**分 类 号 学号 M202072547**

**学校代码 1 0 4 8 7 密级**



**硕士学位论文**

**二值复数神经网络（BCNN）专用加速芯片的研究与设计**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **学位申请人** | **：** |  | **高余敬** |  |  |
| **学科专业** | **：** |  | **电子信息** |  |  |
| **指导教师** | **：** |  | **童乔凌 教授** |  |  |
| **答辩日期** | **：** |  | **XXXX年XX月XX日** |  |  |

**A Dissertation Submitted in Partial Fulfillment of the Requirements**

**for the Professional Master Degree**

**[Research](https://www.bing.com/dict/search?q=research&FORM=BDVSP6&cc=cn)****[and](https://www.bing.com/dict/search?q=and&FORM=BDVSP6&cc=cn)****[Design](https://www.bing.com/dict/search?q=design&FORM=BDVSP6&cc=cn) of Bin**[**ary**](https://www.bing.com/dict/search?q=Binary&FORM=BDVSP6&cc=cn)**[Complex](https://www.bing.com/dict/search?q=complex&FORM=BDVSP6&cc=cn)****[Neural](https://www.bing.com/dict/search?q=neural&FORM=BDVSP6&cc=cn)****[Networks](https://www.bing.com/dict/search?q=networks&FORM=BDVSP6&cc=cn) (BCNN****[)](https://www.bing.com/dict/search?q=%29&FORM=BDVSP6&cc=cn)  Dedicated Acceleration Chip**

**Candidate : Gao Yujing**

**Major :** [**Electronic**](https://www.bing.com/dict/search?q=electronic&FORM=BDVSP6&cc=cn)[**Information**](https://www.bing.com/dict/search?q=information&FORM=BDVSP6&cc=cn)

**Supervisor : Prof. Tong Qiaoling**

**Huazhong University of Science and Technology**

**Wuhan 430074, P. R. China**

**October, 2022**

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保 密□，在 年解密后适用本授权书。

本论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名： 指导教师签名：

日期： 年 月 日 日期： 年 月 日

# 摘 要

近年来，随着人工智能的飞速发展，深度神经网络（Deep Neural Networks，DNN）在诸多领域都有着广泛应用，如目标检测、人脸识别、图像视频超分处理等。由于DNN算法具有存储密集和计算密集两大特性，使得目前的DNN模型无法在边缘侧充分地训练或者推理，而是只能通过与云端设备协调的方式进行工作。但是随着自动驾驶、无人机导航等技术的发展，大量的数据需要在本地完成处理，以确保实时、隐私和安全，显然现有的该类边缘智能设备还不足以满足需求。为此，本文基于软硬件协同的设计思想提出并实现了一种面向二值复数神经网络（Bin[ary](https://www.bing.com/dict/search?q=Binary&FORM=BDVSP6&cc=cn) [Complex](https://www.bing.com/dict/search?q=complex&FORM=BDVSP6&cc=cn) [Neural](https://www.bing.com/dict/search?q=neural&FORM=BDVSP6&cc=cn) [Networks](https://www.bing.com/dict/search?q=networks&FORM=BDVSP6&cc=cn)，BCNN）的专用加速器芯片，经测试该芯片具有低功耗、高能效等优势。本文的主要研究工作有如下几点：

1. 研究BCNN算法的网络结构以及数据计算特性，针对网络结构中的卷积层、批标准化层、二值化层等分别设计了相应的优化计算模块，例如使用简单的popcount(xnor)计算替代复杂的乘法计算、使用简单的阈值比较计算替代复杂的批标准化操作等等，节省了硬件开销的同时还提高了计算效率。
2. 研究BCNN算法中二值复数的数据特点，设计了新兴的数据存储方式，极大地提高了存储资源利用率并减少了内存访问次数。
3. 研究并设计RISC-V扩展指令集，并基于该扩展指令集以及上述工作内容设计了面向BCNN的专用协处理器；该协处理器采用访问执行解耦合架构，内置一块面向二值复数计算且算力强大的脉动阵列阈值单元和两块片上存储，可高效完成BCNN算法中二值复数层计算任务。
4. 将设计的专用协处理器与RISC-V开源核Rocket集成得到一个完整的专用加速器芯片硬件系统，并完成其软件系统系统设计。
5. 搭建软硬件协同验证平台，对专用加速器芯片进行验证与优化；使用Vivado工具对优化后的芯片进行性能评估，并通过FPGA实现该专用加速器芯片。

实验结果表明，本文实现的面向BCNN的专用加速器芯片计算性能可达3.3TOPS，功耗仅为2.076W，基本满足边缘智能设备低成本、低功耗、高能效的需求。

关键词： 边缘智能；二值复数神经网络；RISC-V；协处理器；脉动阵列阈值单元

# Abstract

As

**Key words:**

目 录

二值复数神经网络（BCNN）专用加速芯片的研究与设计 I

A Dissertation Submitted in Partial Fulfillment of the Requirements II

for the Professional Master Degree II

摘 要 I

Abstract II

1 绪论 1

1.1 研究背景与意义 1

1.2 研究现状 6

1.3 本文主要研究内容 17

1.4 本文的章节安排 18

2 CNN与BCNN算法概述 20

2.1 引言 20

2.2 卷积神经网络及其硬件加速原理 20

2.3 BCNN算法概述 27

2.4 本章小结 34

3 面向BCNN算法的协处理器设计 35

3.1 引言 35

3.2 协处理器设计架构 36

3.3 指令集（ISA）设计 40

3.4 SATU设计 41

3.5 内存设计及数据存储方式 53

3.6 本章小结 58

4 BCNN专用加速器的系统设计 59

4.1 引言 59

4.2 开源处理器Rocket概述 60

4.3 系统硬件架构设计 61

4.4 系统软件架构设计 65

4.5 本章小结 68

5 仿真与性能评估 69

5.1 引言 69

5.2 功能仿真 70

5.3 性能评估 75

5.4 本章小结 78

6 总结与展望 80

6.1 总结 80

6.2 展望 81

致 谢 83

参考文献 84

附录1 攻读硕士学位期间取得的学术成果 88

附录2 部分程序代码 89

# 绪论

## 研究背景与意义

随着互联网和大数据时代的到来，近年来在生活中的各个领域，人们越来越离不开各类智能电子设备，例如人脸识别、智能制造、智慧医疗、自动驾驶等，因而人工智能的研究也成为了众多科研工作者研究的焦点。在众多的研究成果中，深度神经网络（Deep Neural Networks，DNNs）由于其突出的性能表现，在许多计算机视觉领域得到了广泛的应用，如图像分类、动态追踪、目标检测等[1,2]。



图 互联网和大数据时代的到来[3]

为了研究人脑行为，上世纪六十年代，科学家提出了深度神经网络的概念，试图通过仿真学研究人脑架构，模拟人脑的活动机理，并使用数学来进行抽象表示，最终采用计算机软件仿真来实现多个方面的应用[4]。而深度学习（Deep Learning，DL）一词最初在 1986 被引入机器学习（Machine Learning，ML），后来在 2000 年时被用于人工神经网络（Artificial Neural Network，ANN）。深度学习作为机器学习的一个子领域，通过使用多层次的非线性信息处理和抽象，用于有监督或无监督的特征学习、表示、分类和模式识别，而ANN是深度学习的最常用形式。第一代人工神经网络由简单的感知器神经层组成，只能进行有限的简单计算。第二代使用反向传播，根据错误率更新神经元的权重。其后，多种和各异的神经网络也出现了，如前馈神经网络 （FNN)、卷积神经网络 （CNN)、循环神经网络 （RNN) 等。利用局部连接、共享权重、池化和多层使用这四种基本思想的构成了卷积神经网络（CNN），2012年Krizhevsky 等人 提出了基于深度卷积神经网络 （CNN) 架构， AlexNet，使CNN在DL 上获得了重大突破，并快速的演化为深度卷积神经网络（DCNN），继而以监督、非监督、半监督或强化学习的形式被广泛应用于各个领域。无论是何种形式何种目标任务，现阶段的DL从本质上都可视为是在海量数据的基础上通过“训练”完成一个复杂的数据拟合过程，所以对数据量及运算能力提出了极高的要求[5]。

作为推动人工智能发展关键驱动力的大数据，其数据源正在从超大规模云数据中心转移到到日益普及的终端设备，即数据越来越多的在网络边缘产生。 在5G和移动物联网大规模商用及传感器技术的快速演进的推动下，移动物联网设备的数量与类型激增，从而产生了大量持续感知设备物理环境的多模态数据（音视频、环境要素、姿态与位置等），将海量与多类型的数据传输到云计算中心进行计算不仅对网络容量，云计算基础设施带来了严峻的挑战，并且带来了较高的传输延迟，并不能满足自动驾驶、目标跟踪等新型应用的需求，而且会带来了保密与隐私问题；在此情况下，边缘计算和人工智能相结合所形成边缘人工智能（Edge AI），提出尽可能的靠近数据源和最终用户完成计算任务，从而解决上诉问题[3,6]。

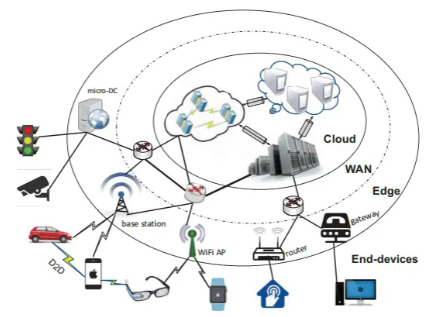


图 边缘计算[3]

边缘人工智能，即完全在边缘设备上执行DCNN模型的训练与推理的方式代表着需要具有更高计算能力的边缘设备，一经提出便引起了工业界和学术界的关注。例如全球最具权威的IT研究与顾问咨询公司Gartner发布的hype cycle就预测了未来5到10年边缘智能作为新型技术将进入稳定产出阶段。包括Google, Microsoft, Intel和 IBM在内的这些巨头公司，也在着手布局他们的边缘智能设备。而边缘智能设备不但需要高能效、高吞吐量、低时延，还需要低功耗和高面积效率，因为绝大部分的边缘设备是使用电池进行供电的。且随着边缘智能的发展，边缘设备的系统也越来越复杂，简单的软件优化或是硬件设计都是无法完成目标的，要想解决这一问题，就需要从根本上把软件的灵活性和硬件的高效性紧密结合，通过系统的统筹和软硬件的协同设计，达到灵活和高效的融合，最大程度的满足算法的迭代需求和硬件的高效设计的需求。因此我们这篇文章的工作就是采用“软硬件协同”的设计思想，旨在从软件和硬件两个角度出发来研究分析并提出一款高能效和低功耗的专用边缘设备。

首先从软件算法角度出发，目前来说，在深度神经网络算法的研究中主要存在以下方向：一是从架构层出发，采用新的AI模型和算法来替代当下常用的监督学习DNN算法。Graphcore就提出目前的先训练后推理的方法并不合理，应当采用可以学习并在部署后能够持续进化的一种“学习系统”，例如采用无监督学习算法，就不需要对数据加标签进行训练，或者采用仿真计算、神经形态计算等等。另一个研究方向从现有的深度学习算法的结构出发，进一步提高能效和减低功耗，通过优化深度神经网络结构，模型裁剪[7-10]与核稀疏化[11,12]以及参数量化加速[13,14]等方式来实现，例如在训练时将权重和激活值的数值精度从原来的32位降到16位或者8位，在推理的过程中采用更低的8位或者4位，甚至1~2位，这就大大降低了功耗和计算量，这个研究方向具有很高的理论研究价值和应用价值[15]，在针对这一研究方向也有众多的研究成果中，作为深度学习“三大泰斗”之一的Yoshua Bengiod提出的二值神经网络（BNNs）凭借其单比特神经元特性在众多的成果中脱颖而出。二值神经网络通过将每个神经元的比特数压缩到极限来满足受限的硬件开销，即BNN每层的输入、权重和激活层的每个元素都仅使用单个二值化的数值（0或者1，分别代表-1或+1）来表示[16]。

二值神经网络对于嵌入式应用具有如下的几个优势：（1）计算效率：二值网络具有极强的硬件友好性，通过将原先的32或者64比特的神经元映射到单比特，而原先的全精度计算也相应地映射为单比特的异或运算（xor)外加一个计“1”（popcount）操作[16]，将计算效率提升了10倍以上[17]。从内存角度来说，使用单比特替代32比特的单精度浮点数或16比特的半精度浮点的使用，二值神经网络更是大幅度提升了内存的利用率和吞吐量[18]。（2）低硬件开销：由于硬件逻辑结构简单和极低的内存需求，二值神经网络的硬件开销远远小于其他的深度神经网络[19,20]。（3）能效比：因为其低硬件开销和功耗低等特性[21]，基于二值网络的硬件对移动设备的电池也是十分友好的。（4）鲁棒性：由于二值权重的空间离散性，二值神经网络具有更好的鲁棒性[22,23]。由于这些优势，二值神经网络已经被广泛运用，如自动驾驶[24]、智慧农业[25]、COVID面部识别[26]、图像增强[27]等[28]。

日历

中度可信度描述已自动生成

图 二值神经网络的卷积过程

虽然二值神经网络具有以上众多优点，但是由于二值化过程中信息的丢失和模型的压缩，导致BNNs的精度损失也很严重。所以自从二值神经网络提出后，大量的研究人员也在致力于提高该网络的精度[29-38]。Yanfei Li等人结合了复数神经网络的研究成果，在2021年提出的二值复数神经网络便是其中一项重要的贡献[28]。

起初Chiheb Trabelsi等人提出的复数神经网络（Deep Complex Networks，DCN）的提出本意在于修正普通的DNNs网络[39]，因为大多数DNNs网络都是采用实数来表示输入和权重，与复数网络（输入、输出和权重都是用复数来表示）相比，不论是表示能力、泛化能力还是抗噪声能力都略逊一筹[40]。大量的实验结果也证明了在相同大小的网络中，复数网络具有更高的准确性。且另一个引人注目的优势便在于复数网络可以携带相位信息，在声波的傅里叶表示[41]，PolSAR图片[42]等应用中积极重要。

而结合了二值神经网络和复数神经网络的二值复数神经网络既继承了BNN网络的计算效率、低硬件开销、高能效比、高鲁棒性等优势的同时，又结合DCN网络提高了BNN网络的精度[28]。在二值复数神经网络中，每层的输入、权重和输出都使用二值化的复数值来表示，{1+i,1-i,-1+i,-1-i}中某一个，即每个神经元使用双比特来表示（一个表示实数部分，另一个表示虚数部分）。与BNN网络相同，二值复数卷积操作依旧可以使用xor-popcount运算替代了大量的乘加计算。

为解决如何实现高能效和低功耗的边缘智能设备问题，在上文中我们从软件角度出发，引出了二值复数神经网络这一方案，下面我们将从硬件角度开始分析。目前市场上的人工智能设备都是通过CPU、GPU、FPGA和ASIC以及他们的各种组合来实现的，Intel、Geogle、NVIDIA、Qualcomm和IBM等公司也推出或者正准备开发新的产品。一般来说一个边缘设备至少需要一个CPU来完成系统控制，当有大量数据需要并行处理时，比如神经网络的卷积、池化等计算，就需要特定功能的协处理器来帮助实现，所以协处理器的设计也是边缘智能设备设计的重中之重。下面我们针对CPU、GPU、FPGA及ASIC这四种不同架构的芯片分别讨论。



图 CPU、GPU、FPAG、ASIC灵活性和性能对比[25]

（越左侧灵活性越高、性能越低，反之性能越低，灵活性越高）

由于CPU的普及和通用编程能力，深度神经网络起初研究时都是在CPU上实现的，但是CPU并不是理想的实现神经网络的硬件，因为CPU最突出的功能是可以复杂的控制流，但神经网络的运行过程需要的数据流的计算，几乎不需要控制。而且随着卷积神经网络层数不断增加，需要处理的数据量不断膨胀，加之访存速度受限（“冯诺依曼瓶颈”），现有的CPU已经无法满足需求。而NVIDA提出的图形处理器（GPU）[24]和Geogle提出的张量处理器（TPU）[43]采用SIMD（单指令流多数据流），能够快速完成浮点密集型计算，特别是GPU架构发展非常迅速，从一开始的可重构，到当下可编程的大规模并行协处理器，这使得它非常适合深度神经网络计算。现在具有超过1000个处理核和超大容量的片上内存的CPU，其功耗也只有几瓦，但是对于边缘智能设备来说，这样的能效还是不足的。

现场可编程门阵列（FPGA）是“可重构”芯片的一种，具有模块化和规则化的架构，主要包含了片上存储、可编程逻辑块以及用于连接逻辑模块的可重构互连层次结构。FPGA最大的优势就是灵活性，可以在运行之前和运行期间对硬件进行重构。当下最先进的FPGA可以完成约十亿个逻辑门复杂度的SoC，且时钟频率可以接近吉赫兹（GHz）范围，因此可以在几瓦的功耗下达到GFLOPS数量级的算力。专用集成电路（ASIC）是指为特定用户和特定电子系统的需求而设计、制造的专用电路。用ASIC来将人工智能算法“硬件化”，可以带来高性能、低功耗等优点，但是缺点也是突出的，因为ASIC芯片的开发需要很高的成本，设计完毕后的流片需要一大笔费用，这笔费用一般不会低于1000万美元，这对开发芯片的公司来说不仅门槛高，商业风险也很大。且ASIC芯片一旦开始批量生产就无法再对内部电路进行修改，这对迭代过程迅速的人工智能算法来说也是不合适的。为了避免这些风险，我们通常都是采用灵活性更高的、成本更低、开发周期更短的FPGA来实现边缘智能设备。

综上所述，为了解决将DNNs部署到边缘智能设备这一问题，我们从软硬件两个角度出发，结合并提出了面向二值复数神经网络的专用加速器，并通过FPGA实现这一解决方案。且目前为止，虽然二值复数神经网络的算法层面的研究已经逐步展开，而针对二值卷积神经网络的加速器的研究工作还没有完全开展，我们的工作刚好弥补了这一空缺。

## 研究现状

### 二值复数神经网络的研究现状

二值复数神经网络的发展时间较短，由浙江大学的Yanfei Li等人在2021年首次提出，所以面向该网络的研究成果还不是很多。二值复数神经网络紧密联系了二值神经网络和复数神经网络，故BCNNs的研究现状与二者是密不可分的，可以看作是二者拓展并交叉的新一领域，故讨论BCNN的研究现状自然也可等价为研究BNNs和DCNs两个神经网络的现状。

早期在研究压缩模型的尺寸和计算量的工作中，M. Rastegari等人和M. Courbariau等人分别提出了XNOR-Net[29]和二值权重网络（BWN）[44]。在此工作的基础上，2016年，二值神经网络概念首先由图灵奖获得者且被誉为世界AI三巨头之一的Y. Bengio等人在论文[44,45]中提出，这两篇文章[29,44]中提出了BNN的基本组成部分，包括（1）二值化函数及其通过直通式估算器（STE）的近似梯度；（2）批量归一化（Batch Normalization，BN），这对于BNN是否能够收敛至关重要；（3）必须保证第一层和最后一层的全精度，因此这两篇文章也被公认为是二值神经网络发展的奠基石。随后A. G. Anderson和C. P. Berg在2017年通过理论计算解释了为什么BNN可以有效的逼近DNN网络：首先经过二值化后的二值向量保留了DNN的实数向量在高维度几何空间中方向；其次通过batch-normalization后的二值点乘运算(popcount(xnor))保留了原始DNN点乘的性质；再者第一层的实数卷积层可以将输入图像嵌入到高维度的二值空间，然后再通过二值运算来有效处理[46]。

虽然BNN在压缩模型的尺寸和计算量上效果显著，且具有极好的硬件友好性，但是BNN一经提出还是受到了广泛的争议，其中最重要的原因便是精度损失，这主要是因为，输入二值化和二值活化带来的信息丢失以及权重二值化造成的模型容量的降低；其次还因为现有的网络结构和训练方法主要是针对实数值的DNNs，无法平滑地在BNN中使用。所以相应地，科研人员为了提高BNN的训练精度开展了大量的工作，主要包括以下方向：

1. 减少信息丢失。例如早在2015年时M. Rastegari就在文章[29]中提出可以通过添加增益项（例如scaling factors)更好地近似DNN激活函数来实现，增益项可以基于输入的统计数据[30,31]或者在训练过程中逐步学习[32,33]来提取。
2. 增强BNN模型容量。例如S. Zhu等人提出的BENN[37]和J. Bethge等人提出的Group-Net[36]网络中就证明了可以通过在网络使用多个BNN组件来实现模型容量的提升。此外还可以通过使用稍微多一点的比特数来表示一个神经元（其中每个比特都表示一个基）来实现模型容量的提升，这些基可以固定为2的幂值（如1，2，4，8…）[30]，或者像残差基一样动态可调[31]，也可以在训练中学习得到[32]。
3. 设计BNN特定的网络结构。当下大多数现存的网络模型和结构都是针对DNN设计的，所以BNN研究人员也开始着手设计面向BNN算法的网络结构，例如J. Bethge 等人提出的ResnetE与BinaryDenseNet网络[38]和J. Bethge等人提出的MeliusNet[36]。其中BinaryDenseNet通过使用更多的跳跃链接shortcut来复用信息，进而保存网络结构中丰富的信息流；而MeliusNet，如下图所示，保存了第一层256个通道的数据流的全精度， 并创新性的提出了two-block结构（一个dense block和一个improvement block）来学习得到独立的另外64个通道的数据，并与原始的256个通道数据结合，通过这个方式，可以降低BN造成的信息丢失。

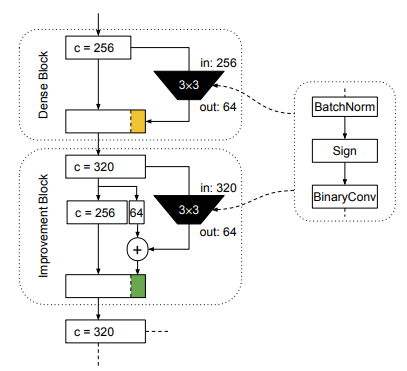


图 MeliusNet的two-block结构[36]

1. 改善BNN的训练方法。目前改善BNN训练方法主要通过修正sign激活函数和STE梯度估算器，例如M. Alizadeh于2018年在文章[47]中提出的“second-moment”技术来调整学习率，进而确保了在BNN训练过程中STE应用的成功率。无独有偶，同一年S. Darabi等人[33]提出了一种类似 Swish 激活函数的变体来代替 STE 机制，通过这一变体函数可以获得更有效的反向传播函数。2019年，Lahoud等人在文章[48]中提出在训练的开头使用一个平滑的激活函数，然后在训练中不断地将其锐化得到一个二值表示的并可替代sign的激活函数。Hou等人在文章[49]中讨论了损失感知的二值化技术，展示了一种具有对角Hessian近似的近似牛顿算法，该算法可以直接最小化二值化权重的损失。除了修正sign函数和STE，还有部分研究人员也致力于改善BNN训练中的其他方面，例如W. Tang等人[31]试图通过在训练使用特殊的正则化项来激励潜在的浮点数变量接近+1和-1，Y. Umuroglu等人[50]在BN和activation之前添加一层池化层。

针对BNN展开的研究工作还有很多，这里不再赘述，详细可参考T. Simons[51]和H. Qin[52]等人对BNN的综述文献，下面我们将展开对复数神经网络的研究现状的介绍。复数通过实数轴和虚数轴将一位实数线（即-∞到∞）扩展到二维平面。虽然现实世界并不存在复数，但是复数凭借其独特的性质和计算特性可以有效地修正实数表达能力不足的问题，尤其是在表达含有相位的信息时。例如在物理学中，因为经过傅里叶变化后的系数都是复数，所以复数更适合表示波的信息；在神经学科中，D. P. Reichert和T. Serre早在2013年就提出使用复数来表示神经元节律[53]，因为神经元节律就是以发射率和相位信息为基础的，且对神经元交流至关重要;在地球学中，Y. Cao等人于2019年就提出了相比单纯的SAR图像，PolSAR 图像可以提供更全面、更可靠的信息，因为PolSAR图像的散射特性就是使用复值偏振散射矩阵来描述，其中每个元素的幅度对应于电磁波从目标到雷达的背向散射强度，相位对应于在传感器平台和目标之间距离[42]；在生物医学科学中，H.-S. Choi等人使用复数有效地处理相位信息，极大地促进了 MRI 图像的重建[41] 。

由于复数超强的表达能力和处理复数输入信号的需要，长期以来人们一直在致力构造复数神经网络，这段历史可以甚至可以追溯到1990年代[54,55]。最新且最具代表性的复数神经网络的研究成果当属C. Trabelsi等人提出的深度复数网络（DCN）[39]，在该文章中，作者提出了复数神经网络的构建方法，其中包括复数卷积、复数批标准化和复数权重初始化策略等等。DCN不仅考虑到了复数输入和权重的实部和虚部的相关性，还证明了它在分类任务上的有效性，实验数据表明，仅具有相关实数网络一半容量的DCN网络就可以实现相当或者更强的性能。

基于BNN和DCN的研究成果，Y. Li和T. Geng等人在2021年提出了二值复数神经网络（BCNN），文章[28]中BCNN系统有效地结合了BNN和DCN，Y. Li等人还针对NIN-Net和ResNet18等模型在CIFAR-10数据集上分别比较BNN、DCN和BCNN网络的参数量和精度（如下表）。从表 中数据可知，针对NIN-Net和ResNet18网络，相比于BNN，使用BCNN模型大小几乎不变，而精度分别提升了1.85%和0.52%。另外与DCN相比，BCNN与BNN都可以大幅度的而降低计算成本，这对于低成本、低功耗、高效能的边缘设备来说是极具吸引力的，从表 中可见，在精度损失可接受的条件下，针对NIN-Net和ResNet18，BCNN相比DNN参数分别减小了20和30倍。

|  |  |  |  |
| --- | --- | --- | --- |
| 网络 | 类型 | 参数量 | 精度（%) |
| NIN-Net | DNN | 3.69M | 89.64 |
| BNN | 0.192M | 85.77 |
| BCNN | 0.187M | 87.62 |
| ResNet18 | DNN | 42.63M | 93.02 |
| BNN | 1.39M | 90.67 |
| BCNN | 1.39M | 91.19 |

表 BNN、DCN和BCNN网络的参数量和精度对比[28]

### 边缘智能的研究现状

目前主流的边缘智能设备主要是深度学习加速器，一般应用在智能手机、无人机、物联网、自动驾驶汽车等，该方向的发展大致可以分为两个阶段：第一阶段是在2013到2015年，这一阶段众多巨头公司，如英伟达、谷歌、英特尔和微软等开始陆续介入并开始探索如何从硬件角度实现网络模型，谷歌TPU的架构与设计也是在这一阶段完成的；第二阶段即2016年至今，这一阶段是实际应用阶段，许多高性能、低功耗和低成本的移动端和边缘侧设备就是在这一期间落地的。

根据美国市场调研公司Tractica的预测，深度学习加速器的市场规模在2025年将达到663亿美元（如下图 所示），其中云端训练、云端推理、边缘推理各占1/3，其中主要份额还是由“1+3”四家巨头公司占据。下面我们首先介绍4家大公司的边缘智能成果，再介绍国内外一些研究机构和初创公司的研究成果。

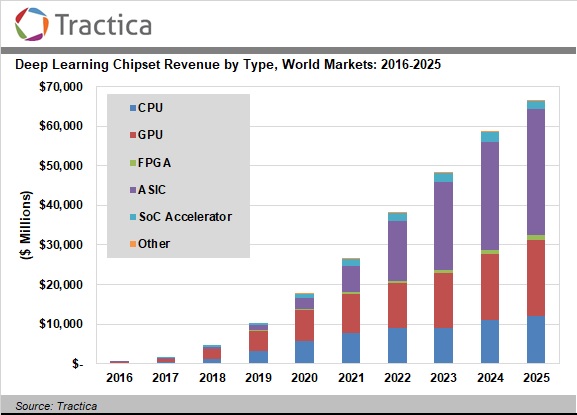


图 深度学习加速器营收增长预测（来自Tractica）

谈及深度学习加速器就不得不提GPU最早的发明者英伟达，正是有了GPU作为作为硬件基础，深度学习算法的优势才得以体现。除了GPU（数据中心）外，英伟达还在边缘侧计算提出了众多优秀的应用解决方案，特别是自动驾驶汽车领域。例如英伟达在2017年针对边缘侧汽车应用推出了Xavier芯片（裸片如下图），其中集成了8个64位ARMv8-A内核和512核Volta架构GPU，除此之外还有一个用于AI处理的8位整数矩阵运算电路。Xavier拥有超过90亿个晶体管，每秒可以运行30万亿次计算（30TOPS），功耗仅为30W。



图 Xavier芯片版图（来自Wikichip）

谷歌于2016年推出的TPUv1峰值运算速率更是达到了92TOPS，当年击败韩国顶级棋手李在石、2017年击败世界围棋冠军柯洁的AlphaGo,就搭载了TPUv1。其中巨大的MAC阵列核心包含了256x256个乘积累加运算器，可在一个一个周期内完成64k次乘加运算。作为巨头之一的英特尔公司也在边缘智能领域紧追不舍，于2017年收购了Mobileye，并开发了自动驾驶汽车的专用芯片，其量产的EyeQ4性能为2TOPS，功率为6W，而2020年量产的EyeQ5性能更是达到了12TOPS，功率仅为5W。此外英特尔也在FPGA领域做了很多前沿创新研究工作，以便于更好地用于AI。例如采用嵌入式多芯片互连桥接（Embedded Multi-Die Interconnect Bridge,EMIB）[56]技术在FPGA周围组合多个人工智能芯粒（如下图 所示），封装之后就可以作为很多应用的AI引擎。而微软公司推出的用于加速DNN推理的Brainwave项目也打破常规，没有选择ASIC而选用了FPGA来实现，不仅可以快速开发后重新映射到FPGA，还可以与迭代迅速的AI算法的要求保持同步。该系统的核心是一个高性能、精度自适应的FPGA软处理器，性能可达39.5TOPS。

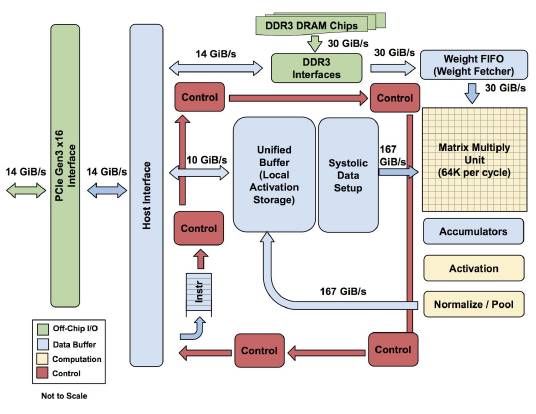


图 谷歌TOPv1结构框图[43]

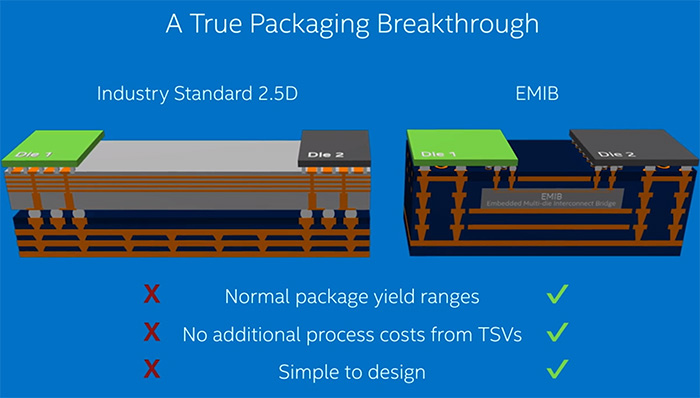
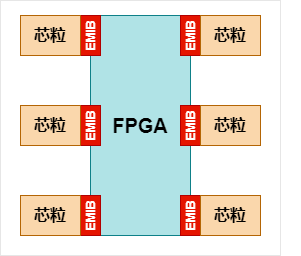
 

图 EMIB技术以及采用EMIB技术将FPGA与芯粒组合在一起[56]

除了上述几个巨头公司的突出贡献外，国内外学术界也有众多独创性和前瞻性的研究成果。例如麻省理工学院研究团队针对移动端设备设计的低功耗图像识别芯片：Eyeriss（发布于2016年）和Eyeriss2（发布于2018年）[57]，架构可见下图。两款芯片都采用了可重构架构，针对不同的CNN优化了包括加速器芯片和片外存储在内的整个芯片系统。其中Eyeriss处理卷积层的速度为35帧每秒，功率为278mW，不及移动GPU功率的十分之一，在台积电65nm工艺下，芯片面积也仅有3.5mm2。Eyeriss2在Eyeriss的基础上引入了非常灵活的分层网格网络，可适用于不同类型数据的复用和带宽需求，进而提高了资源利用率，此外Eyeriss2还可以利用稀疏模型提高处理能效和速度，运行速度可达到Eyeriss的12.6倍，而功耗却仅为其1/4。

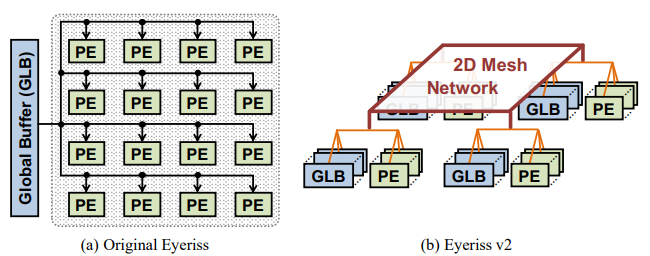


图 Eyeriss和Eyeriss2架构的比较（图a为Eyeriss，图b为Eyeriss2）[57]

近乎同时于麻省理工学院的Eyeriss系列，比利时天主教鲁汶大学分别在2016年和2017年推出了Envision芯片的v1和v2版本[58]，这两款芯片采用了当时所有最先进的优化技术，如网络压缩、降低精度、网络稀疏性等。Envision采用了28nmFDSOI（具有体偏压技术）工艺技术，通过调节体偏压，再度提高了能效。与其他的AI芯片相比，Envision的面积与成本非常具有吸引力，仅为1.87mm2，非常适合在物联网、带AR功能的可穿戴设备等方向中应用。该款芯片在1V电源电压下可以运行到200Mhz，性能可达75GOPS，且最大限度地降低了卷积层的能耗，在人脸识别任务中功耗为 6.4mW的情况下，每帧图像耗能仅为6.2uJ，相比其他芯片，可节省97.5%的能耗。

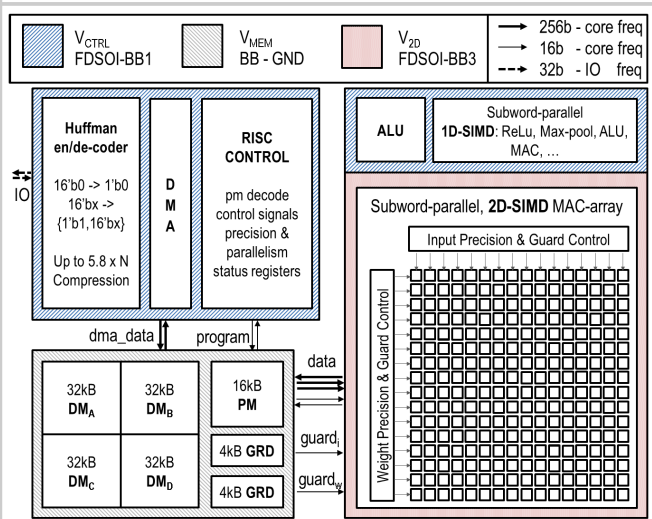


图 Envision的顶层框架[58]

虽然利用稀疏性等技术可以提高速度，但是DRAM的读取耗时常常是PE计算耗时的两个数量级，因此斯坦福大学在2016年提出了“深度压缩”方法并研发了“高能效推理引擎”（Efficient Inference Engine, EIE）芯片[59]，通过修剪冗余连接和多个连接共享权重，使得大型DNN（如VGG和AlexNet等）可以完全运行于片上SRAM，此举让功耗降低至原来的1/120，处理能力更是达到了102GOPS。EIE芯片目前主要应用于自动翻译，以每秒1.88x104帧速度处理AlexNet全连接层时，功率为600mW，仅为GPU的1/3400、CPU的1/24000。

图示

描述已自动生成

图 基于压缩DNN的高效推理引擎（EIE）[59]

虽然上述这些深度学习加速器的商业价值和学术价值都很高，但是我们也发现研发一款功能完备、高性能、高能效、低能耗的芯片的研发周期长、成本昂贵和研发难度大等问题。一款芯片从立项到落地，需要一个乃至多个团队少则一两年，多则三五年的筹备和研发，其中的开销更是不计其数，为了解决这一问题，现在很多高校研究团队也在针对深度学习加速器研究特定的生成器[60-65]，顾名思义，即采用高参数化、模块化的硬件生成器去实现满足需求的深度学习加速器，极大地缩短了研发周期，降低了成本。例如TVM团队研发的，被称之为“开源深度学习芯片栈”的开源项目VTA[60]， R. Venkatesan等人在2019年提出的模块化的神经网络加速器生成器MAGNet[61]， 以及美国伊利诺伊大学、IBM中国研究院等联合提出的基于FPGA的DNN推理加速器DNNBuilder[63]。DNNBuilder项目只需要设计、生成、执行三步即可自动生成一款基于FPGA的高性能的DNN推理加速器，可以快捷地部署到云端或者边缘端的FPGA上。这类加速器生成器的研究还有很多，从结构上分析，总体来说可分为两类，一种是采用脉动阵列的计算方式（类似于谷歌的TPU[43]），一种是采用并行向量的计算方式（类似于Brainwave[66]和NVDIA[67]）。

上述众多的加速器生成器虽然都可以帮助用户轻松的例化生成一个DNN加速器，但是离广泛应用还有一定的距离，因为这些生成器只考虑到了加速模块本身，而没有从系统级去考量，这就导致生成的硬件加速器缺乏一个全栈式的编程接口来对加速器施加控制，在SoC集成过程也十分困难。为此伯克利大学的Hasan Genc等人推出一款“全栈式”软硬件协同的加速器生成器——Gemmini[68,69],该项目在Github上开源，不仅拥有一套灵活可配的高参数化、模块化的硬件模板，多层软件栈，还有一套完备的SoC集成环境，可以在FPGA或者ASIC上快速生成系统完备的DNN加速器电路。与其他的加速器相比，Gemmini还支持多数据流可配、易于编程（保证加速器可应用与多类应用场景）、与RISC-V环境完美兼容等众多优势。Gemmini生成的加速器现已成功在台积电16nm FinFET和英特尔22nm FinFET Low Power（22FFL）工艺下流片，从物理层证明了其可实现性，经过测试Gemmini加速器在DNN加速任务中的速度可达到CPU的2670倍，与当下最先进的商用加速器已相差无几。下表 中从软硬件和系统支持等层面对当下最先进的几款深度学习加速器生成器的性能作出对比。

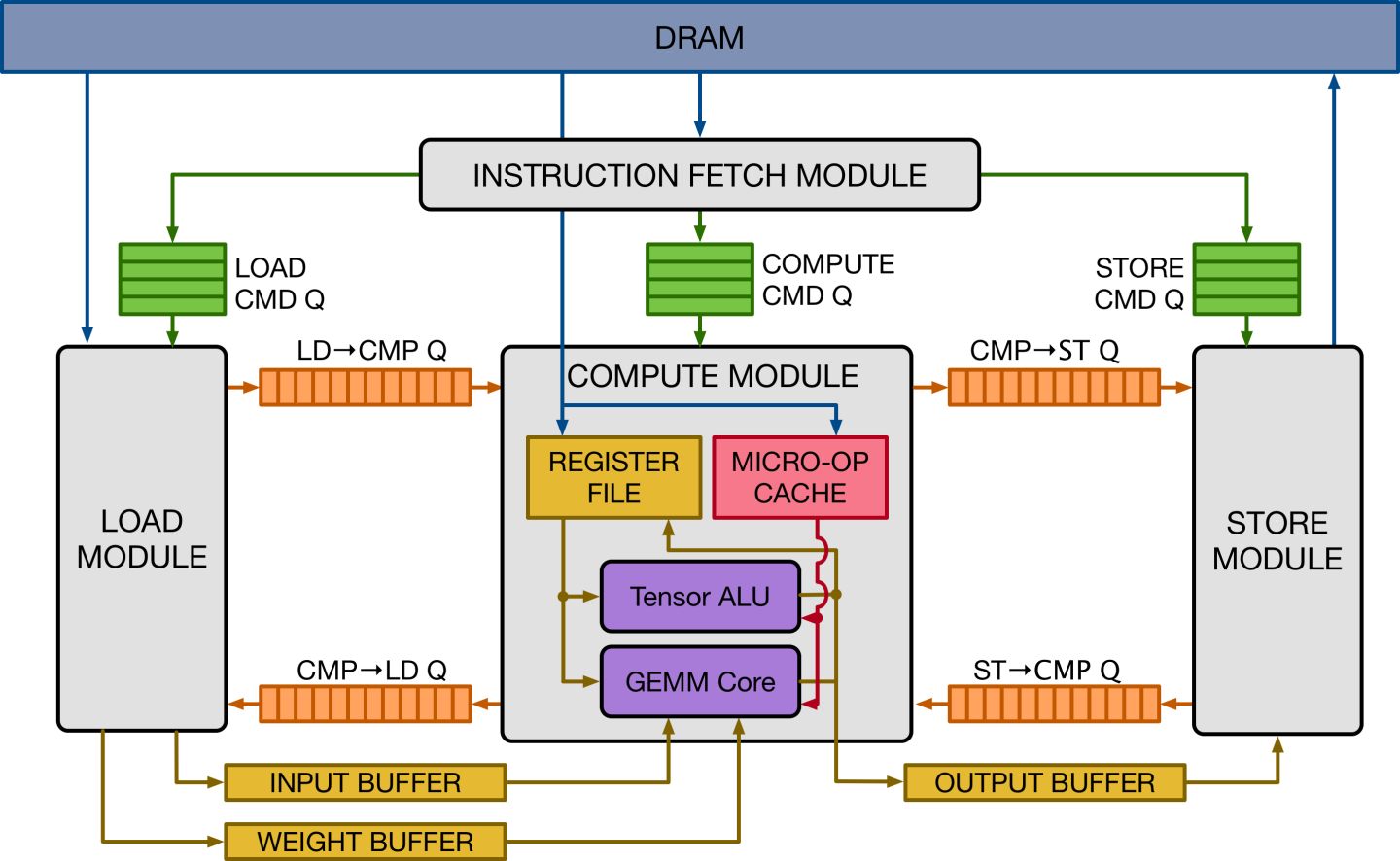


图 VTA架构[60]

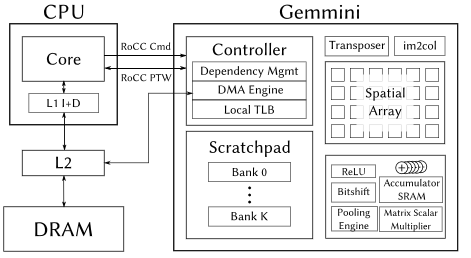


图 Gemmini整体架构[68]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 属性 | NVDLA | VTA | DNNBuilder | DNNWeaver | Gemmini |
| 硬件架构支持 | 数据类型 | Int/Float | Int | Int | Int | Int/Float |
| 数据流 | × | × | √ | √ | √ |
| 空间阵列 | vector | vector | systolic | vector | Vector/systolic |
| 直接卷积 | √ | × | √ | √ | √ |
| 编程支持 | 软件生态 | Complier | TVM | Caffe | Caffe | ONNX/C |
| 虚拟内存 | × | × | √ | × | √ |
| 系统支持 | 全SoC | × | × | × | × | √ |
| OS支持 | √ | √ | × | × | √ |

表 各类加速器生成器性能对比表[68]

近年来边缘侧智能设备的市场发展势头正盛，学术界的研究也百花齐放，但是该方向的设计还面临许多艰巨的挑战，且很多算法本身的效率还存在极大的提升空间。为了达到高效能、低功耗，大家都试图在新型的硬件架构上实现推理任务，有些公司甚至直接采用了新的计算范式，例如存内计算、模拟计算等等。我们相信边缘智能生态必将进一步完善，各类应用也将进入爆发阶段。

## 本文主要研究内容

本文以将深度神经网络部署到高能效、低功耗的边缘智能设备中为目的，致力于解决深度卷积神经网络部署到边缘端设备时计算资源、存储资源不足，能效低等问题，结合软硬件协同设计思想，研究并设计了二值复数神经网络的专用加速器。与传统的算法专用加速电路不同，本次设计的专用加速器，不仅包括了针对二值复数神经网络加速任务的协处理器，还将其集成到一块完整的SoC中（其中包含RISC-V核Rocket,DRAM以及数据Cache等），并在此基础上实现了专用于协处理器的扩展指令，实现了“全栈式”的软硬件协同。

后续章节首先从卷积神经网络层面进行分析，研究卷积神经网络的相关概念，并以“权重固定型”（weight-stationary）数据流的脉动阵列为例说明如何从硬件角度实现卷积神经网络的加速。之后从二值复数神经网络的算法层面着手分析，研究算法本身的并行结构、数据特点，以及二值复数的计算、象限二值化、复数批标准化、二值化等概念，其中首要在于分析二值复数神经网络的具体模型以及各层功能与具体实现，并基于“软硬件协同”设计理念分别提出硬件实现方案。

在有了上述算法研究的基础上，我们将设计出适用于各类二值卷积神经网络算法的协处理器。基于“软硬件协同”思想，我们首先构思了协处理器的指令集架构，但是为了更好的说明该指令集架构，我们在文章中先对硬件设计进行说明。在硬件层面我们首先设计协处理器的数据通路，并最大限度地提高数据通路的并行性；其次是针对协处理器的内存与数据存储方式设计，我们创新性的提出适用于二值复数和脉动阵列的类NHWC并行存储方式，极大地减少了数据访问次数和提升了数据计算的并行性；紧接着是针对计算核心脉动阵列的设计，基于谷歌TPU中的脉动阵列和上述的存储方式，我们提出了脉动阵列阈值单元（Systolic-Array-Threshold Unit，SATU），SATU的设计借鉴了Yaman Umuroglu等人提出FINN框架中的“矩阵向量阈值单元（Matrix–Vector–Threshold Unit，MVTU）”，不仅将整数或浮点数的乘加运算同步为多比特并行的二值复数的xor-popcount运算，避免了复杂的乘法运算，最大限度的降低了硬件开销，且利用多比特并行特性，可在单次运算中完成32个通道（其中16个为复数的实部通道，另外16个为复数的虚部通道），最大化提高了设计的能效；而且在精度损失可忽略的条件下，使用多个比较器替代了复杂的batch normalizatio计算，解决了二值复数神经网络中BN层计算复杂的问题[50]。s

与其他神经网络加速器设计不同，在完成上述的协处理器设计后，我们又将协处理器与主处理器在RISC-V环境下进行集成，并提出了面向二值复数神经网络的加速器的系统硬件整体架构，而后针对完成的SOC系统进行软件设计，即开发相应的C函数库，在此函数库的基础上我们进而完成了二值复数神经网络在Zero\_DCE++应用的软件实现。

将上一步使用C语言完成的软件算法编译并在我们设计的专用加速器上运行，这一步也相当于完成了加速器的功能仿真验证。完成相应的功能仿真后，在FPGA板上实现加速器硬件电路，并进行资源评估。

## 本文的章节安排

本论文利用“软硬件协同”设计思想，设计了面向二值复数神经网络的专用加速器系统电路和底层函数库，在软硬件的基础上实现了Zero\_DCE++算法。

以下是本文的章节结构安排：

1. 绪论：本章主要通过阐述二值神经网络和复数神经网络的研究背景和研究现状来引入对二值复数神经网络的研究，并结合对边缘端加速设备的研究提出了本次课题的研究内容，最后介绍了本文的章节安排和主要内容。
2. 本章主要为卷积神经网络和二值复数神经网络的理论概述，我们首先阐述了卷积神经网络的基本原理和脉动阵列加速神经网络原理，其次我们重点说明了二值复数神经网络的组成、二值化原理和二值复数计算原理，并介绍了矩阵向量阈值单元，为后面的硬件设计铺垫。
3. 本章主要针对二值复数神经网络设计了专用的协处理器，我们首先提出了本次协处理器设计架构，并基于RISC-V扩展指令设计了协处理器指令集，之后结合了脉动阵列和MVTU设计了面向二值复数神经网络计算的SATU（脉动阵列阈值计算单元），之后针对二值复数的数据特点，我们提出并设计了“row-addressed”型片上存储系统和新型的类NHWC存储方式。
4. 在第三章协处理器设计的基础上，本章进一步实现了二值复数神经网络的专用加速器的系统设计。为此，我们首先介绍了RISC-V开源环境，包括开源处理器Rocket等；之后我们介绍了专用加速器的系统硬件架构设计，并介绍了系统的多级分布式存储系统和RoCC接口；在指令集架构的基础上，我们又完成了加速器软件系统的设计并通过C语言从指令集层面实现Zero\_DCE++算法的二值复数卷积运算部分。
5. 本章主要是对前面实现的专用加速器进行功能仿真与性能评估。为此我们搭建了专用加速器系统级的RTL验证平台，首先从指令集层面验证了加速器协处理器的功能，之后又从系统层面上实现Zero\_DCE++算法，并与GPU中运行结果对比，证明了本次设计的专用加速器可以准确的完成二值复数神经网络的加速任务；之后在FPGA板上实现本次设计，并进行性能与资源评估。
6. 本章中我们总结了本次工作的研究内容和创新点，回顾了工作的研究意义，并对后续设计的完善与迭代做出了展望。

# CNN与BCNN算法概述

## 引言

卷积神经网络（Convolutional Neural Network, CNN）作为深度学习领域最具代表性的网络模型，由于其较强的特征提取和分类识别的能力，在图像处理领域应用极其广泛。1998年，杨立昆在其LeNet5的文章中[70]首次提出CNN概念，但是受限于当时硬件算力的不足和神经网络无法实现稳定训练等问题，直至2012年之后才逐步推广。而目前现有的大型数据中心虽然基本可以满足当下最复杂的CNN的计算任务，但是当应用场景推向边缘端时，算力就明显不足了。

为此人工智能算法工作者提出了众多解决方案，例如采用无监督学习算法、优化深度神经网络结构（例如模型裁剪、核稀疏化、参数量化加速等）等。其中包括Y. Bengio等人提出的BNN，通过将激活值、权重值都压缩到1比特，并提出相应的popcount(xnor)计算方式替代全精度浮点数的乘加运算来实现卷积运算，大大降低了内存和算力的需求，且具有极高的硬件友好性，方便了神经网络模型在资源受限的边缘设备上部署，但BNN因其精度损失也常为人们所诟病。而后Yanfei Li等人在DCN和BNN工作的启发下，系统的结合了DCN和BNN的优势，进而提出了BCNN。BCNN将激活值和权重值都使用二值复数来表示，利用复数的更强的表达能力降低了二值网络的精度损失，还保留了BNN网络低硬件开销、低功耗、实时性和硬件友好性，对于边缘智能设备来说，非常具有吸引力。

面向上述简介，本章主要介绍了卷积神经网络的概念及其硬件加速原理、BCNN算法及其计算原理，为后续的工作奠定理论基础和设计依据。

## 卷积神经网络及其硬件加速原理

本小节首先对卷积神经网络的原理进行概述，介绍了卷积神经网络的主要结构组成，并详细的说明了其中最核心的卷积层；紧接着我们又以”权重固定“型脉动阵列为例详细说明卷积神经网络的硬件加速原理。

### 卷积神经网络原理概述

针对特定的应用场景提出一个CNN一般需要三个步骤：层次结构定义、网络模型训练、推理。其中CNN的层次结构通常可以分为输入层、卷积层、池化层、全连接层和输出层，按照不同应用场景需求对这些主要层进行合理的排列组合和复用，再拼接或者叠加，即可生成特定的卷积神经网络。网络层次结构定义完成后，就需要开始对网络进行训练，获得最优权重和其他需要的参数，这一部分计算量最大、耗时最久，一般在数据中心（云端）完成。训练完成后得到的最优网络模型就可以对输入数据进行推理计算，即完成相应的特征提取和分类识别任务。

如图所示，以最经典的LeNet5为例介绍CNN的网络结构。LeNet5的网络结构很精简，仅有一个输入层、两个卷积层、两个池化层、两个全连接层和一个输出层，结构顺序如下：输入层→第一个卷积层→第一个池化层→第二个卷积层→第二个池化层→第一个全连接层→第二个全连接层→输出层。LeNet5是典型的前馈神经网络，每个功能层（如池化层、卷积层等）都有一个输入特征图和输出特征图，输入数据在通过输入层进行预处理后依次通过各层进行处理，最后经全连接层计算并将结果给到输出层，在此之后发展的各类CNN也大都在此基础上衍生的。下面主要介绍一下卷积层和激活函数层。

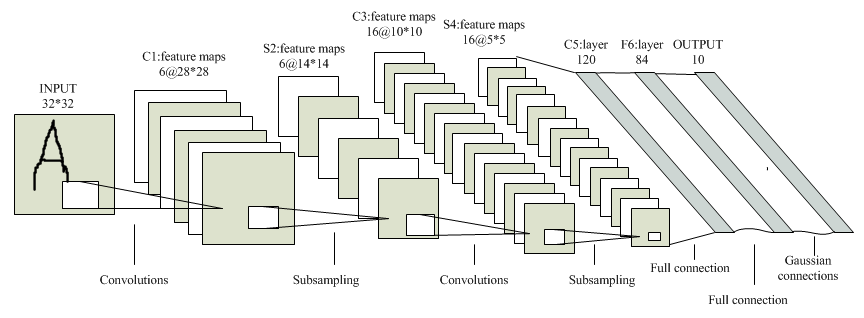


图 LeNet5网络结构[70]

1. 卷积层处理

卷积层作为卷积神经网络中运算的核心层，通过多次卷积计算来提取输入特征图的关键特征，并生成输出特征图。如下图所示，卷积层通常采用二维卷积计算，以每个像素点为中心，对其邻域进行卷积计算，即加权求和并加上偏置后输出结果。加权求和过程中的权重来自于卷积核，不同卷积核会提取不同的特征信息，如有的卷积核对图像的形状信息敏感，有的卷积核对颜色信息敏感，因此一个CNN中通常包含多个卷积核。

卷积计算就是对输入特征图进行补零（Padding）后，让卷积核的权重矩阵在输入特征图中滑动，每次取一个与权重矩阵相同尺寸的子矩阵与其对应的元素进行点积计算（相乘并累加）。如下图所示，首先对输入特征图进行补零操作，然后在补零后的特征图上的“左上角”取一个尺寸3×3的特征子矩阵（与卷积核尺寸相同），并与卷积核对应的元素进行点积运算，即，在加上偏置即可得到输入特征图的第一个元素为-2。之后进行行滑动，即在输入特征图中右移两个步长，取另一个特征子矩阵并进行点积运算得到第二个元素值-2，依次类推完成一行的数据计算后再计算下一行的数据，直至一面特征图计算完成。特别指出对输入特征图进行补零后不仅可以保证输出特征图和输入特征图大小一致，还可以有效地保护特征图的边缘特征信息。

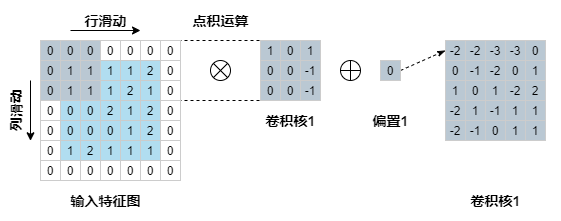


图 卷积计算示意图

而整个卷积层的计算中包括对所有的输入特征图（激活）、卷积核（权重）和输出特征图（输出）等的点积计算，不单单是上图 中所示的2维计算，而是达到了7个维度的计算空间，且卷积层还需要为卷积核数据重用提供平移不变性，如下图 所示，图中R和S表示卷积核的高和宽；C表示卷积核或者输入特征图的通道数；X和Y以及X’和Y’分别表示输入和输出特征图的宽和高；K表示卷积核数量或者输出特征图的通道数；B表示批量大小。。

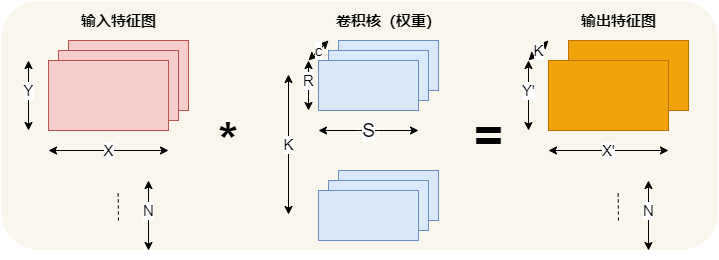


图 卷积层计算示意图（7维计算空间：R×S×X×Y×C×K×N）

1. 激活函数层。

在完成卷积计算并输出输出特征图后，此时的输出特征图的特征值还需要进行激活处理，理想情况下是通过阶跃函数完成，但由于其不光滑且不连续，现实中很难实现，所以一般通过sigmoid函数、ReLU函数、Leaky ReLU函数、tanh函数等激活函数来近似阶跃函数。激活函数可以对输出特征值进行过滤，保证信息的有效性，并且在神经网络中引入了非线性因素，增强了CNN对非线性特征的拟合能力。

### 脉动阵列加速神经网络原理

本段主要通过介绍谷歌公司的张量处理器（TPU）中脉动阵列的计算方式，来说明硬件加速神经网络计算的原理。脉动阵列将卷积计算转换为矩阵乘法运算，并采用并行结构和数据复用等技术，不仅加速了神经网络的计算，也减少了计算单元对带宽的需求。如下图所示，脉动阵列的核心部分是一个二维3×3的计算单元（processing elements，PE）阵列，其中每一个PE都可以在一个内完成一次乘加运算，且每个PE仅与相邻的PE通过横向或者纵向的数据通路实现数据传递。

以图 中卷积计算为例，下面采用卷积核权重固定，输入特征值横向传播，部分和纵向传播的方式为例，来解释脉动阵列实现卷积加速的原理。如图所示，首先将卷积核的权重矩阵静态地存储在脉动阵列的PE中，然后将输入特征图子矩阵按行展开，每一行都错开一个时钟周期，并等待依次进入脉动阵列。

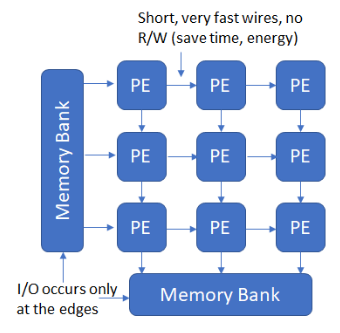


图 3×3脉动阵列结构示意图

开始计算的第一个周期，第一个子矩阵的第一个元素首先进入脉动阵列并计算得到第一个部分和；继续第二个周期子矩阵元素向右滑动，部分和向下滑动，计算得到第二个部分和。以此类推，输入的输入特征图子矩阵沿着脉动阵列的行方向水平移动，不断开始与不同卷积核的点积运算，并得到对应的部分和；而每个PE得到部分和都沿着脉动阵列的列反向垂直移动，并于下一个PE在下一周期计算的部分和累加，直至完成一个卷积核的运算得到一个输出特征图的元素。如下图脉动阵列计算的第9个周期，得到这一个卷积结果后，会将该结果送往存储单元中，等待下一次的计算处理。

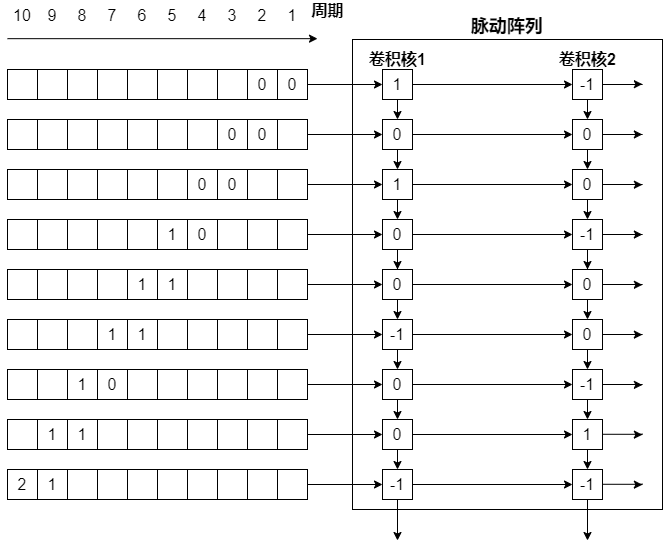


图 脉动阵列计算第0个周期

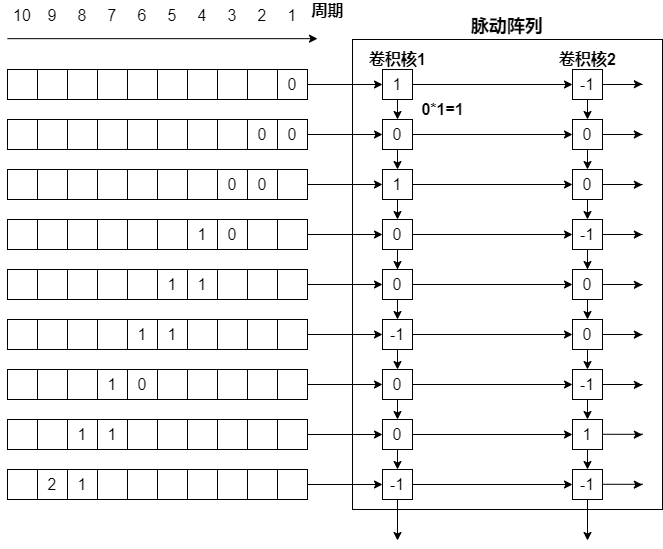


图 脉动阵列计算第1个周期

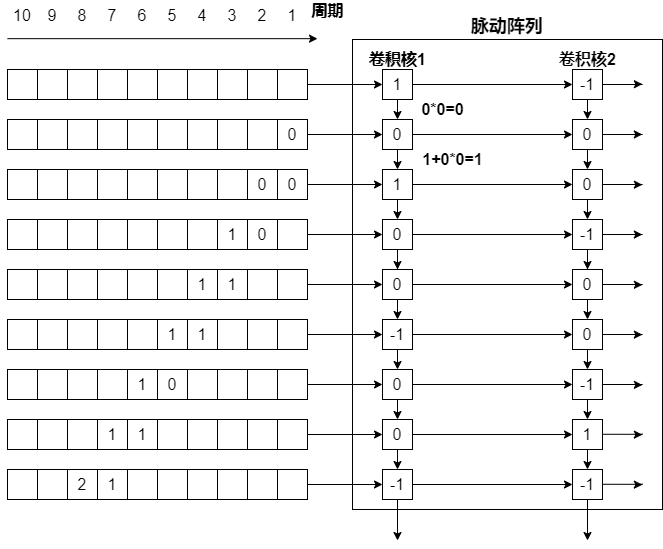


图 脉动阵列计算第2个周期

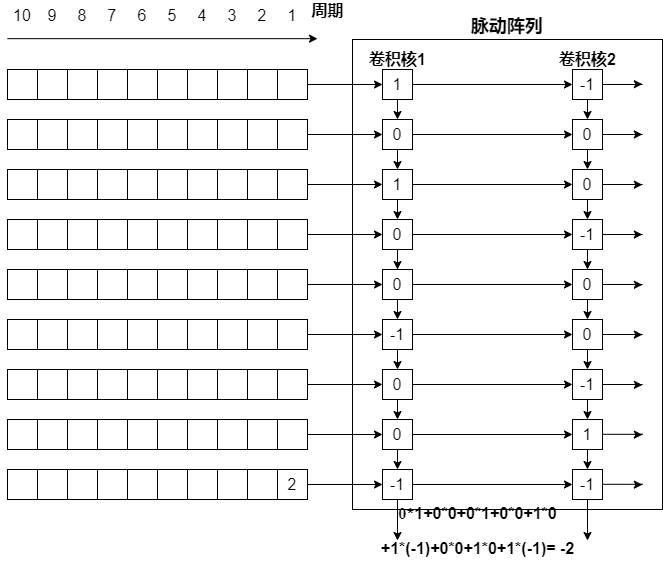


图 脉动阵列计算第9个周期

## BCNN算法概述

本节主要介绍BCNN算法结构及其相关基础理论，首先我们对二值复数进行定义，并讨论了二值复数如何进行点积计算；其次我们又说明了复数的二值化过程以及复数的批标准化（Batch Normalization，BN）；基于上述理论我们提出了BCNN算法的网络结构；最后我们介绍了FINN中提出的矩阵向量阈值单元（MVTU），并提出采用了MVTU后的BCNN算法的网络结构。

### 二值复数及其点积计算

与复数相同，一个二值复数也是由实部和虚部组成，定义其形式为，其中，均属于{+1，-1}，因此仅存在四种可能值：{-1 - i，-1 + i，1 - i，1 + i}。

下面对二值复数的点积运算进行介绍，假设二值复数为激活输入，二值复数为权重，是全精度复数输出。偏置一般也是一个全精度复数，但是此处为了简洁，我们在讨论中忽略偏置的作用。二值复数的点积运算遵循复数计算规则，形式如下式2-1，其中，，， ∈ {+1，-1}。

(2-1)

和BNN的二值点积运算相比，一个BCNN点积运算需要结合4个二值点积和2个额外的实数加法。利用矩阵运算表示，形式如下式2-2：

(2-2)

### 象限二值化

在BNN算法中，二值化是将一个全精度实数转换为一个二值数（即+1或者-1）的过程，这一过程通常可视为一个针对BNN的非线性激活函数。二值化可以通过“确定”和“随机”两个方式[16,45]实现。随机的实现方式可以提供一个更高的精度，但是在实现时却开销很大，而确定的实现方式仅需要一个简单的sign函数（如下式2-3），所以在实际应用中通常选择确定的实现方式。

(2-3)

因为sign函数在0点出不可微分，且梯度恒为0，所以直接做后向传播是不可行的。所以在BNN的研究过程中，研究人员提出了直通式估算器（Straight-Through Estimator，STE）方法来做后向传播，如下式2-4：

(2-4)

其中r是全精度实数输入，是二值输出，Loss是损失函数值，是一个剪裁的阈值（通常设置为1）。sign函数的梯度一般直接设置为一个恒等函数，当输入值变得极大时，这个阈值会将梯度取消，这对算法的优化过程很有用处。

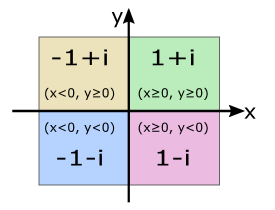


图 BCNN算法中的象限二值化[15]

对于二值复数而言，二值化是将一个复数转换到二值复数的形式（{-1 - i，-1 + i，1 - i，1 + i}中的某一个值）。为此我们提出象限二值化的概念，如图 所示，即象限二值化的输出仅取决于输入的复数位于二维笛卡尔坐标系的哪一个象限。从数学上分析，复平面是由实轴 x 和正交虚轴 y 确定的复数的几何表示，其中两个轴将平面划分为四个象限，每个象限由两个半轴界定。给定四个象限和四个二值复数值，这就很自然地可以将每个象限和某个二值复数值联系在一起。

从本质上来说象限二值化解耦了复数的实部和虚部，所以可以将两个部分作为普通二值化单独处理。对于前向传播来说，二值化的过程如下式2-5：

(2-5)

对于后向传播来说，可以通过两个STEs来得到二值化的梯度，并应用在两个独立的全精度变量x和y上，如下式2-6。这样做保证了二值化过程的简单性，进而方便了硬件实现和内存存储。且为了提高精度，现有工作还提出了二值化函数的各种变体，例如缩放因子[29-33]、近似sign函数[33]等等。但是根据文章[38]，这些变体函数的使用并没有显著的精度提升。

(2-6)

与之前的深度学习算法每个神经元采用32比特的全精度浮点数表示相比，BNNs中每个神经元仅采用1比特表示，不仅将内存和带宽降低了32倍，还极大的提高了计算效率。因为原先DNN中32比特或者64比特的点积运算，现在只需要一个单比特的异或(xnor）运算加上一个“记1”（popc）运算就可以实现（如下式2-7），计算效率至少提高了10-16倍：

(2-7)

popc(xnor)运算的提出，极大地提高了BNN和BCNN算法的硬件友好型，如下式2-8可见，此时一个BCNN的点积运算可以通过4个BNN点积运算（popc-xnor）外加两个全精度加法来实现。而且从硬件角度来说，这四个popc(xnor)运算可以并行执行，所以理论上来说完成一个BCNN点积运算的时间约等于一个BNN点积运算的时间。

 (2-8)

### 复数高斯批标准化（CGBN)

为了加快训练的收敛速度和提高训练的精度，深度学习算法的研究人员提出了Batch normalization（BN)[28]这一技术。在全精度的DNN算法中，首先需要使用BN将输入数据归一化在均值为0、方差为1的正态分布中。然后再通过一个可学习的增益参数来调整归一化后的输入数据，并加上一个可学习的偏置，如下式2-9，其中r是输入数据，是这一批数据的均值，是这一批数据的方差，是可学习的增益参数，是可学习的偏置参数，是一个极小数（避免分母为0）。

(2-9)

对BNNs而言BN层更是至关重要。BN层不仅可以对输入数据进行归一化，且通过学习得到的增益参数和偏置还极大地增加了模型的容量，提高了BNN的学习能力。如果没有BN层，BNNs的训练甚至不会收敛[45]。

与上式2-9不同，将复数输入批标准化为正态复数分布要更加复杂，因为除了要对均值和方差进行归一化外，复数神经网络中的BN还需要确保实部和虚部的方差相等。在DCN中，复数的BN会被当作一个二维的白化变换——通过它们沿实部和虚部的方差的平方根来缩放复数输入。这个可以通过以 0 为中心的数据乘以协方差矩阵的逆平方根来实现的，如下式2-10，其中z是复数输入，E(z)是z的均值，V是2×2的协方差矩阵。增益参数是一个 2×2的正半定矩阵，且具有三个自由度（和相等)；偏置参数也是一个复数。，和都初始化为0，和都初始化为。

从公式2-10中，我们发现复数BN引入了矩阵求逆和矩阵求均方根计算，这些对于硬件来说开销是很大的。除此之外，直接使用这个复数BN会导致训练精度低，甚至无法收敛等问题。所以BCNN研究人员提出了新型的BN技术——复数高斯批标准化（Complex Gaussian Batch Normalization，CGBN)。

(2-10)

我们的目标是将输入复数信号归一化为标准复数正态分布 (CN) 。 标准复正态随机变量，也称为标准复高斯随机变量，是一个复随机变量z，其实部和虚部都是独立的正态分布随机变量，均值为0，方差为1/2。 在数学形式中，意味着：

因此，我们可以将输入复信号的实部和虚部分别归一化为均值为零且方差为 1/2 的正态分布,如下公式2-11所示:

(2-11)

且增益参数和偏置参数是可学习的复数值，所以CGBN如下公式2-12所示：其中增益参数和偏置参数都是在训练中学习得来的，被初始化为，被初始化为。

 (2-12)

与公式2-10 相比，CGBN通过避免计算协方差矩阵的逆平方根，显着降低了计算复杂度。并且实验数据表明，使用CGBN可以获得更快的收敛速度[15]。

### BCNN的网络结构

下图 对比了BCNN和原始CNN的网络结构，其中图a表示原始的CNN的网络结构[28]，由卷积层、偏置层、非线性激活函数层和池化层按顺序组成；而BCNN的网络结构不同于原始的CNN的，如图 的图b所示 ，卷积层之后紧跟着的是池化层和BN层，在BCNN中偏置层可以删除且不造成精度损失，最后是二值化层。对于BCNN来说，要想模型可收敛，BN层是必须存在的。

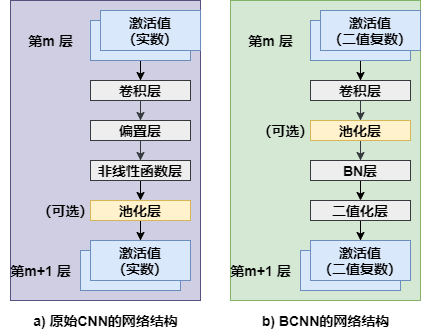


图 BCNN与原始CNN网络结构对比[28]

（图a为原始CNN的网络结构，图b为BCNN的网络结构）

对于具有三通道（RGB)作为输入的图像，初始输入仅包含实部。 为了生成复数输入，BCNN需要一个两层残差卷积神经网络来学习得到虚部。 生成复数输入的网络如图 所示。

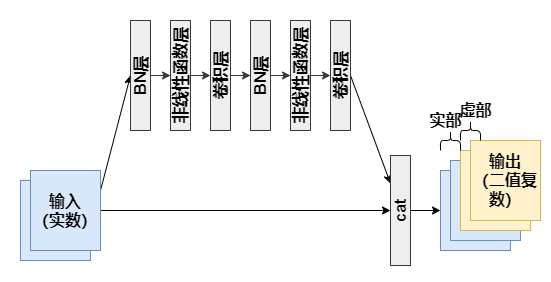


图 复数虚部生成层[28]

### 矩阵向量阈值单元（MVTU）

矩阵向量阈值单元（Matrix-Vector-Threshold Unit，MVTU）由Yaman Umuroglu等在文章[50]中提出，旨在通过MVTU单元实现BCNN结构中卷积层、池化层、BN层与二值化层的功能，其中卷积层由矩阵向量乘加运算完成，池化层通过布尔运算中的或运算完成，BN层和二值化层通过阈值比较完成。特别指出，采用阈值比较法代替原先的BN层和二值化层，可以在精度几乎无损失的前提下，极大地降低了硬件开销，文献中的数据表明，对于一个16比特的点积输出值，使用FPGA实现原先的BN层和sign函数（二值化层）需要2个DSP，55个FF和40LUT，而使用阈值比较法只需要6个LUT。下面对阈值比较法做出详细解释：

假设是神经元k的卷积层输出，是该神经元在训练中学习得到BN的相关参数。所需要的BN-Sign的输出由公式2-13、2-14可得。

(2-13)

(2-14)

(2-15)

下图 以三个神经元A、B、C为例表明了卷积输出和BN-Sign的输出的关系，根据参数值，绘图可能向左或向右移动，或水平翻转，但输出激活值变化的阈值点始终存在。计算等式2-15可得。为了使得阈值恒为正值便于计算，可以将计算得到的阈值与神经元的扇入S求和后再平均获得。通过该技术，我们仅使用一个无符号数的比较就避免了在神经网络的推理过程中进行复杂的BN层计算。

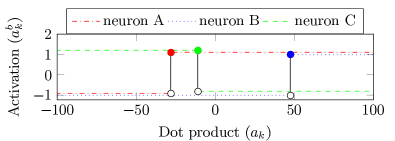


图 三个采用BN的二值神经网络的输入与输出，

为了更好的说明，输出上稍加了偏置[50]

采用了阈值比较法的MVTU结构如下图 所示，MVTU的内部含有一个输入和输出缓冲器和一个计算单元（Process Element，PE）阵列（每个PE都含有多个SIMD通道）。其中PE和SIMD通道的个数都是可配置的，配置参数由所需的吞吐量决定。卷积运算所需的权重矩阵保存分布在PE之间的片上存储（On-Chip Memory，OCM）中。每个PE每次接收完全相同的控制信号和输入向量数据，每个输入向量与矩阵向量中的不同部分分别进行乘加计算。且架构中的PE既是权重固定型（因为权重都固定保存在PE的存储中）也是输出固定型（因为每个popcount计算的结构也保存在PE的存储中）。

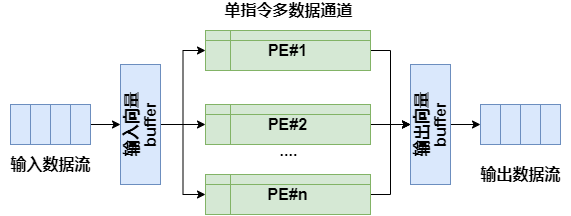


图 MVTU的整体框架图[50]

图 显示了 MVTU 中PE的数据路径。 它用于输入向量和权重矩阵的点积计算，并将结果与阈值进行比较，产生单比特输出。 点积计算本身是两个向量之间的乘法累加运算，可使用 XNOR 门实现。 随后，通过POPCOUNT计算XNOR的结果中“1”（也可以配置为“0”）的个数并添加到累加器寄存器中。 当完成所有点积计算的累加，就会对其进行阈值处理。 累加器、加法器和阈值存储器均为 T 位宽，可按比例缩小至 以节省更多资源，其中Y为该神经元输入的位宽。

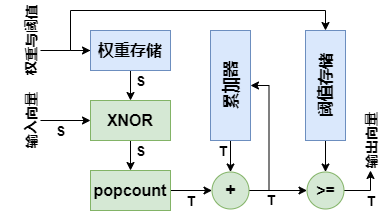


图 MVTU中PE的数据流示意图[50]

## 本章小结

本节主要介绍CNN及其硬件加速原理和BCNN结构及其相关基础理论，为后续的电路设计奠定了理论基础。首先我们阐述了卷积神经网络原理，并以脉动阵列为例介绍了卷积神经网络的硬件加速原理；其次我们对二值复数进行定义，并讨论了二值复数如何进行卷积计算；紧接着又说明了复数的二值化过程以及复数的BN；基于上述理论我们介绍了BCNN算法的网络结构；最后我们介绍了FINN中提出的矩阵向量阈值单元（MVTU）以及其中PE的结构。

# 面向BCNN算法的协处理器设计

## 引言

近些年前，基于深度神经网络的AI芯片（又被成为深度学习加速器）行业正在蓬勃发展。目前主流的AI芯片，主要用于云端（数据中心）和边缘测（智能手机、无人机、自动驾驶汽车等），面向不同应用场景的设计需求各不相同，设计的产品也是各异的，但设计方法和流程都大同小异。总的来说，一款深度学习加速器的设计流程一般分为五个步骤：基本思路、数据集、神经网络算法与架构、芯片架构设计、电路实现，如下图 所示。

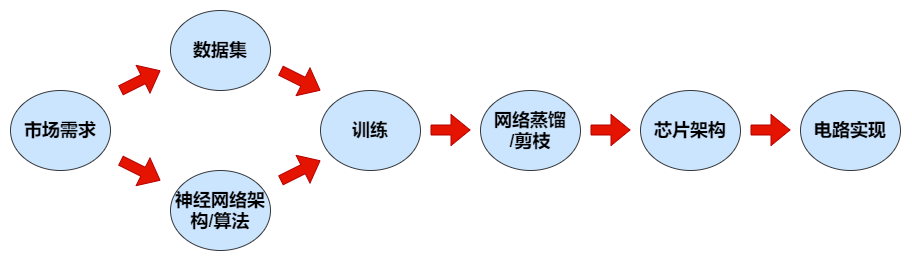


图 AI芯片的设计流程

其中基本思路阶段就是根据芯片的应用场景，对整个研发阶段进行一个统筹的规划，之后准备好数据集，并选择合适的神经网络架构和算法，前面这一部分都属于软件部分的工作，而后续的芯片架构设计和电路实现就是硬件部分的工作，这个流程就是一个标准的“软件定义硬件”流程。而在硬件电路的设计过程中，考虑到电路的实际应用，又不免需要采用“软硬件协同设计”的设计思想。由此可见，当下的深度学习加速器的设计是区别于传统的数字电路设计的，设计人员无法再独立地专注于算法或者硬件电路，而是需要统筹兼顾。

本文也是受此启发，从软硬件协同的角度提出了面向BCNN算法的人工智能加速器的解决方案，本章主要介绍人工智能加速器的协处理器部分的设计，主要包括协处理器设计的目标与整体架构、扩展指令集设计、脉动阵列阈值单元设计（SATU）、内存与数据存储方式设计。

## 协处理器设计架构

根据上述的AI芯片的设计流程，在确定了神经网络架构和算法后，就需要对加速器的硬件架构进行设计。芯片架构的设计就是将神经网络算法映射到硬件设计中，主要包括计算引擎设计和存储系统设计两个部分，往下又可划分为多个任务，例如计算单元的设计与优化、计算单元阵列以及网络拓扑的设计与优化、高效合理的存储系统设计、充分利用数据并行性和可重用性以及确定数据流（权重固定或输出固定等）等等。

经过第二章的铺垫后，我们已经对BCNN算法的架构和理论基础有了一定的了解。所以本节就针对BCNN算法和架构和特点，并借鉴了众多了现存的深度学习加速器，提出了一种新型的面向BCNN的专用加速器的系统架构。

### 设计架构

本次设计的BCNN加速器的协处理器采用了软硬件协同的设计思想，主要包括计算引擎、存储系统、数据通路以及各控制模块的设计。下图为本次设计的加速器的协处理器的整体架构框图，该协处理器的架构参考了Thierry等人设计的Versatile Tensor Accelerator（VTA）[60]，采用了访问执行解耦合（Decoupled Access/Execute）架构，这就意味着“内存访问”和“执行”指令可以在硬件的不同区域同时执行，该机制可以极大地提高指令执行效率。我们把硬件分解为三个控制模块，分别是负责处理“excute”指令的Execute Controller模块、负责处理“load”指令的Load Controller模块和负责处理“store"指令的Store Controller模块。



图 面向BCNN的深度学习加速器的协处理器的架构框图

除了这三个控制模块，本次设计的架构也有与VTA架构的不同之处，例如VTA架构中使用一个取指令模块（Instruction Fetch Module），直接与片外DRAM交互并取指令，这样的设计虽然简洁方便，但直接截断了与CPU集成的可能，无法将协处理器在边缘智能设备中实际应用。为此我们引用并设计了保留站（Reservation Station）模块，保留站技术是计算机科学中常用技术，主要用于解决连续进入指令流水线的指令之间的数据和资源的相关性问题。我们引入的保留站模块可以同时缓存多条指令，并监测不同类型的指令是否存在资源冲突，一旦检测到某一条指令与其他控制器中的指令间没有依赖关系，就将其送往对应的指令队列中，解决了相关性问题；除此之外，本次设计的保留站还含有RoCC接口，便于后续协处理器与主处理器的集成，极大提高了硬件的可实用性。

与VTA架构另一个不同之处在于，VTA架构中忽略了片上存储SRAM的使用，直接在DRAM进行数据的访存，极大的降低了硬件的执行效率且增加了功耗，而我们为了提高计算的执行效率和数据的复用性，在架构中添加了片上存储SRAM，其中包括简单的单端口SRAM组成的Databuffer、和双端口SRAM构成且拥有累加功能的Accumulator。且设计针对的BCNN算法，将每个像素的权重参数和输入数据压缩至2比特（实部和虚部各1比特），极大缩小了对存储空间的需求，在实现高吞吐量的同时降低了成本和功耗，也为我们将激活和权重数据迁移至片上SRAM存储提供了理论依据。

上文中我们对比了本次设计与VTA架构的两点不同，而在指令保留站完成对指令的预处理并将各指令按类别（如数据Load、数据Store、计算执行）分配到相应的队列后，各个模块会对队列中指令依次进行译码和执行，这与VTA架构是相同的。下面我们分别简要地介绍三个控制模块（Execute Controller模块、Load Controller模块、Store Controller模块）的结构和功能。

1. Execute Controller模块

该部分主要负责执行“execute”类型的ISA指令，例如矩阵乘法。该模块主要包含脉动阵列阈值单元（Systolic Array Threshold Unit，SATU）和循环展开单元（Loop Unroller Unit，LUU），其中循环展开单元用于处理长指令（CISC）并将其展开为多条精简指令（RISC），便于后续的计算执行；脉动阵列阈值单元作为本次设计的核心，参考自TPU中的脉动阵列和FINN中的矩阵向量阈值单元（MVTU），可同时完成DIM\*DIM个元素的点积运算（其中DIM指脉动阵列中每行或者每列计算单元的个数），且其中各个计算单元采用了简单的popcount(xnor)计算代替了复杂的乘加计算，极大地提高了数据的并行计算能力和硬件友好性，进而提高了计算效率；其中阈值单元使用DIM个比较器即可完成复杂的BN和二值化的过程，极大地降低了硬件开销。

虽然Execute Controller模块的功能比较复杂，但是其状态机十分简洁，只有“等待指令”、“preload数据”和“计算”三个状态。状态机的默认状态为“等待指令”状态，在收到数据预取指令后进入“preload数据”状态并预取数据，等待数据预取完成后又返回“等待指令”状态；再收到数据计算指令后，进入“计算”状态，通过SATU单元完成卷积计算和二值化过程。状态机设计之所以简单，是因为我们将复杂的逻辑功能直接映射成硬件电路。

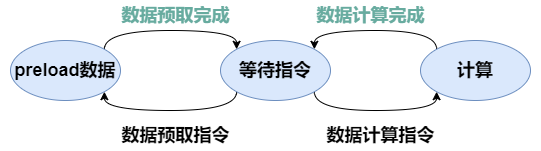


图 ExecuteController模块状态机

1. LoadController模块

该部分负责执行从DRAM向协处理器的Databuffer或者accumulator传输数据的指令。该模块主要包含一个DMA和数据传输的控制寄存器（如读取数据的行数、列数、步长、源地址、目的地址等）。特别指出，DMA可以对虚拟地址进行操作，从指令保留站中获取地址（虚拟地址）后会将其送入高速缓存（Translation Lookaside Buffer，TLB），经过映射得到DRAM的物理地址。如果在TLB未命中后，该虚拟地址将直接转回主处理器中的页表查找模块（Page Table Walker，PTW）来获取物理地址。

LoadController模块的功能也是由状态机控制，状态机的默认状态是“等待指令”状态，在该状态下可以完成对控制寄存器的配置；当存储器准备就绪后，如果DMA未有请求任务则进入“等待DMA请求”状态，如果DMA有请求任务则直接进入“load数据”状态，并通过DMA将DRAM中源地址对应的数据送到本地存储（Databuffer或者Accumulator）中目的地址对应的位置。直至最后一行数据传输完成，则返回“等待指令”状态。

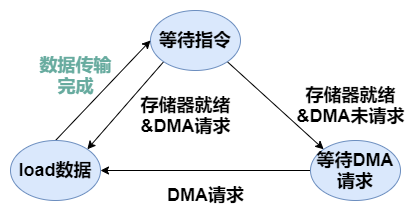


图 LoadController模块状态机

1. StoreController模块

该部分负责执行从协处理器的Databuffer或者Accumulator向DRAM传输数据的指令。结构和状态机都与LoadController模块类似，这里不再赘述。

## 指令集（ISA）设计

指令集指的是计算机中央处理器机器码所使用的指令的集合以及其背后的寄存器体系、总线设计等逻辑框架，作为软件设计和硬件设计的基础和纽带，指令集架构在整个设计过程中的地位可谓是举足轻重。常见的指令集架构大体可分为复杂指令集（CISC）和精简指令集（RISC）[71]，本次协处理器设计中涉及的指令集是依据RISC-V指令集架构中的扩展指令集设计的，且利用了单指令多数据（SIMD）的形式，主要包括三个类型的指令：数据移动指令、配置指令、计算指令。

### 数据移动指令

数据移动指令主要分为“mvin”和“mvout”两个，其中mvin指令主要负责将DRAM中的数据按照要求传输到协处理器中的片上存储中（Databuffer或者Accumulator），mvout指令则相反，主要负责将片上存储的数据传输到DRAM中。经过保留站处理后，mvin指令将送往Load Controller模块，mvout指令将送到Store Controller模块。

为了更好的说明数据移动指令，以下图mvin指令为例进行说明，该指令从DRAM中起始地址x出发，连续读取DIM个数据，作为一行数据送往SRAM中地址为Z的那一行；之后跳过步长mm\_stride直接到地址x+mm\_stride，并再次读取DIM个数据，送往SRAM中地址Z+1的那一行，以此类推，读取完设置的行数后，则说明该指令任务完成。

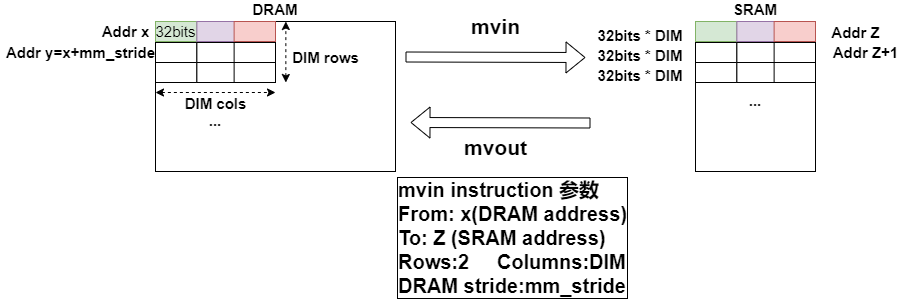


图 mvin指令的示意图

### 计算指令

计算指令，顾名思义，即完成一次卷积计算，包括对数据的移动和计算，这一过程功能较为复杂，如果使用一条指令来完成该功能，那指令长度势必非常长，且在译码过程中也会耗时很久，所以我们将该计算指令分为preload、compute.preloaded、compute.accumulated三个子指令，这三个子指令都将送往Execute Controller模块中依次执行。

以权重固定的脉动阵列为例，preload指令会提前加载特定数量的权重数据。compute.preloaded指令会将存储于SRAM中的输入激活值和偏置数据依次送往脉动阵列中PE的乘加计算输入端口，并与之前固定的权重数据完成popcount(xnor)计算，每列的计算结果经过不同的移位寄存器缓冲后进入阈值单元（也可以选择不通过阈值单元，例如BCNN算法中的倒数第二层输入是二值复数，而输出是全精度数），在完成阈值比较-拼接后，同步输出到Databuffer或者Accumulator中。computed.accumulated指令格式与compute.preloaded类似，不同的是在computed.accumulated前，不需要重新preload权重矩阵，而是与上一次矩阵运算preload的权重矩阵进行计算。

### 配置指令

配置指令是对上述两类有数据操作的指令进行参数配置，可分为config\_ex、config\_mvin、config\_mvout三种。其中config\_ex是对计算指令进行参数配置，包括是否需要激活函数、输入和权重矩阵是否转置、输入矩阵输入时步长等等，该指令将送往Execute Controller模块中。而config\_mvin和config\_mvout主要是配置数据在片外存储DRAM和片上存储SRAM传输时的步长等，将分别送往Load Controller模块和Store Controller模块进行译码与执行。

## SATU设计

当下的深度学习加速器一般都是面向DNN实现的，而DNN中最主要的且最关键的组件就是卷积层，这一点在BCNN中也不例外。下表展示了某款加速器芯片在实际应用中的计算量统计结果，由表中可见，在芯片执行算法的过程中，卷积计算占了总计算量的98.1%，所以深度学习加速器的设计步骤中，计算引擎的设计十分关键。

|  |  |  |
| --- | --- | --- |
| 计算类型 | 计算量（MOPS） | 占比 |
| 卷积 | 34,275 | 98.1% |
| 反卷积 | 576 | 1.6% |
| ReLU | 123 | 0.2% |
| 池化 | 13 | 0.1% |

表 某加速器芯片的计算量实际测试结果[50]

从基本需求出发，卷积层的本质就是要完成对数据的加权求和，这一过程就需要MAC（乘加器）来完成。AI芯片中常以TOPS来衡量性能，即该芯片可实现的峰值吞吐量，因为芯片电路中大多数的操作是MAC，即一次乘法伴随一次加法，所以TOPS也常用下面公式 来计算。所以为了充分利用性能，芯片设计中就需要最大限度地保证在运行时间每一个PE处于忙碌状态，这就是实现高吞吐量的关键。基于此想法，芯片研究人员开始重新使用20世纪80年代比较流行的脉动式阵列，将MAC处理单元分布到复杂且精细的脉动数据流中，例如2015年谷歌的TPU中就采用了这一方法，目前这一方法也广泛应用于深度学习加速器领域。

(3-1)

本节设计的脉动阵列阈值单元中也是采用了脉动阵列的方法，利用数据计算的并行性和连续性，提高了计算效率并降低了功耗。该单元针对BCNN算法进行设计，不同于平常的PE中采用了全精度浮点数的乘加计算，脉动阵列阈值单元中的PE采用了二值复数的popcount(xnor)计算，能效性和硬件友好性得到了大幅提升。而在脉动阵列附近增加的阈值单元，可直接对卷积结果进行BN和二值化过程，通过添加一系列简单的[comparator](https://cn.bing.com/dict/search?q=comparator&FORM=BDVSP6&cc=cn)，而避免了复杂的bn和sign计算，降低了芯片的功耗和成本。

### SATU结构设计

脉动阵列阈值单元（SATU）是为二值复数神经网络加速器芯片提供强大算力的核心单元，通过该单元可快速地将二值复数输入特征图输入到脉动阵列中并进行卷积计算得到二值复数输出特征图或者全精度输出特征图，结构框图如下图所示，主要包括专用于二值复数点积计算的脉动阵列、权重输入FIFO、输入激活值FIFO、阈值FFO、输出激活值FIFO、片上存储和Threshold模块。SATU结构设计参考自MVTU，细节可见于2.3.5节。

OCM作为SATU的数据来源，从DRAM中以矩阵向量的形式缓存输入、权重、阈值数据到片上存储空间，在译码到预取或计算指令后通过相应的FIFO将权重或待计算数据送往脉动阵列中的指定位置，在脉动阵列完成矩阵乘法（如果脉动阵列为输出固定模式，则可能还需要在脉动阵列内部的Compare Splicer模块完成阈值计算）后保存中间数据或者最终输出激活值；得到最终输出激活值后，可通过“mvout”指令将结果传输回DRAM（如果脉动阵列为权重固定模式，则可能还需要送往threshold模块完成阈值比较-拼接计算）。

通过以上对数据通路的描述中，我们不难发现本次设计的SATU中存在两套完成 阈值比较-拼接计算的通路，这是因为为了适应不同的模型和微架构需要，本次设计的协处理器中SATU的脉动阵列满足权重固定（weight stationary）和输出固定（output stationary）两种模式，这两种模式可以通过配置指令完成选择；所以为了适配两种数据流模式，我们提出了两套完成阈值比较-拼接计算的方案。阈值比较计算我们在2.3.5节作出过解释，而之所以还需要进行拼接操作，是因为我们的设计中采用了类NHWC的新型数据存储方式（详见3.5.2），即多个通道的对应像素点的数据存储在一个存储单元中，所以为了保证每层的输出特征图可以直接作为下一层的输入特征图，我们就要通过位拼接操作将阈值比较后多通道的二值复数的输出结果归约起来，恢复为类NHWC的存储方式。

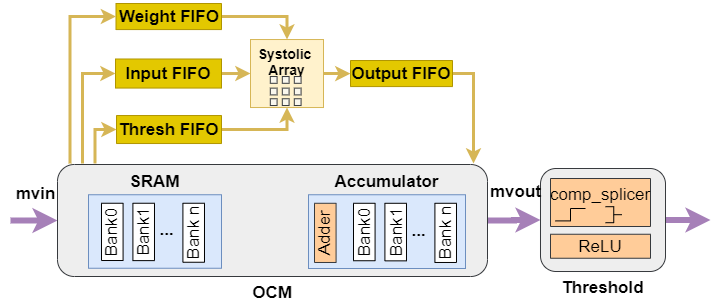


图 SATU的结构框图（OCM为片上存储，Threshold阈值单元）

以上我们介绍了SATU的主要结构和数据通路，下面我们针对其中的脉动阵列的设计细节展开讨论，其中包括此次设计的新型脉动阵列的结构描述以及两种数据流模式下的脉动阵列的运行机制。

1. 新型脉动阵列的结构

脉动阵列作为协处理器计算核心——SATU的核心计算模块，为协处理器提供了强大的并行“乘加”计算能力，主要用于处理深度神经网络中大量的矩阵乘加计算，结构图如下图 所示。该结构的设计参考了谷歌TPU的脉动阵列模块，但是与其相比又存在不同之处。

* 一方面在于TPU中采用的脉动阵列单元尺寸为256×256，这对边缘智能设备来说太庞大了。因为TPU主要是用于云端上神经网络的训练和推断任务，而本次设计的边缘智能设备主要用于完成二值复数神经网络的推理任务，所以对算力的要求远小于TPU，且出于低成本、低功耗的考虑，我们只需要将阵列的尺寸设置为16×16；
* 另一方面为了让输出固定模式下的脉动阵列完成卷积计算后，可以直接进行阈值比较计算并将得到的多个单比特二值结果进行“拼接”，再将结果输出到存储单元，我们在脉动阵列中又额外的增加了Compare Splicer模块，如下图 所示，通过config\_ex指令可配置是否进行阈值比较-拼接计算；
* 再者由于本次设计是面向BCNN算法，所以PE中的核心计算范式需要从整数或者全精度的乘加运算修改为二值复数的popcount(xnor)计算，具体见3.4.2节。

除了上述区别外，新型脉动阵列的结构与传统的结构无本质区别，每个PE通过流水线寄存器和相邻的PE连接，每个周期完成一次popcount(xnor)运算，计算结果从一个PE通过寄存器传输到另一级PE，并自动完成乘加计算，相比于CPU中的标量计算，功耗可以降低10到15倍。在脉动阵列的输入和输出端口还包含了一系列的移位寄存器，用于实现数据的同步。

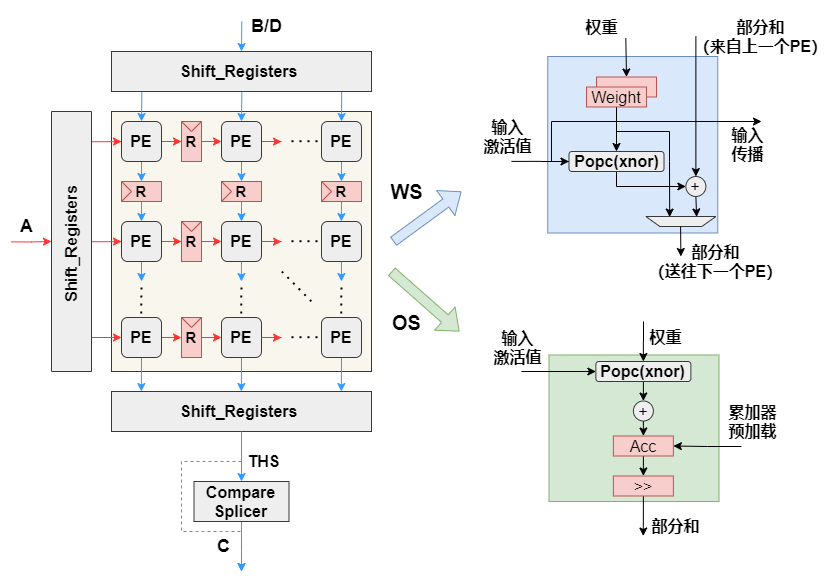


图 面向BCNN的新型脉动阵列结构示意图

1. 不同数据流模式下的运行机制

新型脉动阵列结构支持权重固定和输出固定两种数据流模式，下面以矩阵乘加计算（如下公式3-2）为例，从ISA角度简要介绍一下两种模式下新型脉动阵列的运行机制，如图 所示。公式3-2中A为输入特征图矩阵，B为权重矩阵，C为输出特征图矩阵，因为BCNN算法中没有偏置层，所以在公式3-2中没有偏置矩阵，并且我们假设在此之前我们已经将相应的输入激活值、权重矩阵数据已经通过mvin指令存储进Databuffer中。

(3-2)

首先介绍输出固定模式下的ISA运行机制：

1. 首先将矩阵A和矩阵B通过compute.preloaded指令按行依次加载进入脉动阵列，不断向前传播，在PE中完成popcount(xnor)计算并将结果与之前的部分和累加;
2. 将每个PE中计算得到的新的部分和固定保留在各自的PE中；
3. 重复前两个步骤直至完成矩阵A和矩阵B的乘加计算，之后将输出特征图C读出到Databuffer中，过程中根据功能控制信号THS（Threshold，由config\_ex指令配置）判断是否需要进行阈值比较-拼接计算。

继续介绍权重固定模式下的ISA运行机制：

1. 首先通过preload指令将矩阵B预加载进入相应PE的缓存器中并固定；
2. 紧接着通过compute.preloaded指令将矩阵A按行送入脉动阵列中并不断向前传播，此过程中每个周期每个PE完成一次popcount(xnor)计算并与上一级PE传播来的部分和进行累加，然后将得到的部分和结果向下一级传播；
3. 部分后不断传输并累加经过最后一级PE后进入Accumulator，等待后续的累加操作；
4. 重复步骤二、三直至完成矩阵乘法运算后，将Accumulator中的输出特征图C通过数据移动指令返回DRAM中，此过程中可通过地址信号的比特29（可参考3.5.1中的寻址策略）判断是否需要进行阈值比较-拼接计算。

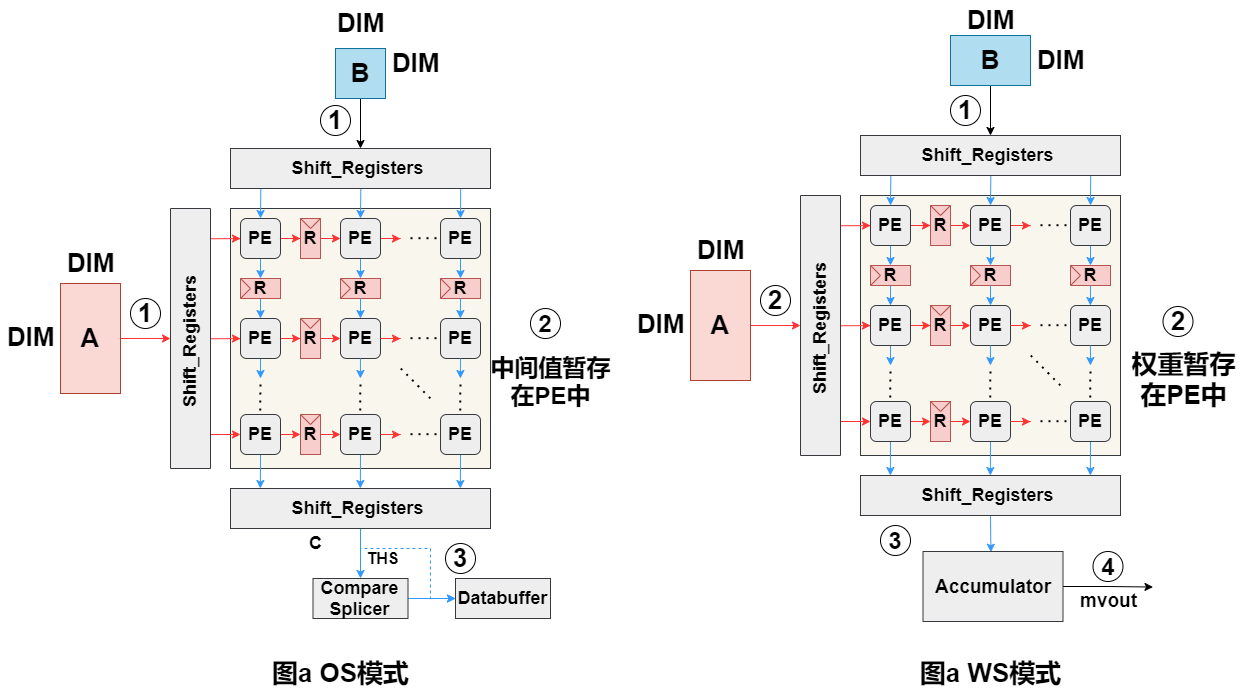


图 不同数据流模式下的运行机制

（图a为输出固定模式下，图b为权重固定模式下）

### 复数卷积运算单元设计

上一小节中我们从SATU出发，先说明了SATU的整体结构和数据通路，然后进一步阐述了针对BCNN设计的新型脉动阵列的结构以及其不同模式下的运行机制，这一小节我们将深入到脉动阵列内部的PE中，从电路层面阐述不同模式下PE的结构和运行机制，并介绍多通道并行的popcount(xnor)计算模块。

1. 输出固定模式的PE

首先介绍输出固定模式 下的PE，其电路结构如图 所示。每个PE中包含了一个32位（其中16位为实部，另外16位为虚部）的popcount(xnor)计算模块、两个32位（其中16位为实部，另外16位为虚部）的进位保留加法器（CSA）、两个寄存器、两个外部控制逻辑块以及三个二选一选择器。之所以使用32位的加法器就是为了避免计算过程的精度损失。除此之外还有一些其他的组件，例如双缓冲（double buffer）和外围逻辑电路（peripheral logic，PL)。双缓冲主要用于缓冲输入数据便于后续的计算，且通过双缓冲可以在当前计算周期未完成的情况下将先前的计算结果从脉动阵列输出，实现了non-stop计算，极大的提高了计算效率。外围逻辑电路可用于对输出结果进行移位操作，以达到缩减位宽的作用。为了减少关键路径的延迟，每个PE都拥有两个加法器、寄存器和两套独立的数据通路，每个通路负责一个寄存器。因此每个寄存器有一个独立的加法器和一个外围逻辑电路。针对两条数据通路，我们在电路的输入端和输出端各添加两个二选一mux和一个二选一mux，电路还有一个1比特控制信号PROP（Propagation）作为mux的选择信号，通过此三者可选择哪条通路用于计算，哪条通路用于传播。

根据3.4.3节中对输出固定模式下脉动阵列的ISA运行机制的描述，以计算公式3-2的矩阵运算为例，在计算开始的第一步，需先将全零的偏置矩阵D从in\_d端口预加载到PE的缓冲器中，此时PROG信号配置为低电平，矩阵D中元素通过传播通路进入寄存器C2，又从out\_c端输出进入下一个PE，经过不断传播，矩阵D按需分布在脉动阵列中。紧接着输入特征图矩阵A和权重矩阵B分别从in\_a和in\_b端口进入PE，此时PROG信号跳变为高电平，A和B的对应元素进入BC\_POPC(XNOR)进行复数popcount(xnor)计算的同时，又分别从out\_a和out\_b端口输出并通过pipeline寄存器进入下一级PE。BC\_POPC(XNOR)模块完成计算后将结果送入缓存buffer中，随后缓存buffer又将数据送往两个加法器中，分别与寄存器C1与C2暂存的累加值进行累加，但只有左侧通路新的累加值能返回寄存器C2。在完成式3-2的矩阵乘加运算得到输出矩阵C（此时矩阵C的各元素存储在各个PE的C2寄存器中）后，如果不需要在结果C上进行累加计算，则PROP信号恢复为低电平，脉动阵列即按行将每行PE中C2寄存器的累加值通过out\_c端口输出到片上存储Databuffer中，至此，一次矩阵运算才算完成，如果需要在结果C上进行新的矩阵运算并累加，则PROP信号维持高电平，并继续从in\_a和in\_b端口传入待计算数据，直至所有计算完成。

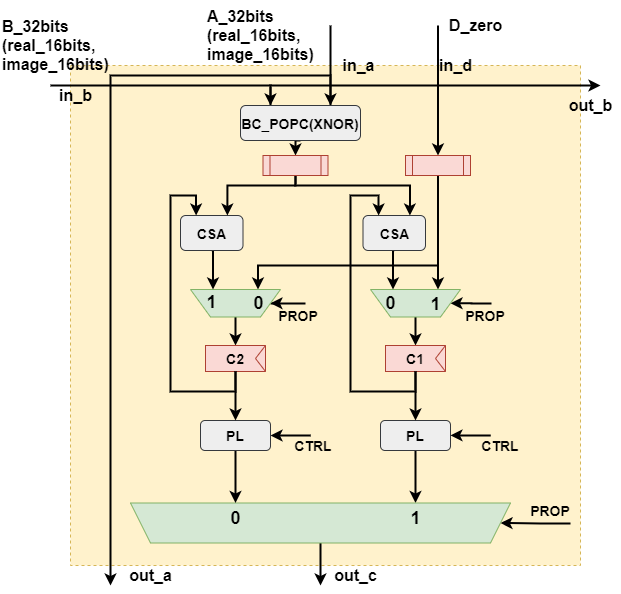


图 输出固定模式下PE的电路结构

1. 权重固定模式的PE

介绍完输出固定模式下PE的结构和运行机制后，我们展开对权重固定模式的PE的描述。与输出固定模式相同，权重固定模式也有两条数据通路，通过PROP信号控制两条通路的使用顺序。不同的是权重固定模式下的PE少了一个加法器，多了一个二选一mux，通过该mux选择寄存器C1或者C2保存的权重值与in\_a输入的激活值进入BC\_POPC(XNOR)模块进行二值popcount(xnor)计算。

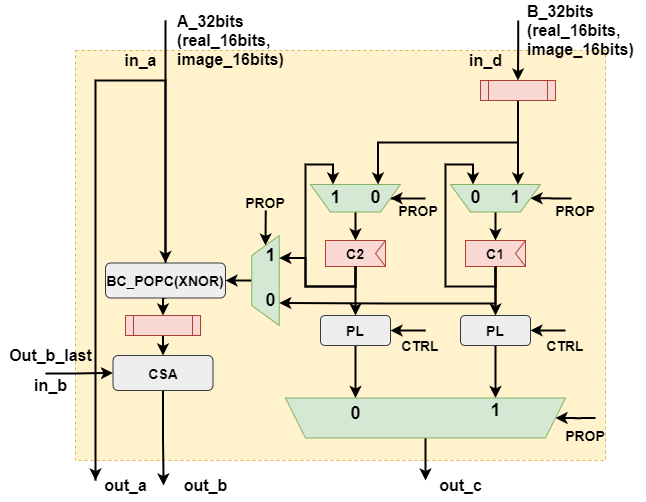


图 权重固定模式下PE的电路结构

权重固定模式下PE进行矩阵乘加运算的工作机制与输出固定模式下的存在一定的区别， 默认PROG信号为低电平，首先需将权重矩阵B（而非偏置矩阵）从in\_d端口预载入到寄存器C2中存储；完成预载任务后，PROG信号跳变为高电平，开始传输输入特征图矩阵A按列传入脉动阵列第一列的的PE中，不断向前传播，并与C2寄存器中保留的权重值进行popcount(xnor)计算，之后与in\_b端口输入的上一级的部分和进行累加，完成上述过程后，PE的计算结果会从输出端口out\_b输入给下一级PE或者Accumulator中，此时（即使上一个矩阵计算还没有完全完成）可以直接开始下一个矩阵计算的预载，PROG信号保持不变，新的权重矩阵B通过in\_b端口传输进入寄存器C1中，待预载完成（此预加载过程也可在前一个矩阵计算未完成的同时进行加载），PROP信号跳变为低，后续计算过程与前面描述的并无差异。当然如果不需要预载新的权重矩阵，也可以直接从in\_a端口传入新的输入矩阵A此时PROP信号保持不变。由上述的过程描述，我们可以发现，整个计算过程中的预载入、计算、累加计算等步骤都伴随着PROP信号的跳变。

1. BC\_POPC(XNOR)运算模块

上述中我们介绍了不同数据流模式下PE的结构和运行机制，然后不论是哪一种模式，都离不开BC\_POPC(XNOR)模块提供的二值复数的popcount(xnor)运算支持。在2.2.1节我们介绍了二值复数以及其点积计算的理论知识，而二值复数的数据特点与计算的独立性就决定了我们可以硬件设计中使用大量的并行性设计，进而提高计算效率、节约存储资源和减少访存次数。基于这两点出发，我们提出了新型的存储方式（见3.5.2节）与新型的计算电路结构。

二值数据的popcount(xnor)运算早已在BNN相关的算法和硬件设计中普及，所以我们将“复数运算”和popcount(xnor)运算的概念相结合得到了二值复数的点积运算中，但这样的并行性在多通道的卷积运算中还是不够的。所以在此基础上，我们创新性的提出多通道“同步”存储和“同时”计算的二值复数点积运算。

以两个通道的同一像素点为例，假设输入激活值为（1，-1）、（-1，1），权重值为（1，1）、（-1，1）,若采用普通设计：

1. 首先完成第一个通道的计算，即
2. 继续计算第二个通道，即
3. 求和得到结果，即

而如果采用新型计算结构，

1. 首先使用“0”替代“-1”，即像素点（1，-1）使用（1，0）表示，则我们直接得到输入激活值（10，01）与权重值（10，11），其中10、01等表示将二值复数实部和虚部分别并行存储的输入值，可看作2’b10、2’b01等。
2. 根据公式 3-3、3-4分别计算得到结果（2，-2）。

(3-3)

(3-4)

令式中popc(xnor)计算结果为*x*，之所以还需要计算，是因为我们使用“0”替代了“-1”，所以计算结果中“-1”的个数应为“通道数-*x*”（此处通道数为2），则多通道合并的popc(xnor)计算结果为（*x*-(通道数-*x*)=2\**x*-通道数）。

由上述论证，我们将通道数推广，设为*chs*，经过并行存储后的输入特征值A为，权重B为，其中、、、分别表示实部和虚部，其位宽等于通道数*chs*。则A\*B计算结果如下式3-5、3-6所示：

实部： (3-5)

虚部： (3-6)

本次设计中我们将通道数推广到16，则BC\_POPC(XNOR)模块的电路结构如图 所示，输入特征值A与权重值B进入模块后将其实部和虚部分别存储在相应的寄存器中，紧接着并行完成四个16位的popcount(xnor)计算（包括乘2与减通道数的运算，不过这两个运算很简单，只需要通过移位和一个减法器即可实现)，再根据复数运算的特性将相应的popcount(xnor)计算结果相加或者取反相加即可得到最终结果。

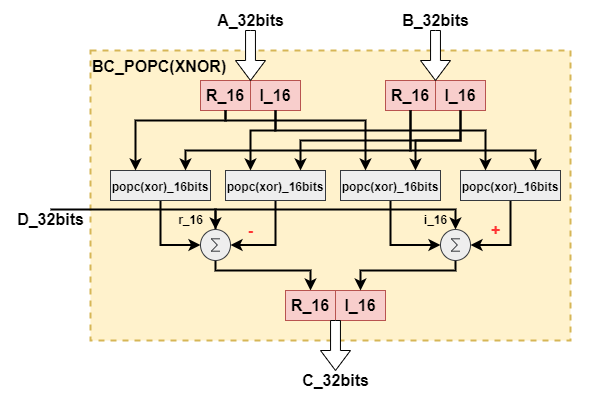


图 32bits二值复数popcount(xnor)计算单元

使用上述的结构的复数卷积运算单元，不仅极大的降低了硬件资源开销，还提高了计算的并行度，减少了访存次数，进而提升了计算效率。例如16个通道的卷积运算，原需要16次卷积运算并求和，而现在只需要一次卷积运算即可完成，不考虑访问内存的时间，计算效率就已经提升了16倍，如果考虑上访问内存的时间，计算效率会提升更多。

### Compare Splicer模块设计

在本文的2.3.5节我们解释了，BN和二值化过程是BNN以及BCNN算法中必不可少的，而这两个过程即使是在推理任务中也需要大量复杂的浮点运算，硬件实现时更是需要耗费大量的资源，所以我们参考了MVTU结构，提出了使用阈值比较（Threshold compare）计算替代BN和二值化过程，并且在训练和推理任务中都得到不错的效果。因为本次设计专注于在边缘侧设备的推理任务，所以我们在训练中得到每层的阈值，并保存在存储系统中，当需要进行阈值比较计算，则将阈值读取并送往相应的Comparator即可。

当脉动阵列所有通道的计算结果并行进入上述的Comparator后，此时我们将得到16个二值复数，如果不加处理直接送入片上存储Databuffer或Accumulator，结果将以NHWC存储方式进行存储，并不符合我们提出的类NHWC存储类型（见3.5.2节），这就不便于下一层卷积计算，所以必须要将16个通道的实部和虚部分别拼接得到一个32比特（其中16位为实部，另16位为虚部）的输出特征值。

经过上述分析，我们设计了Compare Splicer模块，其电路结构如图 所示，电路实现也十分简单，只需要一系列的比较器和位拼接即可完成，用极少的硬件开销即可实现复杂的BN和二值化过程。

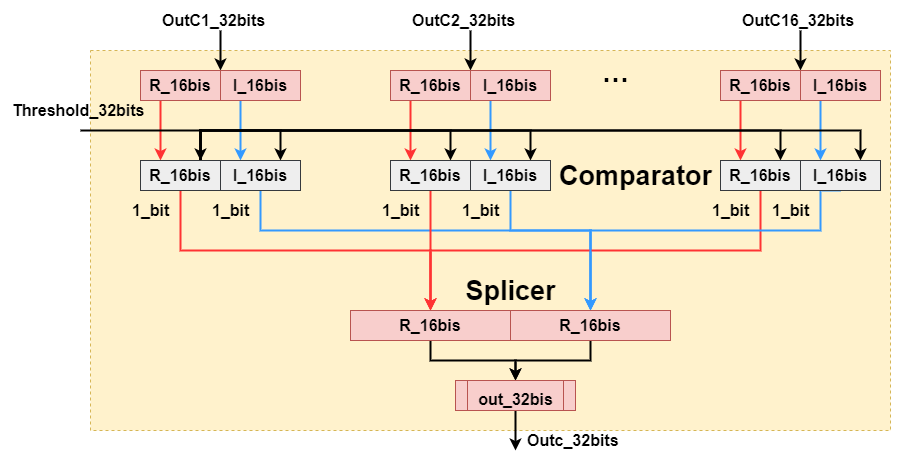


图 Compare Splicer模块结构示意图

## 内存设计及数据存储方式

除计算引擎的设计外，设计一个高效且合理的存储系统也是边缘智能设备乃至整个深度学习加速器领域的关键点。一个加速器的存储架构至少要满足以下两个条件：

1. 内存足够大，能够存储神经网络中尽可能多（乃至所有）的初始输入、权重、中间激活值等，减少对片外存储的访问；
2. 带宽足够宽，因为深度学习相关的都是数据密集型的应用，所以输出结构必须要有足够大的带宽向脉动阵列传输权重和激活值或者接收输出激活值，并跟上脉动阵列的执行速度；

显然易见，存储系统的高效性是提高加速器系统吞吐量和计算效率的关键因素，而在实际应用中我们也要考虑到设计的合理性，例如我们不能为了高效性而在存储系统中全部采用昂贵的片上SRAM存储，这样势必增大了芯片的面积和成本，这就不满足了边缘智能设备的设计中低成本的要求，例如表 中所示的以某款加速器协处理器的面积分布，仅320KB的片上存储就约占了芯片总面积的67%；同理我们也不能为降低成本而全盘使用片外DRAM，因为实验证明了神经网络计算过程中的绝大多数功耗都是来自DRAM访问，而不是乘加计算本身，且DRAM的访问延迟过高，也会限制计算引擎的执行效率，这对低时延、低功耗的边缘智能设备来说也是致命的。因此我们必须要采用分布式多级存储系统，即通过使用一块或者两块DRAM和少量的分布式本地SRAM来实现较高的MAC利用率，从而在实现高吞吐量的同时降低成本和功耗。

|  |  |  |
| --- | --- | --- |
| 组件尺寸 | 面积（um2） | 占系统百分比（%) |
| 脉动阵列（16×16） | 116K | 11.3 |
| Databuffer（256KB) | 544K | 52.9 |
| Accumulator（64KB) | 146K | 14.2 |
| CPU | 171K | 16.6 |
| 总计 | 1029K | 100 |

表 某款加速器系统的版图面积分布表[68]

本节中设计的存储系统不仅采用了分布式多级的方式，且针对二值复数神经网络的数据还提出了新型的存储方式，提高了内存空间的利用率和计算的并行性，满足了二值复数神经网络计算中需要快速提取和传输高并发数据的需求。

### 内存设计

想要协处理器中的SATU能够达到理想的算力，必要条件就是能够保证输入数据准确且及时地传输到计算单元中。本次设计的分布式多级存储系统包含了片外存储、cache以及片上存储（如图 中图a所示），本小节仅详细说明了片上存储部分。片上上存储主要包括单端口Databuffer、双端口Accumulator和逻辑控制电路三个部分。

作为片上存储系统的主要存储单元，Databuffer和Accumulator用于存储脉动阵列的输入和输出，输入数据一般存储在Databuffer中，而中间值（partial sums）和最终结果（final results）一般存储在accumulator中。二者都采用了“row-addressed”的结构，即每个地址对应的memory的一行，而每行的位宽为DIM个元素宽度之和，其中DIM为脉动阵列每行或者每列PE的个数，利用此结构每次内存访问都是一次性读取或者写入DIM个元素，减少了内存访问次数。Databuffer和Accumulator的用途不同，其每个元素的宽度也可以根据算法应用的需求而定，因为本次BCNN算法应用中，因为每个layer的通道数都是16的倍数，且针对的是二值复数，所以我们将二者的单个元素位宽都配置为32位。下面说明一下Databuffer和Accumulator的电路实现和寻址策略。

1. 电路实现

先说明电路实现，Databuffer由多个 banks组成，bank结构十分简单，仅由一块单端口SRAM和输入输出队列构成。相比之下，Accumulator的结构稍显复杂：除了两块双端口SRAM和输入输出队列，还包含了一系列的加法器去支持数据累加操作，如下图 图b所示。

而之所以Databuffer使用单端口sram，而Accumulator使用双端口sram，是因为Databuffer在算法运行过程的load、prelaod和excute阶段的每个周期只需要完成写数据或者读数据任务，而不存在同时进行读写两个任务的机会。例如load阶段我们将数据从DRAM加载进入Databuffer，此阶段的每个周期内只需要完成写数据任务；perload阶段我们将数据从Databuffer预加载进入脉动阵列，此阶段的每个周期只需要完成读数据任务；与preload阶段相同，excute阶段将矩阵乘加计算所需的输入数据依次从Databuffer按行读取进入脉动阵列，也不存在读数据任务，所以为了节约资源，Databuffer只需要使用单端口SRAM即可实现。Accumulator的工作机制与Databuffer则有些许不同，例如在权重固定模式下的脉动阵列，每完成一次矩阵乘加计算的结果将进入Accumulator并与目标地址保存的原值进行累加，这就需要在每个周期同时完成一次读数据任务、累加计算任务、写数据任务，虽然这三个任务可以通过寄存器缓存并使用状态机依次完成，但是在边缘智能设备这类实时性要求较高的应用场景中，这种方式显然是达不到算力要求的，所以我们采用了流水线形式将 三个任务并行执行，极大的缩短了算法的运行时间，因此accumulator也必须要使用双端SRAM实现。

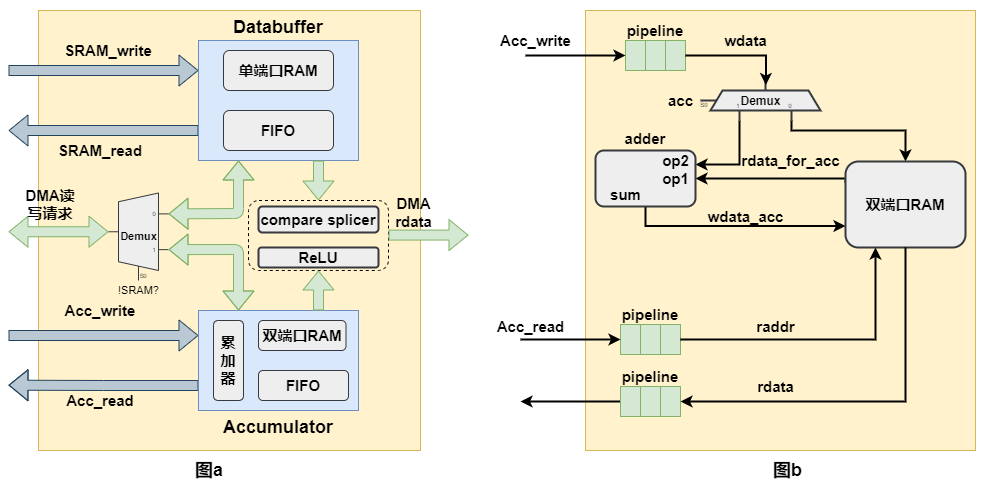


图 片上存储示意图（图a是片上存储整体结构框图，图b是Accumulator结构框图）

1. 寻址策略

在算法运行过程中，我们需要根据数据需要将数据传输到不同存储单元，例如将输入激活值送往Databuffer，中间值送往Accumulator，且对于Accumulator来说，还需要增加部分功能信号对其功能进行控制，为了数据传输不会产生紊乱和错误，就需要指定一个准确有效的寻址策略，还应该基于此要求，我们设计了以下寻址策略，如图 所示。

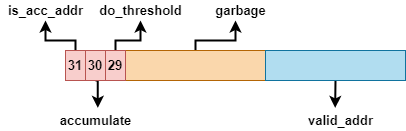


图 寻址策略示意图

内存的地址宽度为32比特，其中高三位是默认的功能位，且具有特殊意义，其中：

* 第31位（地址最高位）用于区别当前地址属于哪个存储单元，为1时表示寻址Accumulator，为0时表示寻址Databuffer；
* 第30位在寻址Databuffer或者读Accumulator时忽略。而当我们向Accumulator写数据时，第30比特位作用很关键；如果需要覆盖该地址之前的数据则可将该比特位置0，如果想要与该地址之前的原值累加，则需要置1。
* 第29位在寻址Databuffer或者写Accumulator时忽略。而当我们从Accumulator读数据时，如果要将读数据通过Compare Splicer模块进行处理时则需将该位置1，如若不处理则置0。

地址信号的低位是寻址的有效地址位，其宽度由Databuffer和Accumulator中存储行数较多的一方决定，假设Databuffer的bank数为4，每个bank的行数4096，则有效地址位的宽度。除功能位和有效地址位外，其余的地址信号都是无效位，默认为0，不需要处理。但是存在一种特殊情况，在处理执行指令时，如果指令中32位地址信号全为1时（包括无效位），则不需要去内存中读取数据，内部会自动向脉动阵列中传输全零数据。正常情况下，当地址信号进入三大控制模块（Execute Controller模块、Load Controller模块和Store Controller模块）后功能位首先被译码，并将有效地址映射到对应的Databuffer或者Accumulator的特定位置完成读写任务。

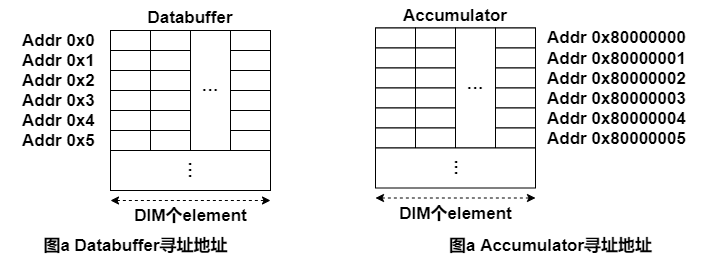


图 Databuffer与Accumulator寻址地址示意图

### 二值复数数据新型存储方式

BCNN算法中数据类型主要是二值复数，该数据类型与普通的整数和浮点数有些许区别，如果不采用合适的存储方式，必将造成存储空间利用率低，数据访问次数过多等问题，进而导致资源和能耗的浪费。基于此，我们针对二值复数提出了一种类似NHWC形式的新型的存储方式，如下图 所示。

无论逻辑表达上是几维的数据，在计算机中存储时都是按照一维形式来存储的。例如普通的NHWC格式的物理存储方式，即先取C方向数据；然后W方向；再H方向；最后N方向。此硬件平台采用的数据通道格式与存储方式，虽然类似于NHWC，但是还是有一定区别：在先取Channel方向的数据后，不是分别存储在单个的存储单元，还是进行合并后存储在同一个存储单元。之所以选择这种存储方式，原因如下：

1. 二值神经网络的每个通道中每个像素点的数据位宽仅为2比特（实部和虚部各1比特），若使用32比特的存储空间去存储2比特的数据，显然是十分浪费且不合理的；
2. 将单bit的数据合并存储后，可以减少memory的访问次数，极大的增加访问效率和计算效率，还降低了多次数访问带来的功耗浪费；
3. 另外一个优势，在于二值复数计算时不同通道计算结果的独立性，与SATU中脉动阵列配合后，一次数据读取可以完成多个通道的计算，且通过popcount计算，避免了重复累加的过程，可完美地实现多通道的并行处理。

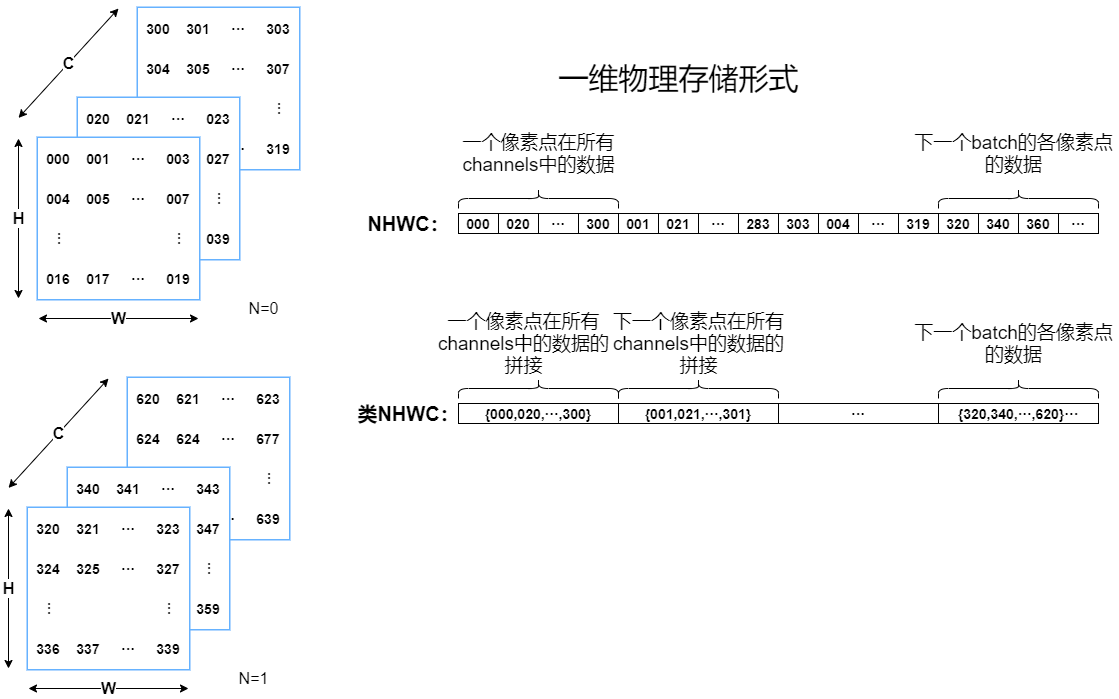


图 类NHWC与NHWC存储方式对比示意图

## 本章小结

本章中我们由浅入深、由整体到局部地介绍了面向BCNN的加速器的协处理器的设计。我们从“软硬件协同”的设计思想和“高性能、低功耗、低成本”的设计目标出发，首先介绍了协处理器的硬件整体架构和指令集架构的设计，接下来我们创新地提出了SATU（脉动阵列阈值计算单元）作为协处理器的计算核心，又分别详细地阐述了SATU中复数卷积运算单元和Compare Splicer模块的设计，最后一节我们介绍了协处理器片上存储的设计，其中包括我们提出的适用于二值复数的新型数据存储方式和“row-addressed”的内存设计。

# BCNN专用加速器的系统设计

## 引言

在近几年的自研人工智能芯片热潮中，众多研究团队、独角兽公司试图在边缘侧智能设备方向上有所突破，其中就包含了众多AI算法专用IP核的开发，但是想要这些IP核落实到实际应用，就不可避免的要嵌入完整的SoC系统中，例如苹果手机中最大的应用处理器芯片就是一块含有AI处理核的SoC。在第三章中，我们面向BCNN算法，从硬件电路角度设计了一款专用于BCNN算法加速推理任务的协处理器并提出了相应的指令集架构，与其他AI专用IP核相同，要想该协处理器能够应用在边缘设备，就必须要将其集成到SoC中。

而在AI芯片的开发过程中，特别是在边缘侧，降低功耗、减小芯片面积几乎是目前所有AI芯片都亟待解决的课题。相比手机、个人电脑等对处理器性能有极高要求的设备来说，边缘智能设备的关注点主要在处理器的控制能力而非其微弱的算力，所以在满足项目要求的前提下，大部分公司都开始尽量选择开源的处理器和工具链，而近些年兴起的RISC-V也逐渐备受青睐。

RISC-V最早起源于2010年美国加州大学伯克利分校Krste Asanovie教授主持的一个关于开源计算机系统的研究项目。RISC-V是个自由开放的指令集，其标准化工作主要由RISC-V基金会主持，该组织的会员数目前已超过100个并在不断增加。无须向基金会支付授权费用，任何想要使用RISC-V指令集设计实现处理器的公司以及个人都可以不受限制地使用其设计、制造和销售RISC-V芯片和软件。在RISC-V问世之初，移动端设备主要都是ARM处理器把持，PC的市场也主要由Intel公司的x86处理器占据，正因上述的自由开放，近些年RISC-V不断攻城略地，生态环境不断发展，给这两大巨头带了危机感，迫于形势，两大巨头也先后投入了对RISC-V的研究。



图 RISC-V基金会及其成员[71]

目前网络上利用RISC-V设计的开源项目数量众多，其中包括BOOM、Rocket、PicoRV32、RI5CY等等优秀的产品。本次面向BCNN算法的加速器的设计中，我们就使用了Rocket开源处理器，在不影响加速器性能的前提下，大大降低了加速器的设计成本。所以本章中我们首先对Rocket进行介绍，并完成Rocket与第三章中设计的协处理器的集成工作。

## 开源处理器Rocket概述

本小节我们将展开对开源处理器Rocket的介绍，因为本次加速器的设计工作重点在于面向BCNN算法的协处理器的设计，所以本节中我们只对Rocket做一个整体的概述，而不关注其设计细节。

Rocket由RISC-V的“摇篮”——美国加州伯克利大学设计，是一款32/64位（可配）、顺序执行、五级流水线的RISC-V处理器。Rocket处理器还包含了分支预测缓存（Branch Prediction Buffer，BTB）、分支历史表（Branch History Table，BTB）、返回地址栈（Return Address Stack，RAS）等模块单元，所以可以有效完成分支预测的功能。除此之外，Rocket还支持分页虚拟内存，所以可以很好地移植Linux操作系统。

不同于传统的处理器设计，Rocket处理器采用一门新的语言Chisel（Constructing Hardware in an Scala Embedded Language）编写，这也是加州伯克利基于Scala语言设计的一种开源的硬件描述语言。Chisel充分利用了Scala的优势，将面向对象（object orientation）、类型参数化（parameterized types）、函数式编程（functional programming）、类型推断（type inference）等软件语言概念引入了硬件描述语言，进而提升了硬件描述的抽象级别，并给与了硬件设计人员更强大的硬件开发能力。使用Chisel编写的硬件电路，可以通过firrtl工具编译得到对应的Verilog设计，还可以得到对应的C++模拟器。Rocket使用Chisel编写，就可以很容易得到对应的软件模拟器。同时，因为Chisel是面向对象的，所以Rocket的很多类可以被其他开源处理器、开源SoC直接使用。

Rocket处理器已流片11次之多，其准确性不言而喻，而与同为标量处理器的ARM Cortex-A5在同时采用台积电40nm工艺的情况下，其性能也毫不逊色，如下表 所示。从对比中我们不难发现，Rocket处理器面积仅为ARM Cortex-A5的一半，功耗甚至不到其一半，而其性能却更优。

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ARM Cortex-A5** | **RISC-V Rocket** | **Ratio** |
| 寄存器宽度 | 32 | 64 | 2 |
| 主频 | >1Ghz | >1GHz | 1 |
| Dhrystone | 1.57DMIPS/MHz | 1.72DMIPS/Hz | 1.1 |
| 面积（不包含Cache） | 0.27mm2 | 0.14mm2 | 0.5 |
| 面积（包含16KBCache） | 0.53mm2 | 0.39mm2 | 0.7 |
| 动态功耗 | <0.08 mW/MHz | 0.034 mW/MHz | >0.4 |

表 Rocket与 ARM Cortex-A5在台积电40nm工艺下的性能对比

## 系统硬件架构设计

在完成BCNN协处理器的基础上，我们通过与Rocket开源处理器集成，得到了面向BCNN算法的加速器，其结构如图 所示。其中s加速器协处理器通过RoCC（Rocket Custom Coprocessor Interface)接口与Rocket处理器完成指令和数据通信，当Rocke处理器取指令并经过初步译码后得出该指令属于RoCC指令，即可通过RoCC结构将指令送往协处理器并译码执行。

如果该指令为load或者store指令，需要与片外存储DRAM交互，且指令中的DRAM物理地址对应的虚拟地址在协处理器映射未命中时，协处理器会见该虚拟地址返回到Rocket并取回对应的物理地址。在取得准确的物理地址后，load controller或者store controller中的DMA将会通过TileLink总线（TileLink是由伯克利大学孕育的芯片公司SiFive提出的一套全新的芯片级总线互连标准，可以为SoC的通用处理器、协处理器、DMA等设备提供一个具有高吞吐率、高速和低延迟传输特性且可扩展的片上互联方式）向L2 Cache读取或写入数据。如果该指令为配置或者计算指令，则只需要在协处理器内部即可完成处理。

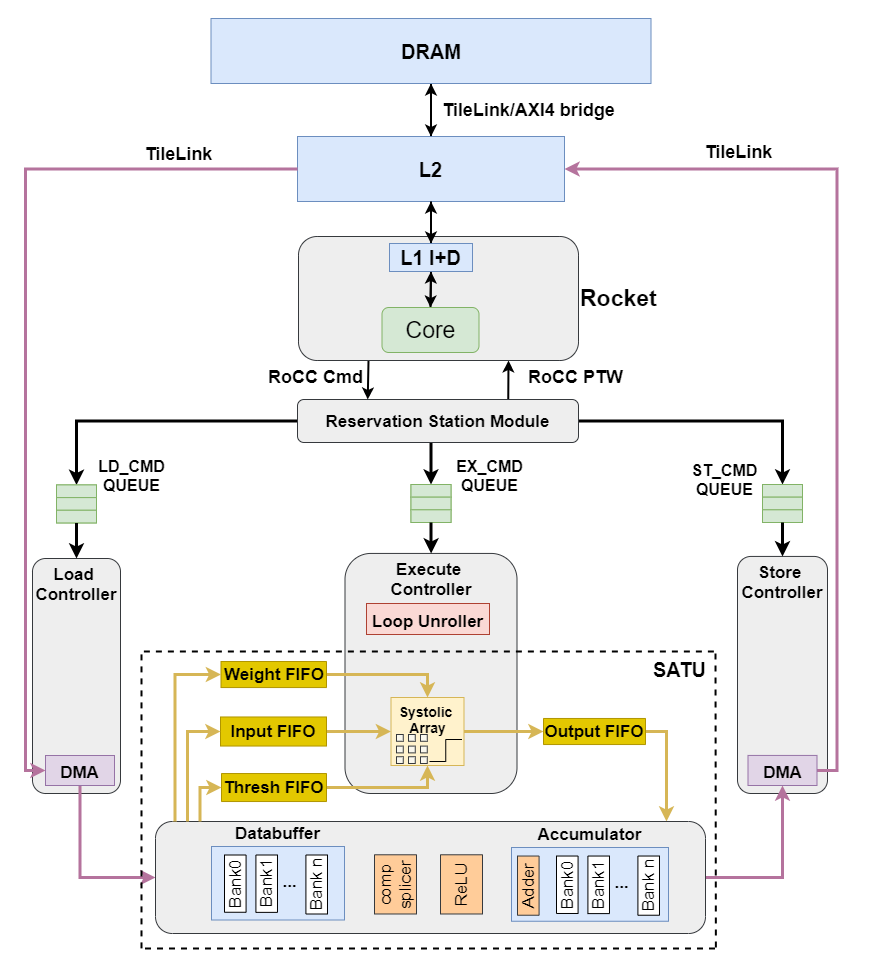


图 面向BCNN的加速器整体框图

（其中包括Rocket处理器、BCNN专用协处理器、DRAM以及Cache）

### 多级分布式存储系统

在第三章我们提到了BCNN加速器的多级分布式存储系统，当时只是说明了其分布式的特征，而本段我们将对“多级”的特性进行详细的描述。片外存储DRAM、L2 Cache、L1 Cache、协处理器中的片上存储（Databuffer和Accumulator）以及设计中的缓冲buffer、队列等共同构成了整个加速器的多级分布式存储系统。片外存储DRAM作为整个加速器的“仓库”，存储了与算法有关的所有初始数据和指令，由于其成本低廉，存储容量也最大，但是DRAM的读写速度远远小于CPU和协处理器的运行速度，这就使得系统要花费大量的时间在访问内存上，从而导致系统整体性能的降低。所以为了解决系统中数据读写速度不匹配的问题，我们在电路系统中引入了高速缓冲器，即L1 Cache与L2 Cache。

Cache一般由访问速度更快的SRAM构成，为了让论文结构完成，本段先对Cache的工作原理做一个简单介绍。Cache可分为标记存储器和数据存储器两个基本组成部分，其中标记存储器负责存储Cache的控制位的块地址标签，控制位可控制Cache的读写操作，地址标签保存Cache中各块的地址，该地址指与主内存映射的块地址，并且与Cache数据存储器中的数据一一对应。当主处理器或者协处理器读取数据时，可先将物理地址通过数据总线传输到Cache中，并于Cache中的块地址标签进行比较。如果相符合，则表示命中，只需要将Cache中对应的数据通过数据总线传输给需求方。如果不相符合，则表示未命中，此时需要将物理地址送到主存储器中，将数据读取并缓存在Cache中，再由Cache传输给需求方。可见提高Cache命中率就可以提高系统性能。而提高命中率一般可通过适当增加Cache容量、改善映射方式和块替换方式等方法来实现。由于篇幅限制，此处就不再赘述了。

L2 Cache位于与L1 Cache、协处理器与主内存之间，是整个存储系统的“中转站”和“交通枢纽”。L2 Cache通过TileLink和AXI4总线的桥接电路与DRAM控制器（图 中未标明）连接并访存DRAM中的存储数据；通过TileLink总线与协处理器中的DMA连接，完成协处理器与主存储器的信息交互。L1 Cache直接例化在Rocket处理器中，位于CPU和L2 Cache之间，分为数据缓存和指令缓存两个部分，因为内置在主处理器中，所以L1 Cache的容量和结构对CPU的性能影响最大，但是受CPU管芯面积限制，L1 Cache一般会做的稍微小一点。在此特别，DMA的数据通道宽度、TileLink总线宽度（SystemBusKey）、Cache数据通道宽度（cacheblockbytes）等这些参数都是紧耦合的，可通过在Rocket的参数中进行配置。

协处理器中的片上存储在此不做介绍，可参考3.5节。

### RoCC接口

上文中我们介绍了加速器的多级分布式存储系统，本小节我们将详细介绍Rocket处理器和BCNN协处理器之间的通信接口——RoCC接口。在第三章介绍协处理器的设计时，为了便于后期的集成工作，我们在协处理器中预留了RoCC接口。RoCC接口旨在扩展Rocket处理器并方便处理器和协处理器之间解耦通信，如图 所示，RoCC接口可以分解为多个子接口，特别是连接了处理器核和协处理器的cmd（Command）子接口，通过该接口主处理器可以向协处理器发送指令，协处理器则可以通过resp（Response）接口向主处理器进行反馈或者返回数据。为了允许协处理器访问内存，RoCC 接口还提供了 mem\_req和 mem\_resp到数据Cache的子接口。

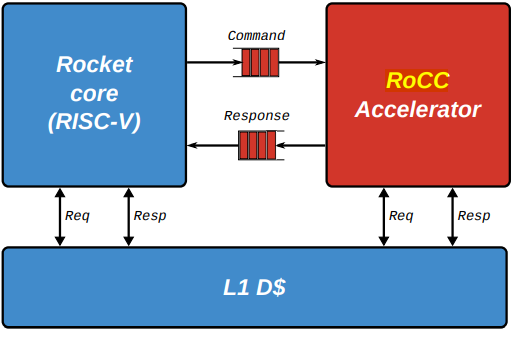


图 Rocket核与协处理器通过cmd与resp接口连接

除了这四个通道之外，RoCC 接口还提供了很多扩展子接口，可用于实现高级功能，例如可以将协处理器与 FPU 连接、与CPU共享PTW、与片外存储系统直接交互等等。 RoCC 接口还提供了一些状态和中断信号，用于主、协处理器的信号同步或标志信号错误。

## 系统软件架构设计

在本节之前，我们设计了面向BCNN算法专用加速器的协处理器并与RISC-V处理器集成得到了一个完成的加速器系统，为BCNN中的卷积神经计算性能的提升提供了硬件基础。但是对于一个神经网络的算法，从神经网络模型的实现到实际硬件电路的运行（包括计算流和数据流），这一过程需要一个多层次的软件结构进行管理。

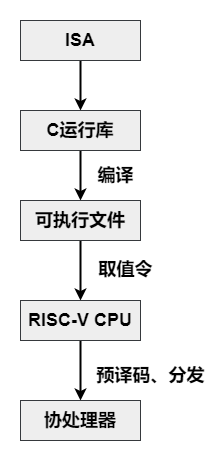


图 系统软件架构流程图

所以为了让面向BCNN的专用加速器能够启动并发挥强大的性能，我们提出了一套系统软件架构，如下图 所示。首先我们结合了3.3节的指令集架构，从底层出发，通过C APIs进行编程，得到了加速器的C运行库；在C运行库中，我们可以使用C语言调用库中的函数来完成需要的算法；在完成算法的代码后，紧接着我们通过GCC工具链将C运行库编译为机器可执行的文件；得到可执行文件并将其加载进入硬件系统的存储器后，剩下的就是专用加速器执行的过程，RISC-V处理器从存储器中取指令并译码执行，如果某条指令在简单的预译码后被判定为属于协处理器的扩展指令，则该指令会通过RoCC接口送往协处理器并继续译码执行。

上文中我们简单地阐述系统软件架构的流程，虽然每个流程我们都一笔带过，但实际上每个流程都涉及了众多的原理，在此处就不一一详解，下面对其中的GCC工具链、C运行库以及编译过程进行简单介绍。

### GCC工具链简介

工欲善其事，必先利其器。本次软件架构设计中离不开各类工具的辅助，例如编译过程中使用的RISC-V GCC（GUN Compiler Collection）工具链。GCC工具链是多个程序的集合，包括GCC、Binutils、GDB和C运行库等。

其中GCC（GNU C Compiler）是编译工具，本文中我们就是通过RISC-V专用的GCC编译器将C语言编写的程序转成加速器可执行的二进制代码，RISC-V GCC不仅支持本地编译，即在某个平台上编译该平台可运行的程序，还可以进行交叉编译，即跨平台编译其他平台运行的程序。

Binutils是一组二进制程序处理工具的集合，这些工具的功能都十分强大，在RISC-V开发和调试起到了至关重要的作用，其中包括addr2line（用于将程序地址转成其所对应的程序源文件及所对应的代码行）、objcopy（用于转换文件的格式，例如将.bin翻译为.elf，或将.elf翻译为.bin等）、ld（用于链接）、as（用于汇编）、ar（用于创建静态库）、readelf（用于显示ELF文件的信息，因为常见的编辑器无法直接查看ELF文件）、objdump（主要用于反汇编）等等工具。

GDB（GUN Project Debugger）主要用于调试C、C++、Ada等编程语言编写的程序，通过该工具我们不仅可以查看处理器和程序的运行状态（包括处理器中通用寄存器的值、存储器的值、变量值等），还可以修改这些状态。

C运行库又称为C运行时库（C Run Time Library, CRT），GCC编译工具的运行离不开C运行库的支持。为了解释C运行库的作用，我们需要回顾C语言标准，C语言标准主要包括两个部分：一部分描述C的语法，另外一部分描述C标准库。其中C标准库由多个标准头文件组成，每个头文件中都定义了相应的函数、类型声明、变量和宏定义等，例如常见的printf函数，其原型就定义在stdio头文件中。特别指出C标准库仅针对C语言本身，并没有提供实现，且与即将要运行C语言的平台无关，所以要在一个平台上运行C语言，我们就必要实现面向该平台的C运行库。

### 编译过程简介

C语言编译是一个非常复杂的过程，其中涉及但不仅限于编译器知识、硬件知识、工具链知识，此处简而言之，编译过程是将高层的C/C++语言编写的程序转为处理器能够执行的二进制代码的过程，其中包括预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）、链接（Linking）四个步骤[71]。

预处理主要是对C语言程序代码做一些文本处理，例如删除所有的#define、注释，展开宏定义，处理条件预编译指令（例如#if、#ifdef、#else等）、保留#pragma编译器指令以及添加行号和文件标识等操作。

在完成预处理后就可以对文件进行编译，即进行一系列的词法分析、语义分析、语法分析以及优化处理后生成相应的汇编代码。紧接着就是汇编过程，该过程会对编译得到汇编代码进行处理并生成处理器能够识别的指令，这些指令通常保存在.o文件（目标文件）中。相对于编译过程来说，汇编过程较为简单，因为每一条汇编语句都一一对应一条指令，所以只需要通过调用Binutils中的汇编工具as，对照处理器指令和汇编指令的对照表进行逐句翻译即可。

注意到虽然汇编后的指令已经能被处理器识别，但是在链接之前还是不能直接运行的，要想成为能够被加载的可执行文件，还需要通过调用Binutils中的链接器ld来链接程序运行依赖的其他库文件、引导程序和所有需要的目标文件，最终生成一个ELF（Executable and Linkable Format）格式的可执行文件，至此一个编译过程才算完成。

### 面向BCNN协处理器的C运行库

基于上述理论，我们使用C语言编写了面向BCNN协处理器的C运行库，编译获得可执行文件并在加速器系统中顺利执行。

基于C标准库，我们首先定义了协处理器RoCC接口的扩展指令格式，如下代码4-1所示，该定义采用了内联汇编的方式，不仅提高了程序的速度，还降低了内存需求，其中asm用于内联汇编表达式的声明，volatile为可选参数，用于向GCC编译器声明该内联汇编表达式不可优化。

#define ROCC\_INSTRUCTION\_0\_R\_R（x, rs1, rs2, func7)                                   \

  {                                                                                  \

    asm volatile(                                                                     \

        ".insn r " STR(CAT(CUSTOM\_, x)) ", " STR(0x3) ", " STR(func7) ", x0, %0, %1" \

        :                                                                            \

        : "r"（rs1), "r"（rs2));                                                       \

  }

代码4-1 协处理器扩展指令底层内联汇编代码

基于上述定义，我们又展开了对协处理器指令集架构中的mvin、mvout、preload、compute.preloaded、compute.accumulated等指令的定义，其中也包括XCUSTION\_ACC、ADDR\_LEN、k\_MVIN等宏定义声明，详细代码见附录2。在前面这些工作的基础上，我们可以精准地控制协处理器的功能，并完成我们需要的数据传输和数据计算任务；另外我们还可以通过声明新的变量和参数并调用这类底层指令去实现更为高级的函数接口，方便用户直接调用。

## 本章小结

本章我们首先引入了对RISC-V生态链和RISC-V开源核Rocket的介绍；基于Rocket处理器和第三章设计的协处理器，我们通过集成得到了完整的面向BCNN的专用加速器的硬件系统；同时我们又设计了软件系统，其中包括RISC-V GCC工具链、面向协处理器ISA的C运行库以及一整套开发流程，至此我们完成了面向BCNN的专用加速器软硬件全系统的设计工作。

# 仿真与性能评估

## 引言

在第三章和第四章的工作中，我们完成了面向BCNN专用加速器的硬件系统和软件系统的设计，在本章中我们将继续基于“软硬件协同”的思想对整个加速器系统进行仿真验证[72]，并利用Vivado工具对加速器的硬件电路进行性能评估。

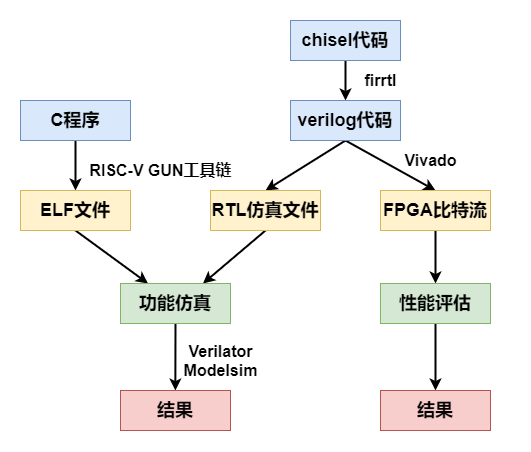


图 5-1 仿真与性能评估流程图

图5-1中展示了本次功能仿真与性能评估的流程，其中功能仿真的软件部分我们在4.4节已经做了介绍，这里不再赘述；因为我们的硬件电路设计没有使用传统的Verilog硬件描述语言而是使用了Chisel语言（该语言的优势可参考4.2节的介绍），所以在进行“软硬件协同”仿真前，我们需要通过Firrtl（Flexible Internal Representation for RTL)工具将chisel代码转译为Verilog代码，便于后续的仿真与性能评估。得到Verilog语言描述的硬件电路后，我们就可以搭建Testbench测试文件、例化待测试模块、写入ELF文件并通过仿真工具Verilator实现功能仿真；因为设计中含有一个功能准确完善的RSC-V处理器，所以我们可以通过主处理器比较仿真结果（完成数据传输或计算任务并写回DRAM的值）和预期结果是否一致来判定设计的正确性；也可以通过Modelsim查看仿真的波形文件，追踪内部信号，查看信号值与信号间的逻辑关系，进而判定设计的正确性或进行debug与设计优化。

除了设计的功能仿真，我们还可以使用Vivado套件对Verilog代码进行综合、布局布线、统计资源消耗以及评估设计性能等，并输出比特流文件，将比特流文件烧录进入FPGA中。

本小节中我们简单介绍了功能仿真与性能评估的流程，下面两节我们将详细描述本次设计的面向BCNN的专用加速器的功能仿真和性能评估结果。

## 功能仿真

本次设计的协处理器中主要用于加速BCNN算法中二值复数的卷积计算，且脉动阵列的维度为16，所以功能仿真中我们主要测试了不同尺寸矩阵的数据移动、矩阵卷积计算、卷积结果累加、阈值比较-拼接运算等功能。

1. 矩阵尺寸为16×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为16×16，即卷积核大小为4×4、卷积核个数为16；A为输入激活矩阵，尺寸为16×16；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行了阈值比较-拼接运算）；其输入A、B及输出C如下图所示：

测试步骤：

1. 随机化获取矩阵A、B数据，存储在DRAM中；
2. 将矩阵A、B数据”mvin”至Databuffer中；
3. 将权重矩阵B的数据”preload”至脉动阵列中；
4. 将输入矩阵A的数据依次送入脉动阵列并计算，计算结果会送往Accumulator中；
5. 将Accumulator中的最终计算结果”mvout”至DRAM中的二维数组C（可选择是否进行阈值比较-拼接运算）；
6. 将数组C中数据与预期数据进行比对，得出结论。

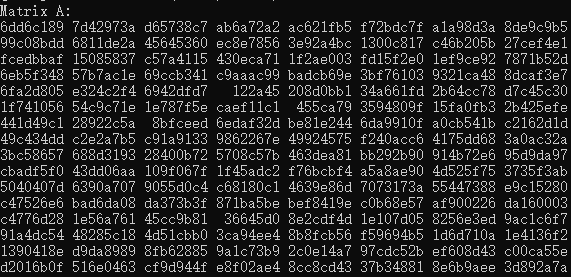


图 输入矩阵A部分数据



图 输入矩阵B部分数据

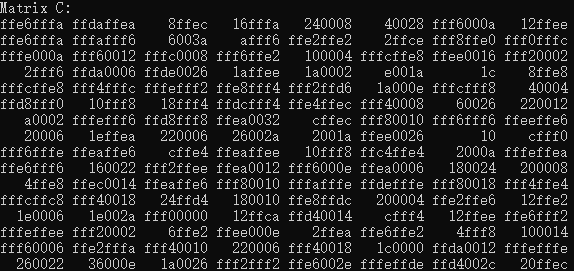


图 输出矩阵C（未进行阈值比较-拼接运算）部分数据



图 输出矩阵C（进行了阈值比较-拼接运算）

由图 、图 与图 （输出矩阵中元素高16位为实部、低16位为虚部，二者为有符号数，若最高位为1，则表示结果为负数）中数据所示，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为4×4情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。仿真波形如下图 所示，其中图 表示矩阵A、B中的数据按行进入Databuffer，并被按行读取进入脉动阵列中；图 表示权重矩阵B的数据缓存进入PE的寄存器，输入矩阵A依次通过各PE并与权重矩阵B完成popcount(xnor)和累加计算；图 表示矩阵A、B的计算结果依次进入Accumulator并被按行读取返回DRAM或进入Compare Splicer模块；图 表示上述的计算结果经过阈值比较-拼接计算后返回DRAM。

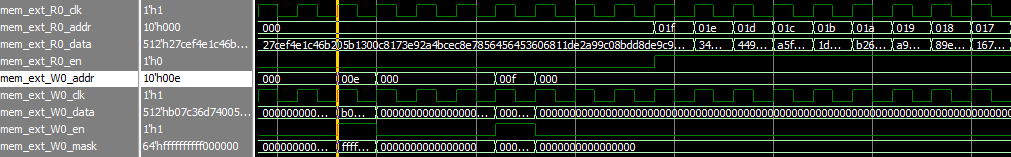


图 数据“mvin”至Databuffer并读取进入脉动阵列

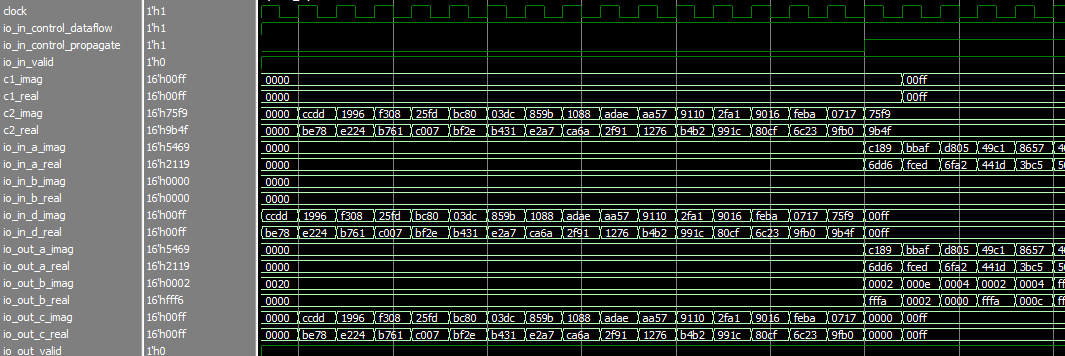


图 矩阵A、B数据进入脉动阵列并进行计算

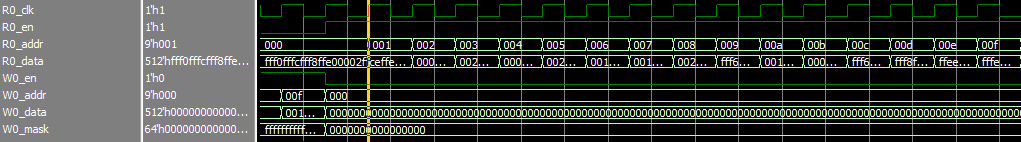


图 矩阵C数据从Accumulator中“mvout”

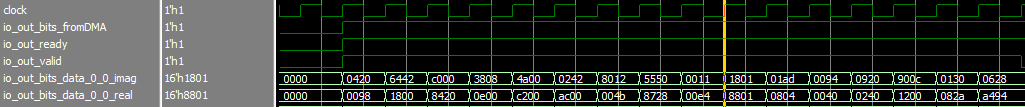


图 经过Compare Splicer模块的输出

1. 矩阵尺寸为9×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为9×16，即卷积核大小为3×3、卷积核个数为16；A为输入激活矩阵，尺寸为16×9；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行阈值比较-拼接运算）；其输入A、B及输出C如下图所示，波形文件不做展示：

测试步骤与上述测试步骤相同，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为3×3情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。

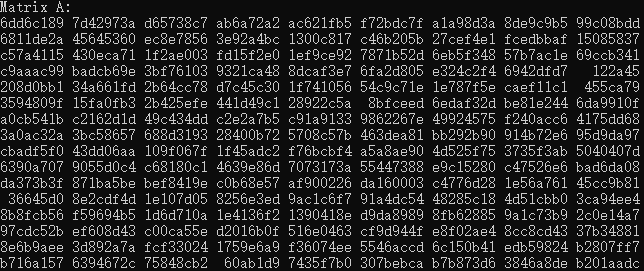


图 输入矩阵A部分数据

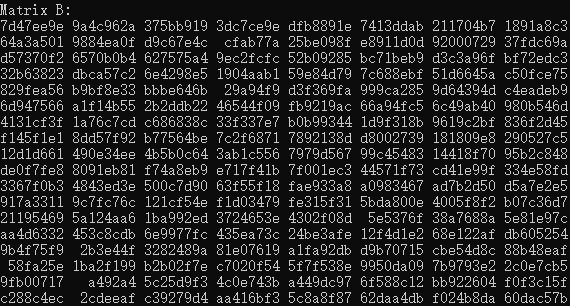


图 输入矩阵B部分数据

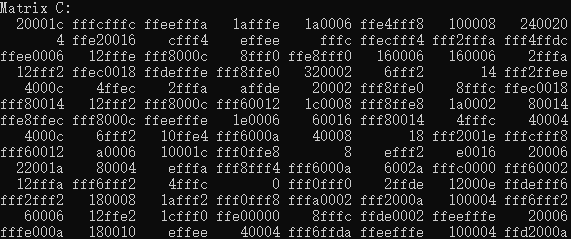


图 输出矩阵C（未进行阈值比较-拼接运算）



图 输出矩阵C（进行了阈值比较-拼接运算）

1. 矩阵尺寸为1×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为1×16，即卷积核大小为1×1、卷积核个数为16；A为输入激活矩阵，尺寸为16×1；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行阈值比较-拼接运算）；其输入A、B及输出C数据、波形文件在此不做展示，测试步骤与上述测试步骤相同，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为1×1情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。

1. Zero\_DCE算法验证

Zero\_DCE算法是一款轻量型的微光图像增强算法，可以在各类不同的灯光条件包括不均匀和弱光情况对图像进行提亮处理，具体可参考论文[73,74]，我们首先将算法推广到BCNN，训练得到最优权重参数，并将参数加载至协处理器的片上存储；之后利用C运行库实现了Zero\_DCE算法，并联合仿真得到图像增强后的结果，如下图 所示：

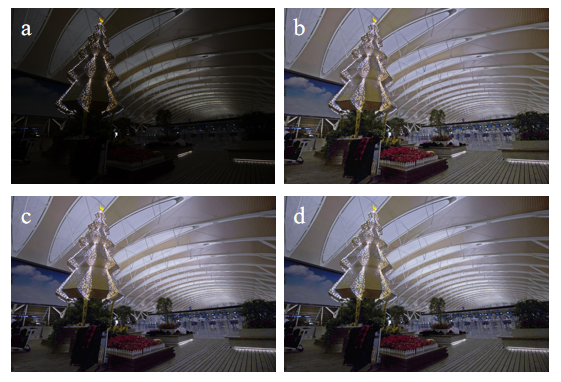


图 Zero\_DCE图像对比（图a为原始图片，图b为全精度DNN算法增强后结果，

图c为BNN算法增强后结果，图d为BCNN算法增强后结果）

综上所测，我们判定本次设计的面向BCNN的专用加速器可以准确有效地完成BCNN算法的加速任务。

## 性能评估

本次性能评估使用了创龙TLZ7xH-EVM开发板（如图 所示）和Vivado 设计套件。其中TLZ7xH-EVM开发板基于创龙SOM-TLZ7xH核心板（如图 所示）设计，该核心板是一款基于Xilinx Zynq-7000 系列XC7Z045/XC7Z100高性能处理器设计的异构多核 SoC，处理器集成 PS 端双核ARM Cortex-A9 + PL 端 Kintex-7架构28nm可编程逻辑资源。Vivado 设计套件由Xilinx公司推出，主要用于FPGA平台的开发和调试，其中包括RTL代码综合、资源评估、布局布线、生成比特流等功能。



图 TLZ7xH-EVM开发板



图 SOM-TLZ7xH开发板核心板

在整个加速器系统的Chisel代码转为Verilog代码后，我们利用Vivado套件对电路进行综合与实现，系统时钟设置为100Mhz，其中系统整体资源占用如表 所示，协处理器的资源占用如表 所示，脉动阵列模块的资源占用如表 所示。观察表 ，我们发现由于采用了popcount(xnor)计算，面向BCNN的脉动阵列模块只占用了少量了LUT和Register而不需要DSP等资源，具有极高的硬件友好性。

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 173079 | 218600 | 79.18 |
| Register | 118416 | 437200 | 27.09 |
| F7 Mux | 1964 | 109300 | 1.80 |
| F8 Mux | 270 | 54650 | 0.49 |
| BRAM | 369 | 545 | 67.71 |
| DSP | 164 | 900 | 18.22 |
| BUFGCTRL | 3 | 32 | 9.38 |

表 系统整体逻辑资源占用表

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 126288 | 218600 | 57.77 |
| Register | 96841 | 437200 | 22.15 |
| F7 Mux | 481 | 109300 | 0.44 |
| F8 Mux | 71 | 54650 | 0.13 |
| BRAM | 192 | 545 | 25.23 |
| DSP | 149 | 900 | 15.56 |
| BUFGCTRL | 0 | 32 | 0 |

表 协处理器逻辑资源占用表

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 74885 | 218600 | 34.26 |
| Register | 42341 | 437200 | 9.68 |
| F7 Mux | 0 | 109300 | 0 |
| F8 Mux | 0 | 54650 | 0 |
| BRAM | 0 | 545 | 0 |
| DSP | 0 | 900 | 0 |
| BUFGCTRL | 0 | 32 | 0 |

表 脉动阵列模块逻辑资源占用表

除了资源占用外，我们还对整个电路系统的功耗和性能进行了评估，如下图 所示，在100Mhz频率下，电路的整体功耗为2.076W，其中动态功耗和静态功耗分别为1.903W和0.172W。由于本次设计的加速器面向的是二值复数神经网络的加速任务，采用了大量的并行结构，计算方式也与传统的整型或浮点型乘加计算方式有别，所以其性能的评估方式不同，假设电路工作在100Mhz频率，加速器中的脉动阵列含有16×16（共256）个计算单元，每个计算单元可在一个周期内完成32个通道的popcount(xnor)计算、复数运算（一次加法与一次减法计算）以及累加计算，所以本次加速器的性能为，即约3.3TOPS。

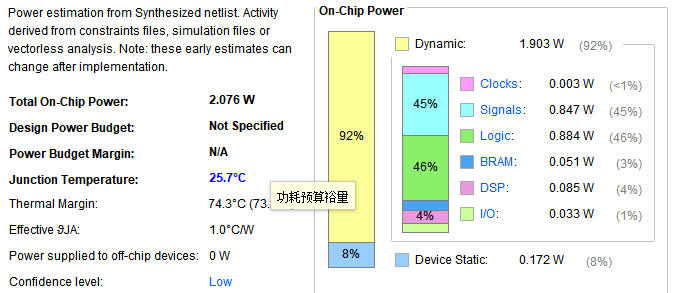


图 FPGA功耗分析报告

基于上述分析，我们针对性能、功耗以及能效三个方面，将本次设计的专用加速器与1.1.2小节介绍的几款边缘侧神经网络加速器芯片进行对比，对比结果如表 所示。由表中数据分析可得本次设计的BCNN专用加速器在能效比并不逊色于几款商用加速器，且我们还应该知道这几款商用加速器芯片都是采用ASIC制造的，我们的BCNN专用加速器仅使用FPGA实现。

|  |  |  |  |
| --- | --- | --- | --- |
| 加速器 | 性能（TOPS） | 功耗（W） | 能效比 |
| Xavier | 30 | 30 | 1 |
| EyeQ4 | 2 | 6 | 0.33 |
| EyeQ5 | 12 | 5 | 2.4 |
| Envision | 0.075 | 0.2 | 0.375 |
| BCNN 加速器 | 3.3 | 2.076 | 1.6 |

表 BCNN加速器与几款边缘侧商用加速器芯片对比

综上所述，我们评估BCNN专用加速器的资源占用、性能、功耗、能效并与其他边缘端神经网络加速芯片进行了对比，可见我们本次设计的加速器能够满足边缘智能设备低成本、低功耗、高能效的要求。

## 本章小结

本章中我们搭建了RISC-V软件编译环境和仿真平台，并在编译环境中添加了协处理器的扩展指令，进而对BCNN专用加速器系统进行了功能仿真和电路优化。完成电路优化并确定电路功能准确后，我们又使用Vivado设计套件对加速器的资源占用、性能、功耗以及能效进行评估，并在创龙TLZ7xH-EVM开发板上综合实现了加速器电路。最后我们将面向BCNN的专用加速器与其他几款商用神将网络加速器芯片进行对比，结果表明本次设计具有良好性能，且具有低成本、低功耗、高能效等优势，适用于边缘侧智能设备。

# 总结与展望

## 总结

随着神经网络不断向边缘智能设备端发展，为了解决深度卷积神经网络部署到边缘端设备时计算资源、存储资源不足，能效低等问题，基于“软硬件协同”的思想，我们在本文中设计并实现了一款面向BCNN的专用加速器，该加速器主要由RISC-V处理器Rocket核和BCNN专用加速协处理器组成，二者的所有代码均由Chisel语言实现。我们在加速器的协处理器中央内置了一块SATU，该单元采用了脉动阵列结构，且大量应用了并行化设计，不仅可以高效地完成二值复数卷积计算，且具有低功耗、高能效等优势，非常适用于边缘智能设备。本文主要工作内容如下：

1. 阐述二值神经网络和复数神经网络的研究背景和研究现状，进而引出对二值复数神经网络的研究，并结合对边缘智能设备的研究提出了本次课题的研究方向与内容。
2. 阐述了卷积神经网络的基本原理和硬件加速的实现方法，其次我们重点说明了二值复数神经网络的组成、二值化原理和二值复数计算原理，并介绍了MVTU，为后面的专用协处理器的设计铺垫。
3. 设计了面向BCNN的专用加速器的协处理器，首先我们提出了本次协处理器设计架构并设计了协处理器指令集，之后结合了脉动阵列和MVTU设计了面向二值复数神经网络卷积计算的SATU（脉动阵列阈值计算单元）；又针对二值复数的数据特点，提出并设计了“row-addressed”型片上存储系统和新型的类NHWC存储方式，不仅提高了存储资源利用率还极大地提高了计算效率。
4. 在完成协处理器设计后，我们将协处理器与开源处理器Rocket核集成，进而实现了面向BCNN的专用加速器的系统设计。除此之外，我们还介绍了RISC-V开源生态链、开源处理器Rocket、加速器系统的多级分布式存储系统和RoCC接口；在指令集架构的基础上，我们又完成了加速器软件系统的设计并通过C语言从指令集层面实现Zero\_DCE++算法的二值复数卷积运算部分。
5. 搭建了软硬件协同验证平台，并对专用加速器进行功能分析、仿真与优化；在此之后，我们又基于Vivado工具对专用加速器电路进行性能、功耗与能效评估，并利用创龙TLZ7xH-EVM开发板实现了面向BCNN的专用加速器系统电路。

## 展望

本文提出、设计并在FPGA中实现了面向BCNN的专用加速器，主要用于完成边缘侧BCNN算法的推理任务，不仅具有超高的计算性能，还同时兼备低功耗、高能效等优势，达到了边缘智能设备的设计需求，但仍然存在优化和改进的空间：

1. 在前文中，我们提出BCNN算法的第一层和最后一层需要全精度计算，但是我们专用加速器的协处理器仅支持二值复数计算，后续可以在协处理器的SATU中扩展，搭配多个全精度整数与浮点数脉动阵列计算单元，并在扩展指令集中添加数据流向选择功能，如图 所示，提高片上系统算力的完备性。

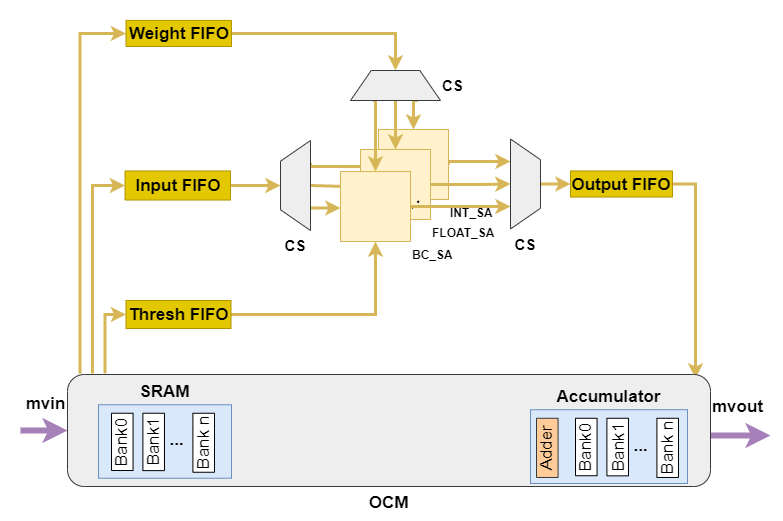


图 多核SATU单元

1. 在第五章的功能仿真验证中我们发现，当卷积计算中卷积核大小小于4×4，即矩阵运行时脉动阵列中有部分PE处于“闲置”状态，但数据信号和控制信号仍通过该PE，造成不必要的功耗与带宽浪费并降低了计算效率。为解决非标准维度的矩阵计算问题，后续添加动态配置PE状态功能，即在对“计算”指令译码过程中判断待计算矩阵的维度是否为标准维度，若不是，则根据指令中的行列信息，随时地动态地配置PE状态，关闭部分PE功能。
2. 本文设计的协处理器内部不具有im2col功能，在进行卷积计算的行滑动和列滑动操作时需要通过软件完成，后续可以添加im2col模块，搭配DMA模块，从硬件角度自动完成数据的加载任务，极大地提高计算效率。

# 致 谢

# 参考文献

[1] J. M. R. Lopez, K. Heider, J. Scheffran. Frontiers of urbanization: Identifying and explaining urbanization hot spots in the south of Mexico City using human and remote sensing %J Applied Geography, 2017, 79:

[2] 赵永强, 饶元, 董世鹏, 张君毅. 深度学习目标检测方法综述 %J 中国图象图形学报, 2020, 25(04): 629-654

[3] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, J. J. P. O. T. I. Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing, 2019, 107: 1738-1762

[4] 侯宇青阳, 全吉成, 王宏伟. 深度学习发展综述 %J 舰船电子工程, 2017, 37(04): 5-9+111

[5] X. Wang, Y. Zhao, F. J. I. J. O. M. L. Pourpanah, Cybernetics. Recent advances in deep learning, 2020, 11: 747-750

[6] Y. Han, X. Wang, V. C. M. Leung, D. T. Niyato, X. Yan, X. J. I. C. S. Chen, et al. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey, 2020, 22: 869-904

[7] H. Li, A. Kadav, I. Durdanovic, H. Samet, H. P. Graf. Pruning Filters for Efficient ConvNets. %J CoRR, 2016, abs/1608.08710:

[8] J.-H. Luo, J. Wu. AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference %J Pattern Recognition, 2020, 107(prepublish):

[9] Y. Yao, X. Wu, L. Zhang, S. Shan, W. Zuo. Joint Representation and Truncated Inference Learning for Correlation Filter based Tracking. 2018

[10] L. Jian-Hao, Z. Hao, Z. Hong-Yu, X. Chen-Wei, W. Jianxin, L. Weiyao. ThiNet: Pruning CNN Filters for a Thinner Net. %J IEEE transactions on pattern analysis and machine intelligence, 2019, 41(10):

[11] W. Wen, C. Wu, Y. Wang, Y. Chen, H. Li. Learning Structured Sparsity in Deep Neural Networks. %J CoRR, 2016, abs/1608.03665:

[12] L. Jiang, H. Zhang, L. J. a. S. P. Yu. Robust Time-Frequency Reconstruction by Learning Structured Sparsity, 2020:

[13] T. Geng, C. Wu, C. Tan, B. Fang, A. Li, M. Herbordt. CQNN: a CGRA-based QNN Framework. in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), 22-24 Sept. 2020: 1-7

[14] J. Choi, P. I.-J. Chuang, Z. Wang, S. Venkataramani, V. Srinivasan, K. J. A. Gopalakrishnan. Bridging the Accuracy Gap for 2-bit Quantized Neural Networks (QNN), 2018, abs/1807.06964:

[15] 冯佳玮. 基于FPGA的二值卷积神经网络加速研究与实现: [硕士学位论文]. 哈尔滨理工大学, 2022

[16] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. J. a. L. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1, 2016:

[17] A. Li, T. Geng, T. Wang, M. C. Herbordt, S. Song, K. J. J. P. O. T. I. C. F. H. P. C. Barker, Networking, Storage, et al. BSTC: a novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets, 2019:

[18] A. Li, S. J. I. T. O. P. Su, D. Systems. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs, 2021, 32: 1878-1891

[19] T. Geng, T. Wang, C. Wu, C. Yang, W. Wu, A. Li, et al. O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning, 2019:

[20] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, R. Shi, et al. O3BNN-R: An Out-of-Order Architecture for High-Performance and Regularized BNN Inference, 2021, 32: 199-213

[21] T. Geng, T. Wang, C. Wu, C. Yang, S. Song, A. Li, et al. LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism, 2019, 2160-052X: 9-16

[22] N. Narodytska. Formal Analysis of Deep Binarized Neural Networks. in: IJCAI:

[23] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, S. Sagiv, T. Walsh. Verifying Properties of Binarized Deep Neural Networks. in: AAAI:

[24] G. Chen, H. Meng, Y. Liang, K. J. I. T. O. P. Huang, D. Systems. GPU-Accelerated Real-Time Stereo Estimation With Binary Neural Network, 2020, 31: 2896-2907

[25] C.-H. J. I. A. Huang. An FPGA-Based Hardware/Software Design Using Binarized Neural Networks for Agricultural Applications: A Case Study, 2021, 9: 26523-26531

[26] N. Fasfous, M. R. Vemparala, A. Frickenstein, L. Frickenstein, W. J. I. I. P. Stechele, D. P. S. Workshops. BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices, 2021: 108-115

[27] Y. Ma, H. Xiong, Z. Hu, L. J. I. C. C. O. C. V. Ma, P. R. Workshops. Efficient Super Resolution Using Binarized Neural Network, 2019: 694-703

[28] Y. Li, T. Geng, A. Li, H. J. A. Yu. BCNN: Binary Complex Neural Network, 2021, abs/2104.10044:

[29] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. in: ECCV:

[30] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, Y. J. A. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients, 2016, abs/1606.06160:

[31] W. Tang, G. Hua, L. Wang. How to Train a Compact Binary Neural Network with High Accuracy? in: AAAI:

[32] X. Lin, C. Zhao, W. J. A. Pan. Towards Accurate Binary Convolutional Neural Network, 2017, abs/1711.11294:

[33] S. Darabi, M. Belbahri, M. Courbariaux, V. J. A. Nia. BNN+: Improved Binary Network Training, 2018, abs/1812.11800:

[34] M. Ghasemzadeh, M. Samragh, F. J. I. T. a. I. S. O. F.-P. C. C. M. Koushanfar. ReBNet: Residual Binarized Neural Network, 2018: 57-64

[35] B. Zhuang, C. Shen, M. Tan, L. Liu, I. D. J. I. J. C. V. Reid. Structured Binary Neural Networks for Image Recognition, 2022, 130: 2081-2102

[36] J. Bethge, C. Bartz, H. Yang, Y. Chen, C. J. I. W. C. O. a. O. C. V. Meinel. MeliusNet: An Improved Network Architecture for Binary Neural Networks, 2021: 1438-1447

[37] S. Zhu, X. Dong, H. J. I. C. C. O. C. V. Su, P. Recognition. Binary Ensemble Neural Network: More Bits per Network or More Networks per Bit?, 2019: 4918-4927

[38] J. Bethge, H. Yang, M. Bornstein, C. J. I. C. I. C. O. C. V. W. Meinel. BinaryDenseNet: Developing an Architecture for Binary Neural Networks, 2019: 1951-1960

[39] C. Trabelsi, O. Bilaniuk, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, et al. Deep Complex Networks, 2018, abs/1705.09792:

[40] S. Wisdom, T. Powers, J. R. Hershey, J. Le Roux, L. E. J. A. Atlas. Full-Capacity Unitary Recurrent Neural Networks, 2016, abs/1611.00035:

[41] H.-S. Choi, J.-H. Kim, J. Huh, A. Kim, J.-W. Ha, K. J. A. Lee. Phase-aware Speech Enhancement with Deep Complex U-Net, 2019, abs/1903.03107:

[42] Y. Cao, Y. Wu, P. Zhang, W. Liang, M. J. R. S. Li. Pixel-Wise PolSAR Image Classification via a Novel Complex-Valued Deep Fully Convolutional Network, 2019, 11: 2653

[43] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. S. Bajwa, et al. In-datacenter performance analysis of a tensor processing unit, 2017: 1-12

[44] M. Courbariaux, Y. Bengio, J.-P. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. in: NIPS:

[45] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. J. A. Bengio. Binarized Neural Networks, 2016, abs/1602.02505:

[46] A. G. Anderson, C. P. J. A. Berg. The High-Dimensional Geometry of Binary Neural Networks, 2017, abs/1705.07199:

[47] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, Y. Gal. An Empirical study of Binary Neural Networks' Optimisation. in: ICLR:

[48] F. Lahoud, R. Achanta, P. Márquez-Neila, S. J. A. Süsstrunk. Self-Binarizing Networks, 2019, abs/1902.00730:

[49] L. Hou, Q. Yao, J. T.-Y. J. A. Kwok. Loss-aware Binarization of Deep Networks, 2017, abs/1611.01600:

[50] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, et al. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference, 2017:

[51] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, N. J. A. Sebe. Binary Neural Networks: A Survey, 2020, abs/2004.03333:

[52] J. Gao, B. Deng, Y. Qin, H. Wang, X. J. I. G. Li, R. S. Letters. Enhanced Radar Imaging Using a Complex-Valued Convolutional Neural Network, 2019, 16: 35-39

[53] D. P. Reichert, T. J. C. Serre. Neuronal Synchrony in Complex-Valued Deep Networks, 2014, abs/1312.6115:

[54] G. M. Georgiou, C. Koutsougeras. Complex domain backpropagation.

[55] T. Kim, T. J. N. C. Adalı. Approximation by Fully Complex Multilayer Perceptrons, 2003, 15: 1641-1666

[56] L. C., B. J., Z. Z. A 256Gb/s/mm-shoreline AIB-Compatible 16nm FinFET CMOS Chiplet for 2.5D Integration with Stratix 10 FPGA on EMIB and Tiling on Silicon Interposer %J Proceedings of the Custom Integrated Circuits Conference, 2021, 2021-April:

[57] Y.-H. Chen, T.-J. Yang, J. S. Emer, V. J. I. J. O. E. Sze, S. T. I. Circuits, Systems. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices, 2019, 9: 292-308

[58] B. Moons, R. Uytterhoeven, W. Dehaene, M. J. I. I. S.-S. C. C. Verhelst. 14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI, 2017: 246-247

[59] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network, 2016: 243-254

[60] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, A. J. A. Krishnamurthy. VTA: An Open Hardware-Software Stack for Deep Learning, 2018, abs/1807.04188:

[61] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. R. Fojtik, et al. MAGNet: A Modular Accelerator Generator for Neural Networks, 2019: 1-8

[62] J. Cong, J. J. I. a. I. C. O. C.-a. D. Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation, 2018: 1-8

[63] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. W. Hwu, et al. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs, 2018: 1-8

[64] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, 2017: 1-6

[65] Y. Wang, J. Xu, Y. Han, H. Li, X. J. N. a. E. I. D. a. C. Li. DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family, 2016: 1-6

[66] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI, 2018: 1-14

[67] G. Zhou, J. Zhou, H. J. T. I. I. C. O. a.-C. Lin, Security,, Identification. Research on NVIDIA Deep Learning Accelerator, 2018: 192-195

[68] H. Genç, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration, 2021: 769-774

[69] A. Gonzalez, J. Zhao, B. Korpan, H. Genç, C. Schmidt, J. C. Wright, et al. A 16mm2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET, 2021: 259-262

[70] Y. Lecun, L. Bottou, Y. Bengio, P. J. P. I. Haffner. Gradient-based learning applied to document recognition, 1998, 86: 2278-2324

[71] D. A. Patterson, J. L. Hennessy. EBOOK : Computer Organization and Design; the Hardware / Software Interface, RISC V Edition.

[72] 周围. 基于FPGA的深度学习软硬件协同设计的仿真实现: [硕士学位论文]. 电子科技大学, 2022

[73] C. Guo, C. Li, J. Guo, C. C. Loy, J. Hou, S. T. W. Kwong, et al. Zero-Reference Deep Curve Estimation for Low-Light Image Enhancement, 2020: 1777-1786

[74] C. Li, C. Guo, C. C. J. I. T. O. P. A. Loy, M. Intelligence. Learning to Enhance Low-Light Image via Zero-Reference Deep Curve Estimation, 2022, 44: 4225-4238

# 附录1 攻读硕士学位期间取得的学术成果

# 附录2 部分程序代码

**宏定义声明：**

#define k\_CONFIG 0

#define k\_MVIN 2

#define k\_MVOUT 3

#define k\_COMPUTE\_PRELOADED 4

#define k\_COMPUTE\_ACCUMULATE 5

#define k\_PRELOAD 6

#define XCUSTOM\_ACC 3

#define ADDR\_LEN 32

**mvin指令声明：**

#define mvin(dram\_addr, sram\_addr, cols, rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, dram\_addr, \

((uint64\_t)(rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(cols) \

<< ADDR\_LEN) | (sram\_addr), k\_MVIN)

**mvout指令声明：**

#define mvout(dram\_addr, sram\_addr, cols, rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, dram\_addr, \

((uint64\_t)(rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(cols) \

<< ADDR\_LEN) | (sram\_addr), k\_MVOUT)

**preload指令声明：**

#define preload(BD, C, BD\_cols, BD\_rows, C\_cols, C\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(BD\_cols) \

<< ADDR\_LEN) | (uint64\_t)(BD), ((uint64\_t)(C\_rows) \

<< (ADDR\_LEN + 16)) | ((uint64\_t)(C\_cols) << ADDR\_LEN) \

| (uint64\_t)(C), k\_PRELOAD)

**compute.preloaded指令声明：**

#define compute\_preloaded(A, BD, A\_cols, A\_rows, BD\_cols, BD\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(A\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(A\_cols) << ADDR\_LEN) | \

(uint64\_t)(A), ((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(BD\_cols) << ADDR\_LEN) | (uint64\_t)(BD), \

k\_COMPUTE\_PRELOADED)

**compute.accumulated指令声明：**

#define compute\_accumulated(A, BD, A\_cols, A\_rows, BD\_cols, BD\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(A\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(A\_cols) << ADDR\_LEN) | \

(uint64\_t)(A), ((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(BD\_cols) << ADDR\_LEN) | \

(uint64\_t)(BD), k\_COMPUTE\_ACCUMULATE)

**卷积计算代码：**

conv(int batch\_size, int in\_channels, int in\_dim,

        int out\_channels, int kernel\_dim,

        int out\_dim,

        int stride, int padding,

        elem\_t input[batch\_size][in\_dim][in\_dim][in\_channels],

        elem\_t weights[out\_channels][kernel\_dim][kernel\_dim][in\_channels],

        acc\_t bias[out\_channels],

        elem\_t output[batch\_size][out\_dim][out\_dim][out\_channels]) {

    for (int b = 0; b < batch\_size; b++) {

        for (int orow = 0; orow < out\_dim; orow++) {

            for (int ocol = 0; ocol < out\_dim; ocol++) {

                for (int och = 0; och < out\_channels; och++) {

                    acc\_t result = bias[och];

                    for (int krow = 0; krow < kernel\_dim; krow++) {

                        for (int kcol = 0; kcol < kernel\_dim; kcol++) {

                            for (int kch = 0; kch < in\_channels; kch++) {

                                int irow = orow \* stride + krow - padding;

                                int icol = ocol \* stride + kcol - padding;

                                elem\_t pixel = irow < 0 || irow >= in\_dim ||

                                    icol < 0 || icol >= in\_dim ? 0 : input[b][irow][icol][kch];

                                result += weights[och][krow][kcol][kch] \* pixel;

                            }

                        }

                    }

                    output[b][orow][ocol][och] = result;

                }

            }

        }

    }

}