    end
    卸载 FATFS 文件系统;
    关闭 SD 卡设备;
```

3.2 通信方案

在 Zynq UltraScale+ SoC 上各硬件组件通过连接到总线上来实现彼此之间的通信和数据交换。在总线中，硬件组件的通信是基于某种标准化协议进行的，这样就可以确保不同的硬件组件在总线上具有相同的接口和协议，从而实现互通性和兼容性。因此针对不同的传输需求，本文的 FPGA 部分需要构建控制逻辑状态机，封装出不同的总线协议接口实现 PS 部分与 PL 端的通信。

3.2.1 数据传输方案

本文设计的数据传输方案主要考虑的优点是可以配合 PL 端设计的数据分块和循环展开方案来进行相应的访存优化。当并行输入通道并且展开输入窗口循环时，每一个时钟周期产生的元素都是有效的，这样可以产生最少的中间结果，减少数据的重复传输；同样地，对于矩阵和列向量乘法的部分，展开不同列中元素与列向量不同行中的元素也能够减少中间结果的产生。此外，对于输入数据和权重数据的复用，主要是通过输出窗口间循环的展开来产生的，输入数据的复用率越高，通过数据传输方案进行重复传输数据的也将随之变得越少。在理想状态下，

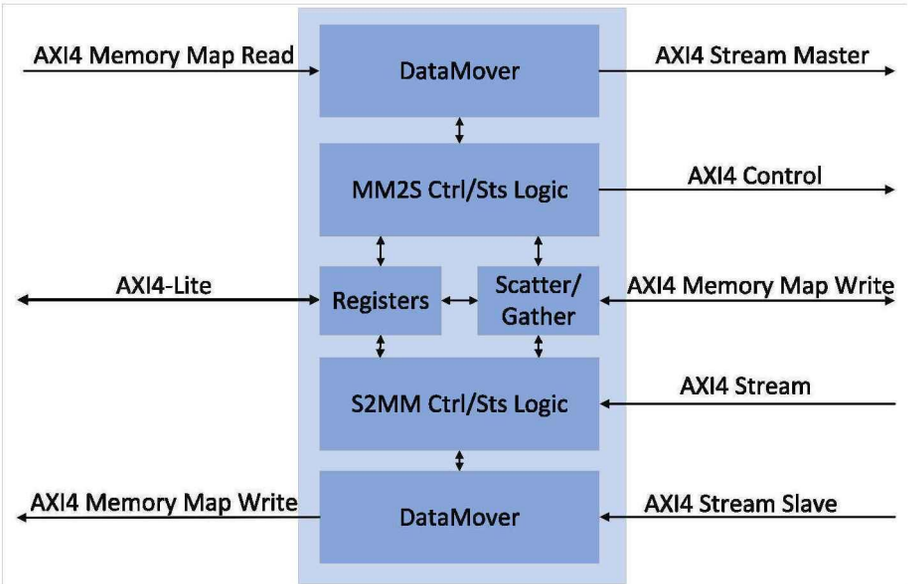


图 3-3 AXI DMA 运行框图

分块策略可以按照原式特征图的大小来进行片上的缓冲，此时可以实现每一个输入数据和权重数据都只访问一次片外存储，即只通过数据传输方案进行一次传输。但是受制于片上资源的限制，循环展开和循环并行策略通常没有办法在 PL 端实现完全的缓存，往往需要根据资源情况进行不同维度的展开。因此本文设计选定的数据传输方案需要逻辑简单，访存容易，能够进行连续传输。选择通过 AXI DMA 来进行数据传输。

Zynq Ultrascale+ 上的 AXI DMA 是一种专用的 DMA 控制器，它可以通过 AXI 总线实现高速数据传输，减少 CPU 与 DMA 之间的通信量，实现数据传输的高效率。通过配置控制寄存器的方式对其进行 AXI DMA 是通过寄存器对其进行传输的参数和通道配置。在工作时，AXI Master 向 AXI DMA 发起传输请求，AXI DMA 控制数据从 Memory 传输到 AXI Master 或从 AXI Master 传输到 Memory。传输时，数据从 Memory 经过 DMA 控制器通过 AXI 总线传输到 AXI Master 或从 AXI Master 传输到 Memory，达到高速数据传输的目的。AXI DMA 的运行框图如3-3所示。如图中所示其主要由以下几个组件构成：

1) DMA 控制器：控制 DMA 读写操作的整个流程，包括设置传输的起始地址、传输大小、方向等。
2) AXI 协议转换器：负责将 AXI4-Memory Map 或 AXI4-Lite 协议转换成 AXI4-Stream 协议，用于与 DMA 控制器进行数据传输。
3) Scatter Gather 控制器：支持 Scatter Gather DMA 模式，将一个大的数据块分割成多个小块，由 DMA 控制器进行传输。

在图3-3所示的框图中，对于数据传输部分，DMA 控制器接收到传输请求后，将 AXI4-Memory Map 获得的数据转换成 AXI-Stream 协议从而传输到 PL 端。因此在加速器的 PL 端，本系统需要设计支持 AXI-Stream 协议的接口来实现同 DMA 的数据交互。在加速系统的 PL 端，针对当前接口的传输需求，基于 verilog HDL 语言将 S_AXIS 接口封装为如表3-1的接口。

表 3-1 S_AXIS 接口信号定义

信号	描述
S_AXIS_tdata	输入数据
S_AXIS_tlast	数据边界
S_AXIS_tvalid	主端握手
S_AXIS_tready	从端握手

该接口的通信逻辑由状态机实现，如算法3-3所示，在使能信号拉高后会开始进行准备信号和有效信号的握手，握手成功的信号标志着当前的输入数据有效。因此会将本次传输的数据写入到片上的存储中，在计算好延时后输出写完成信号，用于开启后续相应的计算逻辑。最终，在加速器系统中 AXI DMA 的架构如图3-4所示，其中 M_AXI_MM2S 从 PS 端的 DDR 中读取数据；M_AXI_MM2S 则负责搬运数据到 PS 端的 DDR 中；M_AXIS_MM2S 将需要的输入数据传入加速器 PL 端封装好的接口中；S_AXIS_S2MM 接收从 PL 端加速模块计算得到的结果。在 Xilinx Vitis 中编写 DMA 运行的算法逻辑如算法3-2所示。

算法 3-2 AXI DMA 的工作逻辑

Input: 输入数据地址，输出数据地址，数据长度

Output: 数据被复制到输出地址

1 打开 DMA 的通道;
2 配置 DMA 的输入输出地址和数据长度;
3 配置 DMACR 寄存器;
4 将输入数据地址写入 AXI DMA 的控制寄存器;
5 将输出数据地址写入 AXI DMA 的控制寄存器;
6 将数据长度写入 AXI DMA 的控制寄存器;
7 启动 DMA 传输;
8 等待 DMA 传输完成;
9 关闭 DMA 通道;

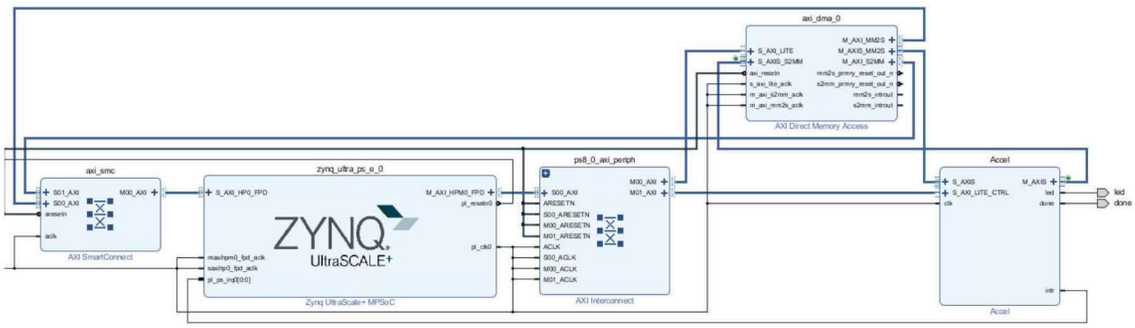


图 3-4 AXI DMA 系统框架图

通过本节的设计，优化了硬件中对于填零操作所需的硬件模块。当需要进行输入数据的补零操作时，PS 端部分只需每次在接收到 S_AXIS_tready 信号拉高后，判断输入特征图当前行的传输工作是否完成，便可以在数据传输的一开始和最后进行相应的补零操作。相较于传统的硬件加速系统设计中通过实现 padding 模块进行补零操作的设计[49]，本文通过软硬件部分的协同，节省了相应的硬件资源。同时传统的硬件加速器实现中，padding 模块需要增加多级流水线来实现数据不同大小的补零，并为不同的补零操作设计不同的状态机状态。当前的设计相较而言优化了硬件系统部分流水线的效率，提高了吞吐率。

3.2.2 命令控制方案

传统的硬件设计方案中，受制于 HDL 语言的限制，其控制逻辑通过比较固定的状态机来进行实现，存在着灵活性较低的弊端。通过参数进行修改的方式尽管可以一定程度上解决这一问题，但是当对不同网络进行部署时，涉及到的功能更改和调整往往需要进行大量的重新设计和重新实现。同时，由于硬件设计中的并行性，调试问题可能涉及信号传播、时序问题、时钟领域交叉等会带来重新验证和调试上的困难。这通常会增大再次部署时的开发周期，对于较大型的硬件电路设计来说，再次进行电路的实现和验证时间成本很高。所以本文所设计的支持较大规模神经网络部署的硬件加速器对此方面进行了相应的改进。

改进的思路出发点主要是使用软件开发的控制逻辑可以比 HDL 设计的控制逻辑更灵活地进行功能修改和调试。因此，本文的设计中针对传统硬件设计中这种重配置后开发周期长，灵活性低的缺陷，结合开发板的处理器设计了一套 32 位的命令集控制逻辑。以此控制逻辑作为硬件加速器的命令控制方案，提高灵活性，降低重配置后的开发周期。实现的方式是通过 AXI-Lite 协议进行硬件系统部分多组寄存器的写入，每个寄存器的 32 位数中从不同地址读出不同位宽的数据，解读为不同的指令。通过指令中的数据来控制硬件计算模块中的数据通路，从而实现

算法 3-3 S_AXIS 接口通信逻辑

```
Input: 输入信号 en,S_AXIS_tlast,S_AXIS_tvalid
Output: 输出信号 wr_en,wr_done,S_AXIS_tready
if state == IDLE then
    if en then
        next_state = READ;
    else
        next_state = IDLE;
    S_AXIS_tready = 0;
else if state == READ then
    if S_AXIS_tlast then
        next_state = FINISH;
    else
        next_state = READ;
    wr_en = S_AXIS_tready & S_AXIS_tvalid;
    S_AXIS_tready = 1;
end
else if state == FINISH then
    next_state = IDLE;
    S_AXIS_tready = 1;
    延迟两拍后拉高 wr_done;
end
else
    next_state = IDLE;
```

对当前计算模块的控制。AXI-Lite 是 ARM 公司推出的一种轻量级的总线协议，很适合用于连接较小、低带宽的外设，主要应用在系统中对于处理器（例如 ARM Cortex-A 系列处理器）与周边 IP 的控制、配置和状态读取等场景，常用于配置和控制寄存器的读写操作。因此，可以看到如图3-3中所示，AXI DMA 就是通过 AXI-Lite 协议对自己的配置寄存器进行数据的传输的。CPU 通过该协议将数据地址、数据长度的参数写入到 AXI DMA 的控制寄存器中，从而将高速数据传输交给 DMA，减少 CPU 自身的负担，提高系统的性能和效率。所以，首先需要在加速系统的 PL 端部分基于 verilog HDL 语言实现的 S_AXI_LITE_CTRL 接口封装为如表3-2的接口。

表 3-2 S_AXI_LITE_CTRL 接口信号定义

信号	描述
S_AXI_LITE_CTRL_awaddr	写地址
S_AXI_LITE_CTRL_awprot	写通道保护
S_AXI_LITE_CTRL_awvalid	主端写地址握手
S_AXI_LITE_CTRL_awready	从端写地址握手
S_AXI_LITE_CTRL_wdata	写数据
S_AXI_LITE_CTRL_wstrb	写数据字节有效位
S_AXI_LITE_CTRL_wvalid	主端写数据握手
S_AXI_LITE_CTRL_wready	从端写数据握手
S_AXI_LITE_CTRL_bresp	写响应
S_AXI_LITE_CTRL_bvalid	从端写响应握手
S_AXI_LITE_CTRL_bready	主端写响应握手
S_AXI_LITE_CTRL_araddr	读地址
S_AXI_LITE_CTRL_arprot	读通道保护
S_AXI_LITE_CTRL_arvalid	主端读地址握手
S_AXI_LITE_CTRL_arready	从端读地址握手
S_AXI_LITE_CTRL_rdata	读数据
S_AXI_LITE_CTRL_rresp	读响应
S_AXI_LITE_CTRL_rvalid	主端读数据握手
S_AXI_LITE_CTRL_rready	从端读数据握手

该接口接收 ARM 核传输的命令的逻辑为如3-4所示的伪代码算法，在其中省略了原代码中用于表示接收成功后的回应信号，实际上回应结果的信号也是需要进行相应的握手并且确认传输成功的过程，只有这样本次传输才会视作成功，传输才会结束。因此，伪代码中省略了这部分的握手逻辑，默认回应的结果也成功完成了握手，传输成功。在 PL 端的 Lite 接口主要是作为 slave 接口负责接收命令，因此伪代码的逻辑中只重点展示了 PS 端进行写数据，然后 PL 端将命令读出的这个过程，故而只展示了写地址通道 (AW 通道) 和写数据通道 (W 通道) 的握手过程。由于其他的命令寄存器实际上也是相同的过程，因此在为代码中只挑选了一个输出信号 ctrl_reg，更多的命令寄存器只需要传入不同的地址 axi_awaddr，通过不同的寄存器进行传输即可。

算法 3-4 S_AXI_LITE_CTRL 接口通信逻辑

Input: 输入信号 axi_awaddr,axi_awvalid,axi_wdata,axi_wvalid,axi_wstrb,

Output: 输出信号 axi_awready,axi_wready,ctrl_reg,axi_bresp,axi_rresp

```
if axi_awready & axi_awvalid & axi_wvalid then
    拉高 axi_awready 进行握手;
    从 axi_awaddr 获取写地址;
else if axi_wready & axi_wvalid & axi_awready & axi_awvalid then
    根据读取到的地址 axi_awaddr，将命令 axi_wdata 写入对应的寄存器
     ctrl_reg 中;
end
else
    拉低 axi_awready;
```

本硬件加速器使用了六个寄存器来实现所需的命令控制，每一个命令寄存器的作用如表3-3所示。

表 3-3 命令寄存器的作用

寄存器	作用
reg1	控制数据的接收和发送，以及数据流
reg2	储存量化所需的参数
reg3	控制行缓冲
reg4	储存权重/特征向量在片上存储中的基址
reg5	储存偏置在片上存储中的基址
reg6	储存输出数据写到 DDR 中的基址

其中第一个寄存器用于使能输入数据（卷积神经网络中是输入特征图数据，长短时记忆网络中是用于相乘的矩阵数据），权重数据（卷积神经网络中是权重数据，长短时记忆网络中是数据重排后的特征向量），偏置数据的接收，以及几个用于控制硬件计算数据流的多路选择器的控制位和乒乓缓冲的切换控制。其他的命令寄存器传输的命令较为简单，因此不做更多的展示。而命令寄存器一的命令较为复杂，详细的命令解析如图3-5所示，通过数据流的控制可以选择不同的计算模式，选择输出结果的存储，也可以选择所需的取数据基址等操作。第二个寄存器用于传输所需的量化参数，即量化伸缩因子和量化移位参数。第三个寄存器用于存储和行缓冲相关的参数，即特征图大小和行缓冲工作模式的指定。第四个寄

图 3-5 reg1 命令解析

存器用于表示权重的基址，由于所占的存储空间较少，且会进行大量的复用，通过片上存储进行存取会提高效率。PL 端的计算模块可以从此处开始读取到权重数据，并通过维护计数器正确地获取到所需的权重数据；相似的，第五个寄存器表示的是偏置数据的基址，第六个寄存器存储的是存放输出数据的基址，PL 端以此将数据发送到 PS 端 DDR 的对应存储空间中进行存储。

传统的硬件加速系统设计中，一般通过状态机的方式来实现控制逻辑。相较于本章提出的命令控制方案来说，存在控制逻辑难以重配置，灵活性较差的缺点[48–50]。因此本文提出的设计精简了硬件系统部分的控制逻辑，提高了灵活性。

3.2.3 中断

硬件加速系统中，需要一个方式使得 PL 端可以通知 PS 端的处理器当前计算流程已经完成，从而进行下一步运算所需的数据传输和使能等操作。相比于轮询等待 IO 或外设的状态变化的方式，中断可以有效地解决处理器大量等待外设返回数据的问题，提高系统的响应速度和效率，使得处理器可以在处理其他任务时同时监控外设状态，加强了处理器的多任务的处理能力。在 Zynq Ultrascale+ 系列开发板上集成了双核 ARM Cortex-A53 处理器和 FPGA 逻辑的 SoC 芯片，通过系统中断在异构处理系统中传达状态、事件、请求和错误。其中断控制器架构如图3-6所示。该中断控制器用于管理从 PS 和 PL 发送到处理器的中断，所有的中断都由唯一的中断 ID 来进行标识。因此本文的设计中选择通过中断控制器采用 PL 到 PS 端的中断来实现此功能。

中断控制器有多个中断信号输入和多个中断信号输出，从 PL 来的中断属于私有外设中断，在 PL 内为低电平有效，但在传输至 PS 端后会被反转，在 PS-PL 接口上属于高电平有效。因此需要做的工作是为对应 PL-PS 中断的中断 ID 注册一个中断处理函数，在中断的高电平到达中断控制器后会调用中断处理函数，从而实现本文所需要的功能。编写一个中断处理的函数，使中断发生时拉高规定的信号；在主程序中只需要编写一个在指定信号拉低时保持等待的处理函数即可，当中断信号拉高时，将其置位

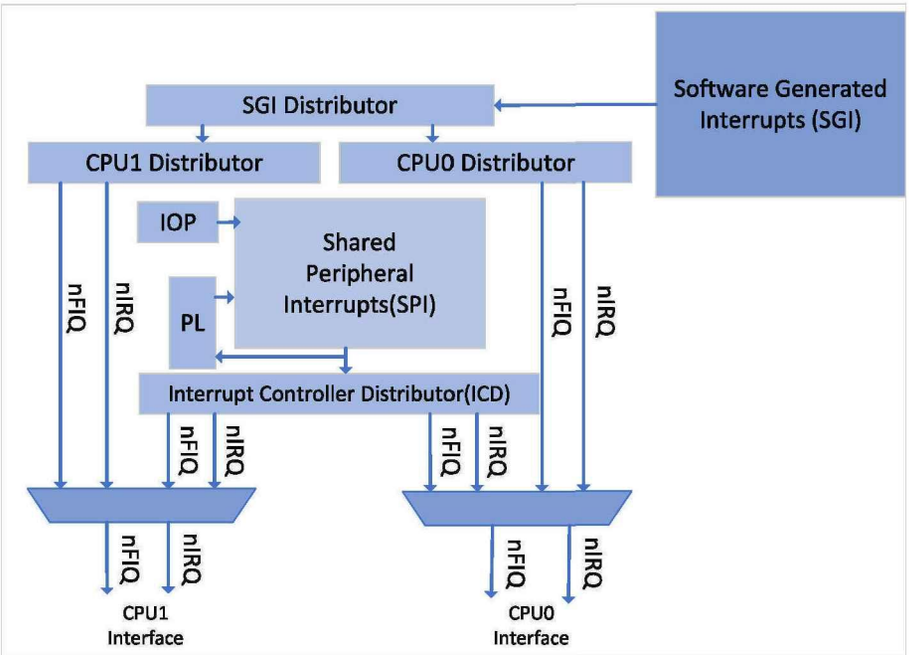


图 3-6 中断控制器架构图

并执行下一步操作。所编写的代码其处理逻辑可以表述为如算法3-5所示。

算法 3-5 中断处理

```
1 定义信号 interrupt;
2 定义中断处理的函数 void callback(){interrupt = 1;}
3 注册该回调函数;
4 执行步骤 1;
5 while interrupt == 0 do
6 |   空等;
7 end
8 interrupt = 1;
9 执行步骤 2;
```

3.3 本章小结

本章主要围绕 PS 端所承担的工作展开，首先介绍了为什么需要使用外部存储，介绍了外部存储中数据是如何进行排布，以及如何从外部存储将数据搬运至 PS 端的 DDR 进行存储；其次，介绍了本文为配合访存优化所选取的传输方案，分析了其具有的优势，并介绍了 PL 端为接收神经网络输入的流式数据所封装的接口和控制逻辑；再次，介绍了本文为了提高功能修改和调整的灵活性所才采用的命

令控制方案，并且介绍了 PL 端为接收命令所封装的接口和控制逻辑；最后，介绍了 PS 端为了快速了解到 PL 端结束当前计算任务所采用的中断，并且介绍了处理中断的程序逻辑。

第四章 可编程逻辑硬件设计

如第三章所介绍的，本硬件加速系统的整体架构可以分为 PS 端和 PL 端两部分，PS 端主要用于实现数据从外存的搬运和计算逻辑的控制，而 PL 端的 FPGA 部分则作为计算模块，通过 RTL 语言实现细粒度的优化。由于网络运算中都是基于循环实现的，因此通过数据重排，循环展开，循环并行的方式在计算模块中提高计算效率。对于片上存储受限的问题，则通过数据分块的方式，平衡计算效率和 BRAM 的使用。最后形成流水线的工作模式，通过优化硬件的设计，减少流水线的空等时间，进一步提高效率。PL 端的硬件架构如下图4-1所示，该架构对应图3-1中的加速器部分，其中的缓冲区部分分别对应图3-1中的输出缓冲与输入缓冲，流水线其余的运算模块则共同构成图3-1中的处理模块。如图4-1所示，运算时的数据通路由多路选择器进行控制，多路选择器的控制位则是来自前述章节中所介绍的存储在寄存器中的命令数据。各个模块之间通过流水的方式进行运算，提高运算效率。本章将分章节介绍优化思路和相应的硬件架构。

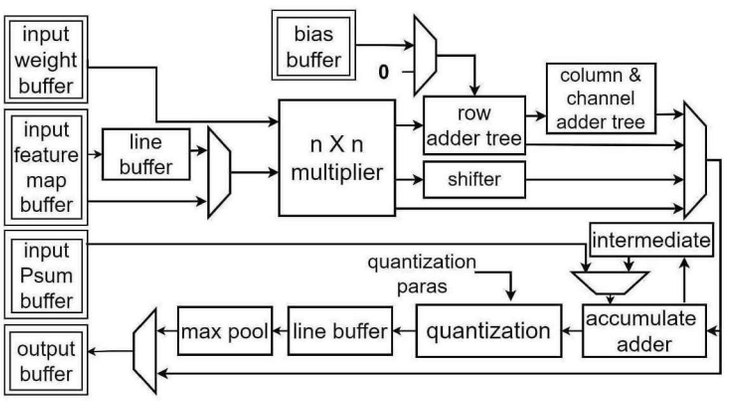


图 4-1 PL 端架构

4.1 循环展开与循环并行计算

对于 CNN 来说，其输出结果中一个元素的计算可以由以下算法4-1的伪代码进行表示。

算法 4-1 卷积运算伪代码

```
1 for (int oc = 0; oc < KN; oc++)//output channel
2 for (int ic = 0; ic < IC; ic++)//input channel
3 for (int ow = 0; ow < OW; ow++)//output window
4 for (int oh = 0; oh < OH; oh++)//output window
5 for (int kw = 0; kw < KW; kw++)//input window
6 for (int kh = 0; kh < KH; kh++)//input window
7 output[oc][ow][oh] += input[ic][ow+kw][oh+kh] * kernel[oc][ic][kw][kh];
```

可以看到朴素的实现实际上主要是基于四层循环的嵌套计算。伪代码中各参数描述如表4-1所示。

表 4-1 卷积计算循环参数描述

数据	描述
KN	输出特征图通道数/卷积核个数
IC	输入特征图输入通道数
OW/OH	输出特征图行和列
KW/KH	输入特征图行和列

因此对于 CNN 的计算而言，其加速主要是通过对于如上所展示的四种循环做相应的优化来实现。首先是输出通道对应的循环，不同输出通道中的输出结果会共用输入特征图的值，因此对于这个循环的并行主要是可以同时与不同卷积核中的权重值并行做卷积运算；其次对于输出窗口和输入窗口的循环，两个循环的优化思路都是基于循环展开，展开循环的累加过程，即同一时间完成足够多的乘法操作，然后进行求和，再进行下一次乘累加，这样可以减少循环执行的次数，减少得到输出的时间开销；最后，对于输入通道的循环，可以将不同输入通道的输入特征图与对应通道的卷积窗口内的值并行计算，从而提高计算效率。

同样，对于 LSTM 涉及到的计算，其中对于 C_t 和 h_t 最终输出的计算是矩阵的按位乘法，所做的只能是尽可能在一个时刻内完成较多的乘法操作从而计算出所需结果；其计算优化时关注的点主要在于每个门中矩阵与列向量的运算，结果中一个元素的计算可以由算法4-2的伪代码进行表示。不难看出其加速的主要思路仍然是基于循环的并行与展开：一是在于列向量中同一位置的元素可以与矩阵中同一列不同行位置的元素进行并行计算，这样可以在一定时间内得到多个有效的

输出结果，最终实现加速计算的效果；另外一方面是在于展开矩阵和列向量中参与计算的元素，同一时刻并行地对展开的元素执行乘法操作，再进行求和，继而进行下一次的乘累加，以此减少循环的次数，更快地计算得到最终的输出。

算法 4-2 矩阵与向量乘法的伪代码

```
1 for (int row = 0; row < ROW; row++)
2 for (int col = 0; col < COL; col++)
3 out[row] += w[row][col] * x[col];
```

基于以上的加速思路，因此提出了本文中所采用的硬件架构，该架构相较于传统的加速器设计能支持更多维度循环展开的运算[50,51]，从循环展开和循环并行的角度去对两种网络进行相应的加速。

首先是最基本的计算单元，在 Zynq Ultrascale+ 系列开发板上最基本的计算资源是 DSP 资源，由 DSP Slice 构成。DSP 资源使用了许多二进制乘法器和累加器，通过 FPGA 的可编程逻辑可以实现定制的、完全并行的算法。而在 Zynq UltraScale+ SoC 上具有许多专用的低功耗 DSP 资源，将高速与小尺寸相结合，同时保留了系统设计的灵活性。该开发板上的 DSP Slice 是通过 DSP48E2 的原语来进行配置和定义的，可以实现成具有不同功能的计算资源。

DSP48E2 的基本架构如图4-2(a)所示，DSP48E2 由一个 27 位前置加法器、27*18 乘法器和一个灵活的 48 位 ALU 组成，ALU 用作后加法器/减法器、累加器或逻辑单元。A 或 B 可以选择为预加法器输入，从而实现更宽位数的数据作为乘数进行乘法操作。预加法器的结果可以发送到乘法器的两个输入端，以提供平方能力。出于节约 DSP 资源的考虑，因此首先应该通过 DSP 资源最大化 8 位定点的运算。片上的 DSP48E2 资源可以支持 27*18 位的乘法运算，且该 DSP 资源本身提供了预加器和累加器等功能。为了兼容两种网络涉及的乘累加、按位乘等多种运算模式，又由于两种网络的绝大部分输入都采用了 8 位定点的量化，因此决定利用 DSP 本身提供的预加器实现一个 DSP 计算两组 8 位定点乘法的方案。该方案中输入数据 A 和 B 共用同一个乘数 C，分别需要与其进行乘法计算。如上图4-2(b)所示，将其中一个 8 位定点数 A 的最高位从 27 位乘数的最高位放置，而将另一个 8 位定点数 B 的最低位从 27 位乘数的最低位开始放置，对应 DSP48E2 基本架构图4-2(a)中的输入口 A 和输入口 D。通过预加器相加后，输入数据将构成一个高 8 位和低 8 位为有效数据而其余位数为 0 的乘数，此时通过寄存器后与另一个乘数 C 进行相乘。最终输出的位宽为 48 位的结果中，AC 的结果在第 33 位到第 18 位之间，AB 的结果在第 15 位到第 0 位之间，产生的结果中 AC 与 BC 得到的结果不

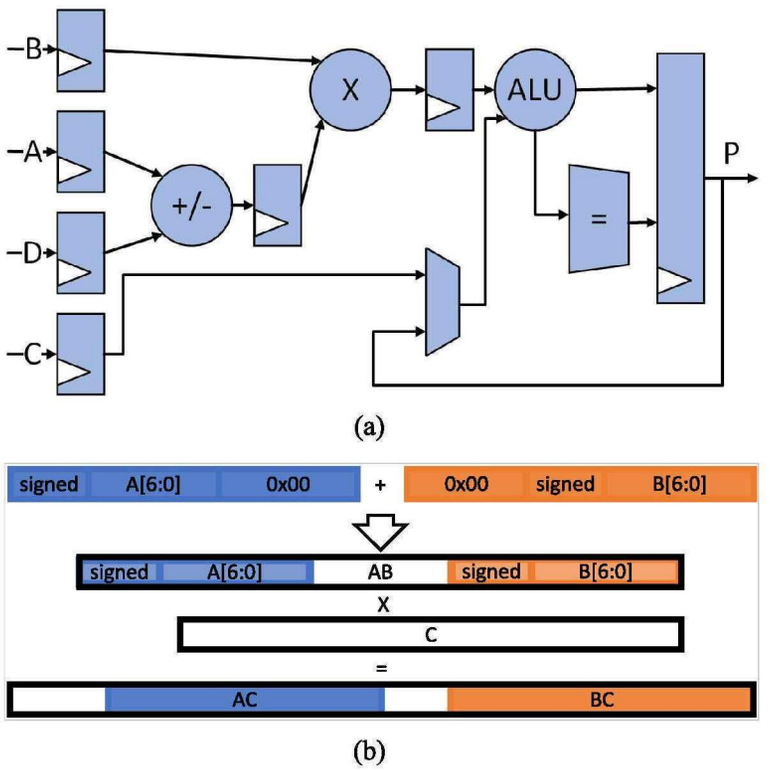


图 4-2 (a)DSP 基本架构图；(b)DSP48E2 的定点乘法。

会产生冲突，从而实现对 DSP 资源的高效利用。

基于上图中的 DSP48E2 的硬件特性和 8 位定点量化的特点，一个 DSP 资源可以同时计算的数据需要有相同的乘数，该优化的思路同样遵循了本文之前所阐述的循环并行的思路。选择对应的循环并行时，对于 CNN 的运算来说即对应输出通道的并行计算，同一个输入特征图的值会与不同的卷积核中的权重相乘进行计算；对于 LSTM 的运算来说，则是输入特征列向量中的一个元素会与权重矩阵中的多个元素进行相乘，产生输出的列向量中不同行的元素，这部分运算可以进行并行计算。如图4-3(a)所示。

由于本硬件设计中基于 DSP48E2 的计算单元只实现了乘法功能，因此为了通过乘累加得到最终结果，本文的硬件系统需要构建加法树将各个乘法单元计算出的结果进行累加。例如现在多数卷积核采用的 3*3 的卷积窗口，输出的 9 个数需要进行相加才作为最终的输出结果。构建的加法树结构如图4-3(b)所示，是一个类二叉树的结构，每个加法器有两个输入，逐级上升两两相加，得到最终的累加结果，对于奇数个加数相加的情况则将第奇数个加数经过 log_2N-1 级寄存器的延时后输入到最后一级加法器中进行相加。

为了支持对于涉及到输入窗口和输出窗口的循环进行展开和并行的优化方案，本硬件系统决定采用行缓冲的结构对计算进行加速。如图 5 所示，通过行缓冲构

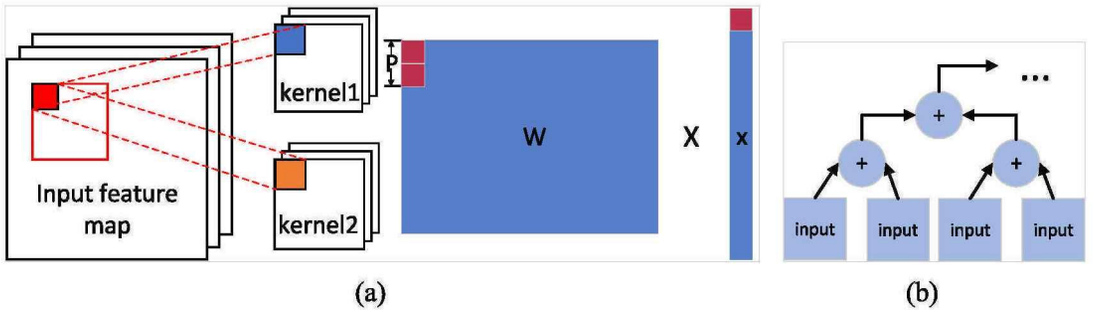


图 4-3 (a) 基于 DSP48E2 实现的并行；(b) 加法树。

建滑动窗口的方式，对于不同大小的输入特征图，通过参数配置的方式，由多路选择器选择实际使用的行缓冲的深度来进行相应的适配。可以看到这种基于滑动窗口的设计模式，展开了输入窗口内的所有元素，如输入特征图大小为 N*N，卷积核大小为 K*K 的卷积运算，在经过 N*K 个时钟周期的填充以后，每个周期都能有一个有效的输出。如图4-4所示是一个 6*6 大小的滑动窗口的有效数据流的例子，通过行缓冲中数据的移位在每个周期形成一个有效的计算窗口。

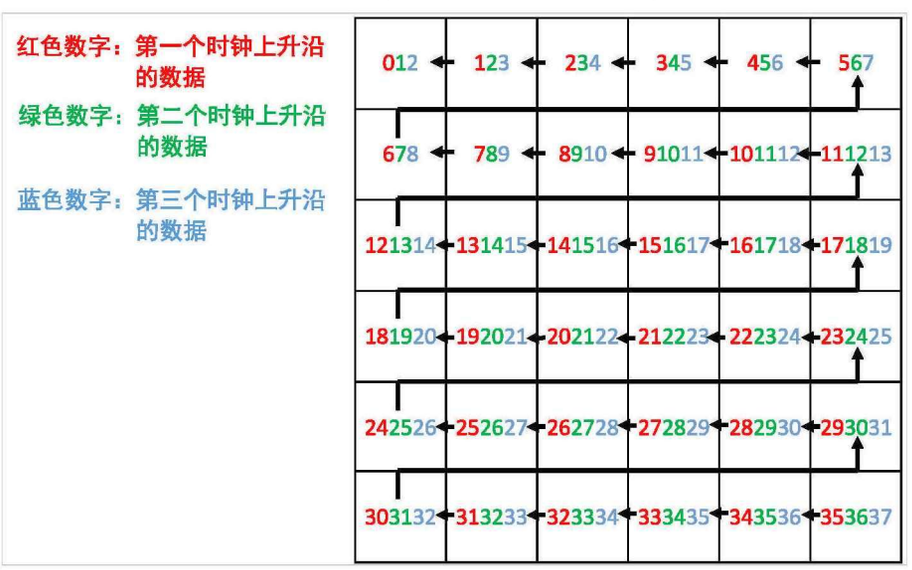


图 4-4 滑动窗口有效数据流

而且实际上，本文当前的设计中，每一个行缓冲对应的输出是独立并行的，因此当出现如图例4-5中展示的情况时，即对于大小为 L 的计算阵列，大小为 K 的卷积核输入窗口，其优点主要体现在当计算模块阵列大小大于单个卷积核计算输入窗口时，即 L>K 时，本文当前的设计在展开并行输入窗口中数据的同时还可以充分地让不同输出窗口间的循环实现数据复用，并行地得到输出。不过此时，为了实现并行，采用的输出必须来自输入数据流，实际上这时没有进行从上一个行缓冲到下一个行缓冲之间的数据流动，行缓冲 1 先流入第 1 行的数据，随后会流入

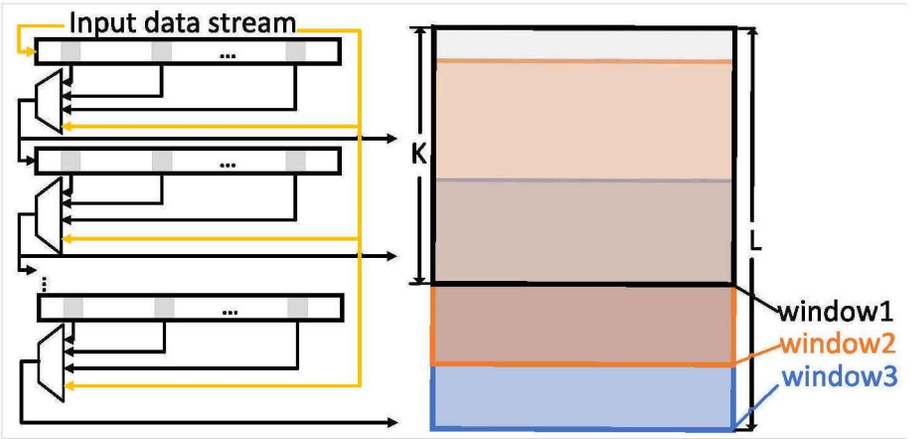


图 4-5 窗口间并行

在本例中，左侧有若干个行缓冲，右侧有三个计算窗口，计算窗口内并行的进行乘法运算，每个周期输出结果。每两行缓冲器之间有一个多路复用器，用于选择当前行缓冲的工作深度，将不同长度的数据作为下一行缓冲的输入。在本例中，三个计算窗口对重叠部分的数据实现了复用，并且并行计算了三个输出窗口中的结果。

第 L+1 行行缓冲。采用基于行缓冲结构的设计，可以通过数据的流动获取到多个输出窗口的数据，而且有效地利用了不同窗口间重叠的数据，提高了数据的利用率，从而实现对于这两个循环中数据的循环展开，提高计算效率。而当计算阵列的大小等于或小于单个卷积核计算输入窗口时，即 L≤K 时，此时则通过行缓冲结构建滑动窗口，此时当前行的行缓冲的数据来自上一行的行缓冲的流出，当前行缓冲的数据同样会流出进入到下一行的行缓冲中。通过这样形成的窗口仍然能够实现对于输入窗口数据的部分展开。因此，从循环展开对运算加速的角度来看，当前设计无论是通过对多个窗口的全展开还是对单个窗口的部分展开都能提高循环累加操作的效率。

对于 LSTM 中涉及到的循环计算的展开操作，由于其计算窗口间的数据不会出现重叠的情况，因此将不适用于行缓冲形成滑动窗口的模式，此时本系统中的多路选择器固定选择行缓冲的输入为输入数据流。这种计算模式下，行缓冲的工作模式等同于通过多组移位寄存器形成匹配计算阵列所需的数据窗口。如图4-6所示，这种设计决定了是通过移位寄存器的数量来弥补流水后数据流可能产生的空缺，否则就会产生计算模块等待数据加载的空等。由于多路选择器此时已经固定，目前对于 LSTM 计算中矩阵阵列和列向量的计算只支持通过固定大小的窗口进行循环计算中的展开，并且需要使 CNN 输入特征图的最大宽度尽可能接近 LSTM 中

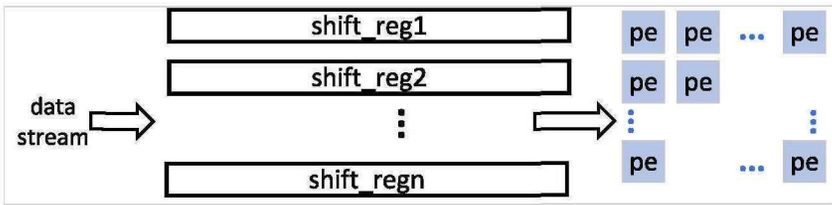


图 4-6 LSTM 中的循环展开

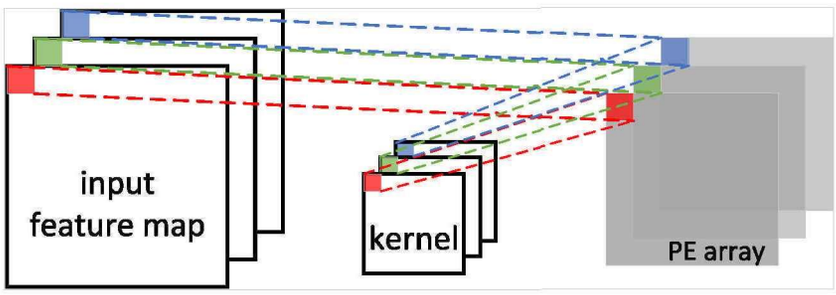


图 4-7 输入通道并行计算

的矩阵宽度。理论上一个计算窗口的大小需要对应相同数量的行缓冲，这对资源是很大的占用，因此为了权衡流水后的吞吐率和存储资源，本文的硬件架构中将用于计算的阵列大小限制在一个较小的值。

最后则是对于 CNN 计算中输入通道的并行，主要是通过多个大小相同的计算阵列来实现，如下图4-7所示，一个计算阵列对应计算一个输入通道的输入特征图和其卷积核计算的结果，同一时刻并行地对不同输入通道中的输入元素和卷积核权重元素进行乘累加操作，最终将多个阵列中的输出结果通过加法树相加从而得到最终输出的部分结果或是最终结果。

4.2 数据分块

由于片上用于生成片上存储的 BRAM 和 LUT 资源都十分有限，且 LUT 不仅仅是生成存储时所需要的资源。因此，对于大多数大规模网络的输入特征图来说，都无法直接通过片上存储进行完整的存储。当前架构对于数据的读取正如第三章中本文阐述的那样，通过 ARM 核将数据从外存 SD 卡上将数据读取到 PS 端的 DDR 中，随后通过总线协议传输到 PL 端。硬件模块读取片上存储的时延明显是小于通过总线协议将数据从 PS 端 DDR 发送到 PL 端的，所以将参与计算的数据放在片上存储中可以显著提高计算效率。因此需要利用本文提出的数据分块方案，将数据分为小块，每个小块都能够存储在本硬件系统的片上存储之中。同时，该数据分块方案能够尽可能减少从 PS 端 DDR 读取重复数据的次数。基于以上的硬件架构设计，本文提出了以下的数据分块方案。

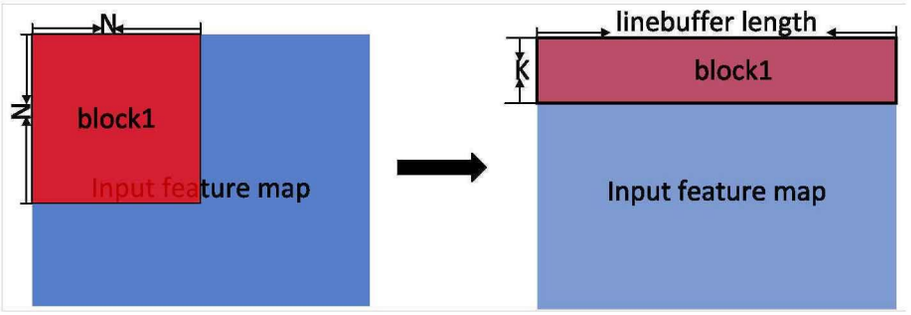


图 4-8 对于 CNN 的数据分块

对于 CNN 的输入特征图来说，由于采用行缓冲形成滑动窗口的方式进行卷积运算，并且行缓冲可以通过参数配置的方式支持不同大小的输入特征图。因此不同于传统的对于输入特征图的固定大小的分块方案[51,52]，本文提出了特定的分块方案，按照卷积核宽度对输入特征图进行矩形的分块。以下图4-8中的一个通道的输入特征图为例。输入特征图按照高度为 K，即卷积核的宽度，宽度为特征图宽度的方式进行分块。目前的神经网络中经常通过 padding 的方式保持边界信息，使输入数据和输出数据的维度保持一致。而采用当前的分块方案使得本系统对输入特征图进行填零的操作变得简单，无需实现专门的硬件模块来进行填零操作。当前分块的方案使得加速系统在读取输入特征图数据时的地址连续，硬件模块无需维护计数器维护相应的状态机等操作，在 PS 端读取数据时即可在通过总线上传输时自动在首尾位置填入零即可。对于 LSTM 的输入来说，由于其输入所用的列向量与此前时刻的输出有关，即 $X=[h_{t-1},x_t]$，如果继续采用 CNN 对于输入特征图的分块方案，那么计算模块会有很长的空等时间用于等待 h_{t-1} 的输出到来，然后才能重新开始新的计算。这样的延迟显然会严重的影响流水的吞吐率，拖累计算效率。因此，不能沿用之前的矩形分块思路，而是提出下图4-9这样的分块方案，同时调整列向量和输入矩阵的数据排布为 $W=[W_x,W_h],X=[h_{t-1},x_t]$。其中分块后的宽度 P，体现的是之前所述的 LSTM 中矩阵与列向量相乘时的循环并行方案。而下一个分块选择相同列位置不同行的块则是为了能够重复利用相同的输入列向量，减少数据的 IO 次数。下面将介绍下前文中未提及的其他重要的模块，所有模块未使用官方所给的 IP 核。因此在设计中考虑了流水线的优化，并且可以通过参数配置的方式动态地选择用于综合实现该模块的资源，最终在实验中体现出本文设计具有灵活性，以及各级流水组合逻辑延时低的优势。

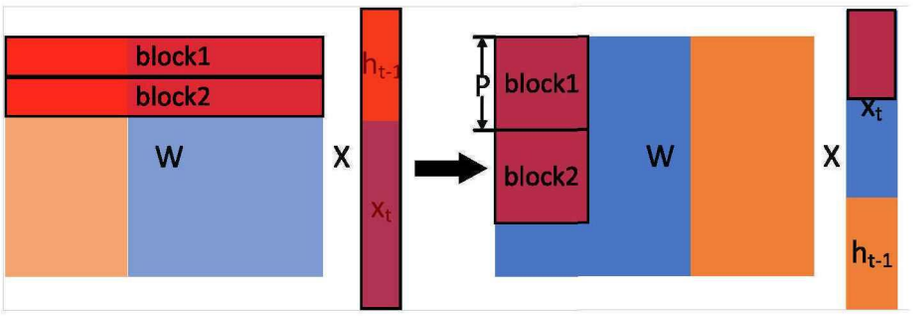


图 4-9 对于 LSTM 的数据分块

4.3 流水线中的重要模块

4.3.1 片上存储

在整体硬件架构中，来自 PS 端的数据会存放在片上存储当中，而所有数据存储所用到的片上存储结构都是基于双口 BRAM 实现的乒乓缓冲结构，它使用两个缓冲区交替存储数据，当一个缓冲区被填满或读取完毕时，数据传输将自动切换到另一个缓冲区。这种技术通常用于解决在读写数据的过程中可能发生的竞争和冲突问题，从而提高数据传输的可靠性和效率。其外部接口如图4-10所示。

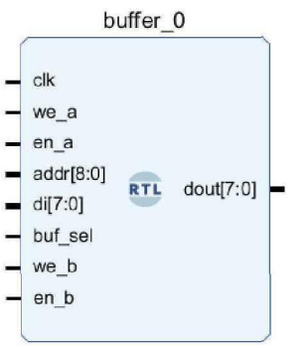


图 4-10 片上存储模块

该模块的输入输出功能如下表4-2所示，其中的使能信号和用于读写进行使能的信号与一般的双口 BRAM 是相同的，而乒乓缓冲的实现主要是通过 buf_sel 信号实现的，本文的设计中人为将一个双口 BRAM 的存储空间一分为二，一半的空间用于读数据，另一半用于存储写入的数据。

表 4-2 片上存储模块信号

信号	描述
clk	时钟
we_a	A 口写使能
en_a	A 口使能
addr	读写地址
di	写入的数据
buf_sel	乒乓缓冲选择信号
we_b	B 口写使能
en_b	B 口使能
dout	读出的数据

该模块可进行配置的参数如下表4-3所示，可以看到数据的位宽，地址的位宽都是可以进行配置的，而最后一个可配置的参数则体现了本设计使用 HDL 语言的优势，可以通过相应参数的配置合理的指导资源的实现方式，具有很强的灵活性。比如当 RAM 较小时，分布式 RAM 在功耗和速度上更有优势；当 LUT 利用率很高时，如果 BRAM 资源利用率不高，可以把分布式 RAM 转换为 BRAM，从而释放出一部分 LUT 资源。

表 4-3 片上存储模块可配置的参数

参数	描述
WIDTH	配置数据的位宽
ADDR_BIT	配置地址的位宽
DEPTH	配置存储所用的深度
RAM_STYLE_VAL	配置合成 RAM 的实现方式

由于地址的位数是可以通过参数配置的，因此下图4-11中以 9 位的地址空间为例，此时传入的 addr 参数为 8 位宽的数据。buf_sel 取 0 时，读地址读取的地址为 {0,addra}，写地址写入的地址为 {1,addrb}；buf_sel 取 1 时，读地址读取的地址为 {1,addra}，写地址写入的地址为 {0,addrb}，这样读写不会发生竞争和冲突，写操作无需等待读操作。通过乒乓缓冲的结构可以提高传输数据的效率，减少当前流水线空等数据的时间。

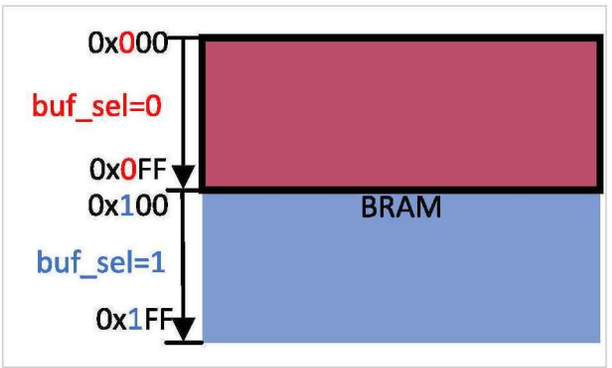


图 4-11 buf_sel 控制信号

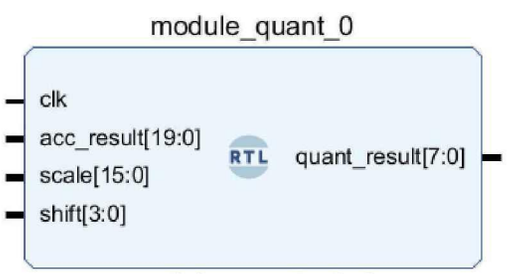


图 4-12 量化模块

4.3.2 量化模块

在前文中提到的量化中，乘累加操作完成后都会需要进一步和与伸缩因子相关的参数 $s_{input} \times s_{weight}/s_{outputput}$ 进行相乘，这个浮点参数可以转化为与整数相乘和右移操作，即转化为 $2^{-n} \times scale$ 的操作。不过针对前文的现有方案，考虑到对于硬件实现上的优化，本文仍然可以选择做出进一步的优化。如式 (2-5) 所示，由于本层的输出 q_{out} 在输出时会和零点进行一次相加操作 $q_{out} = q + Z$，而当需要获得下一层的输入 q_{in} 时，又要进行一次与零点的相减操作 $q_{in} = q_{out} - Z$。在推理过程中可以选择省略输出时与零点相加的操作，这样下一层的输入可以省略一次减去零点的减法操作。这样的设计优化了卷积神经网络推理的过程，省去了一次加法操作和一次减法操作。与之类似的是，在式 (2-5) 所示的 ReLU 计算过程，可以完全等价于式 (4-1)。因此 ReLU 模块的输入同样可以减少一次与零点相加的操作 $q_{out} = Z_{out} + X$。从硬件设计的角度来看这样的优化，减少了设计中的三级流水，对于流水线的运行来说提高了运行效率和吞吐率。

$$q_{out} = \begin{cases} X & X \geq 0 \\ 0 & X < 0 \end{cases} \tag{4-1}$$

该模块如下图4-12所示。 该模块没有可配置的参数，模块中的输入输出信号功能如下表4-4所示。

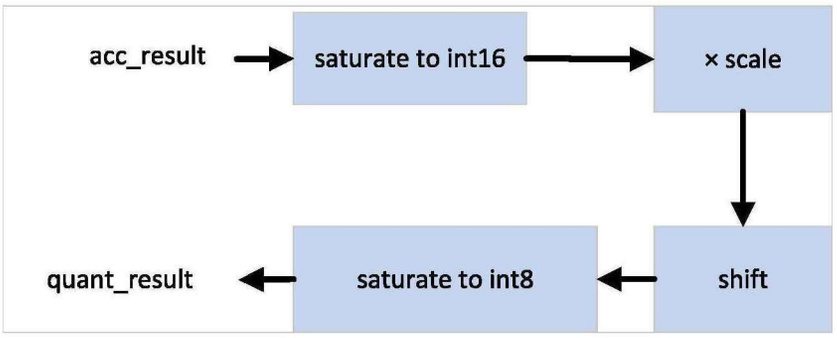


图 4-13 量化模块流水线

表 4-4 量化模块信号

信号	描述
clk	时钟
acc_result	前述模块得到的累加结果
scale	进行相乘的整数
shift	右移的位数
quant_result	输出的量化结果

图4-12中输入数据 acc_result 的位宽体现了本文硬件架构中对于累加操作的优化，由于神经网络中累加计算的累加深度不会太深，因此无需使用 32 位的位宽来存储累加的结果，实际上 20 位的位宽就足够表示[47]。在该模块中 acc_result 首先第一级是截断为 16 位的数据，第二级是与 scale 参数相乘，第三级根据 shift 参数进行右移，最终再次截断为 8 位的输出数据，如图4-13所示。截断操作并不能直接粗暴地只保留低位宽的数据，否则会造成较大的误差。算法4-3以 20 位位宽数据截断为 16 位为例展示量化模块中的截断操作。首先需要判断符号位，对于使 16 位有符号数溢出的正数，将其转换为 16 位有符号数所能表示的最大正数 0x7fff；对于使 16 位有符号数溢出的负数，将其转换为 16 位有符号数所能表示的最小负数 0x8000；其他可由 16 位有符号数表示的数据则简单地截断即可。

算法 4-3 截断操作

Input: 输入信号为 20 位位宽的 [19:0]acc_result

Output: 输出信号为 16 位位宽的 [15:0]out

```
1 if acc_result 符号位为 0 并且第 18 到 15 位不全为 0 then
2 |   out = 0111111111111111;
3 else if acc_result 符号位为 1 并且第 18 到 15 位不全为 1 then
4 |   out = 1000000000000000;
5 end
6 else  out = {acc_result[19] , acc_result[14:0] } ;
```

4.3.3 池化模块

池化是卷积神经网络中常用的一种下采样方法，目的是减少特征图的大小并保留重要信息。在硬件的实现中使用最大池化计算量较少并且可以比较好的节省硬件资源。一个 2*2 大小的池化窗口和步长为 2 的最大池化的步骤如图4-14所示。

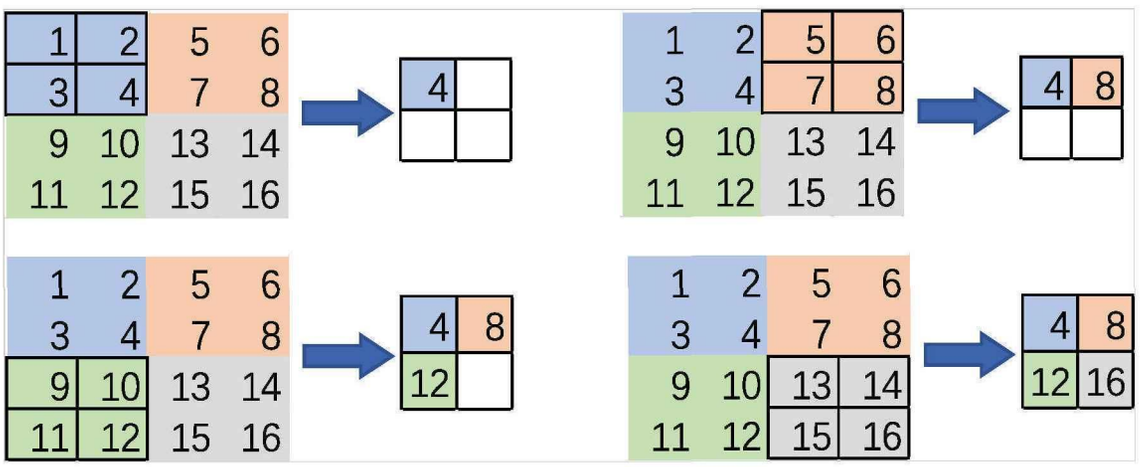


图 4-14 最大池化实例

行缓冲的实现是基于深度可配置的移位寄存器进行实现的，移位寄存器的逻辑较为简单，接口如图4-15所示，该模块的输入输出信号如下表4-5所示。

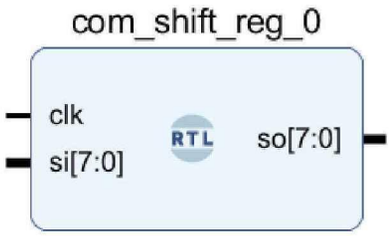


图 4-15 移位寄存器模块

表 4-5 移位寄存器输入输出信号

信号	描述
clk	时钟信号
si	输入数据
so	输出数据

该模块可进行配置的参数如下表4-6所示，可以看到存储的深度可以进行配置，输入数据 si 存储在移位寄存器的最低地址处，随着时钟周期不断移动，最终在最高地址处被读出。最后一个参数用于指定使用何种资源来实现移位寄存器，根据片上资源的使用情况，可以自由的选择使用 FF,BRAM 和 LUT 资源来进行移位寄存器的实现。

表 4-6 移位寄存器可配置的参数

参数	描述
DEPTH	移位寄存器存储深度
WIDTH	输入数据的位宽
SRL_STYLE_VAL	指定实现移位寄存器的资源

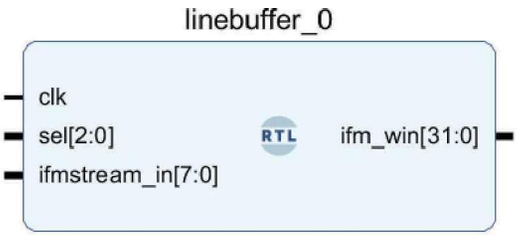


图 4-16 行缓冲模块

表 4-7 移位寄存器输入输出信号

信号	描述
clk	时钟信号
sel	内部数据流选择信号
ifmstream_in	输入数据
ifm_win	输出的窗口数据

行缓冲接口如图4-16所示，该模块的输入输出信号如表4-7所示。一个形成 2*2 池化窗口的行缓冲如下图4-17所示，其中通过 sel 信号选择内部的多路选择器适配不同的特征图数据大小，四个输出寄存器两两一组形成移位组成 2*2 的窗口，最终将窗口的数据输出。

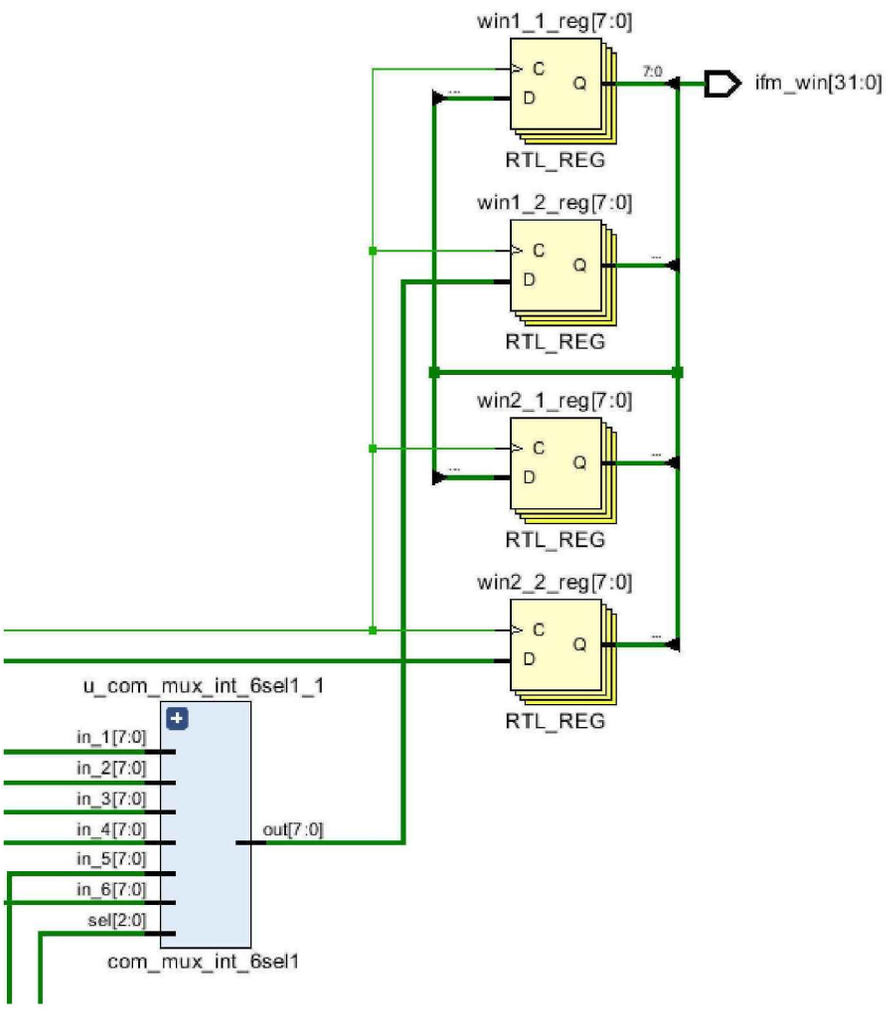


图 4-17 行缓冲形成 2*2 池化窗口

在本文的硬件架构中，通过行缓冲的结构形成 2*2 的池化窗口作为准备数据，

送入到池化模块进行最大池化的操作，接口如图4-18所示，该模块的输入输出信号如下表4-8所示。

图 4-18 最大池化模块

表 4-8 最大池化模块输入输出信号

信号	描述
clk	时钟信号
ifm_win2x2	2*2 窗口数据
ofm_stream	输出数据

该模块将输入的窗口数据通过比较器进行比较，选出当前窗口最大的数据，并将此数据作为输出。

4.4 本章小结

本章主要介绍了 PL 端的设计与架构，由于 PL 端上的存储资源和计算资源都比较有限，因此优化一方面针对计算效率，一方面同样针对资源的利用。首先介绍了如何通过循环展开与并行的方式对神经网络中最主要的计算密集型计算进行加速，同时介绍了如何提高计算资源的利用率。其次介绍了针对设计好的计算模块本硬件加速系统所提出的适配的数据分块方案，尽可能提高数据的复用率，减少通过总线传输重复数据的次数。最后，对于当前架构所形成的流水线进行了介绍，并对其中前文未详细介绍但较为重要的模块进行了介绍。

第五章 实验与分析

在前述两个章节中分别介绍了本文设计的硬件加速器架构的两个部分。基于前文所介绍的硬件系统，本章在 ZCU102 开发板上部署了相应的神经网络，并对其开展了分析、功能性验证与性能测试。

5.1 硬件实验环境准备

本文基于 ZCU102 开发板进行硬件加速器的部署和验证工作，该开发板属于 Zynq Ultrascale+ 系列，属于 Xilinx 推出的高性能开发板，如图5-1所示，该系列的开发板都配备 64 位的处理器作为处理系统的 PS 端，以及具有丰富资源的可编程逻辑，通过丰富的外设接口和总线提供与外界通信的方式。ZCU102 开发板采用了 XCZU9EG-2FFVB1156E 芯片，该开发板的架构示意图如图5-2所示，可以看到不仅有多核的处理器，还有丰富的 PL 端和 PS 端之间通信的接口，有不同功耗不同性能的接口可满足两个部分之间的通信。同时，开发板上还有 4 个 DDR4 SDRAM 总计 32Gb，以及以太网、USB 等高速接口，还有 SD 卡槽、PMOD 等接口扩展其他外设以满足不同的应用需求。该开发板提供了丰富的资源，如图5-1所示。

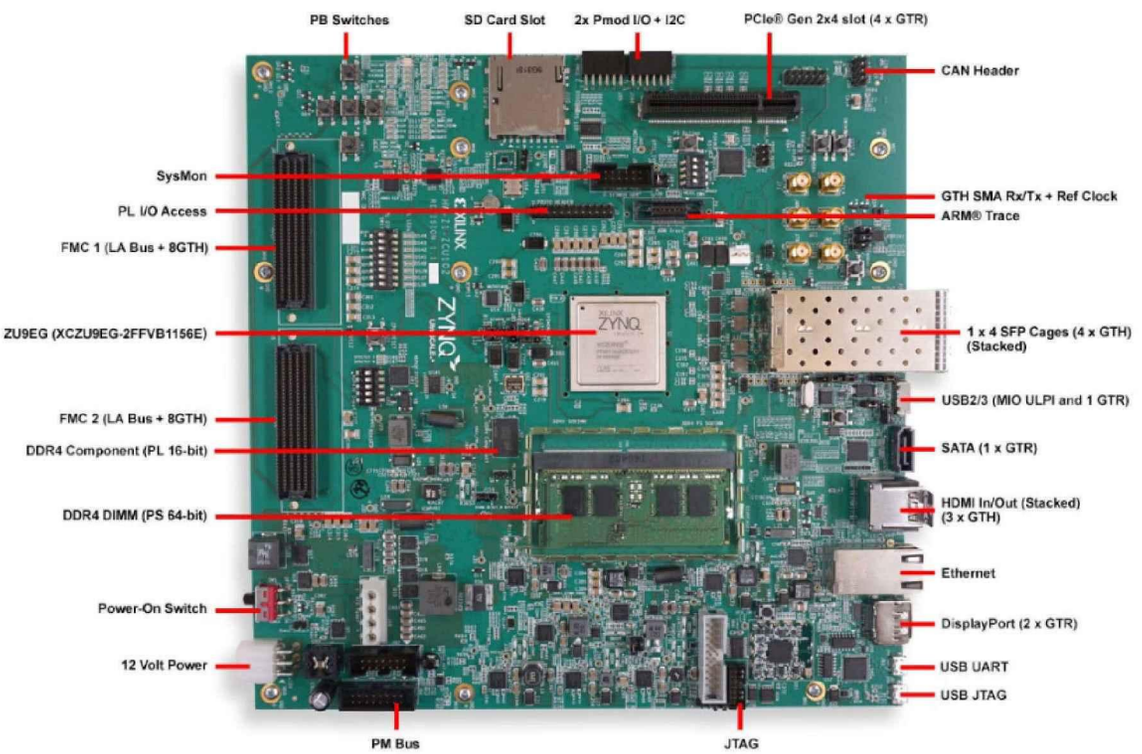


图 5-1 ZCU102 资源分布图

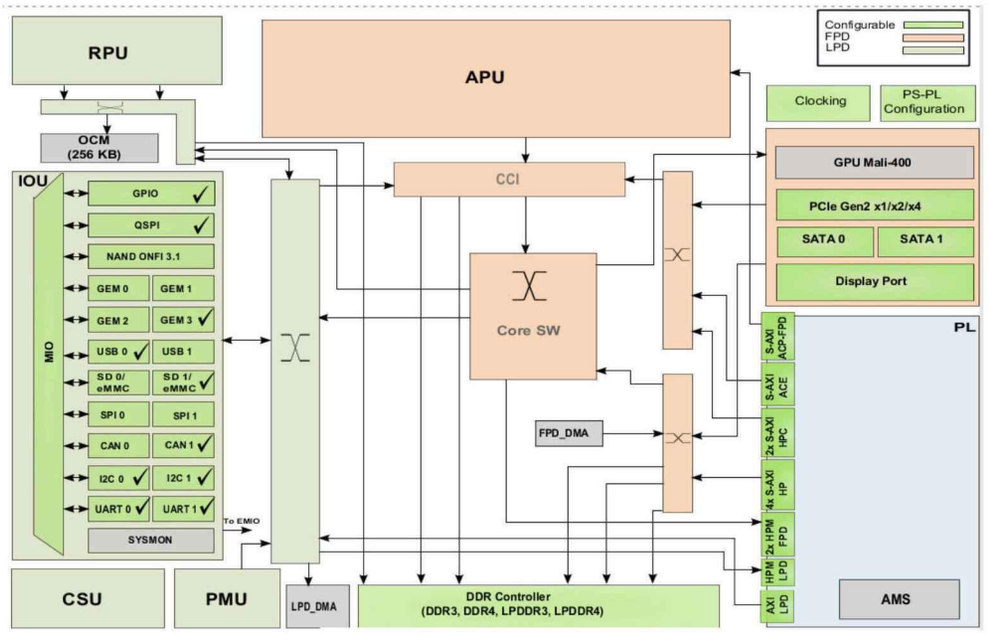


图 5-2 ZCU102 架构图

表 5-1 ZCU102 资源

Resource	
系统逻辑单元 (K)	600
内存 (Mb)	32.1
DSP Slice	2520
最大 I/O 引脚	328

Zynq Ultrascale+ 作为异构的嵌入式平台，可以将其的设计部分分为 PL 端的设计和 PS 端的设计。基于 Xilinx 官方提供的工具链，首先应该使用 Vivado 进行设计，完成对于 Zynq Ultrascale+ 处理器系统 IP 核进行搭建，主要包括 PL 与 PS 之间通信接口的配置，DDR 的使用，中断使能以及输出的时钟频率等。配置好的 Zynq Ultrascale+ 系统 IP 核如图5-3所示。

图 5-3 Zynq Ultrascale+ 系统 IP 核配置

随后通过使用“Create and Package New IP”按钮来将之前设计好的硬件模块封装为新的 IP 核。在创建新的 IP 核之前，需要进行必要的设置，例如确定 IP 核的名称、时序等。之后引入之前设计好的 HDL 文件，通过使用“Package IP”按钮将其打包成 Vivado 支持的格式。在打包过程中，需要为 IP 核指定基本信息，例如名称、供应商、版本号、描述等。此时可以在“IP Catalog”中看到已经打包好的 IP 核，将封装好的加速器 IP 核添加到 block design 界面中进行相应的端口连接，连接完成后验证连接的合理性和正确性，并根据具体应用需要对连接后的 IP 核进行参数配置。配置完成后的系统整体 block design 框图如下图5-4所示。

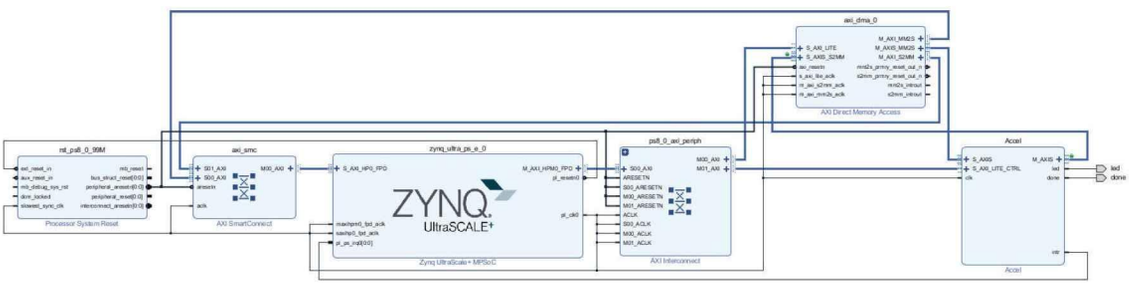


图 5-4 系统 block design 框图

在设计好的硬件模块中，本设计留出了两个接口一个接口 done 用于通过 LED 显示当前的全部计算已经完全结束，其会在最后的计算结束后拉高，点亮 LED；还有一个接口 led，用于简单地帮助上板验证时通过肉眼判断上板是否成功，该接口的目标是驱动一个 LED 并使其随着一定频率进行闪烁。因此，需要对其进行相应的管脚分配。本设计采用的方案是通过约束文件的方式，进行引脚的分配。在左侧工程面板中，右键点击 Constraints，选择 Add Sources 添加约束文件，因为 done 在结束时拉高，因此选择两个高有效的 LED 引脚，约束文件分配的管脚如表5-2所示。

表 5-2 管脚分配

信号名称	管脚	板上参考代号
led	AG14	DS38
done	AE13	DS39

接下来在左侧工程面板中依次执行综合和实现；之后选择 Generate Bitstream，生成比特流文件；然后再在左侧菜单中选择 File，选择 Export Hardware，导出.xsa 格式的硬件描述文件。至此，硬件部分的实验环境准备完毕。

5.2 SoC 部分环境准备

由于使用的是 2020.1 版本的 Vivado，Xilinx 在这个版本的工具链当中使用的是 Vitis，不再支持旧版本的 SDK。因此关于处理系统的部分通过 Vitis 进行准备。打开 Xilinx Vitis 2020.1，基于前一节中的硬件描述文件，创建相应的 Platform Project。配置平台设置后 Vitis 中便提供了 Zynq Ultrascale+ 的相关驱动程序和库文件，可以基于此进行处理器系统部分的代码编写。

输入数据需要进行预处理，其中 CNN 的选择了 VGG-16，它有 13 层卷积层、5 层池化层、3 层完全连接层；LSTM 模型中隐藏单元的数量为 256，LSTM 中每个门的权重矩阵为 1256*256。采用前述章节中介绍的定点量化方案对 VGG-16 和 LSTM-256 进行了定点量化。VGG-16 通过 ISLVRC2012 数据集训练，采用 tensorflow 官方接口进行量化；LSTM-256 通过 AISHELL-2 数据集训练，并在模型定义的前向推理中自定义量化。如表5-3所示，VGG-16 采用浮点推理准确率是 97.26%，采用定点量化后的准确率为 96.99%，损失的精度仅为 0.27%。LSTM-256 采用浮点推理准确率是 68.57%，采用定点量化后的准确率为 68.06%，损失的精度仅为 0.51%。量化后的性能损失可控，能够部署在嵌入式平台上。

表 5-3 推理结果

	浮点推理准确率 (%)	定点量化准确率 (%)	差值 (%)
VGG-16	97.26	96.99	-0.27
LSTM-256	68.57	68.06	-0.51

所需的输入数据按照不同的文件存放在外存 SD 卡中，SD 卡读取和数据的传输前述章节中已经有过介绍，接下来介绍一下功能性和时间性能的测试方案。本文预先通过 cpp 实现了相同的测试网络结构，并将相同输入数据经过网络计算后的正确结果作为一个文件存放在 SD 卡当中，当网络最终的输出从 PL 端传输到 PS 端后，通过代码将比对存储结果的 DDR 空间中的数据和正确结果的数据，如果数据相同则通过 UART 接口打印出正确的反馈，如算法5-1所示。

算法 5-1 功能性验证

```
while done == 0 do
    空等;
end
done = 1;
打开 SD 卡中的存储正确结果的文件;
从 DDR 中相应的地址中读取数据;
while 指针!= EOF do
    移动文件指针，增加 DDR 的地址，比较数据是否相同。
    if 数据不相同 then
        xil_printf(”result wrong”);
end
xil_printf(”result right”);
```

时间性能的测试则是编写代码，指定系统在 UART 接收到键盘的输入后再开始使能运行，此时记录下当前的时间，在接收到最终结束的中断信号后再次记录时间，两个时间的差值即为运算的时间性能，将时间的值通过 UART 接口打印出来，如算法5-2所示。而其他性能相关的参数，如功耗、时序等参数的情况则根据 Vivado 综合实现后的报告进行分析。

算法 5-2 时间性能验证

```
//等待键盘输入
scanf();
//获取初始时间
XTime_GetTime(&T_start);
使能，传输数据;
while done == 0 do
    空等;
end
done = 1;
//获取结束时间
XTime_GetTime(&T_end);
xil_printf(T_end - T_start);
```

5.3 实验结果

5.3.1 资源消耗分析

本文中的硬件加速系统所消耗的资源如图5-5所示，各项资源更具体的数据如下表5-4所示，大多数资源的占用都不大，只占了片上资源的 10% 左右。由于本文的设计中有大量的移位寄存器的使用以及加减移位的操作需要，因此 LUT 的资源占用较为巨大，占比超过了可用 LUT 资源的 30% 为 84128 个。而在移位寄存器和寄存器的合成中同样消耗了 FF 资源，消耗了 40099 个。本文硬件设计中所有涉及到乘法运算的模块都通过 (* use_dsp="yes" *) 的方式指定消耗 DSP 资源来实现，DSP 资源共计消耗 306 个。所有的缓冲模块都通过参数配置的方式 (* ram_style="block" *) 指定由 BRAM 进行相应的合成，从而一定程度上减少了 LUT 和 FF 资源的占用，BRAM 资源共计消耗 146 个。

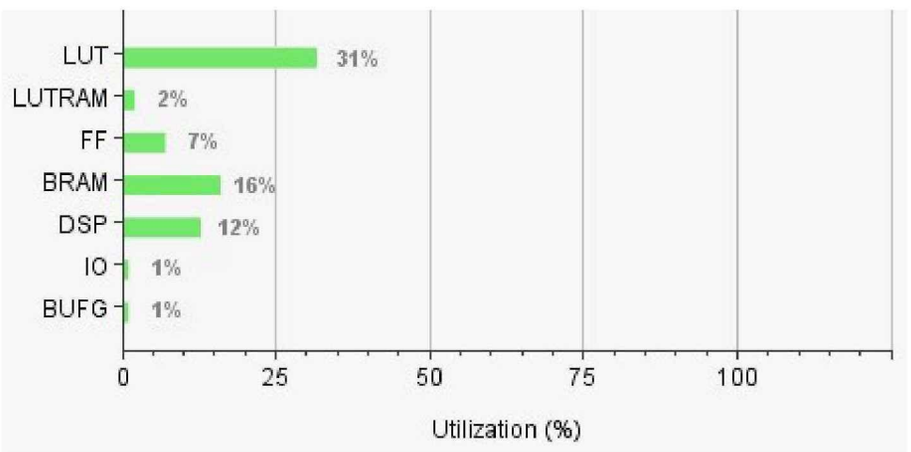


图 5-5 资源消耗

表 5-4 资源消耗情况

Resource	LUT	FF	BRAM	DSP
used	84128	40099	146	306
total	274080	548160	912	2520
utilization	30.69%	7.32%	16.01%	12.14%

5.3.2 功耗分析

本硬件加速系统部署后的功耗如下图5-6所示。在图中展示的功耗中动态功耗为 4.123W，动态功耗占比最高的为 PS 部分超过了 60%，可能是由于 PS 端的处理器部分消耗的功耗占比较大的原因，其余部分单项没有高于 10%。静态功耗为

0.334W，分为 PL 端和 PS 端两部分，静态功耗部分占总功耗部分的占比较小，只有 7%。片上总功耗为 4.457*W*。

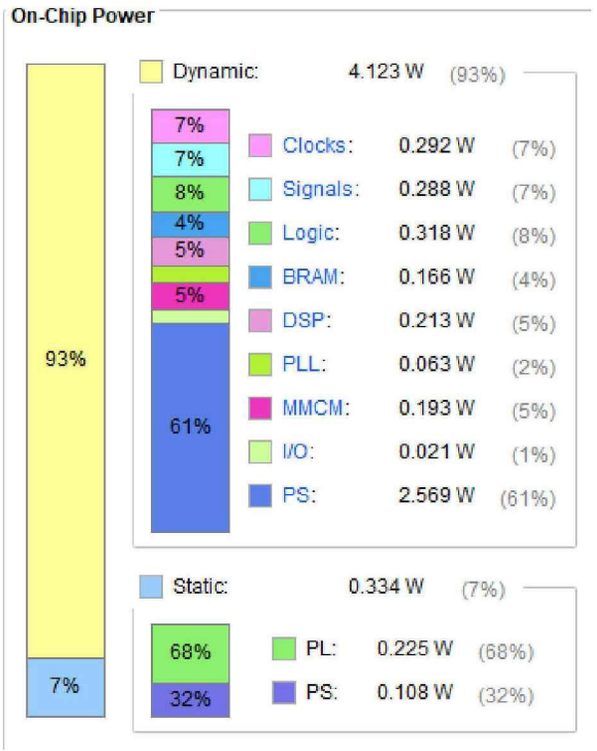


图 5-6 功耗

分别与 CPU 运行 VGG-16 和 LSTM-256 前向推理时的功耗进行比较，如表5-5所示。本设计的能耗比分别是 Intel Xeon E5-2665 的 53.5 和 36.3 倍。

表 5-5 与 CPU 的对比

	CPU	CPU	本文方案	本文方案
平台	E5-2665	E5-2665	ZCU102	ZCU102
测试	VGG-16	LSTM-256	VGG-16	LSTM-256
时钟频率/MHz	2400	2400	250	250
功耗/W	93	93	4.123	4.123
能效比 (GOPs/W)	0.63	0.54	33.71	19.64

5.3.3 时序分析

在 PS 端的输出时钟频率为 250*MHz* 下其时序的分析报告如下。首先是建立时间的时序约束，用于确保数据信号在时钟到达前达到稳定电平，以使数据信号可以被成功采样，该时序约束的最坏路径的时序裕量为 0.041ns，如表5-6所示；其次

是保持时间的时序约束，用于确保数据信号在时钟到达后保持稳定电平，以使数据信号可以被成功采样，该时序约束的最坏路径时序裕量为 0.010ns，如表5-6所示。

表 5-6 建立时间和保持时间约束

	建立时间	保持时间
Worst Negative Slack(ns)	0.041	0.010
Total Negative Slack(ns)	0.000	0.000
Number of Failing Endpoints	0	0
Total Number of Endpoints	262067	261890

最后是脉冲宽度，是要求数据的保持时间必须达到一定长度，以确保下一级电路能够正常采样，该约束的宽度裕量为 0.145ns，如表5-7所示。

表 5-7 脉冲宽度约束

参数	值
Worst Pulse Width Slack(ns)	0.145
Total Pulse Width Negative Slack(ns)	0.000
Number of Failing Endpoints	0
Total Number of Endpoints	65399

可以看出各项约束不存在时序不收敛的情况，数据能够正确地在电路中得到采样，不会出现违例的情况。因此根据报告和后续的实验，验证了本加速器设计可以正确地运行在 250*MHz* 的工作时钟下。

5.3.4 功能和性能验证

本文的硬件结构设计将计算阵列的大小设置为 3*3，输入通道和输出通道的并行度都设置为 8。分别对两种网络的功能和性能进行测试。为了突出针对大规模神经网络的支持，因此测试时选择的网络都为较大的网络，其数据参数不能全部存储在片上的 BRAM 资源当中。分别部署了 CNN 和 LSTM 在硬件平台上以判断加速效果。CNN 的主干是 VGG-16，具有大约 130 万个参数，整个网络包含的运算量为大约 43 *GOP*；LSTM 模型采用 LSTM-256，参数超过 120 万个，该模型的运算量为约 2.6 *GOP*。测试主要包括功能性验证和时间性能，测试方案如章节 5.2 中所述。首先在 Vitis 中通过串口与开发板连接，如图5-7所示。通过比特流将程序烧

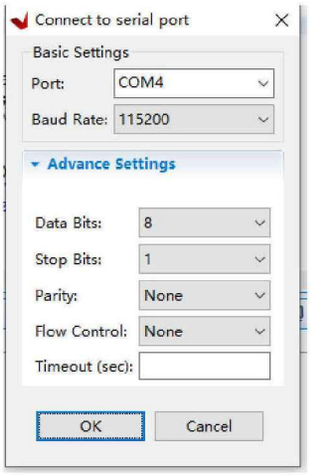
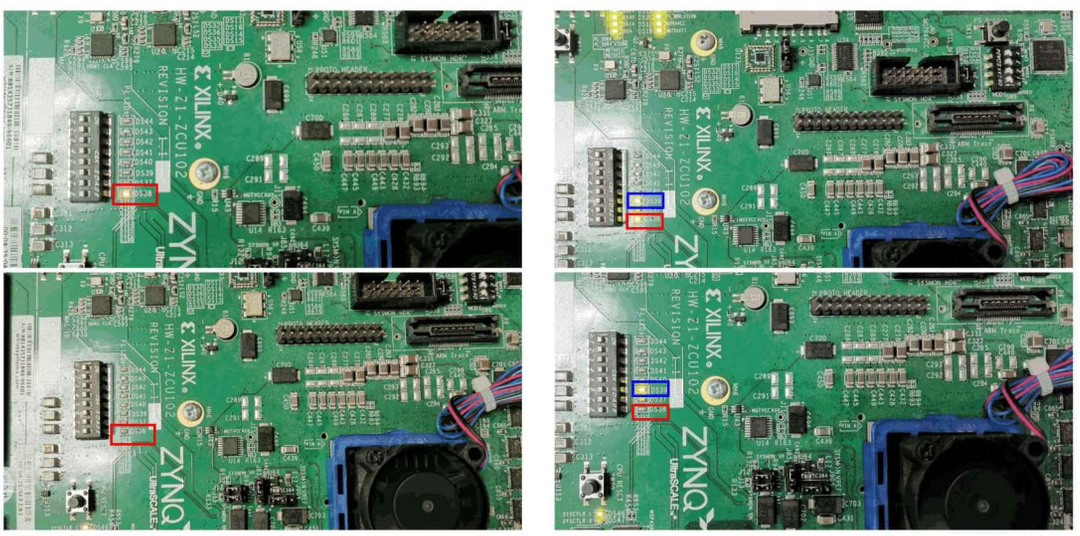


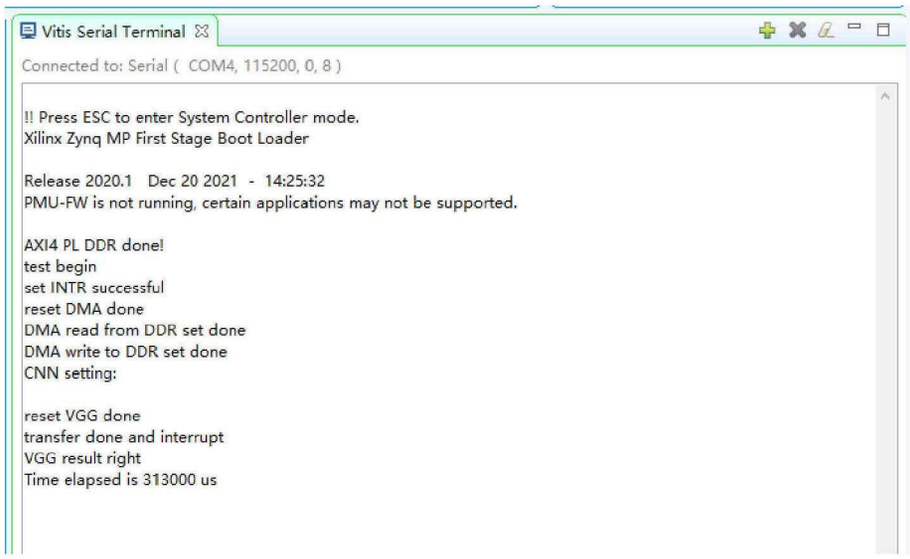
图 5-7 串口连接



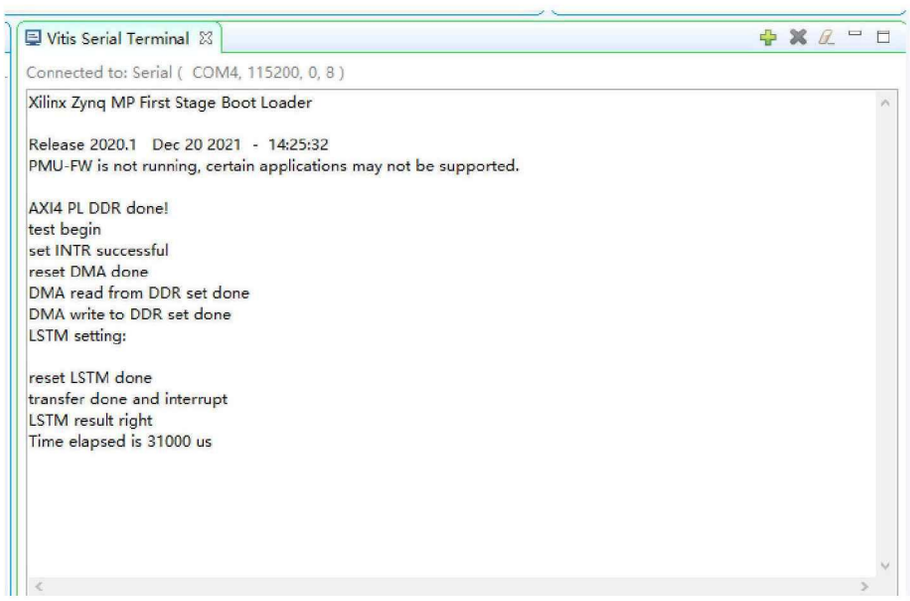
(a) (b)

图 5-8 (a)LED 闪烁；(b)LED 信号正确显示。

制到开发板上，如图5-8(a)所示，led 管脚对应的 DS38LED 灯开始闪烁。如图5-9所示，分别运行两个网络模型，通过键盘键入开始的字符后开始进行数据传输和计算等工作。最终，在串口终端可以看到经过文件与 DDR 中的数据比较后计算结果正确，说明本设计的功能性得到了验证。同时，在串口终端中还能看到 CPU 在接收到中断信号后输出了相应的运行时间。运算结束时，如图5-8(b)所示 done 管脚对应的 DS39LED 灯亮起，DS38 处的 LED 继续闪烁，也说明 FPGA 部分完成了计算。最终总结如表5-8所示，VGG-16 的推理吞吐率为 139GOPS，LSTM-256 的推理吞吐率为 81GOPS。



(a)



(b)

图 5-9 (a)VGG 运行时间；(b)LSTM 运行时间。

表 5-8 网络推理性能

网络	计算量 (G)	推理时延 (ms)	吞吐率 (GOPS)
VGG-16	43	313	139
LSTM-256	2.6	31	81

5.3.5 与其他 FPGA 加速性能的对比

为了验证本文提出的加速器的优点，将其性能与 FPGA 上的现有工作进行了比较，所有加速器的性能结果都是在其所能支持的最高工作频率下的数据。

表 5-9 与 CNN 加速器的对比

	[52]	[53]	[51]	本文方案
FPGA	Virtex-7	Pynq-Z2	Arria-10	ZCU102
时钟频率/MHz	200	100	150	250
DSPs	1436	160	1518	306
GOPs	270	20.53	645.25	139
资源利用效率 (GOPS/DSP)	0.188	0.128	0.425	0.454

如表5-9所示，本文比较了本硬件设计在加速 CNN 时的效果，可以看到对于流水线的优化使得本设计能在更高的时钟频率下运行，可以说明本文设计的流水线分级合理，组合逻辑延迟更低。文献 [52] 使用 16 位定点数的方式进行运算，通过转换为矩阵运算的方式进行加速，这样的方式消耗了大量的 DSP，而每个 DSP 的利用率没有充分地进行挖掘。文献 [53] 通过 HLS 进行加速器的实现，尽管使用的 DSP 很少，但是优化的粒度有限，DSP 利用率较低。文献 [51] 通过充分地优化数据通路，取得了很高的吞吐率，不过在循环展开的维度上的支持少于本文的设计，因此本文的 DSP 利用率更高。通过比较不难看出，本文的硬件架构中通过循环展开和通过原语更进一步的优化对于提高 DSP 利用率是十分有效的。

表 5-10 与 LSTM 加速器的对比

	[54]	[55]	本文方案
FPGA	Stratix-V	Zynq7	ZCU102
时钟频率/MHz	150	142	250
DSPs	2072	900	306
GOPs	316	221	81
资源利用效率 (GOPS/DSP)	0.153	0.246	0.265

本文硬件加速器的 LSTM 加速效果如表5-10所示，其中 GOPS/DSP 的值为 0.265，能够运行的最高工作时钟频率高于其余工作，证明本文提出的数据分块和数据重排的方案降低了流水线中组合逻辑的延迟。文献 [54] 主要在探索通过 RTL

结合 HLS 的方式简化工作量，通过结合已有 IP 核的形式来进行加速器的实现，具体的内部实现未进行优化。因此消耗了大量的 DSP，在计算资源的利用上表现较差。文献 [55] 采用 16 位定点数的推理方式，主要通过优化流水线，将按位乘法与矩阵列向量乘法分为不同的计算模块的方式对流水线计算过程进行优化，但是这相应的降低了 DSP 的利用率，因此该设计的 DSP 利用率低于本设计。性能比较表明，在当前的设计中，基于 DSP 和循环计算特点的量化方案提高了 DSP 的利用率。

5.4 本章小结

本章主要介绍了测试本文硬件加速系统采用的方案。首先介绍了 PL 端的硬件环境部分所做的准备工作，包括 IP 核的封装，block design 的设计直到生成比特流的准备步骤；其次介绍了通过 Vitis 对于处理器系统部分所做的准备工作，主要介绍了代码中对于功能性和时间性能测量的代码逻辑；然后是对本硬件加速系统的各项指标进行分析：本加速系统消耗的片上资源并不多，且由于未使用官方 IP 核，可以移植到大多数 FPGA 加速平台当中；系统的总功耗为 4-5W，其中进行运算的资源所影响的动态功耗较大，占比超过 90%；本系统在 250*MHz* 的工作时钟下时序收敛，可以运行。能够在较高的工作时钟频率上正确运行证明了本文设计中数据分块等对于流水线进行优化的方案是行之有效的。最终通过串口进行了参数量较大的 CNN 和 LSTM 的功能验证和时间性能的测量，本文工作的资源利用效率分别为 0.454 *GOPS/DSP* 和 0.265 *GOPS/DSP*，与前人的工作相比对于该资源的利用率有所提高，从而证明本文中对于 DSP 资源利用的优化有着不错的效果。

第六章 总结与展望

6.1 总结

随着当前对于增加模型的表现力，提高模型的性能，处理复杂的数据和处理大规模数据集的需求不断增加，神经网络模型不断加深和扩展。在 FPGA 这样的嵌入式平台中进行部署，通过利用好网络计算的并行性提高响应速度同时降低能耗和成本，对于神经网络的应用以及发展具有很重要的意义。

本文基于 Xilinx Zynq Ultrascale+ 开发板设计实现了支持量化方案的卷积神经网络和长短时记忆网络的硬件加速系统。主要的工作和实验结论如下：

1) 本文基于神经网络的运算过程和 FPGA 支持并行运算的特性，对目标网络计算密集的部分进行了并行计算的考量。确定了循环并行和循环展开的优化思路，之后结合具体的硬件设计情况进行了相应的实现。
2) 针对片上资源的特性，设计了相应的硬件架构，首先对各个循环的展开和并行进行了相应的支持；其次针对片上计算资源有限的情况，本文优化了单个计算资源的利用效率，使整个 PL 端复用一个计算子模块；最后针对片上存储资源有限的情况，本文通过对数据进行重排，以及相应的数据分块方案减少片上存储数据的重复传输。
3) 系统中的各部分形成流水线的工作模式，通过乒乓缓冲和数据重排等方式对流水线出现的空等时间进行了相应的优化，提高了计算效率。此外，系统中的各模块都支持通过参数进行配置，可以结合不同片上的资源情况和不同网络的结构进行相应的配置，从而实现兼容不同网络，合理分配资源的目标。
4) 在 SoC 的设计部分，本文通过 PL-PS 之间的总线协议由 PS 端进行数据的调配工作，并进行数据通路的选择、使能等工作，实现了将数据存储在外存 SD 卡中，大部分的控制逻辑集中在 PS 端，而 PL 端主要是计算模块的整体系统架构。最终基于此架构进行了系统资源功耗和功能的测试工作。
5) 测试证明本硬件加速器满足低功耗，低资源消耗的离线使用场景需求，同时本加速系统对于计算资源的利用效率在 VGG-16 和 256 个隐藏单元的 LSTM 测试下分别是 0.454 *GOPS/DSP* 和 0.265 *GOPS/DSP*，对比前人的工作有着一定的提升。

6.2 展望

本文的工作仍然存在着许多不足，后续应该在以下方面进行思考和改良：

1) 当前硬件架构中未能实现加法树的复用，当窗口内和窗口间的循环计算的并行度发生变化时，无法通过参数配置的方式进行调整，有待进一步的优化。
2) 当前网络越来越多的引入了残差模块、注意力机制等模块，后续的设计中应该进一步考虑对这些模块的支持。
3) 本文使用的模型轻量化方法都是来自他人的成熟量化方案，应在后续的研究中针对不同类型的神经网络模型，尝试对算法进行优化，以实现更高效的运行。
4) 当前硬件结构的设计中兼容 CNN 和 LSTM 两种神经网络，但实际上对于 LSTM 的优化还存在很大的空间，当前 LSTM 的流水线计算中存在较大的空等时间，后续需要进行进一步的探索和优化。

参考文献

[1] Russakovsky, Olga. Imagenet large scale visual recognition challenge.[J]. International journal of computer vision, 2015, 115(3): 211-252.

[2] Karpathy A, Toderici G, Shetty S, et al. Large-scale video classification with convolutional neural networks[C]. Computer Vision & Pattern Recognition, 2014: 1725-1732.

[3] Merity S, Keskar N S, Socher R. Regularizing and optimizing LSTM language models[C]. 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, 2018.

[4] Wu Y, Schuster M, Chen Z, et al. Google's neural machine translation system: Bridging the gap between human and machine translation[J]. CoRR, 2016, abs/1609.08144.

[5] Liu S, Fan H, Ferianc M, et al. Toward full-stack acceleration of deep convolutional neural networks on fpgas[J]. IEEE transactions on neural networks and learning systems, 2022, 33.

[6] Lemasurier M, Wart A V. Reviews on the visual cortex: A tribute to hubel and wiesel[J]. Neuron, 2012, 75(2): 181-181.

[7] Fukushima K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position[J]. Biological Cybernetics, 1980, 36(4): 193-202.

[8] Lecun Y, Bottou L. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.

[9] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Commun. ACM, 2017, 60(6): 84 90.

[10] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[J]. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014, 1-9.

[11] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[J]. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, 770-778.

[12] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]. Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, 2017: 5998-6008.

[13] Tan M, Le Q V. Efficientnet: Rethinking model scaling for convolutional neural networks[C]. Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, 2019: 6105-6114.

[14] Elman J L. Finding structure in time[J]. Cognitive Science, 1990, 14(2): 179-211.

[15] Jordan M I. Serial order: A parallel distributed processing approach[J]. Advances in Psychology, 1997, 121(97): 471-495.

[16] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural Computation, 1997, 9(8): 1735-1780.

[17] Gers F, Schmidhuber J. Recurrent nets that time and count[C]. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 2000: 189-194vol.3.

[18] Greff K, Srivastava R K, Koutník J, et al. Lstm: A search space odyssey[J]. IEEE Transactions on Neural Networks & Learning Systems, 2016, 28(10): 2222-2232.

[19] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate[J]. CoRR, 2014, abs/1409.0473.

[20] Kim J, El-Khamy M, Lee J. Residual LSTM: design of a deep recurrent architecture for distant speech recognition[C]. Interspeech 2017, 18th Annual Conference of the International Speech Communication Association, Stockholm, Sweden, August 20-24, 2017, 2017: 1591-1595.

[21] Farabet C, Poulet C, Han J Y, et al. Cnp: An fpga-based processor for convolutional networks[C]. 2009 International Conference on Field Programmable Logic and Applications, 2009: 32-37.

[22] Courbariaux M, Bengio Y, David J P. Binaryconnect: Training deep neural networks with binary weights during propagations[J]. Advances in neural information processing systems, 2015, 28.

[23] Courbariaux M, Bengio Y, David J P. Training deep neural networks with low precision multiplications[J]. arXiv preprint arXiv:1412.7024, 2014.

[24] Han S, Pool J, Tran J, et al. Learning both weights and connections for efficient neural network[J]. Advances in neural information processing systems, 2015, 28.

[25] Wan A, Dai X, Zhang P, et al. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions[J]. 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, 12962-12971.

[26] Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network[J]. Computer Science, 2015, 14(7): 38-39.

[27] Chollet F. Xception: Deep learning with depthwise separable convolutions[C]. Proceedings of the IEEE conference on computer vision and pattern recognition, 2017: 1251-1258.

[28] Winograd S. Arithmetic complexity of computations[M]. Siam, 1980.

[29] Chen Y T, Ou Y F, Huang C T. A winograd-based highly-parallel convolution engine for 8-bit cnn acceleration[C]. 2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS), 2022: 395-398.

[30] Ye H, Deng H, Wang J, et al. 3d-nwa: A nested-winograd accelerator for 3d cnns[C]. 2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA), 2022: 143-144.

[31] Sankaradas M, Jakkula V, Cadambi S, et al. A massively parallel coprocessor for convolutional neural networks[C]. 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2009: 53-60.

[32] Li Y, Wen M, Fei J, et al. Tile-sim: A systematic approach to systolic array-based accelerator evaluation[C]. 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2022: 141-143.

[33] Joo J, Yoon M, Choi J, et al. Understanding and reducing weight-load overhead of systolic deep learning accelerators[C]. 2021 18th International SoC Design Conference (ISOCC), 2021: 413-414.

[34] Moon G E, Kwon H, Jeong G, et al. Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(4): 1002-1014.

[35] Kamaleldin A, Göhringer D. A hybrid memory/accelerator tile architecture for fpga-based risc-v manycore systems[C]. 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL), 2022: 300-306.

[36] Chen Y T, Yen Y X, Chen C T, et al. Tile-based architecture exploration for convolutional accelerators in deep neural networks[C]. 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS), 2021: 1-4.

[37] Cao S, Zhang C, Yao Z, et al. Efficient and effective sparse lstm on fpga with bank-balanced sparsity[C]. Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019: 63 72.

[38] Silfa F, Arnau J M, González A. Boosting lstm performance through dynamic precision selection[J]. 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2019, 323-333.

[39] Wang H, Qiu D, Ge F, et al. Implementation of bidirectional lstm accelerator based on fpga[C]. 2022 IEEE 22nd International Conference on Communication Technology (ICCT), 2022: 1512-1516.

[40] Paulin G, Conti F, Cavigelli L, et al. Vau da muntanialas: Energy-efficient multi-die scalable acceleration of rnn inference[J]. IEEE Transactions on Circuits and Systems I: Regular Papers, 2022, 69(1): 244-257.

[41] Que Z, Nakahara H, Nurvitadhi E, et al. Recurrent neural networks with column-wise matrix vector multiplication on fpgas[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2022, 30(2): 227-237.

[42] Cao Q, Balasubramanian N, Balasubramanian A. Mobirnn: Efficient recurrent neural network execution on mobile gpu[C]. Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, 2017: 1-6.

[43] Ardakani A, Ji Z, Gross W J. Learning to skip ineffectual recurrent computations in lstms[J]. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, 1427-1432.

[44] Xilinx. Floating-point operator product guide[J]. LogiCORE IP Product Guide, 2020.

[45] Xilinx. Zynq ultrascale+ mpsoc data sheet: Dc and ac switching characteristics[J]. Zynq Ultra-Scale+ MPSoC DataSheet, 2023.

[46] Jacob B, Kligys S, Chen B, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference[C]. Proceedings of the IEEE conference on computer vision and pattern recognition, 2018: 2704-2713.

[47] Li J, Alvarez R. On the quantization of recurrent neural networks[J]. CoRR, 2021, abs/2101.05453.

[48] 张伟枫. 基于 fpga 的 lstm 循环神经网络加速器设计与研究 [D]. , 2021.

[49] 张家华. 一个基于 fpga 的卷积神经网络加速器 [D]. , 2021.

[50] 黄沛昱. 基于 fpga 的卷积神经网络硬件加速器设计 [J]. 计算机应用与软件, 2023, 40(38-44).

[51] Ma Y, Cao Y, Vrudhula S, et al. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks[C]. Acm/sigda International Symposium on Field-programmable Gate Arrays, 2017: 45-54.

[52] Kala S, Jose B R, Mathew J, et al. High-performance cnn accelerator on fpga using unified winograd-gemm architecture[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019, PP(99): 1-13.

[53] Zhou Z, Liu Y, Xu Y. Design and implementation of yolov3-tiny accelerator based on pynq-z2 heterogeneous platform[C]. EITCE 2020: 2020 4th International Conference on Electronic Information Technology and Computer Engineering, 2020: 1097 1102.

[54] Guan Y, Hao L, Xu N, et al. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates[C]. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017: 152-159.

[55] Que Z, Nugent T, Liu S, et al. Efficient weight reuse for large lstms[C]. 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2019: 17-24.