中国科学技术大学

博士学位论文

基于软硬件结合方法处理稀疏神经网络的不规则性

作者姓名: 周徐达

学科专业: 计算机系统结构

导师姓名: 周学海 教授　王超 副教授

完成时间: 二〇一八年八月六日

University of Science and Technology of China

A dissertation for doctor's degree

Addressing Irregularity in Sparse Neural Networks: A Cooperative Software/Hardware Approach

Author: Xuda Zhou

Speciality: Computer Architecture

Supervisors: Prof. Xuehai Zhou, Prof. Chao Wang

Finished time: August 6, 2018

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____　　签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定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，可以将学位论文编入《中国学位论文全文数据库》等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、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□ 公开　□ 保密（____ 年）

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘　要

神经网络已经在图像识别、目标检测、语音识别和自然语言处理等诸多领域受到广泛应用，并逐渐成为主导算法。然而，随着神经网络的拓扑结构朝着规模不断扩大，层数不断加深的方向发展，庞大的数据和计算量给传统计算平台带来巨大的挑战。虽然稀疏（包括神经元和权值稀疏）能够有效减少神经网络的参数，从而减少数据访问和计算量，但是同时会将稠密网络规则的拓扑结构转化为稀疏不规则的形式，从而阻碍处理平台，包括 CPU、GPU 和加速器充分利用稀疏特性获得性能的提升。

在本文中，我们提出了一个软硬件结合的方法来有效处理稀疏神经网络不规则的问题。

首先，基于大量的实验，我们观察到了局部收敛的现象，即在训练过程中，权值的分布并不是随机的，大的权重往往会聚集成簇。基于这个关键的观察，我们提出粗粒度的剪枝，大幅降低稀疏神经网络的不规则性。我们提出的粗粒度剪枝将多个突触作为一个整体进行裁剪，而不是裁剪单个突触。我们首先将突触分为多个块，当某个块满足特定条件时，该块中的所有突触将从网络拓扑中永久剪除。然后，我们对经过粗粒度剪枝的神经网络进行重训练保证网络的精度。值得注意的是，我们对神经网络迭代使用粗粒度剪枝和重训练，从而获得理想的稀疏度和精度。粗粒度剪枝可以将稀疏神经网络的不规则度平均减少 20.13 倍。同时我们提出了一种新的神经网络压缩算法，算法包括粗粒度剪枝，局部量化和熵编码三个步骤，在 AlexNet 和 VGG16 上分别获得了 79 倍和 98 倍的压缩比，远高于现有的两个最先进的神经网络压缩方法，即 Deep Compression（35 倍和 49 倍）和 CNNPack（39 倍和 46 倍）。

我们进一步设计了一个新型硬件加速器 Cambricon-S，用于有效处理剩余的稀疏神经元和突触的不规则性。新型加速器中最重要的特征是一个中枢神经元选择模块（NSM），它能够有效处理粗粒度突触稀疏。同时，加速器中的突触选择器模块（SSM）,Encoder 和权值解码模块（WDM）能够分别利用神经元稀疏，动态压缩神经元和利用局部量化。与最先进的稀疏的神经网络加速器 Cambricon-X 相比，我们的加速器能够获得 1.71 倍的性能提升，同时减少 1.75 倍能耗。为了减轻程序员的编程负担，我们还提出一个基于库的编程框架。编程框架中的编译器能够应用 loop tiling 和数据重用策略生成高效的加速器指令。

关键词：神经网络，稀疏，压缩，加速器

ABSTRACT

Neural networks have become the dominant algorithms rapidly as they achieve state-of-the-art performance in a broad range of applications such as image recognition, object detection, speech recognition and natural language processing. However, neural networks keep moving towards deeper and larger architectures, posing a great challenge to the huge amount of data and computations. Although sparsity has emerged as an effective solution for reducing the intensity of computation and memory accesses directly, irregularity caused by sparsity (including sparse synapses and neurons) prevents processing platforms, including CPU, GPU and accelerators from completely leveraging the benefits.

In this paper, we propose a cooperative software/hardware approach to address the irregularity of sparse neural networks efficiently.

Initially, based on a wide range of experiments, we observe the local convergence, namely larger weights tend to gather into small clusters during training rather than randomly distributed. Based on that key observation, we propose a software-based coarse-grained pruning technique to reduce the irregularity of sparse synapses drastically. Instead of pruning synapses independently, our proposed coarse-grained pruning prunes several synapses together. The synapses are firstly divided into blocks; a block of synapses will be permanently removed from the network topology if it meets specific criteria. We then employ the fine-tuning approach to retain the network accuracy. Note that we apply the coarse-grained pruning iteratively in training to achieve better sparsity and avoid the accuracy loss. The coarse-grained pruning can reduce the irregularity by $20.13\times$ on average. Then we introduce a novel compression algorithm, a three stage pipeline: coarse-grained pruning, local quantization and entropy encoding, that work together to reduce the storage requirement of AlexNet and VGG16 by $79\times$ and $98\times$, respectively. The compression ratio is much higher than that achieved by two existing state-of-the-art neural network compression methods, i.e., Deep Compression ($35\times$ and $49\times$) and CNNPack ($39\times$ and $46\times$).

We further design a hardware accelerator named Cambricon-S to address the remaining irregularity of sparse synapses and neurons efficiently. The novel accelerator features a central neural selector module (NSM) to leverage coarse-grained sparsity. Additional the synapse selector module (SSM), Encoder and weight decoding moduel (WDM) are used to leverage neuron sparsity, dynamically compress the neurons

and leverage local quantization, respectively. Compared with a state-of-the-art sparse neural network accelerator Cambricon-X, our accelerator is $1.71\times$ and $1.75\times$ better in terms of performance and energy efficiency, respectively. To ease the burden of programmers, we also propose a high efficient library-based programming environment for our accelerator. The compiler is able to apply loop tiling and data reuse strategies for highly efficient instructions.

Key Words: neural networks, sparsity, compression, accelerator

目　录

第 1 章　绪　　论

作为现代人工智能的最重要的分支，人工神经网络（artificial neural network, ANN）近年来获得迅速发展。学术界不断提出新的神经网络理论，算法和结构，提高网络模型的表征能力和性能，从而解决各类人工智能问题；工业界也相继将神经网络应用于目标检测，语音识别，自动驾驶等领域。然而随着神经网络解决的问题越来越复杂，神经网络也朝着规模越来越大，拓扑结构越来越复杂，层数越来越深的方向发展，因此神经网络需要庞大的存储资源，计算资源和能耗完成运算，这不仅给传统的处理平台如 CPU 和 GPU 带来巨大的挑战，而且导致神经网络应用无法部署到嵌入式设备或者移动智能设备中。

研究者从软件和硬件两个方面来来试图解决这个问题。一个方面，近些年来，研究者利用稀疏（sparsity），量化（quantization），矩阵分解（matrix decomposition），熵编码（entropy encoding）等算法缩小神经网络的规模，实现神经网络深度压缩；其中稀疏被证明是其中最有效的一个策略，稀疏对神经网络的拓扑结构进行裁剪，从而减少神经网络参数数量和计算量，使得稀疏神经网络能够部署到嵌入式设备中。另一个方面，研究者提出并设计了神经网络专用加速器，从而代替 CPU/GPU 完成神经网络运算，减少神经网络执行时间，降低能耗。

然而，稀疏技术会使得原始稠密神经网络规则的拓扑结构转变为稀疏不规则的结构，因此使得 CPU，GPU 和部分加速器不能从稀疏特性中获得性能提升；尽管有部分加速器能够支持稀疏，但是效果并不理想。因此本文提出了一种软硬件结合的方法处理稀疏神经网络的不规则性，从算法和架构设计的角度提出多种优化策略，显著提升处理稀疏网络的效果。

本章首先介绍神经网络算法的发展历程和神经网络处理平台的概况。然后介绍本文的主要研究内容及贡献。最后介绍本文的章节安排。

1.1　神经网络算法发展历程

人工神经网络的发展和成熟已经经过了四个时期，每个大约二十年，分别是 1940 年代和 50 年代，1960 年代和 70 年代，1980 年代和 90 年代，2000 年至今。第一个时期主要建立了单个神经元的模型和学习规则。第二个时期主要建立了单层网络的模型和学习规则。第三个时期是多层神经网络蓬勃发展的时期，神经网络开始应用到多个领域。第四个时期神经网络的理论获得长足发展，同时深度神经网络在涌现同时被不断优化和改善，深度神经网络在图像处理，视频处理，自然语言处理等领域有着越来越广泛的应用。

1.1.1　第一个时期（1940 年代到 50 年代）

第一个时期，研究者提出单个神经元的模型对应的学习规则。

学术界认为神经网络领域的先锋是 McCulloch et al. [1]，他首次正式提出了简洁又统一的神经元模型。

1949 年， Hebb [2], Gerstner et al. [3]首次提出了生物神经网络中存储信息的单位–突触（synapse），并提出了学习过程就是修改突触的过程这一条重要的规则，这条规则是目前许多网络学习训练的基础。

1952 年， Hodgkin et al. [4]利用动态方程建立了神经元脉冲发射的模型。

1956 年 Taylor [5]使用联想记忆神经网络（associative memory）的方法模仿了许多不同的认知模型。

1958 年，Rosenblatt [6]提出了一套学习突触权值和神经元阈值的方法，最终完成了一个特定的信息处理任务，因此作者在这篇文章中提出了感知器（perceptron）的原型。

1.1.2　第二个时期 (1960 年代到 70 年代)

第二个时期，研究者重点研究的是单层网络的模型和学习规则。

1960 年， Widrow et al. [7]采用最小二乘法（least mean-square algorithm）提出了线性适应元（adaptive linear element），这是一种自适应的模式分类神经元。

FitzHugh-Nagumo 神经元模型 [8] 被广泛用于模拟感官系统中生物神经元的振荡行为。它使用相空间结合霍奇金-赫克斯利方程（Hodgkin-Huxley equations）的方法将神经纤维的生理状态划分为多个区域（包括活跃，压抑，增强，休息等），形成一个生理状态图，从而帮助研究者观察多种生理现象。

1967 年 Minsky [9]从自动机理论和计算理论的角度深入研究了研究了 McCulloch 和 Pitts 提出的神经元模型。1969 年 Minsky [10]发现了单层感知机在处理信息任务时的局限性，并提出了多层感知机，然而多层感知机的学习规则在当时并没有深入研究，因此阻碍了神经网络的发展。

Cainiello [11]提出当神经元受到的激励处于某个阈值时，该神经元不能被激活。1972 年， Nagumo et al. [12]在此基础上提出了一种非线性的神经元模型，当一个恒定频率的脉冲序列作用于神经元时候，由于脉冲序列的振幅随时间减少，神经元被激活的频率也会随之减少。

1.1.3　第三个时期（1980 年代到 90 年代）

第三个时期，研究者开始重点研究多层神经网络的模型和学习规则。在这个时期，贝叶斯方法，高斯过程，支持向量机（support vector machine，SVM）和多

层神经网络都试图与生物模型结合，从而解释大脑的认知，学习，记忆等功能。

1982 年，Hopfield [13]将统计物理的概念应用于循环神经网络模型，提出了突触对称连接的神经网络，即 Hopfield nets，这是 80 年代神经网络复兴的主要原因之一。这种简单优美的模型将记忆储存为激活值中的模式。

之后，广义 Hopfield 网络（generalized Hopfield network，GHN）用多层神经网络代替了单层神经网络结构，该结构被不断研究和发展。 Zurada et al. [14]证明 GHN 拥有多个稳定状态，能够保证原始 Hopfield 网络的稳定性。因此，各种基于 GHN 的网络不断被提出来解决不同的问题。

1984 年， Hindmarsh et al. [15]提出了 henomenological model，它能够将复杂的微分方程减少到三个，但是这个相对简单的模型能够解释合理地神经元的震荡现象。

1985 年， Ackley et al. [16]提出了玻尔兹曼机（Bolzmann machine），它在信息理论方面有着方唱广泛的应用。玻尔兹曼机基于模拟退火算法（simulated annealing），产生周期性的随机网络，并且根据玻尔兹曼分布模拟产生输入信号。其中， Kirkpatrick et al. [17], Černỳ [18]提出的模拟退火算法是一个基于统计力学来解决优化问题的方法。

1986 年， Rumelhart et al. [19]应用反向传播（back propagation，BP）算法来训练神经网络，从而解决了 Minsky [10]提出的多层感知机难以进行学习的问题。由于多层前馈网络的训练经常采用 BP 算法，人们也常把将多层前馈网络直接称为 BP 网络。BP 网络能学习和存贮大量的输入-输出模式映射关系，而无需事前揭示描述这种映射关系的数学方程。它的学习规则是使用最速下降法 (梯度下降法)，通过反向传播来不断调整网络的权值和阈值，使网络的误差平方和最小。

90 年代有不少研究将神经网络与概率理论相结合，从而诞生了统计机器学习方法（statistical machine learning）。MacKay [20], Bishop et al. [21]将贝叶斯技术（Bayesian technique）应用于到神经网络中，用于解决推理，回归和分类等问题。同时， Williams et al. [22]将高斯方法论（methodology of Gaussian ）引入到神经网络领域。

1998 年，Vapnik [23]提出了支持向量机（SVM）。SVM 的基本想法是利用 kernel 产生非线性表示，从而解决非线性的问题。支持向量机在模式识别，回归和密度估计等方面有着非常广泛的应用，尽管 SVM 被有些研究者被认为是神经网络的一个分支，但是学术界普遍认为 SVM 和神经网络是两个不同的领域。

1.1.4　第四个时期（2000 年代至今）

第四个时期，研究者开始研究深度神经网络（deep neural network, DNN），深度神经网络由多种不同的层构成，输入数据依次通过各个层被逐层处理，最终

被分类, 识别或者检测。深度神经网络还包括卷积神经网络（convolutional neural network, CNN）和循环神经网络（recurrent neural network）；前者广泛应用于计算机视觉 [24-26]，而后者广泛应用于语音处理 [27-29] 或者自然语言处理 [30]。最近轰动全球的成果就是谷歌旗下的 Deep Mind 团队所设计的 Alpha Go [31]，在围棋上先手战胜李世石和柯杰等顶级棋手。下面主要介绍神经网络在图像处理领域和目标检测领域的应用和发展，然后介绍神经网络的低功耗技术。

1. 神经网络在图像分类领域的应用

在 2012 年，深度神经网络在图像识别和分类领域领域取得了非常优秀的效果， Krizhevsky et al. [24]提出的 AlexNet 网络结构，在 ILSVRC-2012 上 top-5 的错误率为 34.2%。AlexNet 的主要结构是 5 个卷积层和 3 个全连接层，其中卷积层用于提取图像的特征，全连接层用来整合卷积层提取出的特征并进行分类。

研究者发现，深度神经网络比浅层神经网络具有更强的表征能力，拥有更高的精度。在 2014 年， Simonyan et al. [25]提出了 VGG16 网络，VGG16 共由 13 个卷积层和 3 个全连接层，13 个卷积层对比 AlexNet 的 5 个卷积层，能够提出去更高层次的特征。VGG16 在 ImageNet 数据集 [32] 上 top-5 的错误率为 25.3%。

然而网络的层数不能无限增加，网络层数过多，不仅会导致梯度消失（vanishing gradient），梯度爆炸（exploding gradient）或者过拟合（over-fitting）的问题，导致神经网络难以进行训练；还会出现网络退化的问题，导致神经网络的表征能力下降，精度降低。因此，研究者提出了不同的网络拓扑结构，试图增加网络的层数，增加网络的表征能力。

Szegedy et al. [33]提出了 GoogLenet 网络，GoogLenet 网络深度达到了 22 层。它最主要的特征在于使用了 inception 结构，从而更好地利用了网络中的计算资源，并且在不增加计算负载的情况下，增加网络的宽度和深度。inception 结构的主要思想在于卷积视觉网络中一个优化的局部稀疏结构怎么样能由一系列易获得的稠密子结构来近似和覆盖。为了避免 patch 对齐问题，作者限制了 inception 结构中滤波器的大小为 1x1，3x3，5x5。最终 GoogleNet 网络在 ImageNet 数据集上 top-5 的错误率为 26.7%。

He et al. [34]在 2016 年提出了深度残差网络（deep residual network），在 ImageNet 中斩获图像分类、检测、定位三项的冠军。ResNet 通过 shortcut 的结构进行参差学习，解决了增加深度带来的退化问题，这样能够通过单纯地增加网络深度，来提高网络性能，作者在 ImageNet 上实验了一个 152 层的残差网络，比 VGG 深 8 倍，最终取得了 3.57% 的 top-5 错误率。

2. 神经网络在目标检测领域的应用

除了图像识别/分类，深度神经网络在目标检测领域也有广泛应用，并且取得了非常理想的效果。2014 年，Girshick et al. [35]提出了 RCNN 网络解决目标检测问题，RCNN 的主要思想是将检测问题转化成为分类问题。RCNN 使用 region proposal 来得到有可能得到是 object 的若干（大概 10^3 量级）图像局部区域，然后把这些区域分别输入到 CNN 中，得到区域的 feature，再在 feature 上加上分类器，判断 feature 对应的区域是属于具体某类 object 还是背景。当然，RCNN 还用了区域对应的 feature 做了针对 bounding box 的回归，用来修正预测的 bounding box 的位置。RCNN 在 VOC2007 上的 mAP 是 58% 左右。

RCNN 存在着重复计算的问题（proposal 的 region 有几千个，多数都是互相重叠，重叠部分会被多次重复提取 feature），于是 Girshick [36]借鉴 He et al. [37]的 SPP-net 的思路提出了 Fast-RCNN，跟 RCNN 最大区别就是 Fast-RCNN 将 proposal 的 region 映射到 CNN 的最后一层卷积层的 feature map 上，这样一张图片只需要提取一次 feature，大大提高了速度，也由于流程的整合以及其他原因，在 VOC2007 上的 mAP 也提高到了 68%。

Fast-RCNN 的速度瓶颈在 Region proposal 上，于是 Ren et al. [26]提出了 Faster-RCNN，将 Region proposal 也交给 CNN 来做。Fater-RCNN 中的 region proposal netwrok 实质是一个 Fast-RCNN，这个 Fast-RCNN 输入的 region proposal 的是固定的（把一张图片划分成 $n \times n$ 个区域，每个区域给出 9 个不同 ratio 和 scale 的 proposal），输出的是对输入的固定 proposal 是属于背景还是前景的判断和对齐位置的修正（regression）。Region proposal network 的输出再输入第二个 Fast-RCNN 做更精细的分类和 bounding box 的位置修正。Fater-RCNN 速度更快了，而且用 VGG 作为 feature extractor 时在 VOC2007 上 mAP 能到 73%。

神经网络在目标检测领域还在持续发展，研究者不断提出新的网络和方法如 Mask-RCNN [38]，YOLO [39] 等来提高网络运行速度，提高网络的精度。

3. 神经网络的低能耗技术

随着神经网络的应用越来越广泛，神经网络的规模不断扩大，层数不断加深，因此未来的大规模网络很难部署到嵌入式系统中。很多研究人员致力于减少神经网络的参数规模，从而减少对存储资源，计算资源和带宽的需求，加快神经网络运行速度，降低神经网络的能耗。研究者提出了许多方法来解决这个问题，主要可以分为低精度计算和裁剪技术。

低精度计算主要将神经元的权值或神经元进行低比特量化，从而降低网络存储需求和运算能耗。低精度量化包括 16 比特定点量化 [40]，8 比特量化 [41]，

二元权值量化 [42]，三元权值量化 [43]，2 的幂次等形式的量化 [44]；在对神经网络参数量化过程中，可能会降低网络的精度，尤其是需要注意两个情况：第一，当采用极低比特量化神经网络时，可能导致精度损失；第二，当对神经网络的权值和神经元同时进行量化时，可能导致精度损失。

神经网络的裁剪技术采用剪枝直接减少神经网络中参数数量。剪枝技术 [45-49] 能够修剪神经网络中不必要的连接。如果神经元或者权值满足某些条件，那么这些神经元/权值将会被修剪，从而降低网络的复杂性，最终能够在不影响精度的情况下获得具有良好泛化性能的精简神经网络。值得注意的是，对神经网络进行裁剪会破坏稠密网络规则的拓扑结构，引入不规则性。

除了上述的两种方法，研究者还采用矩阵分解，例如矩阵因式分解 [50-51] 的对权值矩阵或者卷积核进行分解，在保持模型规则计算的情况下，实现神经网络压缩和加速。

1.2　神经网络计算平台发展

主流的神经网络计算平台是通用处理器，即 CPU 和 GPU。然而 CPU 的性能并不能满足实际的需求，而 GPU 的的能耗过高。因此研究者开始为神经网络算法设计专用的加速器，使得神经网络算法在加速器上既能够满足高性能的需求，又能够满足低能耗的需求。目前有两个流行的加速平台，分别是可编程逻辑门阵列（field-programmable gate array，FPGA）和专用集成电路（application specific integrated circuit , ASIC）。

1.2.1　通用处理器 CPU 和 GPU

CPU [52-53] 和 GPU [54-57] 是两个完成神经网络运算的通用平台。同时，在 CPU 和 GPU 上，有许多针对神经网络的高级编程框架，如 Caffe [58]，Tensorflow [59], MXNet [60], 使得用户能够方便地将神经网络算法部署到 CPU 和 GPU 上进行计算。

Vanhoucke et al. [53]在 CPU 上采用多种策略对神经网络算法进行优化，包括循环展开，使用 SIMD 指令并行累加，调整数据保证内存对齐，使用 8 比特量化减少内存占用等，最终获得了 4 倍的性能提升。

CPU 集群更能够满足大规模网络应用的性能需求。谷歌提出了由数万个 CPU 组成的神经网络训练框架——DistBelief [61]，并用它来训练由数十亿个参数构成神经网络，训练数据集为 1000 万张 200×200 的图片，训练时间长达 3 天，最终在 ImageNet 数据集上获得了 15.8% 的错误率。

在 CPU 集群中，由于核之间的通信延迟高，带宽低，因此核之间的通信会成

为整个系统的瓶颈。GPU 采用完全不同的体系结构，它采用单任务多数据（single program multiple data, SPMD）框架，兼有分布式存储和共享内存的优点，具有非常高的并行性。因此，GPU 对于计算密集（compute intensive）型应用，如 CNN 和 DNN，具有非常高的加速效果，性能完全超过 CPU。2004 年，Oh et al. [62]成功地将神经网络部署到 GPU 进行运算。2010 年， Scherer et al. [55]使用 CUDA 框架将大规模 CNN 部署到 GPU，在训练过程中，每一个 block 负责一个 batch 运算，每一个 thread 负责一个权值的运算，同时利用共享内存重用输入数据或错误信号，从而大大提高内存利用率。更进一步的，不少研究者开始全面探究如何使用 GPU 加速机器学习算法。

然而，CPU 和 GPU 在处理神经网络时的效率并不高，因为这些通用处理器首先考虑的是通用性和普遍性，许多处理逻辑（包括计算和存储）对神经网络计算的支持并不理想。从计算的角度考虑，通用处理器只支持基本的计算，复杂的算术运算是通过一系列的基本计算组成，因此寄存器和内存之间需要进行频繁的数据交换，这不仅降低性能，还会带来大量的能耗。从存储的角度考虑，通用处理器采用冯 · 诺依曼体系中的层次化存储结构，缓存容量通常小于其他专用处理器，例如 NVIDIA K20 的片上缓存只有 $1.5MB$，Intel E78880L 的片上缓存不超过 $50MB$。然后，随着神经网络的迅猛发展，神经网络的参数朝着十亿 [63] 甚至百亿 [57] 发展。有限的片上缓存与超大规模的网路参数之间的矛盾会导致频繁的片上缓存与片外存储之间的数据交换，进一步降低处理器性能，增加能耗开销。

1.2.2　FPGA

FPGA 能够为计算密集型的应用（如 CNN，DNN 等）提供大量的逻辑资源，包括计算资源和存储资源。FPGA 的可编程性和可重构特性允许用户自定义设计，同时能够在很短时间内完成设计评估，从而缩短开发周期，节约开发费用。尽管 FPGA 的性能和能效都比不上 ASIC，但是研究人员仍然可以选择用 FPGA 代替 ASIC 设计神经网络加速器。

2009 年，Farabet et al. [54]在 Xilinx Virtex-4 SX35 平台上设计了一个卷积神经网络处理核，它采用多个 DSP 并行计算多个输出神经元的部分和。但是由于 FPGA 资源的限制，系统只能够运行二维卷积运算，因此无法利用不同卷积核之间的并行性。

随后在 2011 年， Farabet et al. [64]在 Xilinx Virtex 6 平台上设计了基于可扩展的数据流的架构 Neuflow，并且获得了 100 倍的加速比。Neuflow 包含多个 tile，每一个 tile 集成了多个一维卷积运算单元和乘加器（multiply-add accumulation, MAC），从而完成二维卷积运算；多个 tile 可以并行执行二维卷积运算从而完成

三维卷积运算。

Gokhale et al. [65]在 Xilinx ZC706 平台上设计了 nn-X。这是一个可扩展的，低功耗的协处理器，主要用来加速深度神经网络。nn-X 协处理器由 8 个运算单元组成，每个运算单元集成了一个卷积运算单元，池化运算单元和非线性运算单元; 运算单元通过 4 通道的 DMA 与片外 DDR3 内存进行互联。nn-X 的理论峰值能够达到 227GOP/s，实际吞吐率为 200GOP/s；整个设计的功耗为 $8W$，其中核心计算模块的功耗为 $4W$。

Zhang et al. [66]在 VC707 板卡上实现了一个吞吐量为 $61.62GFLOPS$ 的 CNN 加速器，工作频率为 $100MHz$。作者采用 loop tiling 对计算进行划分，并采用 roofline model 定量分析计算资源与带宽需求，从而获得最佳配置方案。Suda et al. [67]在 Zhang et al. [66]的基础上进一步进行研究，他们提出了一个基于 FPGA 资源约束的系统设计空间搜索（design space exploration）方案，最大化 CNN 加速器在 FPGA 上的性能。

Qiu et al. [68]在 Xilinx Zynq ZC706 板卡上实现了一个 CNN 加速器，在 $150MHz$ 的频率下性能为 $137GOP/S$。作者对全连接层的权值进行奇异值分解（Singular Value Decomposition, SVD），从而减少内存访问，同时采用动态精度量化的方法对神经网络参数进行量化，从而减少参数存储量，进一步减少逻辑资源和能耗。

除了 CNN 网络，FPGA 还能够加速其他类型的网络。Rice et al. [69]在 FPGA 上实现了分层贝叶斯网络模型，对比于的 Cray XD1 平台上的纯软件实现能够获得 75 倍。Kim et al. [70]在 FPGA 上实现了深度信念网络模型，对比 CPU 实现能够获得 24 ˜ 30 倍的加速比。

1.2.3 ASIC

ASIC 是设计师自定义功能的定制电路，相比于 CPU，GPU 和 FPGA，它具有体积小，功耗低，可靠性高，性能高等优点。

CNN 这类有高度并行性的计算密集型应用适合部署到 ASIC 上，因此早期的研究主要集中在如何设计加速器来快速实现卷积操作以满足实时处理的需求。Lee et al. [71]提出了一个能够完成任意规模二维卷积运算的架构，其中处理单元采用二维网格结构互相连接。 Stearns et al. [72]提出了 L64240 的过滤器，计算单元的规模为 8×8，工作频率为 $20MHz$。 Kamp et al. [73]提出了一个可编程的规模为 7×7 二维卷积核结构，工作频率为 $20MHz$，内部缓存区的每一行最多能够存储 512 个像素。 Hecht et al. [74]设计了脉动阵列的架构完成二维卷积运算。更进一步的， Lee et al. [75]提出了一个超级脉动阵列的结构完成二维卷积运算，其中阵列单元的输入输出带宽仅仅为 1 比特，能够在没有性能损失的情况下减

少 2/3 的面积。

表 1.1 DianNao Family 详细参数

accelerator	techonology(nm)	area(mm^2)	power(nm)	throughput(GOP/S)	Application
DianNao	65	3.02	0.485	452	neural network
DaDianNao	28	67.73	15.97	5584	neural network
PuDianNao	65	3.51	0.596	1056	machine learning
ShiDianNao	65	4.86	0.320	194	CNN

DianNao Family [76] 是一系列的神经网络加速器，包括 DianNao [77]，DaDianNao [77]，PuDianNao [78] 和 ShiDianNao [79] 。我们在表 1.1列出了它们的详细参数。DianNao 是加速器家族的第一个成员，它能够完成通用的神经网络运算，如 CNN，DNN 等。DaDianNao 是 DianNao 的多核版本，它采用 eDRAM 作为片上缓存尽可能将神经网络的参数存储到片上，从而避免高额的片外访存开销。PuDianNao 是机器学习处理器，它除了能够完成神经网络运算，还支持 KNN，K-Means 等多种机器学习算法。ShiDianNao 是面向嵌入式设备的卷积神经网络处理器，它采用脉动阵列的形式完成神经网络的卷积运算，从而提高数据复用，减少访存能耗，实现端到端的神经网络应用。

除了 DianNao 系列神经网络加速器，最近还涌现出越来越多的加速器，如 Farabet et al. [64]提出了 Neuflow， Chen et al. [80]提出了 Eyeriss，Google 提出了 TPU [81]。它们均采用脉动阵列的形式完成神经网络的运算。

稀疏是减少神经网络参数规模的重要技术，但是同时会将原始规则的计算变得不规则；CPU 和 GPU 对稀疏支持并不理想 [82]，上述的神经网络加速器均无法利用神经网络的稀疏特性。因此近年来研究者着手研究能够支持稀疏神经网络的加速器 [82-87]。它们的优缺点如表 1.2所示。

表 1.2 现有支持稀疏的神经网络加速器的比较

	权值稀疏	神经元稀疏	注释
Eyeriss [83]	✗	✔	只能跳过计算，无法带来性能提升
Cambircon-X [82]	✔	✗	
Cnvlutin [84]	✗	✔	
EIE [85]	✔	✔	只适合全连接层
ESE [86]	✔	✗	只适合稀疏的 LSTM 层
SCNN [87]	✔	✔	需要额外计算部分和的坐标，全连接层效果不理想

1.3　主要研究内容及贡献

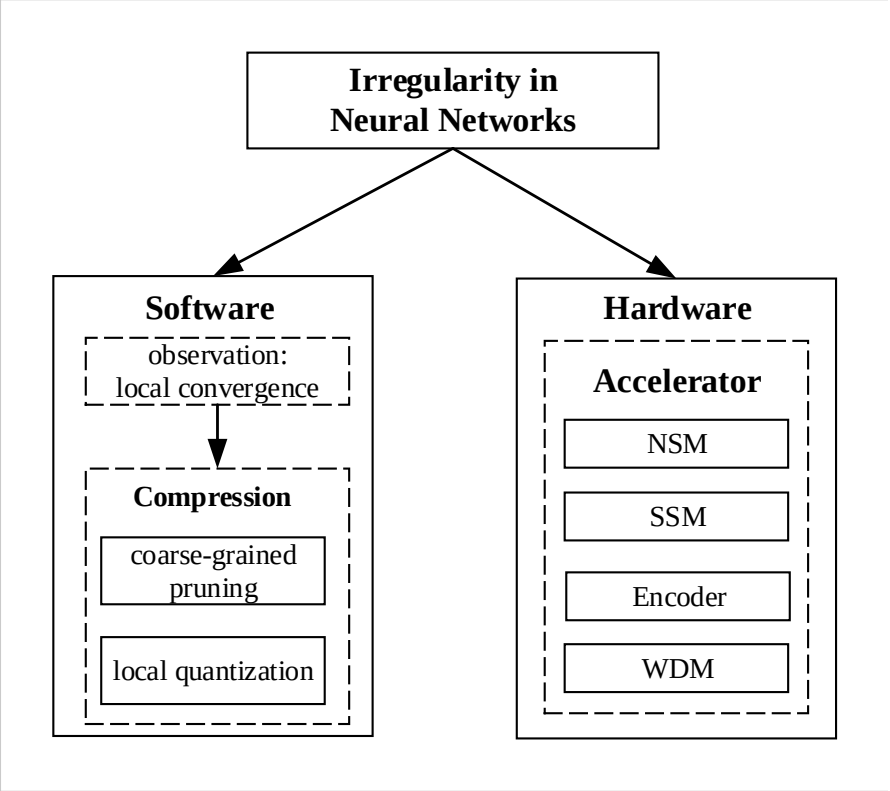


图 1.1　本文主要研究内容和创新点

最近的研究 [44, 48, 88-89] 表明，神经网络可以在保持精度的前提下，通过稀疏，量化，低精度等方式进行深度压缩，从而减少参数存储量，计算量和访存量。然而，稀疏技术会使得原本规则的神经网络计算变得不规则，因此 CPU，GPU 以及一些不支持稀疏的加速器并不能从中受益。尽管，最近出现了一些能够支持稀疏的神经网络加速器 [82-87]，但是挖掘稀疏的效果并不理想，利用稀疏的效率也不高。

本文提出了一种软硬件结合方法（software/hardware cooprative approach）来处理稀疏神经网络的不规则性。我们首先提出了粗粒度剪枝算法，从而减少稀疏神经网络的不规则性，同时我们结合局部量化和熵编码对神经网络进行深度压缩。然后我们设计了粗粒度稀疏神经网络加速器 Cambricon-S，来处理粗细度稀疏的神经网络，新型加速器有高性能和和低能耗的特点。本文的主要贡献如图 1.1所示，可以归纳如下：

1. 本文首次提出采用软硬件结合的方法处理稀疏神经网络不规则性。在软件方面我们提出了一种新的压缩算法对神经网络进行深度压缩，压缩算法中包含的粗粒度稀疏算法能够减少稀疏神经网络的不规则性；在硬件方面，我们提出了粗粒度稀疏神经网络加速器来 Cambricon-S 高效处理粗粒度稀疏，从而获得性能提升，同时降低能耗。

2. 本文观察到了局部收敛（local convergence）的现象，即在训练过程中，神经网络的权值不是一种随机分布的情况，大的权值会聚集成簇，我们通过大量实验证明局部收敛的现象出现在许多的神经网络中。本文根据观察到的局部收敛现象，提出了一种新的神经网络压缩算法。压缩算法包括三个步骤，分别是粗粒度剪枝（coarse-grained pruning），局部量化（local quantization）和熵编码（entropy coding）。粗粒度剪枝将神经网络的权值分为多个权值块，当一个权值块符合某个条件时将从网络拓扑中被完全剪除，粗粒度稀疏的神经网络对比与细粒度稀疏的神经网络拥有更低的不规则度。局部量化将权值分为多个子区域，然后在每一个子区域内分别进行量化，进一步压缩神经网络。之后，我们采用熵编码对神经网络进行无损压缩。新的神经网络算法不仅能够降低稀疏神经网络的稀疏度，还最终获得非常理想的压缩比。

3. 本文设计并提出了首个能够支持粗粒度稀疏神经网络的加速器微结构。该微结构不仅能够处理普通的稠密神经网络，还能够通过打开/关闭稀疏处理模块支持多种稀疏/量化情况，包括神经元稀疏，粗粒度权值稀疏，神经元/权值同时稀疏，局部量化等。新型加速器能够非常高效的利用稀疏和量化，获得非常理想的性能和能耗。

4. 本文针对神经网络加速器的计算特性，设计了专用的性能模拟器。该模拟器能够高速，精准地模拟神经网络在加速器的运行性能。

1.4　论文的组织结构

本文第一章介绍了本文的工作背景。我们主要从神经网络算法和神经网络加速器两方面介绍神经网络的发展简史，基本原理和面临的问题，在此基础上，我们提出了本文的主要研究内容和创新点。

本文第二章介绍了神经网络相关的背景知识。首先我们介绍了神经网络算法，对神经网络的多种不同类型的层计算进行详细分析。然后我们介绍了神经网络的低能耗技术，包括低精度计算技术和裁剪技术。最后我们介绍最新神经网络加速器的相关工作，其中我们重点介绍了稀疏神经网络加速器。

本文第三章提出了一种新的神经网络压缩方法。首先我们观察到神经网络的权值在训练过程中有局部收敛的现象，即大的权值容易聚集成簇。基于局部收敛，我们提出了粗粒度剪枝的策略，将多个权值同时进行裁剪操作；然后我们提出局部量化的方案进一步利用局部收敛，减少表示权值的比特数，进一步对神经网络进行压缩；最后我们利用熵编码对神经网络进行无损压缩。最后通过大量实验证明，新的压缩方法能够对神经网络进行深度压缩，同时减少稀疏神经网络的不规则性。

本文第四章提出了一个粗粒度稀疏神经网络加速器，能够快速处理经过深度压缩的神经网络。首先，我们观察粗粒度稀疏神经网络的特点，提出了设计加速器的三个原则。然后我们根据这三个设计原则，提出了新型加速器的架构，新的加速器能够充分利用神经元稀疏，粗粒度权值稀疏和局部量化，从而提高性能，降低功耗。最后我们为新型加速器设计了专用的编程框架，减轻用户的编程负担。

本文第五章提出了一个新型加速器专用的性能模拟器，我们根据新型加速器的特性，设计了一个性能模拟器代替周期精确模拟器，从而在误差允许的范围内，快速完成对加速器的性能评估。

本文第六章，我们对加速器进行详细的性能和能耗评估，在七个 benchmark 上，我们将 Cambricon-S 与 CPU，GPU，DianNao 和 Cambricon-X 进行性能和能耗的比较。最后我们根据实验结果详细分析了新型加速器能够获得高性能和低能耗的原因。

最后一章我们对本文进行总结，并展望了未来的研究工作。

第 2 章　神经网络简介

2.1　神经网络算法基础

现代神经网络由多种不同类型的层组成，输入数据依次通过各个层被逐层处理，最终被分类, 识别或者检测。在每一层中，神经元接收多个输入进行处理，然后通过连接将输出发送到下一层。神经元之间的连接，即所谓的突触，通常具有独立或者共享的权重。神经网络在图像处理，语音识别，语音合成，自然语言处理等领域有着非常广泛的应用。目前在图像处理应用最广泛的网络是卷积神经网络（convolutional neural networks, 简称 CNNs）和深度神经网络（deep neural networks, 简称 DNNs），它们由卷积层，池化层，归一化层，激活层和全连接层等组成。递归神经网络（recurrent neural networks，简称 RNNs）是一类重要的机器学习技术，专门用于处理顺序数据序列和可变长度数据序列。RNN 在语音识别，自然语言处理，场景语义理解和时间序列分析等方面有着广泛的应用。其中应用最广泛，性能最高的两种类型的 RNNs 是长短时记忆网络（long short term memory, 简称 LSTM）和门控循环单元 (gated recurrent unit, 简称 GRU)。下面将为大家介绍神经网络中集中常见的神经网络层类型。

2.1.1　全连接层

全连接层（fully connected layers）是神经网络算法中常见的一种层类型，主要用来将上一层提取的特征进行组合，综合以及分类，在整个神经网络中起到“分类器” 的作用。全连接层的结构如图 2.1所示，输入层神经元为 m 个，输出层的神经元为 n 个，其中每一个输出神经元与所有输入神经元相连，其结构相当于 n 个 m 输入的感知机，因此全连接层也可以看成是感知机扩展。

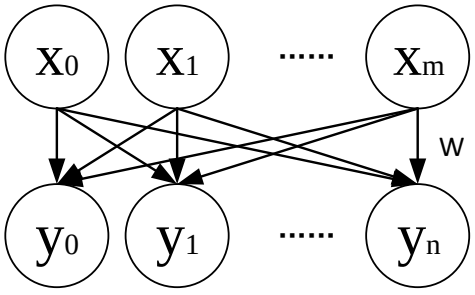


图 2.1　全连接层

全连接层的核心操作操作是矩阵向量乘积，

$$y = W * x + b \tag{2.1}$$

其中 y 是输出神经元向量，x 是输入神经元向量，W 是权值矩阵，b 是输出神经元的偏置向量。因此全连接层的本质是由一个特征空间线性变换到另一个特征空间，且目标空间的任一维都会受到源空间每一维的影响。由于每个输出神经元与所有输入神经元相连接，当神经网络输出和输入规模比较大时，会造成权值规模非常庞大，从而造成网络难以训练，并且会出现过拟合的情况。因此，对于图像处理等这类输入规模庞大的应用，我们不采用全连接层来提取特征，而采用共享权值的卷积层提取特征，全连接层常出现在最后几层，对卷积层提取的特征进行整合和分类。

2.1.2　卷积层

卷积层（convolutional layers）是卷积神经网络的核心层，主要用于提取特征。卷积层基于生物学上感受野（receptive field）的机制而提出的。感受野主要是指听觉系统、本体感觉系统和视觉系统中神经元的一些性质。比如在视觉神经系统中，一个神经元的感受野是指视网膜上的特定区域，只有这个区域内的刺激才能够激活该神经元。在听觉系统中，对于语音则是某一时间戳后的时间段才能激活神经元。卷积层充分借鉴感受野的机制，神经元仅仅能够感受局部特征，而卷积核的大小直接限制感受野的大小，输出神经元只和周围一小块的输入神经元存在连接。局部连接的方法大大减少了卷积层中的权值数量。

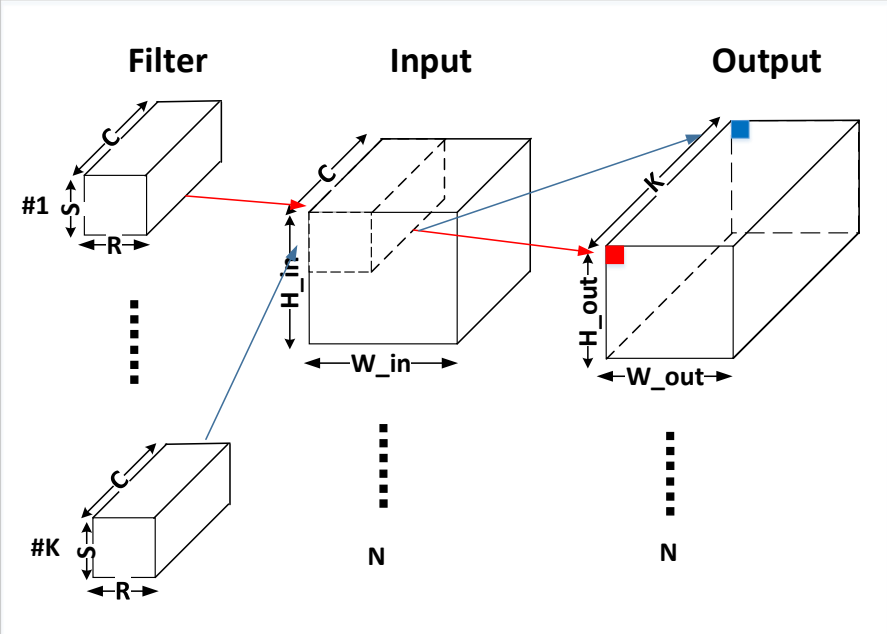


图 2.2　卷积层

卷积层采用了共享权值的方法进一步减少权值的数量，即同一输出特征图共享一组卷积核。同一个输出特征图像上的所有像素点是同一组卷积核在输入图像不同位置上提取到的图像特征, 而这些图像特征具有相同意义。卷积层采用多组不同的卷积核对输入特征图进行卷积, 从而获得多组输出特征图, 即多组不同的图像特征。

卷积层的结构如图 2.2所示。卷积层的核心运算是二维滑动窗口卷积运算，即规模为 $R \times S$ 的卷积核在规模为 $W_{in} \times H_{in}$ 的输入特征图上滑动进行二维卷积，最终产生规模为 $W_{out} \times H_{out}$ 的输出特征图。通常情况下，一个输入包含的特征图不仅只有一个，而是由 C 个组成，即一个输入的规模为 $C \times W_{in} \times H_{in}$，因此我们对每一个输入通道都施加一个二维卷积核，即使用规模为 $C \times R \times S$ 的三维卷积核对输入进行卷积，最终获得一个输出特征图。当我们将采 K 组不同的卷积核作用于相同的输入特征图，将获得 K 个不同的输出特征图。考虑到多个输入的情况，N 个不同的输入，共产生 N 个不同的输出。最终，卷积层的输入规模为 $N \times C \times W_{in} \times H_{in}$，卷积核的规模为 $K \times C \times R \times S$, 输出规模为 $N \times K \times W_{out} \times W_{out}$。

```
1  for n=0 to N
2      for k=0 to K
3          for w=0 to W
4              for h=0 to H
5                  for c=0 to C
6                      for s=0 to S
7                          for r=0 to R
8                              out[n][k][w][h] += in[n][c][w*sw+r][h*sh+s] * filter [k][c][r][s]
```

代码 2.1　七层卷积循环

卷积层的计算由 N，K，W，H，C，R 和 S 这七个变量形成嵌套循环完成，而且这七个变量的所有排列都是合法的。算法 2.1展示了其中一种循环嵌套方式，我们可以用 $N->W_{out}->H_{out}->C->S->R$ 来描述这种循环，其中 sw 和 sh 表示卷积操作的步长。不同的循环方式决定了数据的复用形式和数据流的方式，最终将影响神经网络加速器的设计 [87]。

2.1.3　池化层

池化层（pooling layer）是神经网络中一个重要的层。池化层一般是在卷积层之后，对输入进行非线性降采样，常用的池化做法是对每个滤波器的输出求最大值，平均值，中位数等。池化层的意义主要体现在两个方面：第一，池化层通过对特征图像进行降维操作，能够在保留显著特征的情况下，有效减少整个神经网络所需要的参数量和计算量。第二，池化层能够保证输入的平移不变性

(translation invariant)，这意味着即使图像的像素在邻域发生微小位移时，池化层的输出能够保持不变，从而增强神经网络的鲁棒性，有一定的抗扰动能力。常用的池化层包括最大池化层，平均池化层，ROI（Regions of interest）池化层.

最大池化层的基本思想是在一个特定的数据区域内选择一个最大值作为输出。其计算公式是

$$out_{x,y} = max(in_{x*sx,y*sy} : in_{x*sx+kx,y*sy+ky}) \tag{2.2}$$

其中 kx 和 ky 是池化窗口的大小，sx 和 sy 是池化的步长。通常情况下池化窗口的大小与池化的步长相同，即池化操作的输入并不会重复。后来也出现了数据复用的池化，相邻池化窗口之间会有重叠区域，此时池化步长小于池化窗口的大小。

平均池化层的基本操作与最大池化层类似，它将一个特定的区域内的数据取算数平均作为输出，其计算公式是

$$out_{x,y} = mean(in_{x*sx,y*sy} : in_{x*sx+kx,y*sy+ky}) \tag{2.3}$$

ROI 池化层主要是针对 ROIs 的池化操作，主要应用于目标检测领域的 Fast RCNN [35] 和 Faster RCNN [26] 网络中，它的特点是输入特征图尺寸不固定，但是输出特征图的尺寸固定。ROI Pooling 的输入包含两部分，第一部分是特征图，在 Fast RCNN 中，它位于 ROI Pooling 之前；在 Faster RCNN 中，它是与 RPN 共享的那个特征图。第二部分是 ROIs，在 Fast RCNN 中，指的是 Selective Search 的输出；在 Faster RCNN 中指的是 RPN 的输出，是一系列的矩阵候选框，每一个矩阵候选框用四个坐标和索引来表示。ROI Pooling 的输出则是 batch 个三维向量（$C \times W \times H$），其中 batch 的值为 ROI 的数量。因此 ROI Pooling 的过程可以总结为将大小不同的矩阵框映射为大小固定的矩阵框。

2.1.4　归一化层

随着神经网络的规模不断变大，结构的复杂度增加，神经网络越来越难以训练，同时神经网络越来越容易出现过拟合的现象。归一化层（normalization layer）成为神经网络中不可或缺的一个层，它能够加快神经网络的收敛速度，防止过拟合，降低神经网络对初始化权重的敏感度，提高神经网络的精度。近几年出现了各种各样的归一化方法, 主要包括 LRN（Local Response Normalization）[24]，BN（Batch Normalization）[90]，LN（layer Normalization）[91]，IN（Instance Normalization）[92] 和 GN（Group Normalization）[93] 等。

LRN 是 AlexNet 等网络中使用的归一化方法，它在每一个像素的小领域范围内进行归一化处理，但是这种小邻域的归一化效果有限。

目前主流的归一化方法则更加注重全局范围内的归一化处理，这些全局归一化的方法能够使得我们在训练神经网络时使用较大的学习率，从而加快神经网络训练速度。如图 2.3所示，BN 选择在 batch 维度上进行归一化处理；LN 选择在 channel 维度进行归一化处理；IN 执行类似 BN 的计算, 但是仅仅在单个样本执行归一化；GN 在 IN 的基础上对多个样本进行归一化。下面主要对 BN 的计算方法进行简要介绍。

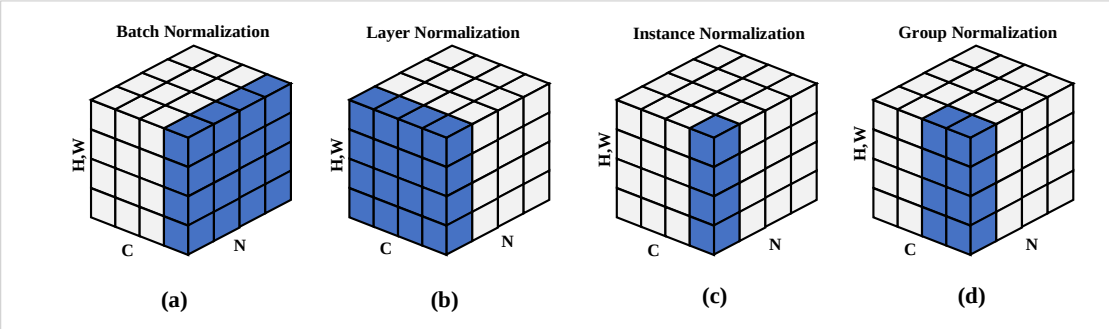


图 2.3　归一化方法。每一个子图显示的是一个 feature map，其中 N 为 batch 轴，C 为 channel 轴，（H,W）为空间轴。蓝色的像素表示归一化的范围。

BN 的计算公式如下所示：

$$\mu_\beta = \frac{1}{N}\sum_{i=1}^{N} a_i \tag{2.4}$$

$$\theta_\beta^2 = \frac{1}{N}\sum_{i=1}^{N}(a_i - \mu_\beta)^2 \tag{2.5}$$

$$\hat{a}_i = \frac{a_i - \mu_\beta}{\sqrt{\theta_\beta^2 + \epsilon}} \tag{2.6}$$

$$b_i = \gamma \hat{a}_i + \beta \tag{2.7}$$

在上述算法中，m 为 batch 数，a_i 是第 i 个 batch 的输入数据，μ_β 和 θ_β^2 分别是 N 个输入的均值和方差，参数 ϵ 是批变化常量，参数 γ 和 β 是训练时需要学习的参数，b_i 是最后归一化后的输出。

2.1.5　激活层

激活层（activation layer）是神经网络能够解决非线性问题的关键，它弥补了神经网络中线性模型表达能力不足。目前主流的激活函数包括 Sigmoid, Tanh, ReLU 以及 ReLU 的变种, 如 PReLU 和 RReLU 等。

Sigmoid，Tanh 和 ReLU 函数的公式分别为

$$Sigmoid(x) = \frac{1}{1+\mathrm{e}^{-x}} \tag{2.8}$$

$$Tanh(x) = \frac{1-\mathrm{e}^{-2x}}{1+\mathrm{e}^{-2x}} \tag{2.9}$$

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{2.10}$$

Sigmoid 函数主要被应用于早期的神经网络中，Sigmoid 函数拥有良好的性质，Sigmoid 函数的值域范围限制在 (0,1) 之间，这与概率值的范围是相对应的，这样 Sigmoid 函数就能与一个概率分布联系起来了; 同时 Sigmoid 函数的导数可以由其本身求得 $Sigmoid(x)' = Sigmoid(x)(1 - Sigmoid(x))$。但是 Sigmoid 函数也有不少缺点，首先 Sigmoid 函数的输出并不是以“0”为中心，这个特性会导致为在后续神经网络的高层处理中收到不是零中心的数据，进而导致训练过程中时的权值更新产生锯齿晃动。同时 Sigmoid 函数具有饱和性，容易出现梯度消失的现象，使得神经网络难以训练。

Tanh 函数是 Sigmoid 函数的改进版本，它的收敛速度比 Sigmoid 函数更快，相比 Sigmoid 函数，其输出以“0”为中心，但是仍然存在饱和性而导致梯度消失的问题的。Tanh 函数和 Sigmoid 函数目前主要被用于基于 LSTM 的 RNN 架构或者基于 GRU 的 RNN 架构中。

ReLU 是目前非常流行的激活函数，它是分段线性函数，所有的负值经过激活后为“0”，而正值保持不变，这种操作被称为单侧抑制。单侧抑制使得神经网络中的神经元也具有了稀疏激活性，尤其体现在深度神经网络模型 (如 CNN) 中，当模型增加 N 层之后，理论上 ReLU 神经元的激活率将降低 2^N 倍，从而更好地挖掘相关特征，拟合训练数据。同时，ReLU 不存在饱和区，因此不存在梯度消失的问题，使得模型的收敛速度维持在一个稳定的状态。实验显示 ReLU 激活单元对比 Tanh 激活单元提升了 6 倍的收敛速度提升 [24]。

2.1.6　LSTM

现代大规模自动语音识别（automatic speech recognition，ASR）系统利用基于 LSTM 的 RNN 作为其声学模型。LSTM 模型由一系列大规模矩阵组成，这是 ASR 的所有步骤中计算量最大的部分。在基于 LSTM 的 RNN 中，时刻 T 的输入取决于时刻 T-1 的输出。一个经典的 LSTM 模型如图 2.4 所示。LSTM 模型包含特殊存储单元（图 2.4中的 $cell$）和三个特殊的门（图 2.4中的 i，o，f），其中 $cell$ 用于存储网络的时间状态信息，门用于执行特殊的乘法运算，包括输入门（input gate）、输出门（output gate）和遗忘门（forget gate）。输入门 i 控制输入到存储单

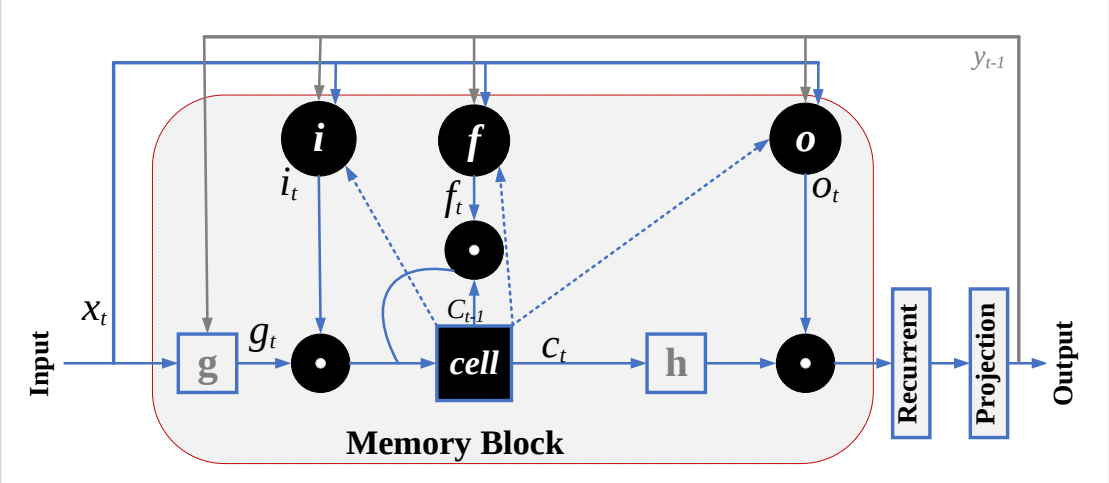


图 2.4　LSTM

元中的输入量；输出门 o 控制输出值；遗忘门 f 自适应地遗忘 $cell$ 中存储的信息，从而控制前一状态对现状态的影响。除了基本的三个众所周知的门和 $cell$ 之外，该 LSTM 模型还引入了窥视孔（peephole）和投影层（projection layer），以便更好地学习。窥视孔将缩放后的 $cell$ 状态添加到三个门，其中缩放尺度由三个对角矩阵决定。投影层线性地将输出转换为低维形式。

LSTM 模型的接收一个输入序列 $X = (x_1; x_2; x_3; ...; x_T)$（其中 x_t 是 t 时刻的输入向量）和上一个状态的输出序列 $Y^{T-1} = (y_0; y_1; y_2; ...; y_{T-1})i$（其中 y_{t-1} 是 t-1 时刻的输出向量）。利用下列公式从 t=1 到 T 计算输出序列 $Y = (y1; y2; y3; ...; y_T)$:

$$i_t = \delta(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i), \tag{2.11}$$

$$i_t = \delta(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i), \tag{2.12}$$

$$f_t = \delta(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f), \tag{2.13}$$

$$g_t = \delta(W_{cx}x_t + W_{cr}y_{t-1} + b_c), \tag{2.14}$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t, \tag{2.15}$$

$$o_t = \delta(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o), \tag{2.16}$$

$$m_t = o_t \odot h(c_t), \tag{2.17}$$

$$y_t = W_{ym}m_t \tag{2.18}$$

其中符号 i、f、o、c、m 和 y 分别是输入门、遗忘门、输出门、$cell$ 状态、$cell$ 输出和投影输出；$\odot$ 表示向量逐元素乘积，W 表示权值矩阵（例如 W_ix 是从输入向量 X_t 到输入门的权值矩阵），b 表示偏置向量。值得注意的是 W_ic、W_fc 和 W_oc 是用于 peephole 连接的对角矩阵，因此它们本质上是向量。δ 是 Sigmoid 激活函数，h 是用户自定义的激活函数，通常情况下会采用 $tanh$ 激活函数。

2.1.7　GRU

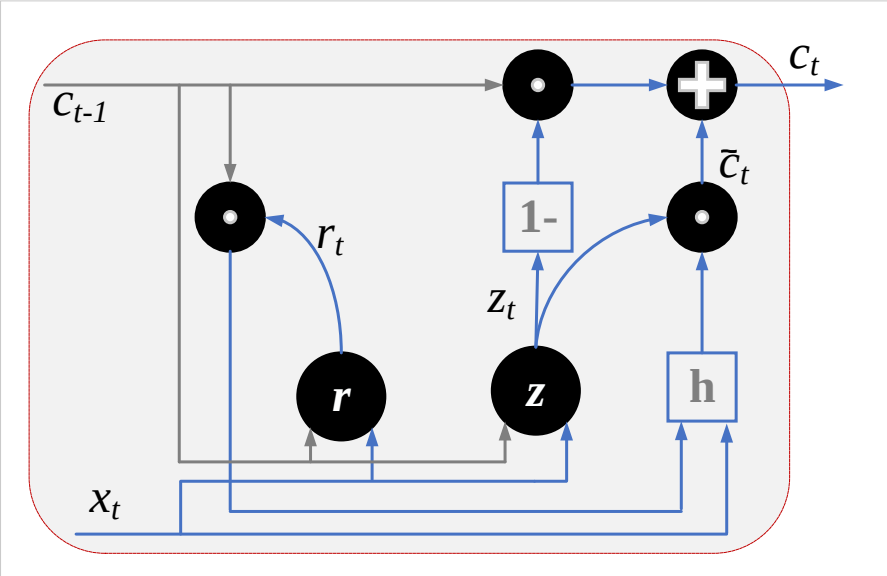


图 2.5　GRU

GRU 是 LSTM 的变体，它把遗忘门和输入门组合成一个更新门（update gate），同时合并了 *cell* 状态和隐藏状态，并进行了一些其他更改。GRU 架构如图 2.5所示。类似地，它遵循如下的公式从 t = 1 到 T 进行迭代计算：

$$z_t = \delta(W_{zx}x_t + W_{zc}c_{t-1} + b_z), \tag{2.19}$$

$$r_t = \delta(W_{rx}x_t + W_{rc}c_{t-1} + b_r), \tag{2.20}$$

$$\tilde{c}_t = h(W_{\tilde{c}x}x_t + W_{\tilde{c}c}(r_t \odot c_{t-1}) + b_{\tilde{c}}), \tag{2.21}$$

$$c_t = (1 - z_t) \odot c_{t-1} + z_t \odot \tilde{c}_t \tag{2.22}$$

其中符号 z、r、$\tilde{c}$、c 分别是更新门、复位门、复位状态和 *cell* 状态；$\odot$ 表示逐元素乘法。W 表示权值矩阵，δ 是 Sigmoid 激活函数，h 是用户定义的激活函数，这里我们使用 $tanh$ 激活函数。值得注意的是 GRU 有两个门（更新门和复位门），而 LSTM 有三个门（输入门、遗忘门、输出门），GRU 中不存在 LSTM 中存在的输出门，而是将 *cell* 状态作为输出。LSTM 中输入门和遗忘门耦合成为 GRU 中的更新门 z，复位门 r 直接作用于到上一个 *cell* 的状态。

2.2　神经网络低能耗的技术

随着神经网络算法被应用于更复杂的处理任务和更广泛的场景中，神经网络的规模也越来越大。最新的 DNNs [57, 63] 需要数百兆字节甚至千兆字节来存储神经元和权值；同时需要数十亿次的乘加操作完成运算。考虑未来神经网络将

朝着规模更大，层数更深的趋势发展，未来的大规模网络很难部署到嵌入式系统中。因此很多研究人员致力于减少神经网络的参数规模，从而减少存储资源，运算资源和带宽需求，提升性能同时降低能耗。目前已经有了一些有效的算法来解决这个问题，主要方法包括神经网络的低精度计算，神经网络裁剪技术和权值矩阵变换。

2.2.1　神经网络的低精度计算

最近研究 [40] 表明，在深度神经网络的训练中，不需要全精度的数据。使用低比特的权值表示，可以显著减少网络存储需求和运算能耗，特别是当使用极低比特位，如二进制/三元权值时。研究显示，非精确的计算（在神经元和权重中加入噪音）具有正则化的效果，从而减少泛化误差 [94]。Gupta et al. [40]使用 16 位定点数字表示来训练网络。Köster et al. [95]提出了动态数据格式（FlexPoint），用来完全替代 32 位浮点格式。Flexpoint 具有一个可动态调整的共享指数，以最小化溢出和最大化可用动态范围。在训练过程中，数据范围会随着训练轮数的增长而连续变化，其中共享指数部分可以根据历史数据进行预测。实验结果显示，采用 16 位尾数和 5 位共享指数的 Flexpoint 格式数据在训练神经网络时，能够获得 32 位浮点类似的精度，并且远远超过 16 位浮点获得的精度。Dettmers [41]使用 8 比特量化神经网络，在保持性能的情况下，加快神经网络训练速度。在 [42] 和 [96] 中，作者表明，深度神经网络可以通过二进制权值进行训练，在某些情况下，它的精度可能超过使用浮点数进行训练的结果。Rastegari et al. [43]首先提出了三元权值网络（Ternary Weight Network, TWN），并在 ImageNet [97] 数据集上取得良好的效果。三元权值网络在 [98-99] 等工作中被不断深入研究，其中 [99] 提出了学习三元值和三元赋值的方法。Zhou et al. [44]提出了 INQ（Incremental Network Quantization），在不降低网络精度的情况下，将神经网络的权值量化为 2^{-n} 的形式。Wang et al. [100]提出了定点分解网络（Fixed-point Factorized Network, FFN），使用定点分解的方式对神经网络权值进行三元化。这些方法可以在 ImageNet 上达到与全精度相当的精度，但是，这些工作只对权值进行量化，而使激活保持浮点格式。

除了对神经网络权值进行量化外，也有许多研究对激活进行量化。通过将权重和激活转换为低比特格式，网络计算需要使用定点操作，从而更有效地节约资源。Hubara et al. [101]中提出的二值神经网络（binarized neural network, BNN），在 CIFAR-10 这样的小数据集上达到了跟浮点数类似的精度。 Rastegari et al. [43]提出了 xnor-net 网络模型，该网络将权值和激活都进行了二值化，在像 ImageNet 这样的大数据集上，xnor-net 比 BNN 更精确，但是，它的精度对比浮点数的情况仍然有很大的下降。 Zhou et al. [102]提出了 doref-net，研究了量化时比特位长度

对权值、激活和梯度的影响。 Cai et al. [103]提出了 HWGQ（Half-wave Gaussian Quantization）的方法对权值和激活同时进行量化。与权值量化相比，激活量化通常会导致更大幅度的精度下降。对 AlexNet 和 VGG16 这样的大型网络进行量化，这些方法会导致精确度下降 5% – 10%。因此，如何用低比特位来量化权值和激活仍然是一个具有挑战性的问题。

2.2.2　神经网络裁剪技术

大规模的神经网络通常是过拟合的，过多的参数会严重影响神经网络的运算速度。因此，我们可以通过裁剪技术修剪不必要的神经元或者权值，从而缩小神经网络的规模，同时减少神经网络的存储需求和运算需求，缓减过拟合的现象。

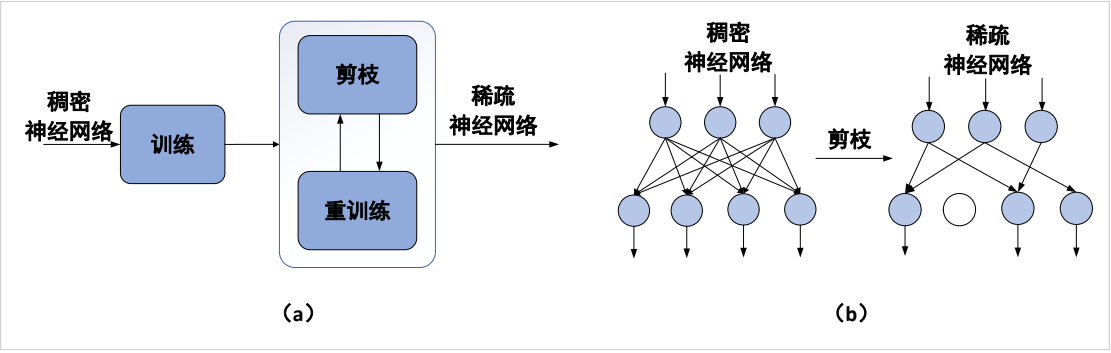


图 2.6　(a) 剪枝流程 (b) 剪枝前后神经网络

Han et al. [48]提出了剪枝的策略来减少神经网络中权值的数量，在不影响神经网络准确性的前提下，能够将神经网络所需要的存储量和计算量减少一个数量级。如图 2.6（a）所示，剪枝策略分为三个步骤：第一步，对神经网络进行训练，找出那些重要的连接；第二步，修剪不必要的连接，即当权值的绝对值小于某个阈值时，该连接将被永久从神经网络的拓扑结构中剪除；第三步，对神经网络进行重训练，微调剩余连接的权值。同时，我们需要迭代执行第二步和第三步，在不影响精度的情况下尽可能提高神经网络的稀疏度。稀疏神经网络的拓扑结构如图 2.6（b）所示，值得注意的是，当某个神经元没有与上一层神经元或者与下一层神经元相连时，该神经元将会被剪除。实验显示，剪枝策略能够将 AlexNet 网络的权值数量减少 9 倍，将 VGG16 网络的权值数量减少 13 倍。

在此基础上，Han et al. [88]进一步提出了 Deep Compression 用来深度压缩神经网络。Deep Comression 由三个阶段组成：剪枝（Pruning），量化（quantization）和霍夫曼编码（Huffman Coding），最终在不影响神经网络的精度的前提下将神经网络压缩了 35 倍到 49 倍。Deep Compression 首先通过剪枝阶段学习网络中重要的连接，删除不必要的连接，这个步骤能够减少 9 倍到 13 倍的权值。然后

通过聚类算法将权值进行聚类，然后量化权值实现权值的共享，这个步骤能够减少表示每个权值的比特数，从 32 比特减少到到 5 比特。最后采用霍夫曼编码进一步无损压缩神经网络，这个步骤能够节省 $20\% - 30\%$ 的网络存储开销。在 ImageNet 数据集上，Deep Compression 能够将 AlexNet 网络压缩 35 倍，将网络规模从 $240MB$ 压缩到 $6.9MB$。同时，Deep Compression 将 VGG16 网络压缩了 49 倍，将网络规模从 $552MB$ 压缩到 $11.3MB$。Deep Compression 将允许网络模型存储在加速器的片上 SRAM 缓存中，而不需要在片上和片外之间反复搬运权值，从而减少片外访存开销。

Wang et al. [89]提出了一种新的有效的 CNN 压缩方法 CNNpack，它在频域对神经网络进行剪枝操作，因此它不仅能够关注小的权值，同时关注所有潜在的对计算结果影响小的连接。CNNpack 借助离散余弦变换（Discrete Cosine Transform，DCT）将空间域的权值变换到频域，然后采用聚类的方法将频域中的权重分解为公共部分和私有部分（残差）。在这两部分中，采用剪枝的策略丢弃大量的低能级的频率系数，能够在不显著降低精度的情况下产生高压缩比。在此基础上，CNNpack 接着采用量化，霍夫曼编码和 CSR 存储的方法进一步压缩神经网络。最终实验实验显示，CNNpack 能够对 AlexNet 和 VGG16 分别压缩 35 倍和 49 倍。

不同于 [48, 88-89] 中的静态裁剪的技术，即符合条件的权值将被用永久从拓扑连接中剪除，Guo et al. [49]提出了一种动态裁剪神经网络的算法（dynamic network surgery），这种算法包含两个不同的操作，即剪枝（pruning）和粘接（splicing）。Splicing 能够在训练稀疏神经网络过程中重新连接被剪除的突触，从而实现动态裁剪。实验结果显示，Dynamic Network Surgery 能够将 LeNet-5 和 AlexNet 的突触数量分别减少 108 倍和 17.7 倍。

除了以上对裁剪权值的策略，还有不少研究工作直接对神经元进行裁剪操作 [104-107]。但是对神经元直接进行剪枝操作会严重降低神经网络的精度，因此不能获得理想的稀疏度。Data-free Parameter Pruning [105] 在 AlexNet 和 LeNet-5 上仅仅能够获得 34.89% 和 83.5% 的稀疏度，远远低于权值剪枝策略 [48] 的 89% 和 92%。Network Trimming [106] 能够在 VGG16 和 LeNet-5 上获得 63% 和 74% 的稀疏度，也低于 [48] 中的 92.5% 和 92%。

2.2.3　权值矩阵变换

除了经典的剪枝策略（包括权值剪枝和神经元剪枝），还有一部分的研究工作基于矩阵分解和因式分解 [50-51, 108]，这些方法可以保持原始模型的规则密集计算结构，在通用处理器上实现神经网络压缩和加速。

Denton et al. [50]利用神经网路线性结构的特性，对神经网络进行适当的低秩分解（low-rank approximation），在精度损失在 1% 的情况下，在很多模型上都

能获得超过 2 倍的加速。 Jaderberg et al. [108]利用张量分解将规模为 $k \times k$ 的卷积核分解为 $k \times 1$ 和 $1 \times k$ 的卷积核，从而减少卷积操作的计算量，加速卷积计算过程。同时 [108] 提出了两种优化方案：一种是利用滤波重构的方法最小化滤波权值误差，另一种是利用数据重构最小化权值误差。 Lebedev et al. [51]则采用 CP decomposition 的方式对大规模神经网络的卷积核进行分解和加速，实验结果显示，在精度损失低于 1% 的情况下，在 AlexNet 网络能够获得 4 倍的加速效果。

2.3　神经网络加速器

2.3.1　现有神经网络加速器架构

由于严峻的能耗约束和高性能要求，定制加速器成为 CPU 和 GPU 等传统处理平台的替代品。近几年出现了数据流和结构各异的神经网络加速器。神经网络加速器按照数据流的形式可以分为两类，分别是基于向量算子（vector operator）数据流的加速器和和基于乘加算子（multiply-and-accumulate，MAC）空间数据流的加速器。

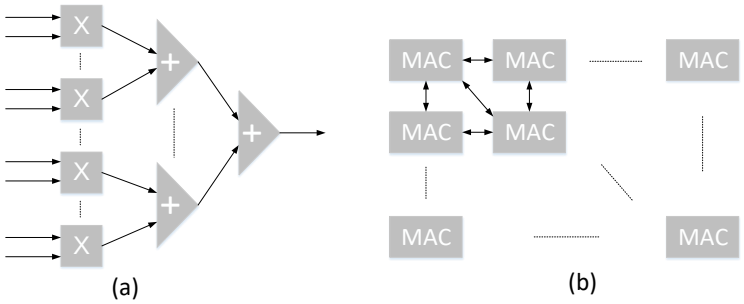


图 2.7　现有的加速器架构.

基于向量算子的加速器将神经网络的运算转化为向量运算，主要是将矩阵操作转化为一系列的向量操作（通常是内积操作）。如图 2.7 (a) 所示的结构中，n 个乘法器和一个 n 输入的加法树就能够完成两个 n 维向量的内积操作。

基于乘加算子空间数据流加速器（如图 2.7（b））通常具有一系列规则分布的处理单元，其中每一个处理单元能够完成一次 MAC 操作。处理单元之间采用一种规律的方式进行连接（如二维矩阵），神经元和权值采用某种规律的方式在处理单元之间进行移动，如经典的脉动阵列形式（systolic design），因此这种形式又称为空间数据流形式。

脉动阵列，本身的核心概念就是让数据在运算单元的阵列中进行流动，减少访存的次数，并且使得结构更加规整，布线更加统一，提高频率。图 2.8描述了是一个规模为 3×3 的脉动阵列完成两个规模为 3×3 的矩阵乘法的过程。阵列

中的每一个处理单元（processing element, PE）由一个乘加器（MAC unit）和寄存器组成，其中乘加器执行两个数的乘法操作，将相乘结果与寄存器中存储的数进行累加，并将计算结果写回寄存器中。值得注意的是，在使用脉动矩阵进行矩阵计算的时候需要对数据调整好形式，按照一定顺序，分时进入脉动阵列。如图 2.8所示，PE 之间存在规则的数据通路，其中 $PE_{i,j}$ 能够从 $PE_{i-1,j}$（即上方邻居）获得矩阵 B 的第 j 列的输入，并在下一个 cycle 将该输入发送给 $PE_{i+1,j}$（即下方邻居）；同时 $PE_{i,j}$ 能够从 $PE_{i,j-1}$（即左方邻居）获得 A 矩阵的第 i 行的输入，并在下一个 cycle 将该输入发送给 $PE_{i,j+1}$（即右方邻居）。

在第一个 cycle，a_{11} 和 b_{11} 分别从左方和上方流入 $PE_{1,1}$，同时 $PE_{1,1}$ 完成 $a_{11} * b_{11}$ 的操作。第二个 cycle，a_{11} 和 b_{11} 分别沿着水平向右和竖直向下的方向流入 $PE_{1,2}$ 和 $PE_{2,1}$，同时 a_{12}，a_{21}，b_{21} 和 b_{12} 分别流入 $PE_{1,1}$，$PE_{2,1}$，$PE_{1,1}$ 和 $PE_{1,2}$；然后 $PE_{1,1}$ 中完成 $a_{12} * b_{21}$，同时将结果与 $a_{11} * b_{11}$ 累加，即 $PE_{i,1}$ 中寄存器的结果为 $a_{11} * b_{11} + a_{12} * b_{21}$，同时 $PE_{1,2}$ 和 $PE_{2,1}$ 寄存器中的结果分别为 $a_{11} * b_{12}$ 和 $a_{21} * b_{11}$。随着两个矩阵的输入不断流入脉动阵列中，最终两个规模为 3×3 的矩阵乘法能够在个七周期内完成。

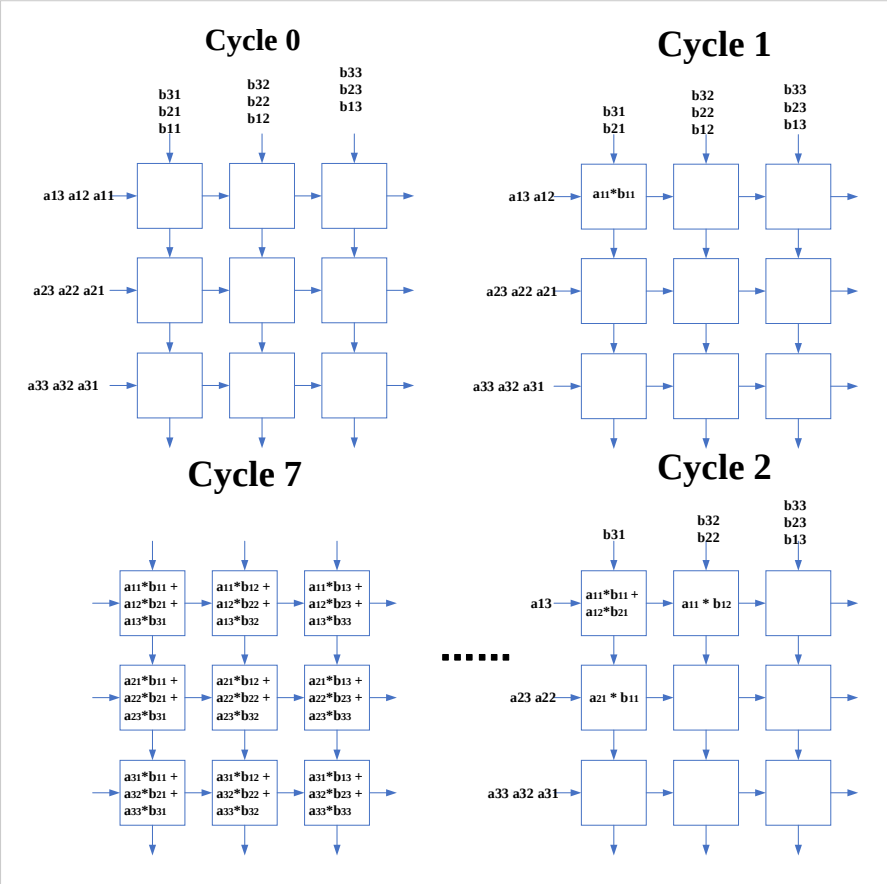


图 2.8　脉动阵列完成矩阵乘法操作

表 2.1总结了近年来出现的神经网络加速器和它们的数据流形式。基于向

量算子的神经网络加速器包括 DianNao [77], DaDianNao [109], PuDianNao [78], Cambricon [110], Cambricon-X [82], Cnvlutin [84], EIE [85] 和 ESE [86]。基于乘加算子空间数据流的加速器包括 ShiDianNao [79], Eyeriss [83], TPU [81], Neuflow [64] 和 SCNN [87]。下面我们将对两类神经网络加速器分别进行介绍。

表 2.1　现有神经网络加速器的数据流形式

数据流	加速器
基于向量算子的数据流	DianNao [77], DaDianNao [109], PuDianNao [78], Cambricon [110], Cambricon-X [82], Cnvlutin [84], EIE [85], ESE [86]
基于乘加算子的空间数据流	ShiDianNao [79], Eyeriss [83], TPU [81], Neuflow [64] SCNN [87]

2.3.2　基于向量算子的神经网络处理器

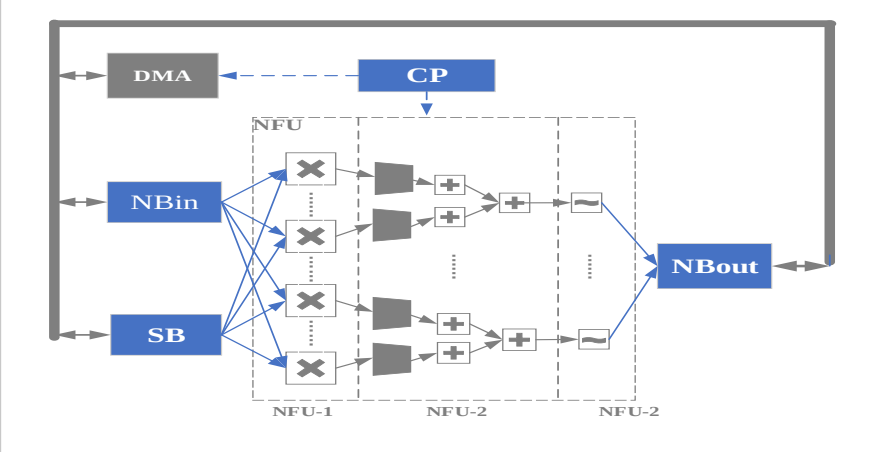


图 2.9　DianNao 加速器结构

DianNao 是世界上首个深度神经网络处理器，它的结构如图 2.9所示。DianNao 包含以下的主要模块：一个控制器（Control Processor, CP），一个神经功能单元（Neural Functional Unit, NFU），三个片上缓存（NBin，NBout 和 SB）和 DMA。CP 采用自定义的指令控制 DianNao 的运行。NBin，NBout 和 SB 分别用来缓存输入神经元，输出神经元和权值。NFU 包含 T_n 个计算单元，每个计算单元包含 T_n 个乘法器和一个 T_n 输入的加法树。在计算过程中每个计算单元共享输入神经元，接收不同的权值，从而计算不同的输出神经元。因此 NFU 最多读取 T_n 个输入神经元，$T_n \times T_n$ 个权值，计算 T_n 个输出神经元。在 TSMC $65nm$ 工艺下，采用 $T_n = 16$ 的配置，DianNao 的面积，功耗和吞吐率分别为 $3.02mm^2$，$485mW$ 和 $452GOP/s$，对比与一个 $2GHz$ 的 CPU，能够获得 117 倍的性能提升，并且降低 21 倍的能耗。

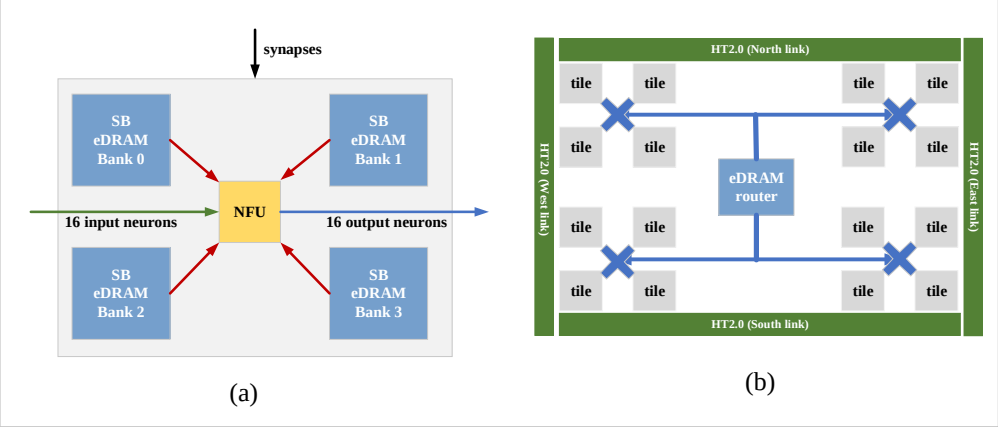


图 2.10　（a）DaDianNao 中一个 tile 的结构（b）DaDianNao 中一个 node 结构

DaDianNao 是 DianNao 的多核版本，如图 2.10（a）所示，DaDianNao 中的一个 tile 相当于一个 DianNao 处理器，它由 NFU 与外围的存储模块构成。进一步的，如图 2.10(b) 所示，多个 tile 通过 fat-tree 进行局部互联组成一个 node，最终多个 node 通过 HyperTransport2.0 互联形成 DaDianNao。DaDianNao 使用 eDRAM 来提供足够的片上缓存用来存储权值和神经元，从而减少片外访存开销，进而有效地处理大规模神经网络。实验显示，在 ST28nm 的工艺下，64 核的 DaDianNao 的面积和功耗分别是 67.73mm^2 和 15.97W，对比 Nvidia K20，平均性能提升 21.38 倍，平均能耗降低了 330.56 倍。

PuDianNao 是一个机器学习处理器，它不仅支持神经网络算法，而且支持 K 最近邻算法（K-Nearest Neighbors，K-NN），K 均值算法（K-Means），线性回归（Liner Regression, LR），支持向量机（Support Vector Machine，SVM），朴素贝叶斯（Naive Bayes，NB）和决策树（Classification Tree，CT）等多种机器学习算法。PuDianNao 深入分析上述机器学习算法，并且提取出机器学习中的核心运算，其中包括向量内积（LR、SVM 和 DNN), 距离计算（K-NN 和 K-Means），计数（CT 和 NB），排序（K-NN 和 K-Means），非线性函数（如 sigmoid 和 tanh）等。PuDianNao 的核心模块是机器学习单元（Machine Learning Unit，MLU），它包括六个流水级，分别为 Counter，Adder，Multiplier，Adder Tree，Acc 和 Misc，通过这六个流水级的组合，PuDianNao 能够完成机器学习的核心运算。实验显示，在 65nm 的工艺下，PuDianNao 的面积和功耗分别为 3.51mm^2 和 596mW，对比 NVIDIA K20 能够提高 1.2 倍的性能并且减少 129.41 倍的能耗。

然而随着神经网络算法的飞速发展，一些新的层类型和新的操作不断涌现，如 ROI Pooling 层，Deconvolution 层等，这又迫使研究者开发新的加速器和指令集来支持神经网络新的特性。为了进一步提高神经网络加速器的通用性， Liu et al. [110]等人提出了针对神经网络处理器的指令集 Cambricon。Cambricon 的主要思想是为神经网络加速器提供一系列的基本算子（即指令），然后通过这些基

本算子逐步搭建完成神经网络的运算。Cambricon 中集成了控制，数据传输，运算和逻辑操作这四种不同的指令，其中运算指令进一步分为矩阵运算，向量运算和标量运算，研究者只需要通过基本指令组合就能完成神经网络的运算。基于 Cambricon 指令集设计的加速器能够比 DaDianNao 拥有更强的通用性，在 10 个 benchmark 中，DaDianNao 只能支持其中的三种，而基于 Cambricon 的加速器能够全部支持。同时对比与 Intel Xeon E5-2620 和 NVIDIA K40，加速器能够分别获得 91.72 倍和 3.09 倍的性能提升。

2.3.3　基于乘加算子空间数据流的神经网络处理器

ShiDianNao 是面向嵌入式设备的神经网络处理器，实现端到端的神经网络应用。ShiDianNao 与 DianNao 最大的不同是 NFU 模块，ShiDianNao 的 NFU 是一个大小为 $P_x \times P_y$ 的二维处理单元阵列（Processing Elements，PEs），它采用脉动阵列的形式完成神经网络矩阵向量运算操作。每个 PE 在每个周期能够完成卷积层、全连接层的一次乘法和一次加法，或者平均池化层的一次加法或者最大池化层的比较操作。每个 PE 能够从右方邻居和下方邻居读取神经元，并且能将部分和传播给相邻的 PE，值得注意的是权值通过广播的形式传送给 PE。这种方法能够充分复用神经元和权值，从而提高性能并降低访存能耗。在 $P_x = P_y = 8$ 的配置和 $65nm$ 工艺下，ShiDianNao 的主频，面积和功耗分别为 $1GHz$，$4.86mm^2$ 和 $320.1mW$，对比 GPU 和 DianNao 能够分别获得 30 倍和 1.87 倍的性能提升，减少 4700 倍和 60 倍的能耗。

除了 ShiDianNao， Farabet et al. [64]提出的 Neuflow， Chen et al. [80]等人提出的 Eyeriss 和谷歌 [81] 的 TPU（tensor processing unit）均采用二维网格的形式排列处理单元，并且采用脉动阵列的形式完成神经网络运算。其中 Google 的 TPU 主要用来加速其第二代人工智能系统 TensorFolow 的运行，并且相比于 K80 能够有 15 倍的性能提升。

2.3.4　稀疏神经网络处理器

虽然上述加速器能够以低能耗实现高吞吐量，但它们不能利用现代稀疏神经网络的稀疏性。最近出现了一些能够支持稀疏的神经网络加速器 [82-87] ，但它们都有各自的优缺点，如表 1.2所示。

Eyeriss [83] 应用游程压缩方法（run-length-compression,RLC）对稀疏神经元进行编码，具体来说，它在架构中加入 RLC Encoder 对输出神经元进行压缩，同时采用 RLC Decoder 对输入神经元进行解压缩，从而减少访问 DRAM 的数据量，进而减少 DRAM 的带宽需求和访存能耗。同时 Eyeriss 在 PE 单元中加入控制门，

当输入神经元为 0 时，PE 单元将被关闭，这样能够跳过不必要的计算，从而减少计算能耗。然而，这两种方法仅仅能够带来能耗的减少，不会带来性能增益。

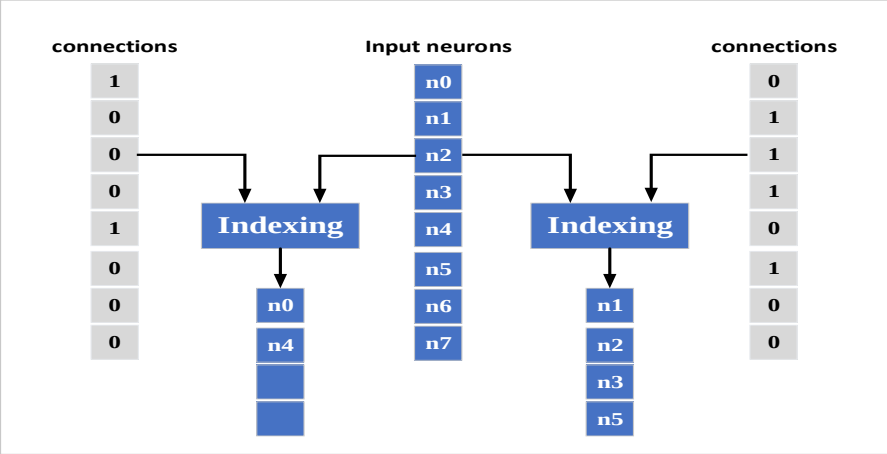


图 2.11　Cambricon-X 中的 IM 模块

Cambricon-X [82] 能够充分挖掘稀疏权值带来的收益，减少能耗和提升性能。Cambricon-X 最主要的特征是索引模块（indexing module，IM），IM 能够通过稀疏的权值索引信息过滤掉不需要参与计算的神经元，从而充分挖掘和利用稀疏神经网络的特性。如图 2.11所示，当权值索引为“10001000”时，IM 会筛选出 $n0$ 和 $n3$ 这两个神经元进行后续计算；当权值索引为“01110100”时，IM 会筛选出 $n1, n2, n3, n5$ 这四个神经元进行后续计算。同时，Cambricon-X 使用步长索引的形式压缩稀疏权值，从而减少权值存储量，减少片外和片上访存能耗开销。Cambricon-X 对比不支持稀疏特性的 DianNao 提高了 7.23 倍的性能，并减少了 6.43 倍的能耗，但是 Cambricon-X 并不能挖掘稀疏神经元带来的收益。

Cnvlutin [84] 能够利用动态神经元稀疏筛选出需要进行计算的权值，获得 1.37 倍的性能提升，但是 Cnvlutin 不能利用权值稀疏特性。

ESE [86] 是实现在 FPGA 上的加速器，它面向的是稀疏的 LSTM 模型，并不适用于稀疏的 CNN 模型；由于 LSTM 模型中使用 Tanh 作为激活函数，因此不存在神经元稀疏的特性，所以 ESE 仅仅能够挖掘权值稀疏的特性。

以上工作只能挖掘权值稀疏性或者神经元稀疏性，但不能同时从两者中受益。EIE [85] 采用稀疏矩阵的行压缩形式（compressed sparse row，简称 CSR）存储稀疏的权值，并且通过 LZND（leading non-zero detection）筛选出非零的神经元，使得使得 EIE 能够同时利用神经元稀疏性和权值稀疏性，对比与 DaDianNao 能够提高 2.9 倍的性能，减少 19 倍的能耗并且缩小 3 倍的面积。但是这种架构仅仅针对全连接层的计算，并不能针对卷积层进行计算。

SCNN [87] 能够同时利用神经元稀疏性和权值稀疏性。SCNN 中的计算单元构成了一个二维网格的形式，每个计算单元执行非零神经元和非零权值的乘法

操作，同时计算非零乘积的坐标。非零乘积通过坐标在二维计算网络进行路由，最终被分配给对应的累加器阵列，每个非零乘积通过读取修改写入操作对包含部分和的本地 RAM 执行累加，最终获得输出神经元。但是这种架构需要不断计算坐标，大大增加了计算成本和存储成本。因此 SCNN 在处理稠密神经网络时的会损失 21% 的性能，同时增加 33% 的能耗；在处理稀疏神经网络时仅仅能够增加 2.7 倍的性能，减少 2.3 倍的能耗。同时 SCNN 对全连接层和规模为 1x1 的卷积层支持并不理想，在这两种类型的层时时只能利用 20% 的乘法器资源。

第 3 章　一种新的神经网络压缩方法

3.1　背景

现代神经网络获得迅猛发展，目前深度神经网络已经在图像识别，目标检测，语音识别，自然语言处理，视频处理领域有了越来越广泛的应用。随着神经网络算法处理问题的复杂度不断上升，神经网络的规模不断扩大，主要体现在两个方面：第一个方面，神经网络的参数（包括神经元和权值）不断增多，例如，AlexNet 的网络规模超过 $200MB$，VGG16 的网络规模超过 $500MB$；第二方面，神经网络的层数不断加深，例如 ResNet 网络的层数可以到达上百层甚至上千层。大规模神经网络对存储，带宽和计算提出了很高的需求，从而使得这些大规模的深度神经网络难以在嵌入式设备，特别是移动设备上进行部署。

第一个问题是存储容量的问题。由于嵌入式设备或者移动设备的存储容量有限，尽管在手机端本地运行深度神经网络能够减少网络带宽，进行任务实时处理，获得更好的隐私保护，但是这些大规模的神经网络需要耗费巨大的存储开销，因此阻碍了深度神经网络应用被部署到移动应用程序。

第二个问题是能耗问题。大型神经网络需要大量的带宽加载神经网络数据（包括拓扑结构，输入，神经元，权值等），同时需要大量的计算资源完成神经网络的运算，这个过程需要耗费巨大的能量。考虑到移动设备电池容量的限制，高能耗的应用，如深层神经网络应用将难以部署。能源消耗主要来源于访存能耗。在 $45nm$ 工艺下，一个 32 位浮点加法需要消耗 $0.9pJ$ 的能量，一次 32 位 SRAM 缓存访问耗费 $5pJ$ 的能量，而一次 32 位 DRAM 内存访问需要 $640pJ$ 的能量，这个能耗几乎是浮点加法操作的 3 个数量级。大型神经网络无法缓存在片上的 SRAM 中，因此需要需要在片外的 DRAM 与片上的 SRAM 之间频繁进行数据交换，从而产生巨大的访存能耗开销。例如我们以 $20fps$ 运行一个拥有 10 亿个连接的神经网络，那么需要为 DRAM 访存提供至少 $P = (20Hz) \times (640pJ) \times (1G) = 12.8W$ 的功率，这个已经完全超过了大多数手机电池能够提供的功率。

3.1.1　稀疏神经网络

虽然现代神经网络在许多应用中占主导地位，但是庞大的参数给计算，内存容量，内存访问和带宽带来了沉重的负担。为了应对这些挑战，研究者从各个方向做出了各种各样的努力，包括算法级 (例如 Drop out [111]、Sparsity [48])，架构级 (例如短位宽运算 [112] 甚至是 1 比特运算 [43]，非精确计算 [113]) 和物理级 (例如，动态电压调整技术 [114])。其中，稀疏技术是目前解决这个问题最有

效的方法之一。

神经网络通常是过拟合的，研究人员在早期工作中已经证明了稀疏性是解决过拟合问题的有效方法。对于现代神经网络，研究者们提出了一系列的训练技术，如稀疏编码 (Sparse Encoder) [115]，自动编解器 (Auto Encoder/Decoder) [116-117]，深度信念网络 (Deep Belief Network, DBN) [118] 等，在不影响精度的前提下裁剪冗余的突触和神经元。最近，研究人员提出了一种新的剪枝方法 [48]，当神经网络中的突触权值的绝对值小于某个阈值时，该突触将从神经网络的拓扑结构中裁剪，这种剪枝策略在 CNNs 上获得了非常理想的稀疏性，在 AlexNet [24] 网络上获得了 88.85% 的稀疏度，在 VGG16 [25] 网络上获得了 92.39% 的稀疏度。在表 3.4，我们报告各个网络详细的剪枝结果，这里，我们定义稀疏度 (sparsity) 为被修剪的神经元/突触占总神经元/突触的比例，同时我们定义稠密度（density）为非零神经元/突触占总神经元/突触的比例。

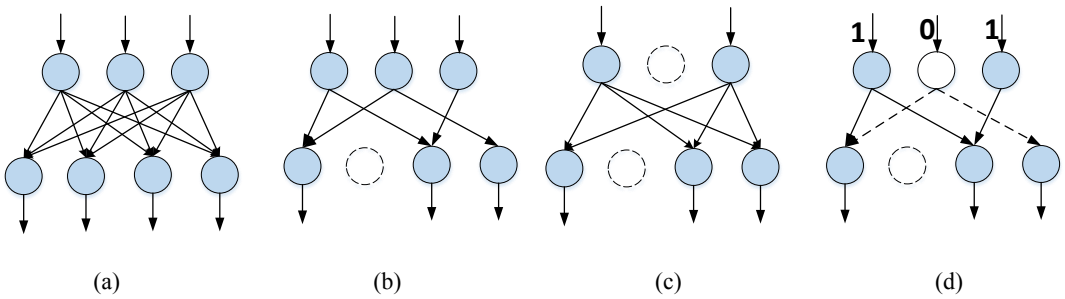


图 3.1　(a) 稠密神经网络 (b) 静态权值稀疏 (c) 静态神经元稀疏 (d) 动态神经元稀疏

如图 3.1所示，我们将神经网络中的稀疏分为两类: 静态稀疏（*static sparsity*）和动态稀疏（*dynamic sparsity*）。

静态稀疏源于裁剪操作，当突触（图 3.1 (b)）或者神经元（图 3.1 (c)）满足某些条件时，它们将从神经网络的拓扑结构中被永久剪除。那么在预测（inference）过程中，网络的拓扑结构并不会随着输入的变化而发生改变，即稀疏神经元或者稀疏权值的位置不随着输入而发生改变，因此这种稀疏又被称为静态稀疏。

当神经元的激活值为 0 时（图 3.1 (d)）会发生动态稀疏，特别是当使用 ReLU 作为激活函数时，在 ReLU 激活之前结果为负的神经元输出值为 0。形式上，ReLU 用如下公式计算输出神经元

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{3.1}$$

其中 x 为 ReLU 激活函数的输入，y 为激活的输出。因此，由于 ReLU 单侧抑制的特性，神经网络的神经中会引入许多 “0”。不同于 静态稀疏，动态稀疏不改变网络拓扑结构，它与输入密切相关，因为不同的输入将导致不同的计算结果，从而导致不同的动态稀疏结果。

值得注意的是，静态稀疏又可以分为静态权值稀疏（*static synapse sparsity, SSS*）和 静态神经元稀疏（*static neuron sparstiy, SNS*），动态稀疏仅由动态神经元稀疏（*dynamic neuron sparsity, DNS*）组成。我们将 静态神经元稀疏（*static neuron sparstiy, SNS*）和动态神经元稀疏（*dynamic neuron sparsity, DNS*）统称为 神经元稀疏（*neuron sparsity*）。

3.1.2　权值编码

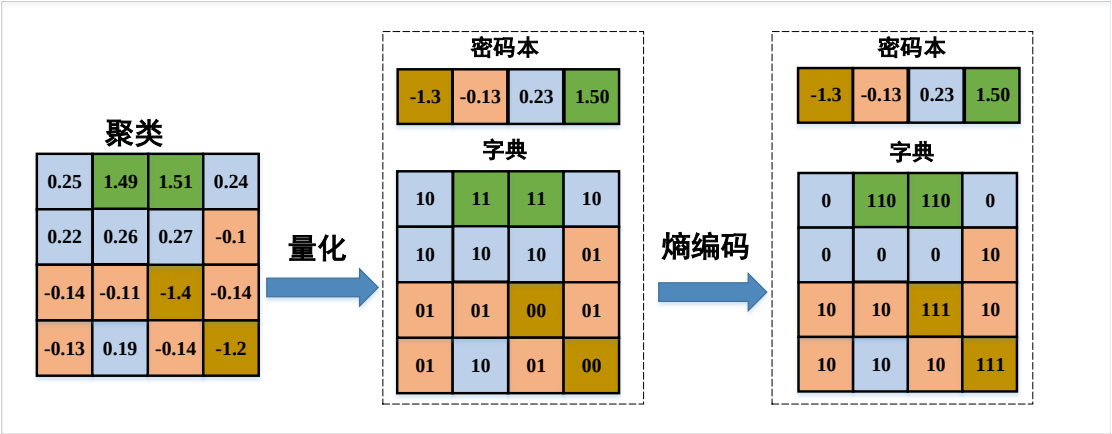


图 3.2　权值编码过程

为了进一步压缩权值数据，研究人员将图像压缩技术，如量化 (quantization) [119]，熵编码 (entropy coding) [120] 等应用到神经网络的领域。Deep Compression [88] 运用细粒度剪枝，全局量化和霍夫曼编码对神经网络进行深度压缩，在基本不损失精度的情况下，在 AlexNet 网络上获得 35 倍的压缩比，在 VGG16 网络上获得了 49 倍的压缩比。另一种有效的压缩方案是 CNNpack [89]，它利用离散余弦变换将空间域的权值变换到频域，然后在频域中裁剪掉低频的信号，然后利用量化和熵编码对神经网络进行深度压缩，最终在 AlexNet 网络上获得了 40 倍的压缩比，在 VGG16 网络上获得了 46 倍的压缩比。我们在图 3.2中描述了权值编码的过程，它由量化和熵编码两个步骤组成。

1. 量化

量化是将连续取值或者大量可能的离散取值近似为有限多个离散值的过程。量化能够减少表示神经网络参数的比特数，当权值的比特数为 t 位时，权值的可能取值为 2^t 个。

在量化过程中，我们利用聚类算法 (例如，K-means 聚类算法) 将分散的权值聚集成 K 个簇，其中值相近的权重将会被聚成一个簇。在图中，16 个权值被分为 4 个簇，其中 (0.25, 0.24, 0.22, 0.26, 0.27, 0.19) 这六个权值被聚为一个簇，(-0.1,

-0.14, -0.11, -0.14, -0.13, -0.14）这六个权值被聚为一个簇，（1.49, 1.51）这两个权值被聚为一个簇，（-1.4, -1.2）这两个权值被聚为一个簇。每一个簇对应一个中心值，该值与簇中所有权重的总距离最小，同一个簇中的所有权值都用它所在簇的中心值来近似表示，例如在图中，(0.25, 0.24, 0.22, 0.26, 0.27, 0.19) 这六个权值聚成的簇的中心点为 0.23，因此我们用 0.23 近似表示这六个权值；同理，我们用-0.13 近似表示（-0.1, -0.14, -0.11, -0.14, -0.13, -0.14）这六个权值；用 1.50 近似表示（1.49, 1.51）这两个权值；用-1.3 近似表示（-1.4, -1.2）这两个权值。因此，我们可以用一个密码本和一个字典来表示所有权值，其中密码本中记录了 K 个簇中的中心值以及每个中心值的索引，例如图例中的密码本（(00, -1.3），（01, 0.13），（10, 0.23），（11, 1.50)）。字典中每个元素只需要 $log(K)$ 位的索引来表示，我们可以通过索引密码本从而获得权值。

由于量化后可能会导致神经网络的精度下降，因此我们需要重新训练密码本。在训练过程中，我们需要对梯度进行约束，即同一个簇中的权值必须获得相同的梯度，以保证最终训练后每个簇中的权值都相同。

2. 熵编码

熵编码即编码过程中按熵原理不丢失任何信息的编码，因此它是一种无损压缩的编码技术。熵解码是熵编码的逆过程，能够完整地解码出原始码流中的信息，常见的熵编码有：香农编码 (Shannon coding)，哈夫曼编码 (Huffman coding) [121] 和算术编码 (arithmetic coding) [122]。信息熵为信源的平均信息量（不确定性的度量），具体来说，符号 α_i 所表示的信息熵为

$$I(\alpha_i) = -\log p(\alpha_i) \tag{3.2}$$

信息流 Y 的包含的信息熵 $H(Y)$ 为

$$H(Y) = -sum_{i=0}^{m-1} p(\alpha_i) \log p(\alpha_i) \tag{3.3}$$

经过量化后，由于密码本中元素的出现概率是不平衡的，因此我们可以采用熵编码（如哈弗曼编码）进一步降低表示权重的比特数。哈弗曼编码方法完全依赖于码元出现的概率来构造整体平均长度最短的编码方法。进行哈夫曼编码的关键步骤是建立符合哈夫曼编码规则的二叉树，该树又称作哈夫曼树。哈夫曼树是一种特殊的二叉树，其终端节点的个数与待编码的码元的个数等同，而且每个终端节点上都带有各自的权值。每个终端节点的路径长度乘以该节点的权值的总和称为整个二叉树的加权路径长度。在满足条件的各种二叉树中，该路径长度最短的二叉树即为哈夫曼树。在使用哈夫曼编码执行对码元的实际编码过程时，通过递归地合并概率最小的码元来构建哈夫曼树。

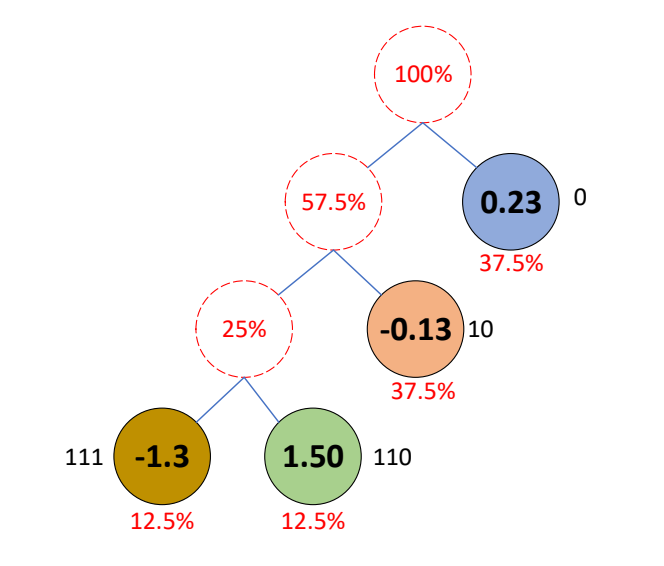


图 3.3　哈弗曼树

以图 3.2为例，四个码元-1.3，-0.13，0.23，1.50 出现的概率分别是 0.125，0.375，0.375，0.125，最终可以构造出如图 3.3所示的哈弗曼树。在哈夫曼树构建完成后，便可以得到每一个码元的哈夫曼编码的码字。具体方法是：从哈夫曼树的根节点开始遍历，直至每一个终端节点，当访问某个节点的左子树时赋予码字 1，访问右子树时赋予一个码字 0（反之亦然），直到遍历到终端节点时这一路径所代表的 0 和 1 的串便是该码元的哈夫曼编码码字。例如，在图中，最终四个码元-1.3, -0.13, 0.23, 1.50 分别被编码成为是 111，10，0，110。

3.1.3　不规则性

尽管稀疏神经网络能够在理论上减少计算量，存储量和数据传输量，但是稀疏会导致原本规则的神经网络拓扑结构转变为不规则的形式，从而使得规则计算模式转变得不规则。

然而，目前的处理平台由于存在诸多问题，无法有效地处理稀疏性。根据 [82]，CPU 和 GPU 并不能很好地支持稀疏神经网络运算，即使使用稀疏矩阵运算库，如 sparseBLAS 或者 cuBLAS，也不能取得非常理想的加速效果，在某些情况下处理稀疏网络的性能甚至比处理稠密神经网络的性能还要差。例如，CPU 平台上，除了 LeNet-5 网络，稀疏的 AlexNet 网络和 VGG16 网络的性能都比稠密网络差，性能平均降低了 2.11 倍。在 GPU 平台上, 运行稀疏神经网络仅仅能提高 23.34% 的性能，对比于稀疏神经网络高达 90.84% 的稀疏度，这个性能提升是非常有限的。因此通用的 CPU 和 GPU 并不能利用稀疏的特性。

尽管目前有不少支持稀疏神经网络的加速器 [82, 84-87]，但是它们仍然不能有效地处理稀疏神经网络带来的不规则性。他们需要显著的稀疏成本，而且不

能完全支持所有稀疏类型，例如 Cambricon-X [82] 中支持稀疏的逻辑占总面积的 31.07%，占功耗的 34.83%，而且只能支持权值稀疏。Cnvlutin [84] 只能利用神经元稀疏。EIE [85] 虽然能够同时利用权值和神经元稀疏，但是它只能适用于稀疏的全连接层，不能运行稀疏卷积层。SCNN [87] 能够在卷积层同时利用神经元和权值稀疏，但是对全连接层的支持并不理想，同时会引入复杂的坐标计算过程。因此，我们需要一种新的神经网络稀疏方法，能够在保持精度和稀疏度条件下尽可能保持神经网络的规则性，减少稀疏神经网络的不规则性，有利于 CPU，GPU 或者加速器能够充分利用稀疏带来的收益。

3.2　局部收敛

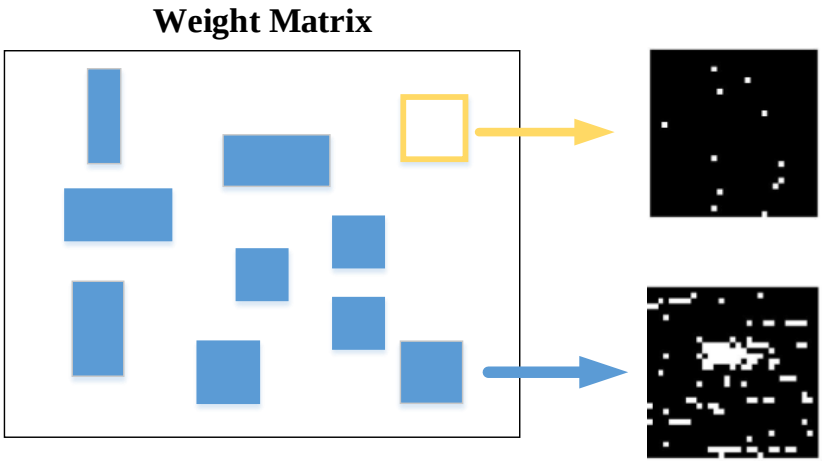


图 3.4　全连接层中的局部收敛现象（白点表示大权值，绝对值大于其余 90% 的权值）

传统的剪枝方法将单个突触当作独立不相关的元素，当突触符合特定的剪枝条件时（如权值的绝对值小于某一个阈值），该突触将会被剪除，但是这种方式忽略了突触之间的潜在关系。我们充分分析了神经网络中的权重分布情况，观察到了一个有趣的现象，即局部收敛（local convergence）。如图 3.4所示，我们使用权值矩阵来表示全连接层的权值，每一行包含与对应输入神经元相连接的所有权值，每一列包含与对应输出神经元相连接的所有权值。我们发现，在训练过程中，权值的分布并不是分布，绝对值大的权值往往会聚集成簇，我们将这种现象称为局部收敛。

我们利用如下的方式证明局部收敛的存在。首先，我们在权值矩阵上设定一个大小为 k 的滑动窗口，然后将该滑动窗口沿着权值矩阵空间维度进行滑动（全连接层权值矩阵为两个维度，卷积层权值矩阵为四个维度），同时统计窗口中较大权值的数量，如果某一个权值的绝对值大于剩余 $m\%$ 的权值，我们将其定义为较大的权值。为了方便阐述，我们将一个具有 x 个大权值的窗口标记为 x 窗

口。我们选择了 5 个代表性的层: AlexNet 网络的 *fc6* 层, VGG16 网络的 *fc6* 层, MLP 网络的 *ip1* 层, LSTM 网络的 W_{ix} 层, AlexNet 网络的 *conv2* 作为驱动示例。我们将前四层网络的滑动窗口大小设为 $k = 4$, *conv2* 层网络的滑动窗口大小设为 $k = 2$，此外，我们将 $m = 90$ 设置为较大权值的阈值。在图 3.5中，我们描绘了一个随机初始化层与训练后的 5 个层中较大权值的累计分布。我们观察到，在初始化层中，同一个窗口中较大权值的数量不会超过四个。但是，这五个训练过的层存在同一个窗口中较大权值的数量超过七个的情况。因此，较大权值在训练过程中趋向于聚集在一起，也就是局部收敛。

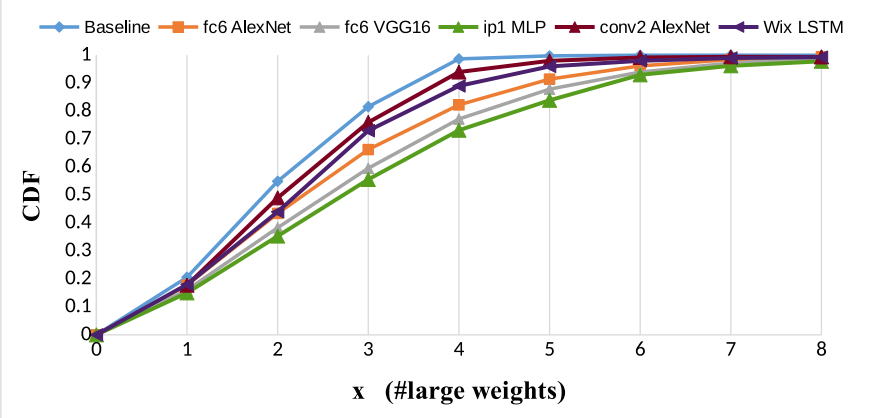


图 3.5　较大权值的累计分布

3.3　压缩神经网络

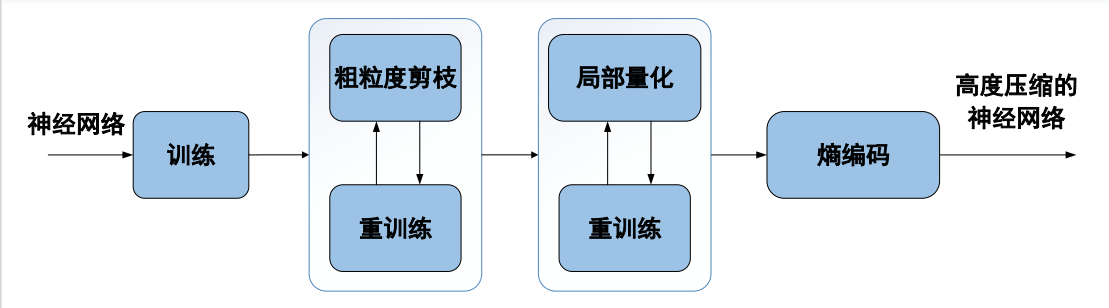


图 3.6　新的压缩神经网络步骤

如图 3.6所示，基于局部收敛的现象，我们提出了一种新的压缩神经网络的方法。整个压缩流程包括三个步骤：粗粒度剪枝（coarse-grained pruning），局部量化（local quantization）和熵编码（entropy encoding）。由于粗粒度剪枝和局部量化后会导致神经网络的精度下降，我们需要对神经网络进行重训练，保证神经网络的精度。下面我们将依次详细阐述这三个步骤。

3.3.1　粗粒度剪枝

我们提出了一种新的剪枝策略：粗粒度剪枝。粗粒度剪枝的核心思想基于神经网络局部收敛的特征，我们将多个突触聚成的簇进行剪枝操作，而不是对单个权值进行剪枝。由于在训练过程中，绝对值较大的权值倾向于聚集成簇，我们重点保留那些拥有大量较大权值的簇，裁剪那些只有少量较大权值的簇，从而在粗粒度裁剪的过程中保持神经网络的表征的能力。

我们首先将权值矩阵分成多个块，如果某一个块符合特定的条件（如权值块中所有权值绝对值的平均值小于某个阈值），这个权值块将在网络拓扑中被剪除。然后我们使用较小的学习率对网络进行重训练，从而保持网络的准确性。在训练过程中，那些被裁剪的权值保持为 “0”。值得注意的是，我们在训练中迭代地应用粗粒度剪枝和重训练，以获得更好的稀疏性，同时避免精度损失。为了清晰地阐述粗粒度剪枝技术，我们使用全连接层（如图 3.7所示）和卷积层（如图 3.8所示）作为驱动示例。

1. 剪枝策略

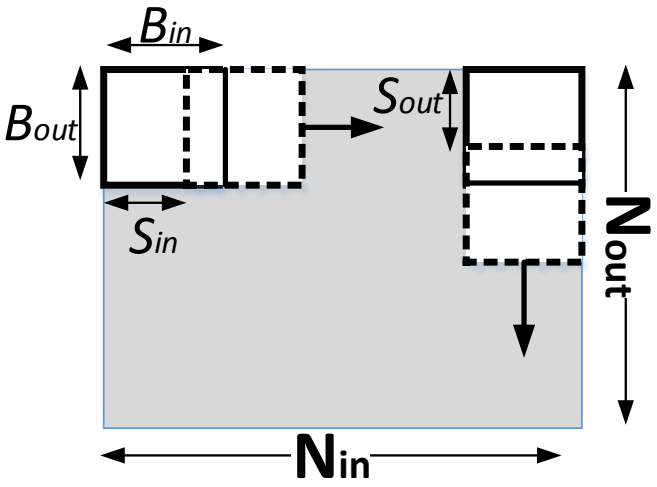


图 3.7　全连接层上的粗粒度剪枝

如图 3.7所示，在一个全连接层中，N_{out} 个输出神经元通过二维权值矩阵（N_{out}, N_{in}）与 N_{in} 个输入神经元相连。在剪枝过程中，我们设定一个大小为 $B_{in} \times B_{out}$ 的滑动窗口，使其沿着权值矩阵的两个维度分别按照 S_{in} 和 $_{out}$ 的步长进行滑动。如果滑动窗口中的权值符合某些条件，窗口内的所有突触会同时被裁剪。值得注意的是，在剪枝过程中，滑动窗口将跳过修剪过的突触块，使所有修剪过的突触块都具有相同的规模，从而简化索引过程。

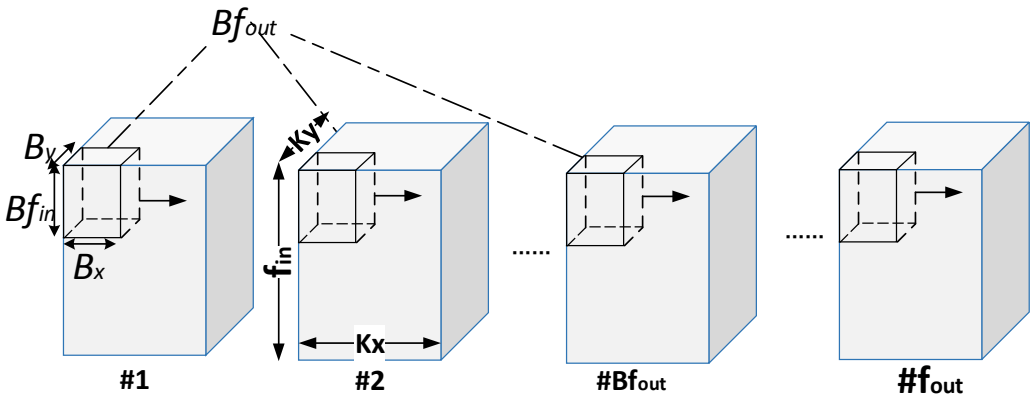


图 3.8　卷积层上的粗粒度剪枝

对于卷积层，输出特征图中的输出神经元通过共享的突触连接到输入特征图中的神经元。因此，卷积层中的权值可以表示为一个四维张量，即 $(N_{fin}, N_{fout}, K_x, K_y)$，其中 $Nfin$ 是输入特征图的数量，$Nfout$ 是输出特征图的数量，Kx,K_y 是卷积核的大小。如图所示，在剪枝过程中，我们设定滑动窗口的大小为 $B_{fin} \times B_{fout} \times B_x \times B_y$，使其沿着权值张量的四个维度分别按照 S_{fin}，S_{fout}，S_x 和 S_y 的步长进行滑动。与全连接层粗粒度剪枝类似的，如果滑动窗口中的权值符合某些条件，窗口内的所有突触会同时被裁剪。在迭代剪枝过程中，滑动窗口将跳过修剪过的突触，使所有修剪过的突触块都具有相同的规模大小，从而简化索引过程。

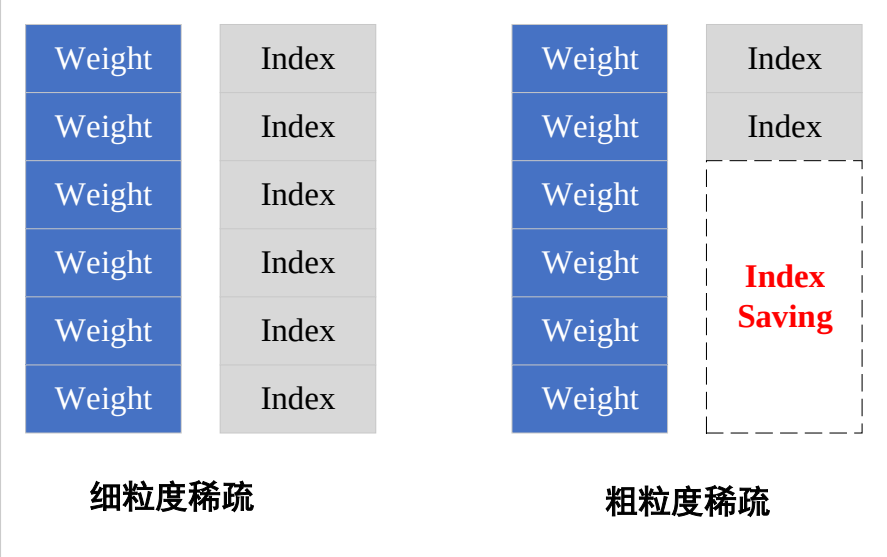


图 3.9　粗粒度剪枝减少非零权值索引信息

粗粒度剪枝策略能够从两个方面带来收益，第一，粗粒度剪枝能够有效降低神经网络的不规则度，我们将在后续详细进行讨论不规则度的定量度量方法；第二，粗粒度剪枝能够有效减少非零权值索引信息，如图 3.9所示，在相同的稀疏度下，粗粒度稀疏神经网络中一个索引可以一个权值块，即多个权值，因此可以大量减少非零权值的索引信息。

2. 剪枝块规模

在粗粒度剪枝过程中，剪枝块的大小会严重影响神经网络的精度和最终的压缩效果。使用过大的块进行剪枝将导致神经网络的表征能力严重下降，无法完全表达神经网络的特性，从而导致网络精度降低。使用过小的块进行剪枝，并不能充分利用局部权值收敛的特性，从而无法获得理想的压缩效果。因此，我们需要寻找最佳的剪枝块大小来权衡压缩比与精度之间的平衡。值得注意的是，如果我们将剪枝块的大小设定为 1，那么粗粒度的剪枝就是 [48] 中的细粒度剪枝。

在本文中，我们为神经网络不同类型的层设定不同的剪枝块的大小。理论上，我们应该为神经网络拓扑结构中的每一个卷积层/全连接层选择不同的剪枝块大小，而不是为不同类型的层选择剪枝块大小。但是考虑到庞大的设计搜索空间和极长训练时间，我们关注的是为网络中不同类型的层设定剪枝块大小。例如，由于卷积层中的权值比全连接层的权值更敏感，因此我们只在卷积层的某个维度上进行粗粒度剪枝。

我们以 AlexNet 为驱动实例，在保持网络准确性的同时，阐述不同剪枝块的规模对神经网络稀疏度和压缩效果的影响。如表 3.1所示，我们将 AlexNet 网络的基准精度设定为 top-1 误差 42.8%，即经过粗粒度剪枝后神经网络的最终 top-1 误差需要小于 42.8%，同时我们使用规模为 $(1, N, 1, 1)$ 和 (N, N) 的剪枝块分别对卷积层和全连接层进行粗粒度裁剪。我们将卷积层和全连接层的权值分别用 8 位和 4 位进行量化，然后用 Huffman 编码进行编码，最终获得神经网络的压缩比。最终 AlexNet 网络在不同剪枝块的情况下稀疏度和压缩比如表 3.1所示，当 N 从 1 增加到 64 时，压缩比 r_c 首先从 40 倍迅速增长到 79 倍，然后迅速下降到 65 倍。当 N 从 1 增大到 16 时，压缩比随之增加，主要原因是随着剪枝块规模增加，非零权值索引所需的存储空间不断减少，因此压缩比上升。当剪枝块大小从 16 增加到 64 时，压缩比随之下降，这是因为过大的剪枝块严重影响了神经网络的表征能力，为了保持相同的精度，神经网络的稀疏度不断降低，卷积层稀疏度从 64.75% 降低到 54.45%，全连接层的稀疏度从 89.95% 降低到 87.83%，从而导致压缩比下降。为了更好地平衡压缩比和精度，我们将卷积层中的剪枝块大小设置为 $(1, 16, 1, 1)$，将 fc6，fc7 和 fc8 的剪枝块大小分别设定为 $(32, 32)$，$(32, 32)$ 和 $(16, 16)$。值得注意的是，使用粗粒度剪枝，非零权值索引所需要的存储空间只需要 $29.38KB$，相比于使用细粒度剪枝后 [88] 的索引（$2.95MB$）减少了 102.82 倍。

表 3.1　不同剪枝块大小情况下 AlexNet 网络的稀疏度和压缩比（C: 卷积层；F：全连接层；S: 稀疏度；r_c 压缩比）

N	1	2	4	8	16	32	64
C:S(%)	64.99	64.99	64.89	64.77	64.75	59.95	54.45
F:S(%)	89.99	89.98	89.98	89.97	89.95	89.91	87.78
Weight (MB)	2.86	2.86	2.87	2.87	2.91	3.01	3.59
Index (MB)	2.95	0.76	0.22	0.07	0.03	0.01	0.005
r_c	40×	64×	75×	79×	79×	77×	65×

3. 最大值剪枝 vs. 平均值剪枝

在粗粒度剪枝过程中，我们可以使用两种不同的剪枝策略，第一种是最大值剪枝策略，第二种是平均值剪枝策略。对于最大值剪枝策略，如果窗口中具有最大绝对值的权值小于预定义阈值 (W_{th})，则该窗口中的所有权值被同时裁剪。对于平均值剪枝，如果窗口中所有权值绝对值的平均值小于预定义的阈值，则窗口中的所有权值被同时裁剪。这两种剪枝策略的主要特性是完全不同的。最大值剪枝表明只有块内部最大的权重才会影响整个块的重要性。平均值剪枝表明，块内的所有权重将对块的重要性产生影响。

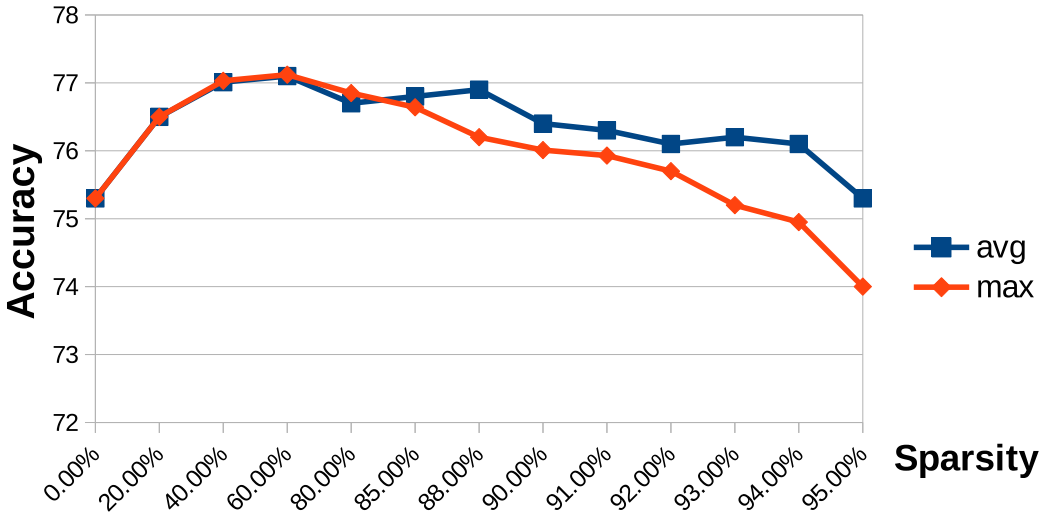


图 3.10　Cifar10 快速模型上的最大值剪枝和平均值剪枝

为了探究这两种剪枝策略的效果，我们 Cifar10 快速模型上分别采用最大值剪枝和平均值剪枝进行粗粒度裁剪和训练，然后统计在相同稀疏度下两种剪枝策略获得的精度。在剪枝过程中我们控制这两种剪枝方法使用相同大小的滑动窗口和步长。如图 3.10所示，当神经网络的稀疏度低于 85% 时，两种策略的精

度相近；当神经网络稀疏度高于 85% 时，平均值剪枝策略比最大值剪枝策略的精度更高；当神经网络稀疏度为 95% 时，平均值剪枝策略比最大值剪枝策略的精度高 1.2%。因此，本文选择了平均值剪枝的策略。值得注意的是，在剪枝初期，即稀疏度较低的情况下，稀疏神经网络的精度高于原始的稠密神经网络，这是因为稀疏能够缓减神经网络的过拟合，从而增强神经网络的表征能力，从而提高精度。

4. 不规则性的评估方法

粗粒度剪枝能够减少稀疏神经网络的不规则性，为了更直观地表现粗粒度剪枝的效果，我们提出了一种简单而有效的方法来定量测量粗粒度剪枝减少的不规则性。我们使用如下公式计算

$$R(Irr) = JBIG(I_f)/JBIG(I_c) \tag{3.4}$$

其中 $R(Irr)$ 表示减少的不规则性，I_f 和 Ic 分别表示经过细粒度剪枝和粗粒度剪枝后稀疏神经网络的索引，$JBIG()$ 表示基于 Joint Bi-level Image Experts-Group [123] 的无损二值图像压缩标准。这种方法基于这样一个事实: 常规数据 (特别是二进制矩阵) 包含了更多的冗余信息，因此可以用更少的数据来表示。因此，我们首先使用直接索引的方式将突触索引转化为二进制图像，即用“1”表示对应的突触存在，“0”表示对应的突触不存在；然后采用 JBIG 压缩二进制图像，压缩后的数据大小在某种程度上可以衡量数据的不规则性。最后，我们通过粗粒度剪枝与细粒度剪枝后压缩索引大小之比来度量降低的不规则性。

5. 神经元稀疏

在粗粒度的剪枝过程中，我们只裁剪突触，并不会对神经元进行直接裁剪，仅仅当某个神经元与其他神经元之间没有任何连接时，该神经元会被剪除，因此静态神经元稀疏的比例并不高。然而，在大型网络中，动态神经元稀疏占了很大的比例，即神经元经过激励后的输出值为“0”的情况。在表 3.2中显示了在各个网络中，权值稀疏度，静态神经元稀疏度和动态神经元稀疏度。在大规模神经网络中，如 AlexNet、VGG16 和 ResNet-152，静态神经元稀疏性非常低，在卷积层中接近 0%，即如果不对神经元进行直接裁剪，基本不会出现静态神经元稀疏的情况。然而，在这三个大规模网络中中，动态神经元稀疏性却很有挖掘和利用前景：在卷积层中，AlexNet，VGG16 和 ResNet-152 的动态神经元稀疏度分别为 37.63%，在 VGG16 为 59.48% 和 50.30%，动态神经元稀疏为提高计算性能和节省能耗提供了很好的契机。

表 3.2　神经网络中的稀疏性（C: 卷积层; F：全连接层; SSS：静态权值稀疏; SNS: 静态神经元稀疏; DNS：动态神经元稀疏）.

–	LeNet-5	MLP	Cifar10	AlexNet	VGG16	ResNet-152
C: SSS (%)	88.98	–	92.08	64.75	64.83	45.69
SNS (%)	25.39	–	11.49	0.00	0.00	0.00
DNS (%)	0.00	–	30.61	37.63	59.48	50.30
F: SSS (%)	91.47	90.13	93.99	89.95	95.16	0
SNS (%)	47.70	40.25	57.73	15.75	22.18	0
DNS (%)	11.50	66.31	19.93	29.27	43.03	24.10

3.3.2　局部量化

应用权值共享和量化的策略可以有效地减少代表权值的比特位 [88]。为了更好地利用局部收敛的特性，我们提出了局部量化的策略。不同于全局量化在整个权值矩阵中进行权值共享，局部量化仅仅在权值矩阵的局部区域中进行权值共享。

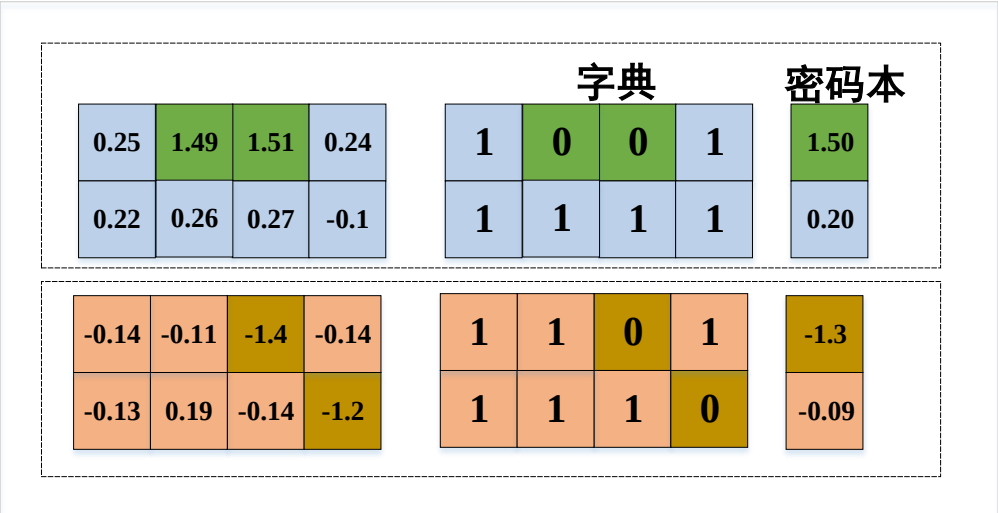


图 3.11　局部量化

如图 3.11所示，局部量化策略首先将权值矩阵划分为两个子矩阵，然后对两个子矩阵分别进行聚类。在每个子矩阵中，权值将被编码成一个密码本和一个字典。在图 3.11的图例中，字典中的每个权值只需要 1 比特进行索引，而使用图 3.2中的全局量化的方法，字典中的每个权值需要值需要 2 比特进行索引。与全局量化相比，局部量化能够利用局部收敛来进一步减少表示权值的比特数，从而获得更高的压缩比。值得注意的是局部量化的开销很小，以 AlexNet 网络的 $fc6$ 层为例, 全局量化需要一个 $128B$ 的密码本和 $2.0MB$ 的字典，其中字典中每

个元素需要 5 比特进行索引; 局部量化过程中我们将权值矩阵分为 64 个小矩阵，因此需要 64 个密码本和 64 个对应的字典，但是字典中每个元素仅仅需要 4 比特进行索引，因此密码本和字典的大小分别为 $4KB$ 和 $1.6MB$，对比与全局量化存储容量减小了 19.81%。

3.3.3　熵编码

熵编码是一种无损的数据压缩策略，它为输入中的每个符号创建并分配唯一的无前缀码。由于每个码字的长度都与该符号出现概率的负对数成比例，所以我们使用更少的比特来编码出现概率更高的符号。目前两种最常见的熵编码技术是 Huffman coding [121] 和算术编码 [122]。实验显示，熵编码能够进一步减少 $20\% - 30\%$ 的网络存储开销。

3.3.4　压缩实验结果

在表 3.3中，我们列出了七个神经网络的详细压缩结果，这七个神经网络包括 LeNet-5 [124], MLP [125]（三层神经网络，包含 300×100 个隐层神经元），Cifar10 快速模型 [126]，AlexNet [24]，VGG16 [25]，ResNet152 [34] 和 LSTM [127]。注意，我们只关注 LSTM 模型中的 LSTM 层。我们在表中详细地罗列了各个神经网络能够获得的稀疏度（Sparsity），经过三个压缩步骤（即粗粒度剪枝，局部量化，熵编码）后神经网络的规模（分别用 W_p，W_q 和 W_c 表示）和压缩比（分别用 r_p，r_q 和 r_c 表示），同时我们还罗列了粗粒度剪枝减少的不规则性（$R(Irr)$）。

1. 压缩比

我们的压缩算法在 AlexNet、VGG16、LeNet-5、MLP、Cifar10、ResNet152 和 LSTM 网络上分别获得了 79 倍、98 倍、82 倍、92 倍、69 倍、10 倍和 77 倍的压缩比。在表 3.4中，我们将压缩方法与现有的两种最先进的神经网络压缩方法进行了比较，即 Deep Compression [88]（细粒度剪枝）和 CNNpack [89]（频域压缩）。我们的压缩方法获得的压缩比几乎是 Deep Compression 和 CNNpack 的两倍，获得这么高压缩比的主要原因是我们的压缩算法能够充分神经网络局部收敛的特性，显著减少神经网络的不规则度，从而显著减少权值索引信息的存储空间。值得注意的是，我们的压缩算法除了能够获得高压缩比，还能获得与 Depp Compression 类似的稀疏度，同时精度损失几乎可以忽略不计。对于残差网络中，即 ResNet152，我们的压缩方法和 Deep Compression 只能获得不到 10 倍的压缩比，远远低于传统的神经网络。出现这种现象的原因是残差网络中包含多种新的特性，包括快捷连接（shortcut connections）和批处理归一化（batch

表 3.3　压缩后神经网络的稀疏度，压缩比和不规则性减少量 (W_p: 粗粒度剪枝后权值规模; W_q: 粗粒度剪枝，局部量化后权值规模; W_c: 粗粒度剪枝，局部量化，熵编码后权值规模; L: *LSTM* 层; C: 卷积层; F: 全连接层; W: 权值; I: 权值索引; r_p: 粗粒度剪枝后压缩比; r_q: 粗粒度剪枝，局部量化后压缩比; r_c: 粗粒度剪枝，局部量化，熵编码后压缩比; $R(Irr)$: 不规则性减少量)。

Model	Sparsity(%)	$W_p(B)$	r_p	$W_q(B)$	r_q	$W_c(B)$	r_c	$R(Irr)$
Alexnet	C: 64.75	W: 25.65M	9×	W: 3.60M	64×	W: 2.90M	79×	101.65×
	F: 89.93	I: 36.73K		I: 36.73K		I: 29.38K		
VGG16	C: 64.83	W: 42.72M	12×	W: 7.80M	66×	W: 5.25M	98×	28.54×
	F: 95.16	I: 202.80K		I: 202.80K		I: 121.68K		
LeNet-5	C: 88.98	W: 135.96K	12×	W: 23.77K	66×	W: 19.01K	82×	8.87×
	F: 91.47	I: 1.67K		I: 1.67K		I: 1.39K		
MLP	C: –	W: 102.54K	10×	W: 19.23K	51×	W: 12.01K	82×	10.41×
	F: 90.13	I: 1.20K		I: 1.20K		I: 0.61K		
Cifar10	C: 92.08	W: 42.08K	13×	W: 8.68K	62×	W: 7.82K	69×	7.61×
	F: 93.99	I: 0.48K		I: 0.48K		I: 0.42K		
ResNet152	C: 45.69	W: 134.10M	1.7×	W: 33.50M	7×	W: 23.44M	10×	13.02×
	F: 0.00	I: 0.49M		I: 0.49M		I: 0.44M		
LSTM	L: 87.94	W: 1.50M	8.3×	W: 191.28K	60×	W: 152.47K	77×	50.51×
		I: 25.96K		I: 25.96K		I: 17.84K		

normalization)，这些特性从而大大缓和了神经网络的过拟合的现象。因此，我们在 ResNet152 网络中只能获得一个非常有限的压缩比。

2. 精度

与表 3.4，我们将我们压缩算法的准精确与 Deep Compression 和 CNNpack 进行比较。我们可以看出，我们的压缩方法造成的精度损失是可以忽略不计的。与参考准确度相比，我们的压缩方法的平均精度损失低于 0.2%，类似于 Deep Compression（0.1%），并且低于 CNNpack(0.7%)。

3. 不规则度

表 3.3还列出了粗粒度剪枝减少的不规则性。值得注意的是，粗粒度的剪枝可以将不规则性平均减少 20.13 倍，剪枝的块越大，不规则性减少的程度就越高。对于小型网络，如 LeNet-5、MLP 和 Cifar10，减少的不规则性平均是 8.80 倍，因

表 3.4　CNNPack, Deep Compression 与我们的压缩方法的对比 (S%: 稀疏度; r_c: 压缩比).

model	Ref	Deep Cmp. [88]			CNNpack [89]			Ours		
	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)
AlexNet	42.78	88.85	35×	42.78	–	39×	41.60 (41.80*)	88.97	79×	42.72
VGG16	31.50	92.39	49×	31.17	–	46×	29.70 (28.50*)	91.93	98×	31.33
LeNet-5	0.80	91.57	39×	0.74	–	–	–	91.40	82×	0.95
MLP	1.64	91.82	40×	1.58	–	–	–	90.13	82×	1.91
Cifar10	24.20	94.98	45×	24.33	–	–	–	92.93	69×	24.22
ResNet152	25.00	45.00	8×	24.40	–	–	–	44.17	10×	25.05
LSTM	20.23	88.47	35×	20.52	–	–	–	87.94	77×	20.72

为我们只能用比较小的剪枝块对小的神经网络进行修剪，以减少精度损失。相反，大型网络，如 AlexNet、VGG16 和 LSTM，减少的不规则性要大得多，平均是 52.71 倍，因为我们可以选用更大的剪枝快对神经网络进行修剪。另外，对于残差网络，考虑到新特性，我们只能用比较小的剪枝块来修剪它，能够减少 13.02 倍的不规则性。实验结果进一步证实粗粒度剪枝可以显著降低网络的不规则性。

4. AlexNet 网络详细压缩特性

表 3.5　AlexNet 压缩特征 (W: 权值; S%: 稀疏度 I: 权值索引).

layer	#W	S%	bits per W	W size(KB)	I size(KB)
conv1	35K	24.9	6.5	20.75	0.80
conv2	307K	65.5	5.6	72.45	3.23
conv3	885K	64.1	5.4	209.37	9.69
conv4	663K	66.6	6.2	167.73	6.76
conv5	442K	65.9	5.5	101.28	4.61
fc6	38M	91.1	3.3	1353.37	1.60
fc7	17M	91.10	3.4	626.69	0.72
fc8	4M	74.8	3.3	415.80	1.97
total	61M	88.9	3.7	2967.44	29.38

此外，我们在表 3.5列出了 AlexNet 网络的详细压缩特性。卷积层和 fc6，fc7，fc8 层的块大小分别为 $(1, 16, 1, 1,)$，$(32, 32)$，$(32, 32)$，$(16, 16)$。我们将卷积层和全连接层的权值分别量化为 8 比特和 4 比特。表中的权值和权值索引采用了都用 Huffman 编码。值得注意的是，与权值大小相比，权值索引大小几乎可以忽略

不计。而在 Deep Compression 中，权值索引信息几乎是总存储容量的 40%，这严重阻碍了更有效的压缩。此外，使用细粒度的修剪时，权值和权值索引分别是 $25.93MB$ 和 $3.24MB$; 使用粗粒度剪枝时，权值大小和索引大小分别是 $25.65MB$ 和 $36.73KB$(比细粒度剪枝减少 90.32 倍)。局部量化进一步将权值大小减小到 $3.6MB$，而全局只能将权值大小压缩到 $4.95MB$。经过熵编码后，索引大小只有 $29.38KB$，对比与 Deep Compression 的 2.95MB 缩小 102.82 倍。受益于粗粒度稀疏和局部量化，我们新的压缩方法可以将 AlexNet 压缩 79 倍，远远高于 Deep Compression（35 倍）和 CNNPack（39 倍）。

第 4 章　粗粒度稀疏神经网络加速器

我们的压缩方法能够对神经网络进行深度压缩，其中粗粒度稀疏能够充分利用神经网络局部收敛的特征，寻找最能够表征神经网络特征的权值簇，从而有效减少稀疏权值的不规则性（平均减少 20.13 倍）; 局部量化进一步利用局部收敛的特性，减少表示每个权值的比特数; 最后熵编码对神经网络进行进一步的无损压缩。深度压缩的神经网络能够减少数据存储需求，减少计算量，减少数据传输量，最终提高计算性能，减少计算能耗。

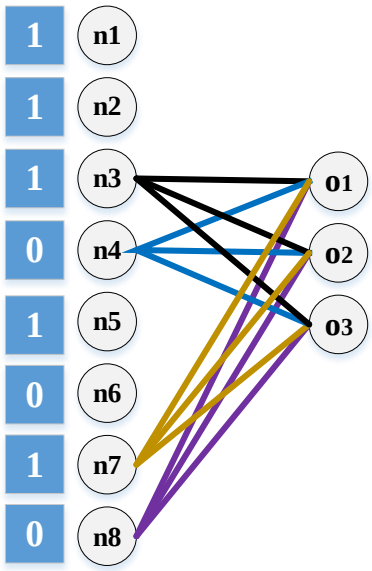
因此我们设计专用的硬件加速器 Cambricon-S，充分挖掘深度压缩神经网络的优秀特性。Cambricon-S 最主要的特征是两个选数逻辑: 神经元选择模块 (neuron selector module, NSM) 和突触选择模块 (synapse selector module, SSM)，分别用来处理静态稀疏和动态稀疏，过滤不必要的神经元和权值，筛选出需要进行计算的神经元和权值。除了两个选数模块，我们设计了 WDM（weight decoding module）和 Encoder 分别支持局部量化和动态压缩神经元。同时，我们设计了一个多发射的控制器，能够同时发射没有依赖关系的指令，从而挖掘 IO 与计算之间的并行性，进一步提高加速器的性能。

为了减轻用户的编程负担，我们为加速器设计了专用的基于库的编程模型。编程模型中集成了加速器专用的编译器，能够将 C++ 高级语言编译成为加速器可执行指令；值得注意的，编译器能够静态分析指令之间的依赖关系，提取出没有依赖关系的指令，使它们能够在加速器中同时进行发射。

本章中我们首先以一个粗粒度稀疏的全连接神经网络为驱动实例，充分分析神经网络加速器的设计原则。然后我们根据这些设计原则设计对应的神经网络加速器。最后我们为加速器设计了一套基于库的编程框架以减轻用户的编程负担。

4.1　设计原则

我们将使用一个经过粗粒度稀疏的全连接层作为示例 (参见图 4.1)，分析粗粒度稀疏神经网络的特性，进而分析出神经网络加速器的设计原则。在图中，我们只关注其中规模为 8×3 的部分连接，我们使用 2×3 的剪枝块对全连接层进行剪枝。由于粗粒度剪枝，输入神经元 $n1, n2$ 与输出神经元 $o1, o2, o3$ 之间的连接被完全裁剪；输入神经元 $n5, n6$ 与输出神经元 $o1, o2, o3$ 之间的连接也被完全裁剪。由于 ReLU 激励函数，输入神经元 $n4, n6, n8$ 的激活为 “0”。我们采用直接索引的方式标志非零权值的信息，即我们使用 “*00110011*” 比特串标志与输出



neuron indexes	input neurons	output neurons			synapse indexes
		o1	o2	o3	
1	n1	0	0	0	0
1	n2	0	0	0	0
1	**n3**	**s11**	**s21**	**s31**	**1**
0	n4	s12	s22	s32	1
1	n5	0	0	0	0
0	n6	0	0	0	0
1	**n7**	**s13**	**s23**	**s33**	**1**
0	n8	s14	s24	s34	1

o1 = n3*s11 + n7*s13
o2 = n3*s21 + n7*s23
o3 = n3*s31 + n7*s33

图 4.1　粗粒度稀疏的全连接层

神经元 $o1, o2, o3$ 对应的突触，其中“0”表示对应的突触被裁剪，“1”表示对应的突触存在。通过观察，我们发现了以下四个粗粒度稀疏神经网络的特性。

首先，多个输出神经元之间共享索引信息。如图 4.1所示，输入神经元 $n1$，$n2$，$n5$，$n6$ 与输出神经元 $o1, o2, o3$ 之间的连接被同时剪除，因此输出神经元之间共享相同的连接拓扑，也就是说它们共享相同的权值索引表示“*00110011*”(图中的“Synapse Indexes”)。尽管输出神经元之间不共享相同的权值，但所选权重的位置是相同的，因此是一种部分共享。

第二，多个输出神经元之间共享输入神经元。如图 4.1所示，最终需要参与计算的神经元是 $n3, n7$，它们的值被输出神经元共享。不失一般性，在完全连接的层中，设定修剪块大小为 (B_{in}, B_{out})，那么 B_{out} 个相邻输出神经元将共享相同的输入神经元。在卷积层中，设定剪枝块的大小为为 $(B_{fin}, B_{fout}, B_x, B_y)$，那么 B_{fout} 个相邻的输出神经元将共享相同的输入神经元。

第三，动态稀疏性能够进一步提高运算效率。在图中，通过挖掘权值稀疏，输入神经元 $n3, n4, n7, n8$ 被筛选出来进行计算，同时由于输入神经元 $n4, n8$ 的激活为零，在运算过程中，它们对输出神经元没有贡献，最后需要进行计算的输入神经元为 $n3, n7$。在图中的实例中，通过挖掘权值稀疏性，共需要进行 12 次乘法，9 次加法完成运算；同时挖掘权值稀疏性和动态神经元的稀疏性，只需要进行 6 次乘法和 3 次加法。对比在稠密情况下 24 次乘法和 21 次加法，分别有 2.14 倍和 5 倍的性能提升。因此，利用动态神经元稀疏性是进一步提高效率的关

键 (在上面的示例中，能够进一步提升 2 倍性能)。值得注意的是，即使考虑到动态神经元稀疏，输出神经元仍然共享索引和所选的输入神经元。

第四，多个输出神经元之间的负载是平衡的，因为它们共享相同的输入神经元。对于图 4.1中的示例，如果考虑静态稀疏，那么每个输出神经元共享相同的输入神经元 $n3, n4, n7, n8$，同时需要四个对应的权值 $S_{Ti1}, S_{Ti2}, S_{Ti3}, S_{Ti4}$ 进行计算，因此每个输出神经元需要进行 4 次乘法和 3 次加法完成运算。如果同时考虑静态稀疏和动态稀疏，那么每次输出神经元共享相同的输入神经元 $n3, n7$，同时需要两个对应的权值 S_{Ti1}, S_{Ti3} 进行计算，因此每个输出神经元需要进行 2 次乘法和 1 次加法完成运算。因此粗粒度稀疏的神经网络可以避免负载不平衡而造成的性能损失 [86]。

因此，在设计加速器时要考虑以下原则，以最大限度地提高加速器的效率：加速器 (1) 能够利用共享的索引信息和共享的输入神经元信息，简化加速器的设计；（2）能够利用动态稀疏性进一步提高效率;(3) 利用相邻输出神经元之间的负载均衡。

4.2　Cambricon-S 的架构

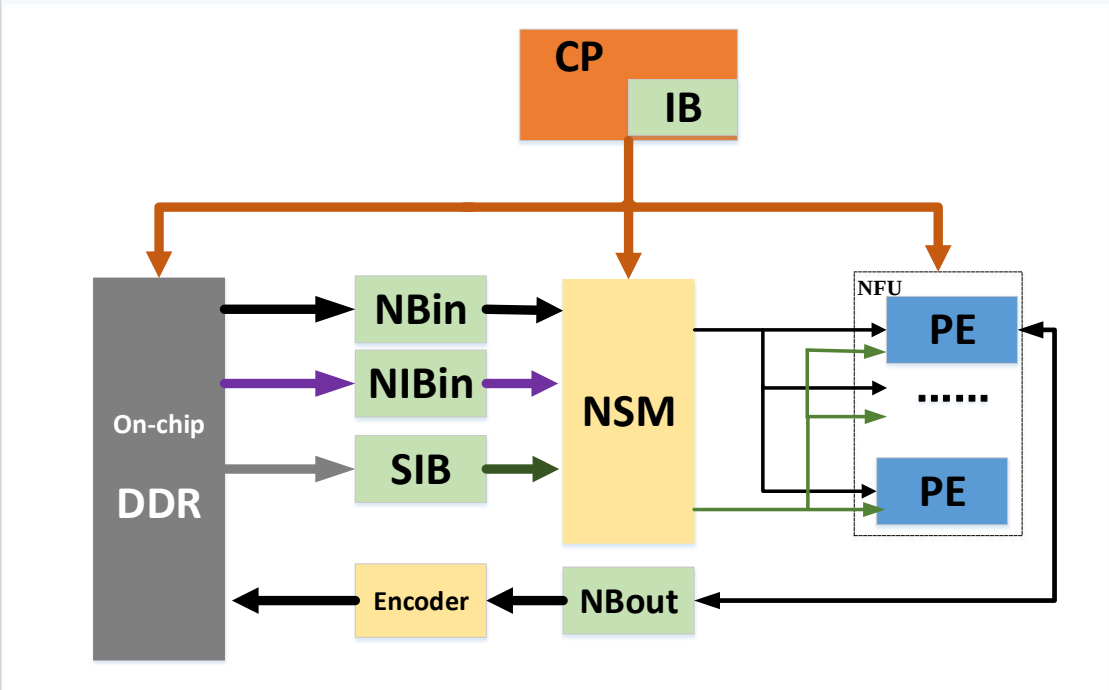


图 4.2　Cambricon-S 整体架构

在本节中，我们将介绍 Cambricon-S 的详细架构，它能够充分利用深度压缩神经网络带来的收益，有效地解决粗粒度修剪稀疏网络的剩余不规则性。

如图 4.2所示，我们展示了加速器架构。按照 4.1的设计原则，我们在加速器

中设计了一个关键模块-神经元选择器模块（neural selector module, NSM），用于处理静态稀疏和共享信息（包括共享索引信息和共享神经元信息）。同时我们设计了一个神经功能单元（neural functional unit，NFU）用于完成神经网络中的核心计算。NFU 中具有多个处理单元（processing elements, PEs）用来并行计算不同的输出神经元。每个 PE 都包含一个本地的突触选择器模块（synapse selector module, SSM）来处理动态稀疏性。动态神经元压缩模块（Encoder）用于动态地将输出神经元压缩成为非零元素/非零元素索引的模式，从而减少 Load/Store 神经元的开销。存储模块需要存储输入神经元，权值和输出神经元，因此我们需要输入神经元缓存（input neuron buffer, NBin），输出神经元缓存（output neuron buffer, NBout）和突触缓存（synapse buffer, SB），考虑到神经元和权值的稀疏性，我们需要两个额外的缓存，即输入神经元索引缓存（input neuron index buffer, NIBin）和突触索引缓存（synapse index buffer, SIB ）分别存储输入神经元索引和突出索引信息。注意，其中 SB 被内置与 NFU 的每一个 PE 中，没有表现在图中。控制模块由控制处理器（control processor, CP）和指令缓存（instruction buffer, IB）组成，CP 有效地将 IB 中存储的各种指令解码为所有其他模块的详细控制信号，我们将 CP 设计为一个多发射的控制器（multi-issue controller），从而挖掘访存与计算之间的并行性，进一步提升加速器的性能，同时我们为加速器定义一个 VLIW（very long instruction word）风格的指令集。

下面我们将从稀疏处理，存储，控制和片上互联四个方面介绍加速器。

4.2.1　稀疏处理

该加速器旨在利用 (1) 静态稀疏性和 (2) 动态稀疏性，以及 (3) 局部量化，从而减少数据存储量，减少数据传输量，减少计算量，从而提升性能，并减少能耗。在加速器中，稀疏性由 NSM，NFU 和 Encoder 这三个模块共同处理。NSM 接收来自 NBin 的输入神经元，来自 NIBin 的输入神经元索引和来自 SIB 的权值索引，筛选出需要进行计算的神经元（静态稀疏），同时生成筛选权值的 synapse flags，然后将筛选出的神经元和 synapse flags 通过 NFU 广播给 PE。每个 PE 中的 SSM 通过 synapse flags 筛选出需要进行计算的权值（动态稀疏），从而避免无用的计算和数据传输，提高加速器的性能。Encoder 用来动态压缩稀疏的输出神经元，从而减少片外访存能耗。此外每个 PE 中集成了一个权值解码模块（WDM），用于解码经过局部量化的权值。

1. Indexing

在详细说明 NSM，NFU，Encoder 之前，我们将详细说明如何在加速器中存储和索引稀疏数据。表示稀疏数据的方法主要有两类，分别是 binary mask 和 numerical indexing，这两类方法都仅存储非零元素，使用不同的方式索引非零元素。Binary mask 的方式主要包括直接索引（direct indexing），我们使用比特串对非零元素进行索引，其中”0” 表示对应的位置的元素为零，”1” 表示对应位置元素为非零。Numerical Indexing 包括步长索引（step indexing），压缩稀疏行（compressed sparse row, CSR），压缩稀疏列（compressed sparse column，CSC），坐标列表查找（coordinate list，COO）等，它们主要使用非零元素之间相对位置（步长）或者绝对位置信息对非零元素进行索引。

由于目前主流的神经网络稀疏度低于 95%，采用 CSR，CSC，COO 等方式的存储开销远远大于 direct indexing 或者 step indexing 的方式；而且考虑到神经元和权值同时稀疏，step indexing 这种方式很难索引到两者同时非零的情况，因此我们采用直接索引的方式存储稀疏神经元和稀疏权值。如图 4.1，我们仅需要存储 $n1, n2, n3, n5, n7$ 这五个神经元和 “*11101010*” 比特串就能够表示稀疏的输入神经元；同时我们仅需要存储 $s11, s12, s13, s14$ 这四个权值和 “*00110011*” 比特串就能够表示与 $o1$ 相连的稀疏突触。值得注意的是，由于稀疏权值是一种静态稀疏，这种编码压缩过程可以离线完成；但是稀疏神经元是一种动态稀疏，这个过程需要在线实时完成，因此我们需要 Encoder 模块对稀疏神经元进行动态压缩。

2. NSM

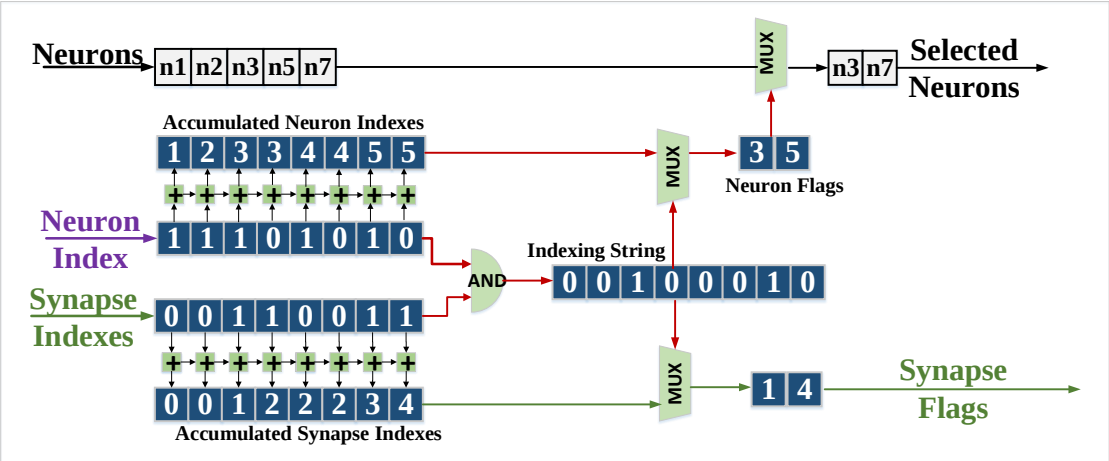


图 4.3　NSM 结构

如图 4.3所示，NSM 模块的输入为输入神经元，输入神经元索引和权值索引，NSM 模块通过神经元索引和权值索引筛选需要进行计算的输入神经元来处

理静态稀疏。为了更有效地利用粗粒度稀疏特性，即多个相邻的输出神经元共享索引和输入神经元，我们设计了一个中心 NSM，它能够广播筛选出的输入神经元信息到多个 PE 中，从而完成输入神经元的共享，保证多个 PE 之间的负载均衡。我们利用图 4.1作为实例来描述 NSM 筛选输入神经元的流程。

在图 4.1中的实例中，NSM 需要筛从 8 个输入神经元中筛选出需要进行计算的神经元 $n3, n7$ 并且生成筛选权值需要的 synapse flags。具体来说，首先 NSM 对 neuron indexes（“*11101010*”）和 synapse indexes（“*00110011*”）执行“AND”操作，从而生成 indexing string （“*00100010*”）。同时 neuron indexes 和 synapse indexes 分别累计自身，获得 accumulated neuron indexes （“*12334455*”）和 accumulated synapse indexes （“*00122234*”），然后分别与 indexing string 执行“MUX”操作，生成 neuron flags（“*35*”）和 synapse flags（“*14*”）。neuron flags 和 synapse flags 中的数字就表示了需要筛选的神经元和权值的最终位置信息。最后 neuron flags 与 neurons 执行“MUX”操作筛选出最终需要进行计算的神经元 $n3, n7$。值得注意的是，筛选出的神经元与 synapse flags 被多个输出神经元共享，因此这些信息最后被广播到 NFU 的各个 PE 中。最终 synapse flags 在 SSM 中被用于筛选出需要进行计算的权值 (见图 4.6(a))。

3. NFU

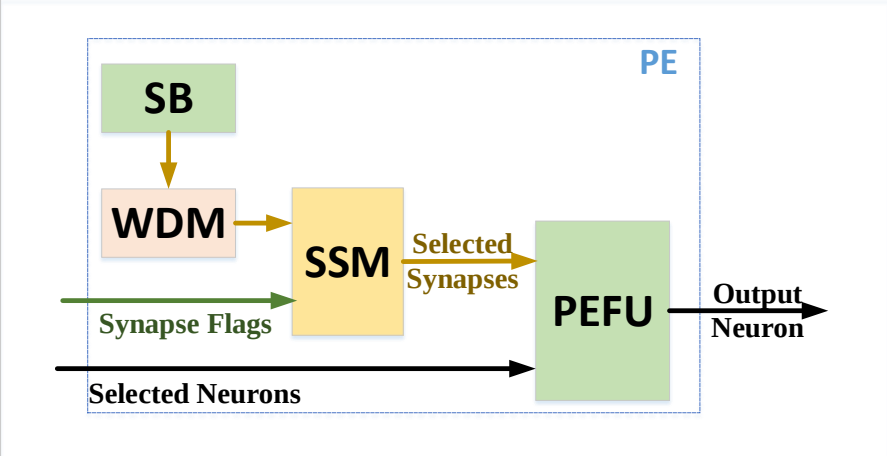


图 4.4　PE 结构

NFU 可以高效地处理神经网络中的所有运算，NFU 包含 T_n 个同构的 PE 来处理动态稀疏性，同时完成神经网络的运算。如图 4.4所示，PE 由一个局部突触缓存（synapse buffer, SB），权码解码模块（weight decoding module, WDM），突触选择模块（synapse select module, SSM）和处理功能单元（processing element functional unit, PEFU）组成。

SB. SB 用来存储 PE 计算所需要的突触。由于每个 PE 处理共享相同的输入

神经元，但是使用独立的权值计算不同的输出神经元。因此我们在每个 PE 中集成了一个本地的 SB，使得每个 PE 能够从本地 SB 中加载权值，从而缓减数据传输中的网络拥塞（network congestion），减少连线开销。

WDM. WDM 用来解码经过局部量化的权值。WDM 中集成了一个查找表（look-up table, LUT），查找表中存储了权值密码本，查找表的输入为权值字典，最终通过查表操作解码出实际的权值。

SSM. SSM 接收权值和 synapse flags，处理动态稀疏，最终筛选出需要进行计算的权值（如图 4.5）。我们利用图 4.1作为实例来描述 SSM 筛选权值的流程。在图 4.1的实例中，对于每个输出神经元 $o1, o2, o3$，第一个和第四个突触 (S_{Ti1}，S_{Ti4}) 是计算所需的两个突触。SSM 接收来自 NSM 的 synapse flags 筛选突触，其中 synapse flags 中包含所需突触位置索引信息（“14”）。如图 4.5所示，SSM 在权值与 synapse flags 之间执行 “MUX” 操作，最终筛选出需要进行计算的权值，即 S_{Ti1} 和 S_{Ti4}。由于每个 PE 接收不同的权值，并且处理不同的输出神经元，我们在每个 PE 内部局部集成 SSM，从而避免高带宽和长延迟。

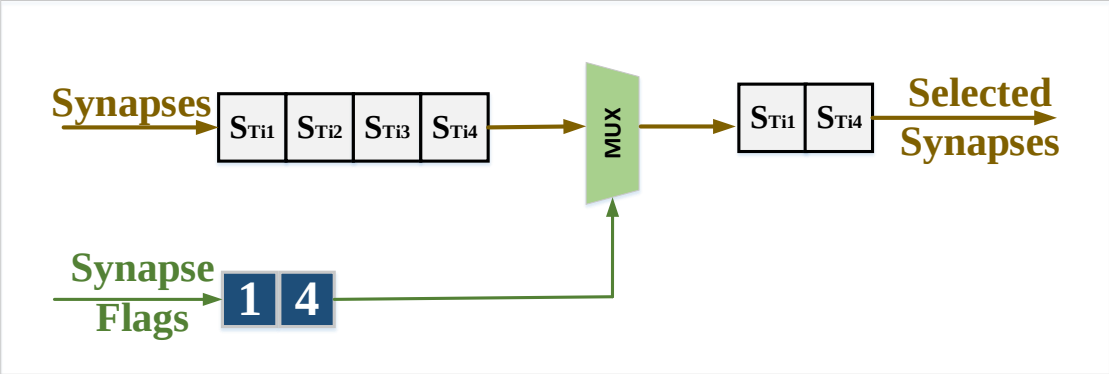


图 4.5　SSM 结构

PEFU. PEFU 用来完成神经网络的核心运算。如图 4.6所示，PEFU 由三段流水结构，分别完成乘法，累加和非线性的运算。每个 PEFU 由 T_m 乘法单元，一个 T_m 输入的加法器树和一个非线性函数模块组成。我们使用分时复用的方法将神经网络映射到 PE，即每个 PE 计算一个输出神经元。理想情况下，如果一个输出神经元需要 M 个乘法操作，那么 PE 需要 $\lceil M/T_m \rceil$ 个 cycle 完成计算元，因为 PEFU 可以在一个 cycle 内完成 T_m 次乘法。最后，NFU 将从 T_n 个 PE 中收集 T_n 个输出神经元的部分和，传输到 NBout 中。

PEFU 中的非线性运算主要是指超越函数（Transcendental Functions）。超越函数是变量之间的关系不能用有限次加、减、乘、除、乘方、开方运算表示的函数。常见的超越函数包括三角函数，指数函数，对数函数，双曲函数等。目前运算超越函数的方法主要有差值查表和迭代两种。其中迭代方法中最有效的方法是 CORDIC 算法。

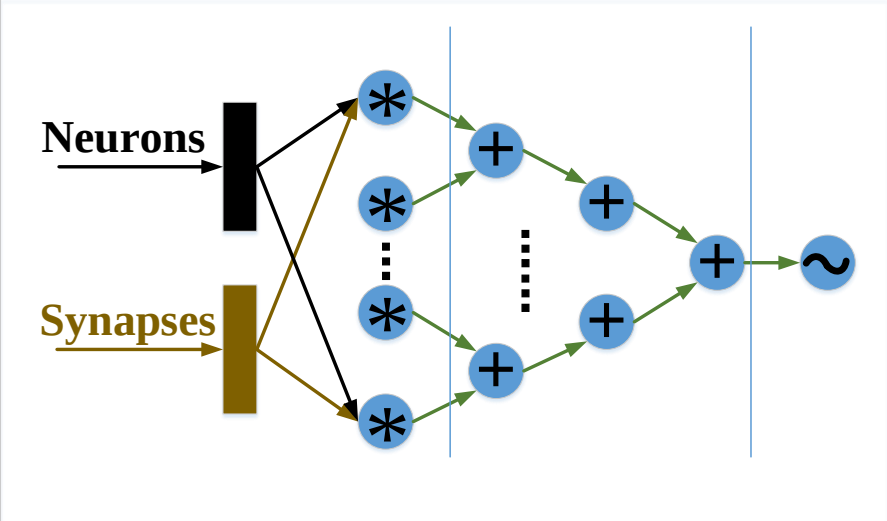


图 4.6　PEFU 结构

CORDIC（Coordinate Rotation Digital Computer）算法即坐标旋转数字计算方法，能够既方便又有效地用来计算三角函数和双曲函数。该算法中涉及到的运算只有加法，减法，移位和查表运算，非常适合用硬件实现。算法由 Volder [128]于 1959 年提出，首先用于导航系统，使得矢量的旋转和定向运算不需要做查三角函数表、乘法、开方及反三角函数等复杂运算。 Walther [129]在 1974 年用它研究了一种能计算出多种超越函数的统一算法。CORDIC 的每一次迭代能够额外获得 1 位的精度，因此迭代次数越多，CORDIC 的精度也就越高。

线性差值查表的方法目前已经在多个神经网络加速器中被应用 [77, 130]，我们将超越函数分为 M 个段，对每一段函数使用线性函数进行拟合获得斜率 a 和截距 b，其中第 i 段可以使用线性函数 $f(x) = a_i \times x + b_i$ 进行近似。

尽管 CORDIC 的精度高于线性差值查表的方法，但是面积和能耗开销远远高于查表的方法。由于神经网络具有很强的鲁棒性，线性差值查表的方法并不会引起网络精度下降 [130]，因此我们选择查表的方法完成非线性运算。

4. Encoder

Encoder 模块与 Nbout 直接相连，用于对输出神经元进行压缩，消除输出神经元中的零元素，最终用 direct indexing 的形式表示稀疏的输出神经元。图 4.7展示了神经元压缩模块 Encoder 的架构。具体来说，Encoder 根据神经元的激活值，生成 synapse indexes（“11101010”），每个索引指示相应的神经元激活是否为零。然后 Encoder 在神经元与 synapse indexes 之间执行 “MUX” 操作筛选出非零的神经元。最终非零神经元与对应的索引 synapse indexes 被传输到片外，然后作为下一层网络的输入被重新加载到 NBin。通过动态压缩神经元，加速器能够显著减少 DRAM 访问神经元的开销，从而减少 DRAM 的访存能耗。

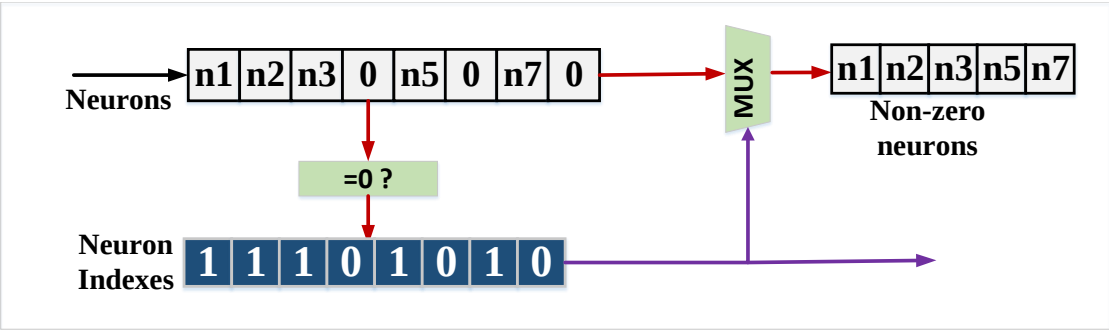


图 4.7　Encoder 架构

4.2.2　存储模块

考虑到不同数据在加速器中的不同行为模式，我们将片上缓存分为三个部分：一个输入神经元缓存（NBin），一个输出神经元缓存（NBout）和 T_m 个权值缓存（SB），分别用来存储输入神经元，输出神经元和权值。考虑到神经元稀疏和权值稀疏的情况，我们增加了两个索引缓存，分别是输入神经元索引缓存（NIBin）和权值索引缓存（SIB）。为了实现输出传输与计算之间的 overlap，我们使用乒乓（ping-pong）的模式管理所有的缓存。下面我们将着重讨论各个缓存的带宽和存储容量。

1. NBin

对于 NBin，我们设置带宽为 $16 \times T_m \times 16$ 比特，设置这个带宽主要是考虑到 PE 的利用率。虽然 PE 共享相同的输入神经元，但是每个 PE 在一个 cycle 需要 T_m 个输入神经元来避免 PEFU 中计算单元的空转。考虑到现有大部分神经网络的稠密度都高于 $1/16$（即稀疏度低于 $15/16$），特别是卷积层，一般稠密度为 30% 左右，因此，我们在一个 cycle 发送给 NSM $16 \times T_m$ 个输入神经元时，就能够保证 NSM 在一个 cycle 能够筛选出 T_m 个输入神经元，从而保证 PEFU 的运行效率。

2. NBout

对于 NBout，我们设置其带宽为 $4 \times T_n \times 16$ 比特，这部分主要是考虑到动态神经元稀疏。如表 3.2考虑到现有大部分神经网络的神经元稠密度高于 25%（即稀疏度低于 75%），当我们在每一个 cycle 发送给 Encoder $4 \times T_n$ 个输出神经元时，就能够保证 Encoder 在一个 cycle 筛选出 T_n 个神经元进行压缩，从而充分利用 NBout 与片外 DDR 之间的带宽。

3. SB

对于每一个 PE 中的 SB，我们设置带宽 $4 \times T_m \times 16$ 比特，保证能够在一个 cycle 读取 $4 \times T_m$ 个权值。考虑到现有大部分神经网络的动态稀疏度低于 75%，$4 \times T_m \times 16$ 比特的带宽能够保证 SSM 在一个 cycle 筛选出 T_m 个权值，从而保证 PEFU 的运行效率。由于权值是以压缩形式（通过剪枝和局部量化）存储在 SB 中，局部量化会导致权值在不同的神经网络和不同层中的位宽不同，例如，在 AlexNet 网络的卷积层中权值被量化为 8 比特，在 MLP 网络的全连接层中权值被量化为 6 比特，在 AlexNet 网络的全连接层中权值被量化为 4 比特。不同的量化比特会导致 WDM 模块庞大的面积/功耗开销，因此我们将权值按照 4 比特进行对齐操作。例如，假设 SRAM 的一行为 128 比特，那么当权值被量化为 16 比特，8 比特和 4 比特时，SRAM 的一行分别能够存储 8 个，16 个和 32 个权值。因此，$T_m \times 64$ 比特的权值能够被 WDM 解码成为 $T_m \times 16$ 个权值（当权值被编码成小于或者等于 4 比特），或者 $T_m \times 8$ 个权值（当权值被编码成大于 4 比特而小于等于 8 比特），或者 T_m 个权值（当权值被编码成大于 8 比特，小于等于 16 比特）。对比于支持所有位宽编码形式的解码模块，我们的 WDM 能够减少 5.14 倍的面积和 4.27 的功耗。

4. NIBin 和 SIB

对于 NIBin 和 SIB, 我们设定其宽度为 $16 \times T_m \times 1$ 比特，因为我们使用直接索引的方式存储稀疏的神经元和权值，索引中的每一比特用于表示对应的元素是否为 “0”。因此,NIBin 和 SIB 则需要 $16 \times T_m \times 1$ 比特的带宽与 Nbin 提供给 NSM 的 $16 \times T_m$ 个输入神经元一一对应。

5. 缓存大小

五个缓存的大小决定了整个架构的整体性能和能耗。尽管有部分研究 [85, 109] 提出，加速器需要提供足够大的缓存来保存神经网络所有的权值和神经元从而避免高昂的片外访存开销，但是这种设计会大大增加延迟，面积，功耗；同时这种设计方案缺少可扩展性，并不能很好地支持新兴的规模更大，层数更深的神经网络。因此我们使用小的缓存以及适当的数据替换策略，将数据分多次传输到片上进行运算，从而权衡加速器的可扩展性，性能和能耗。在探索了不同大小的缓存区后，我们设定 NBin，NBout，SB，SINin 和 SIB 的大小分别是 $8KB$，$8KB$，$32KB$，$1KB$ 和 $1KB$。

4.2.3　控制

CP 用于控制加速器的整个执行流程。它从内部指令缓存中读取指令然后进行解码，从而有效地协调数据摆放形式，内存访问和数据计算的过程。CP 通过监控各个模块对应的状态寄存器来监控每个模块的状态，然后通过设置各个模块的控制寄存器管理各个模块的运行。我们利用基于库的编译器生成高效的超长指令字（Very Long Instruction Word，VLIW）。

1. 状态机

我们为 CP 设计了高效的有限状态机（finite state machine, FSM）。如图 4.8所示，FSM 分为两层控制状态，第一层状态表示神经网络中的宏观操作，如卷积，内积，池化，激活等操作; 第二层状态则是每一个宏观操作中的细节，如对于卷积操作还会有数据预取，计算，写回等不同的状态。分层有限状态自动机能够限制状态机的跳转，从而简化状态机的设计。

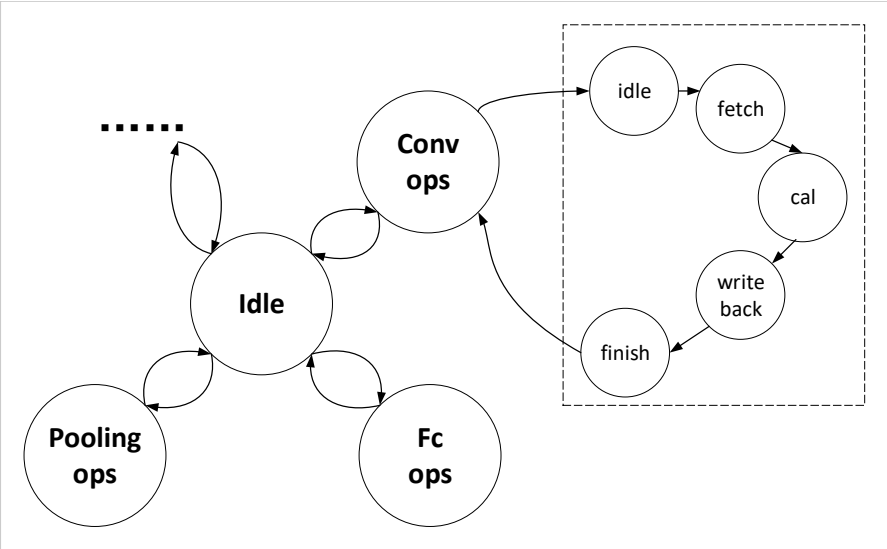


图 4.8　CP 的有限状态机

2. 多发射控制器

单指令发射的控制器会忽视一些潜在的提高加速器性能的因素。单发射的控制器需要等待 IO 指令（Load/Store）将数据完全从片外搬运到片上缓存后才能发射下一条计算指令，即使我们采用 pingpong 的方式管理片上缓存也无法使 IO 指令和计算指令并行执行。一种简单但是有效的解决方案是加速器的控制器能够提前解码下一条指令，如果不与上一条指令不冲突，则发射下一条指令。因此，我们设计了一个多发射控制器，用于充分挖掘 IO 指令与计算指令之间的并行性。

多发射控制器的关键特征是能够在一个周期内发射多条没有依赖关系的指令。一般来说，没有依赖关系的指令包括三个方面，首先 IO 指令与计算指令之间没有依赖关系，可以同时发射执行；第二，由于 Off-chip Memory 和片上 SRAM 均采用双端口的设计模式，因此 Load 和 Store 指令之间没有依赖关系，可以同时发射执行；第三，如果计算指令涉及到涉及不同的计算单元和数据，那么这些计算指令也能同时发射执行。IO 指令主要包括 NBin，SB 以及 NBout 与片外之间的 Load/Store 指令；计算指令指令主要包括矩阵/向量运算，标量运算，逻辑运算等。我们在编译过程中静态分析指令之间的依赖关系，提取出没有相互依赖关系的指令，在执行时同时进行发射。由于多发射控制器在一个周期需要解码/发射多条指令，我们需要为指令缓存（IB）设计更高的带宽，使其能在一个周期为多发射控制器提供多条指令。

4.2.4　片上互联

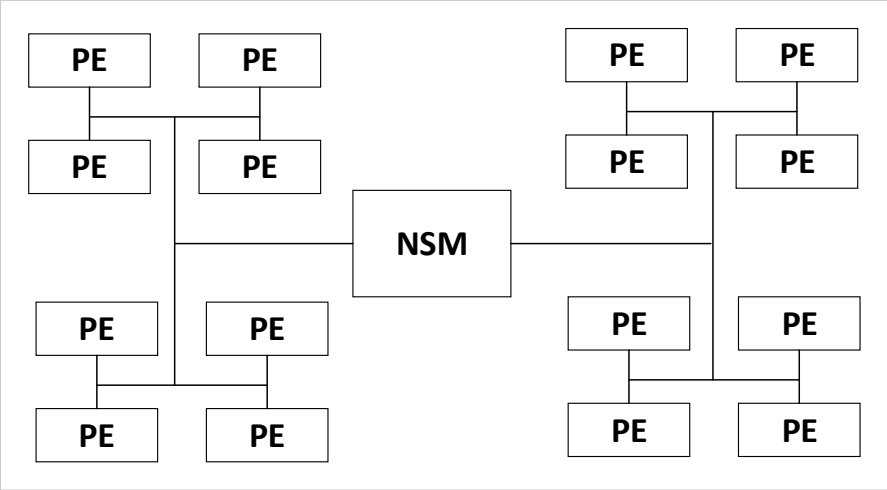


图 4.9　H 树互联结构

如图 4.9所示，我们采用 H 树的拓扑结构连接 NSM 与 NFU 中的 T_n 个 PE。由于 NSM 产生的 selected neurons 和 synapse flags 被所有 PE 共享，而且 PE 之间的负载是一种均衡的状态，因此 NSM 通过 H 树将这些信息通过广播的形式传送给 PE，从而避免 NSM 和 PE 之间由于距离差异引起的不平衡延迟的问题；同时 H 树拥有非常良好的可扩展性，能够非常方便地对 PE 进行扩展，在不改变整体拓扑结构的情况下通过增加 PE 的数量增加加速器的计算能力，从而应对更大规模的神经网络。

4.3　编程模型

4.3.1　基于库的编程模型

考虑到实际应用场景和开发效率，高级编程模型，如 Caffe [58]，Tensorflow [59]，MXNet [60] 等为用户提供了一系列 C++ 的库函数接口，使得用户能够方便地调用这些接口，从而控制 CPU 或者 GPU 完成神经网络的运算。高级编程模型能够大幅度增加用户或者开发人员的编程效率，特别是算法和应用程序开发人员。因此，专用的加速器必须为用户提供类似于高级编程框架的接口，以减轻用户的编程负担。为此，我们提供了基于库的编程模型，其基本思想是为用户提供一系列 C++ 的高级库函数接口，每一个函数对应了神经网络中一个基本操作，如卷积运算，内积，池化等操作。

代码 4.1是卷积层库函数的接口，卷积库接口分为两部分，分别是数据定义和操作定义。用户首先通过 TensorDescriptor 和 FilterDescriptor 定义卷积操作涉及输入/输出和权值; 同时分别使用 setTensorDescriptor 和 setFilterDescriptor 分别描述输入/输出和权值的特性，如稀疏，大小，地址等。然后用户通过 ConvDescriptor 和 setConvDescriptor 分别定义卷积操作描述符和卷积操作的特性，如输入/输入/权值规模，步长，padding 等。最后用户调用 convForward 执行卷积操作。用户通过调用这个库函数接口能够非常方便地最终驱动加速器完成卷积操作。

```
1  // Data Declaration
2  TensorDescriptor input, output;
3  FilterDescriptor weight;
4  setTensorDescriptor (input, ...);
5  setTensorDescriptor (output, ...);
6  setFilterDescriptor (weight, ...);
7  // Operations
8  ConvDescriptor conv;
9  setConvDescriptor (...);
10 ConvForward(conv, input, weight, output);
```

代码 4.1　卷积层库函数接口

我们将库嵌入到深度学习框架 Caffe 中，如图 4.10所示，这一套编程框架对用户完全透明，使得用户能够无缝使用加速器。值得注意的是我们需要为 Cambricon-S 设计专用的编译器（compiler），使其能够将 C++ 源码编译成为加速器中的可执行指令。

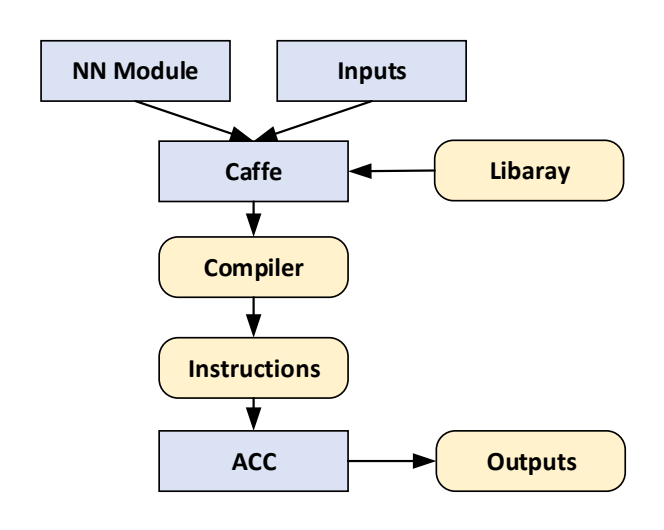


图 4.10　编程框架

4.3.2　编译器

深度学习框架（如 caffe）与加速器之间存在巨大的鸿沟，如何为加速器建立一套完整的编程生态系统是一个巨大的挑战。为此我们需要为加速器设计专用的基于库的编译器，从而将基于库的源文件编译成为加速器中可执行指令。编译的主要挑战来源于加速器中复杂的片上数据管理。

具体来说，第一，神经网络中的数据多样性和特征多样性增加了数据分配和数据搬运的复杂度。由于神经网络中有不同类型的参数，如输入/输出神经元，权值等；同时不同的数据拥有不同的特征，如神经元有动态稀疏的特性，权值有静态稀疏和量化的特性。如何根据数据的特征为不同的数据种类分配数据空间，如何搬运不同种类的数据成为一个巨大挑战，因此，我们为不同的数据使用适当的数据分配和数据搬运策略来提升效率。

第二，有限的片上缓存增大了数据调度的复杂性。考虑到神经网络中庞大的神经元/突触的和加速器中有限的片上内存之间的矛盾，我们需要使用 loop tiling 对数据进行切分，同时使用合适的数据重用策略提升加速器的性能。而不同的 loop tiling 策略和数据重用 (data reuse) 策略会进一步影响片外数据访问，片上数据访问，计算调度，从而影响加速器的效率和性能。因此我们需要在编译期间探索不同的 loop tiling 策略和数据复用策略，选择最优的 loop tiling 方式和最佳的数据重用策略，最大程度挖掘加速器的性能。

我们为加速器设计了一个专用的编译器来弥合深度学习框架与加速器之间的差距。如图 4.11所示，我们显示了如何将基于库的源文件编译成为加速器的可执行代码。编译过程分为两个阶段：数据摆放（data placement）和指令生成（instruction generation）。

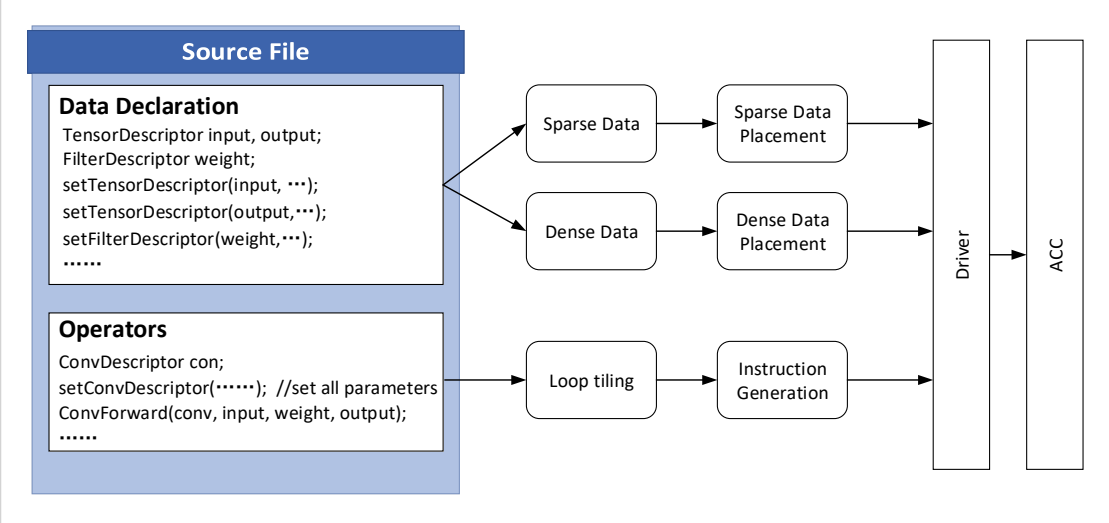


图 4.11　源文件编译成为加速器可执行代码的过程

在数据摆放阶段，源代码中的数据结构被合理地分配到加速器的片上缓存中。我们使用张量（tensor）对输入和输出神经元进行封装，同时我们用过滤器（filter）对权值进行封装，值得注意的是，tensor 和 filter 都是四维的张量。我们使用不同的数据结构封装不同数据类型，是因为不同类型的数据会被分配到加速器不同的片上缓存中（如输入神经元，输出神经元和权值会分别被分配到 NBin，NBout 和 SB 中）。同时考虑到数据稀疏的特性，我们使用 sparse 这个枚举类型作为描述符，并将其添加到 tensor（动态神经元稀疏）和 filter（静态权值稀疏）的属性中。

在指令生成阶段，考虑到有限的片上缓存，我们需要使用 loop tiling 将运算操作（包括卷积操作，池化操作等）分解为子操作，以确保子操作对应的数据能够（即输入/输出神经元，权值）能够填充到有限的片上缓存中。值得注意的是，运算操作对应的 IO 操作和访存操作会被同时生成。最终指令生成器将这些操作转换为对应的运算指令，IO 指令和访存指令。更进一步的，指令生成器会对生成的指令进行静态依赖分析，提取出没有依赖关系的指令并进行标记，在这个过程中，指令生成器主要关注 IO 指令与计算指令之间的依赖关系，Load/Store 指令之间的依赖关系，以及不同计算单元对应的计算指令之间的依赖关系。最终，指令生成器生成加速器可以解码和执行的指令，同时没有依赖关系的指令可以同时发射执行。

4.3.3　loop tiling

Loop tiling 能够对计算进行合理划分，以确保子计算所涉及的数据能够被完全填充到片上缓存中。由于片外访存会花费巨大的能耗（占用超过 90% 的整体能耗），因此 loop tiling 策略优先考虑如何提高片上数据重用性，从而减少片外

数据访存，从而减少加速器的访存能耗。Eyeriss [83] 提出了四种不同的数据重用策略，分别是卷积重用（*convolutional reuse*），卷积核重用（*filter reuse*），输入特征图重用（*input feature map reuse*）和部分和重用（*partial sums reuse*）。考虑到 Cambricon-S 与 Eyeriss 架构上的差异，我们将数据重用策略分为三种类型，即输出重用（*output-reuse*，与 Eyeriss 中的 *partial sums reuse* 对应），突触重用（*synapse-reuse*，与 Eyeriss 中的 *convolutional reuse* 和 *filter reuse* 对应）和输入重用（*input-reuse*，与 Eyeriss 中 *input feature map resue* 对应）；这三种策略分别表示最大程度重用输出部分和数据，突触数据和输入数据。对于 *output-reuse* 策略，在运算过程中，一段输入神经元被加载到 NBin 中，一段权值被加载到 SB 中，然后计算出一段输出神经元的部分和，并将其存储到 NBout 中；在下一个阶段，第二段的输入神经元和权值被加载到片上缓存持续计算 NBout 中输出神经元部分和，这个过程一直循环直至 NBout 中的这一段输出神经元被完全计算完成，最后 NBout 中的输出神经元被存储到片外；因此在 *output-reuse* 策略中，输入神经元和权值会多次从片外加载到片上缓存，而输出神经元部分和会持续存储在 NBout 中直至被完全计算完成。同理，在 *synapse-resue* 策略中，输入神经元和输出神经元部分和会被多次从片外加载到片上，直至片上的权值被完全重用。在 *input-reuse* 策略中，权值和输出神经元部分和会被多次从片外加载到片上，直至片上的输入神经元被完全重用。因此我们在编译阶段需要探索最佳的数据重用策略，减少片外访存的能耗。

不失一般性的，我们以 VGG16 网络的 *conv4_2* 卷积层为例，来计算不同的数据重用策略对片外访存的需求。具体来说，*conv4_2* 的输入规模为 $30\times30\times512$（考虑到输入需要进行 padding），权值的规模为 $32\times32\times512\times512$，输出的规模为 $28\times28\times512$，其中输入神经元和权值的稠密度分别是 40.52%（稀疏度为 59.48%）和 34.20%（稀疏度为 65.80%）（如表 3.2所示），权值和神经元均为 16 比特定点。加速器中 NBin，SB 和 NBout 的大小分别是 $8KB$，$32KB$ 和 $8KB$。在 loop tiling 过程中，我们将数据且分为 $SegN$ 块，其中每一块数据量为 $SegSize$。为了尽可能提高片上缓存的利用率，我们将输出的 channel 维度按照 4×128（$SegN\times SegSize$）的方式进行切分，在 height 维度按照 28×1 的方式进行切分；同时我们将输入的 channel 方向按照 16×32 的方式进行切分。经过 loop tiling 后，输入，权值，输出每一个数据块的大小分别为 $SegSize_{in}=30\times3\times32\times40.52\%$，$SegSize_{syn}=3\times3\times32\times128\times34.20\%$ 和 $SegSize_{out}=28\times1\times128$。值得注意的是，我们对输入数据块和权值数据块分别乘以网络的稠密度来进行近似。因此，使用 *input-reuse*，*output-reuse* 和 *synapse-reuse* 策略，所需要的片外访存的开销可以按照如下公式进行计算：

$$MEM_i = 16 \times 1 \times 28 \times (SegSize_{in} + SegSize_{syn} \times 4 + SegSize_{out} \times 4 \times 2) \times 2 = 68.59MB$$

$$MEM_o = 4 \times 28 \times 1 \times (SegSize_{out} + SegSize_{in} \times 16 + SegSize_{syn} \times 16) \times 2 = 47.85MB$$

$$MEM_s = 16 \times 4 \times (SegSize_{syn} + SegSize_{in} \times 28 \times 1 + SegSize_{out} \times 28 \times 2) \times 2 = 30.03MB$$

最终，*input-reuse*，*output-reuse* 和 *synapse-reuse* 策略需要的片外访存量分别是 $69.59MB$、$47.85MB$ 和 $30.03MB$，因此我们在 *conv4_2* 卷积层上选择 *synapse-reuse* 的策略。

第 5 章　针对新型加速器性能模拟器

我们提出的粗粒度稀疏的神经网络加速器 Cambricon-S 能够充分利用粗粒度稀疏神经网络的神经元/索引共享和负载均衡等特性，同时利用动态神经元稀疏和局部量化进一步减少计算量和访存量，从而提升加速器性能，减少能耗。

我们采用模拟的方法对新型加速器进行性能分析。传统的周期精确模拟器由于性能低，速度慢，无法满足我们实际的需求。因此，我们为 Cambricon-S 设计了一个专用性能模拟器，它能够在误差允许范围内快速进行性能模拟。

优化的性能模拟器基于事件驱动进行设计，它能够隐藏加速器中复杂的设计细节，专注于影响加速器性能的因素，从而快速，准确地预测神经网络在新型加速器上的执行时间。

5.1　背景

计算机模拟是指利用计算机软件开发的模拟器对真实世界过程或者系统进行模拟的行为，进而完成故障分析、测试 VLSI 逻辑设计等复杂的任务,从而为研究人员提供设计指导。计算机系统模拟器是指以在一台计算机上模拟另一台指令不兼容或者体系不同的计算机。如今，计算机系统模拟器已经成为了计算机系统结构领域研究中不可或缺的工具。研究人员通过使用模拟器能够用较低的成本和开销，高效地完成对软硬件的配置和观察，进而为硬软件的设计和优化提供指导。随着计算机行业的迅猛发展，计算机功能变得越来越强大的同时，计算机系统也变得越来越复杂，模拟计算机系统的开销也越来越大，因此如何提升模拟器的性能成为一个亟待解决的问题。研究者一般从准确性，事件开销，内存开销，易用性和可扩展性这几个方面考虑模拟器的性能。

研究者一般采用周期精确模拟器进行硬件的性能模拟。周期精确模拟器需要模拟硬件的每一个模块在每一个时钟周期执行操作的各个细节，包括状态机跳转，寄存器更改，流水级操作等，以保持模拟器与硬件之间的一致性。这种精确模拟会导致巨大的资源，能源和时间开销，进而导致周期精确的模拟器无法满足实际的需求。

周期精确的模拟器按照实现方式可以分为两类，一类是时间触发的模拟器，一类是事件触发的模拟器。其中时间触发模拟器以 cycle 为单位进行模拟，每一个 cycle 会触发一次模拟操作；事件触发模拟器以事件为单位进行模拟，事件通常为用户自定义事件，每一个事件会触发一次模拟操作。如图 5.1所示，一个浮点乘法单元需要产生一个浮点乘法操作，该浮点乘法需要三个周期完成。当我们

使用时间触发的模拟器时，总共需要模拟四个周期的浮点乘法操作，尽管第二个和第三个周期内不需要进行任何模拟操作；当我们使用事件触发的模拟器时，只需要模拟两个周期的操作就能完成。由于神经网络加速器涉及到大规模数据的传输和运算，基于事件触发的模拟器更有性能上的优势。

时间触发

时钟周期	事件
1	乘法操作开始
2	无
3	无
4	乘法操作结束
5	无

事件触发

时钟周期	事件
1	乘法操作开始
4	乘法操作结束

图 5.1　时间触发模拟器和事件触发模拟器

5.2　加速器专用性能模拟器

我们采用基于事件触发的模拟器对 Cambricon-S 进行性能模拟。在设计性能模拟器的过程中，我们采用了如下的策略优化模拟器，在误差允许范围内加快模拟速度，提高模拟器的性能：第一，我们对加速器进行高层次抽象，隐藏加速器中复杂的细节，只关注影响性能的最主要因素，从而简化性能模拟器的设计，加快性能模拟器的模拟速度。第二，我们精简了模拟器中的事件，提取出最重要的事件构建神经网络的执行过程，同时我们使用建模的方法计算事件的执行时间，提高模拟器的准确性。第三，我们根据 loop tiling 将神经网络的执行过程分为多个子运算，然后我们以子运算为单位进行性能模拟，最终通过累加各个子运算的执行事件获得神经网络在加速器上的运行时间。

5.2.1　加速器的高层次抽象

为了简化性能模拟器的设计，加快性能模拟器的模拟速度，我们首先需要对新型加速器进行高层次的抽象，隐藏新型加速器的细节，提取出影响神经网络

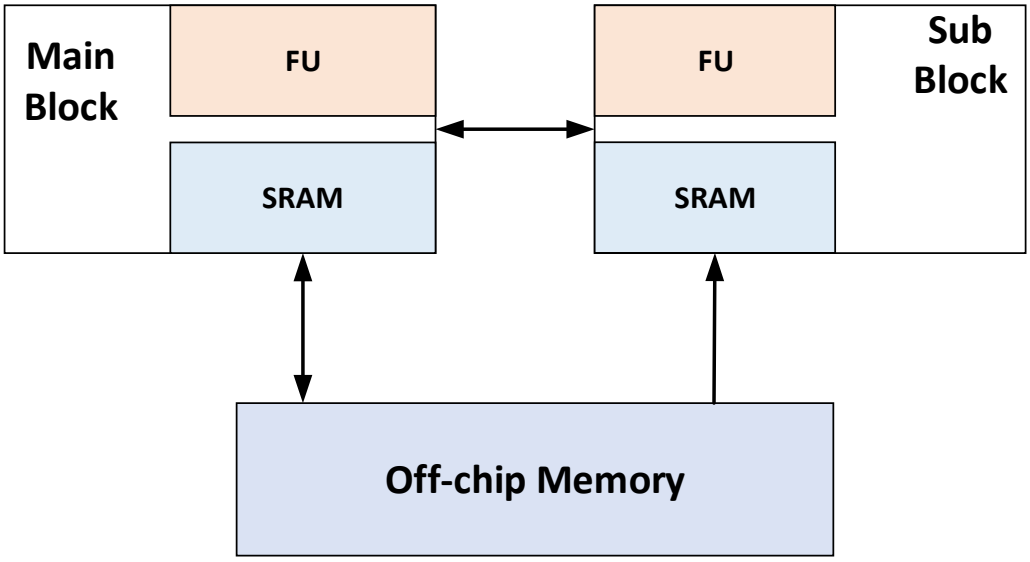


图 5.2　加速器的抽象结构

执行速度的关键因素。由于新型加速器中的大部分模块内部采取流水（pipeline）的方式执行，而且部分模块之间也完成了流水操作，因此我们可以将这些流水的模块整合为一个大模块，从而隐藏新型加速器的设计细节。

如图 5.2所示，新型加速器的抽象结构由三部分构成，分别是 Main Block，Sub Block 和 Off-chip Memory。下面我们将对这三个部分进行详细介绍。

1. Main Block

Main Block 主要作为数据通路同时，完成一部分向量运算和标量运算。Main Block 由片上缓存 SRAM，运算单元 FU（functional unit）组成。Main Block 中的 SRAM 对应了新型加速器中的四个片上缓存，分别是 NBin，NIBin，NBout 和 SIB。Main Block 中的 FU 主要用来完成向量运算，标量运算和非线性运算。其中向量运算包括逐元素相加，逐元素相乘等运算；标量运算包括标量四则运算；非线性运算主要包括指数，双曲函数等超越函数，用来支持激活函数运算。Main Block 中主要完成 Pooling 层，BN 层，ROI Pooling 层和激活层的运算。

值得注意的是，Main Block 的内部集成了新型加速器中的稀疏处理的模块，如 NSM，SSM，Encoder 等，由于选数逻辑与运算逻辑之间为流水的结构，只要我们能够准确模拟运算模块的性能，是否模拟这些模块对最终的结果没有影响，因此我们可以隐藏这些模块的细节，专注于加速器中运算模块的模拟，因此我们将这部分稀疏处理模块隐藏。

2. Sub Block

Sub Block 主要是核心运算模块，主要完成神经网络算法中的矩阵向量运算。Sub Block 由片上缓存运算 SRAM 和运算单元 FU 构成。Sub Block 中的 SRAM 对应新型加速器中的 SB，FU 则对应了 PEFU。神经网络中的卷积层，全连接层和 LSTM 层的大部分运算都在 Sub Block 的 FU 中完成。

3. Off-chip

Off-chip Memoy 存储了神经网络的参数，包括神经网络的拓扑结构，输入，权值，部分和，输出等。由于加速器的片上缓存有限，加速器无法一次性将神经网络的所有参数存储到片上缓存，因此 Off-chip Memory 与片上缓存之间需要由非常频繁的数据交换。

4. 数据通路

Main Block 与 Sub Block 之间存在数据双向的数据通路，从 Main Block 到 Sub Block 方向的通路用来传输输入神经元，从 Sub Block 到 Main Block 方向的通路用于传输输出神经元部分和。

Main Block 与 Off-chip Memory 之间也存在双向数据通路，其中从 Off-chip Memory 到 Main Block 方向的通路用来加载输入神经元，输入神经元索引，输出神经元部分和和权值索引，从 Main Block 到 Off-chip Memory 方向的通路用来存储输出神经元部分和。

Sub Block 与 Off-chip Memory 之间仅仅存在单向的数据通路，从 Off-chip Memory 到 Sub Bloock 方向的通路用来加载权值。

5.2.2　事件分类

设计基于事件触发的模拟器，我们首先需要根据加速器的特性和数据流定义在加速器在运行神经网络过程涉及到的事件。事件有一个四维的描述符，分别是类型，数据，时刻和时间，其中类型用来描述事件需要完成的操作，包括控制（如分支跳转），访存（如 Load/Store 片外 DDR 或者 Read/Write 片上缓存），运算（如矩阵，向量，标量，逻辑运算等）；数据描述事件涉及到的数据，包括数据地址，数据大小，数据类型等信息；时刻描述了事件出触发的时间点；时间描述了事件的执行时间，执行时间可以通过性能分析（Profiling）或者建模（Modeling）的方式完成。

我们在新型加速器的性能模拟器中定义了四个事件，分别是数据加载事件，运算事件，数据存储事件和同步事件，我们将它们依次简写为 Load 事件，Compute

事件，Store 事件和 Synchronize 事件。

1. Load 事件

Load 事件涉及到将数据从 Off-chip Memory 加载到 Main Block 的 SRAM 和 Sub Block 的 SRAM 中；对应到新型加速器中，就是将输入神经元，输入神经元索引，输出部分和，权值和权值索引分别从片外加载到 NBin，NIBin，NBout，SB 和 SIB 中。

2. Compute 事件

Compute 事件涉及到 Main Block 中的 FU 和 Sub Block 中的 FU 完成神经网络的核心运算，神经网络运算包括矩阵运算，向量运算和标量运算。

其中矩阵运算主要包括矩阵向量运算，这部分是神经网络的核心运算，卷积层，全连接层，LSTM 层中大部分运算都是矩阵向量运算。

向量运算包括向量内积，向量逐元素相乘，向量逐元素相乘，神经网络中的池化层，BN 层，LSTM 层，LRN 层等都是涉及到向量运算。

标量运算主要包括标量的四则运算，Faster-RCNN 中的 ROI pooling 层会涉及到这部分的运算。

因此，Compute 事件还能进一步划分矩阵运算事件（Matrix Compute），向量运算事件（Vector Compute）和标量运算事件（Scalar Compute）。

3. Store 事件

Store 事件涉及到将 Main Block SRAM 中的数据存储到 Off-chip Memory；对应到新型加速器中，就是将输出部分和从 NBout 存储回 Off-chip Memory。值得注意的是 NBin，NIBin，SB 和 SIB 这四个缓存不会涉及到 Store 事件。

4. Synchronize 事件

Synchronize 事件是一个同步事件。当出现 Synchronize 事件时，必须要保证 Synchronize 事件之前其他所有事件完成之后才能执行 Synchronize 事件之后的其他事件，也就是说 Synchronize 事件相当与一个同步信号，用来同步 Synchronize 事件之前的其他事件。

5.2.3　事件执行时间

我们采用建模的方式预测各个事件的执行时间，下面我们将详细介绍 Load 事件，Compute 事件和 Store 事件的建模方法。值得注意的是，由于 Synchronize

事件是一个阻塞事件，用于同步之前所有的事件，因此不需要对其进行建模。

1. 对 Load 事件进行建模

Load 事件的执行时间跟数据量和 Off-Chip Memory 带宽有关，具体可以用如下方式进行计算

$$t_{Load} = t_{start} + Data_{Load}/Bandwidth_{Load} \tag{5.1}$$

其中 t_{Load} 是 Load 事件执行时间，t_{start} 是 Off-chip Memory 的启动时间，$Data_{Load}$ 是 Load 事件涉及到的数据量（包括所有输入神经元，输入神经元索引，输出部分和，权值和权值索引），$Bandwidth_{Load}$ 是 Off-Chip Memory 的 Load 带宽。

2. 对 Compute 事件进行建模

Compute 事件的执行时间与运算量和运算单元数量有关，具体可以用如下方式进行计算

$$t_{Compute} = Data_{Compute}/(FU * u\%) \tag{5.2}$$

其中 $t_{Compute}$ 是 Compute 事件执行时间，$Data_{Compute}$ 是 Compute 事件涉及的运算量，FU 是运算单元的数量，$u\%$ 是运算单元的利用率。

神经网络的神经元稀疏度，权值稀疏度，网络拓扑结构（如卷积层的规模，全连接层的规模），数据阻塞等因素都会影响运算单元的利用率，因此运算单元的利用率是一个实时的值。我们采用建模的方法计算运算单元的利用率进行预测，例如对于神经网络的卷积运算，我们会实测多组不同网络规模配置和稀疏度配置下运算单元利用率，然后根据这些数据对运算单元利用率进行建模，最终获得运算单元利用率与网络配置和稀疏度的定量关系。值得注意的是 Matrix Copmute 事件，Vector Compute 事件和 Scalar 事件的执行时间都可以用上述公式进行计算。

3. 对 Store 事件进行建模

Store 事件的执行时间计算方法与 Load 时间类似，执行时间跟数据量和 Off-Chip Memory 带宽有关，具体可以用如下方式进行计算

$$t_{Store} = t_{start} + Data_{Store}/Bandwidth_{Store} \tag{5.3}$$

其中 t_{Store} 是 Store 事件执行事件，t_{start} 是 Off-chip Memory 的启动时间，$Data_{Store}$ 是事件涉及到的数据量（输出部分和），$Bandwidth_{Store}$ 是 Off-Chip Memory 的 Store 带宽。

5.2.4　事件的依赖关系

由于 Cambricon-S 中集成了多发射控制器 4.2.3，因此可以同时发射没有依赖关系的指令。在 Cambricon-S 中，没有依赖关系的指令包括三个方面，第一，IO 指令与计算指令之间没有依赖关系；第二，Load 和 Store 指令之间没有依赖关系；第三，如果计算指令涉及到涉及不同的计算单元和数据，那么这些计算指令之间也没有依赖关系。

对应的，性能模拟器也存在没有依赖关系的事件：第一，Load/Store 事件与 Compute 事件之间没有依赖关系；第二，Load 事件与 Store 事件之间没有依赖关系；第三，Matrix Compute 事件，Vector Compute 事件和 Scalar 事件之间没有依赖关系。没有依赖关系的事件可以同时执行，我们用 Synchronize 事件来同步没有依赖关系的事件。

5.2.5　模拟过程

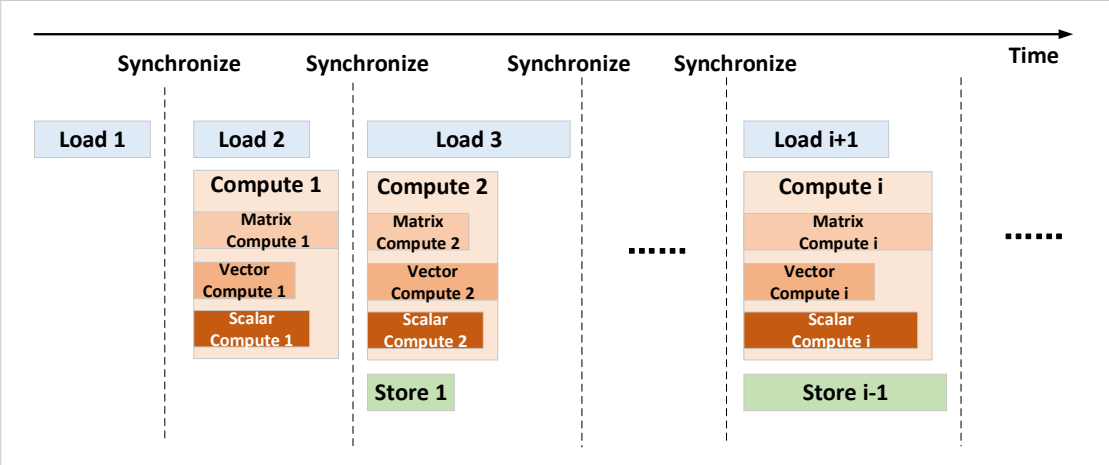


图 5.3　性能模拟器模拟神经网络在加速器上的执行过程

我们使用 loop tiling 的策略将神经网络运算分割成为多个子运算操作，使得每一个子运算对应的数据能够被完整加载到片上缓存中。假设神经网络某一层的运算被切分成为 *N* 个子运算，同时 *N* 个子运算对应 *N* 个数据块。如图 5.3描绘了模拟器模拟神经网络在加速器上执行，同时统计运行时间的过程。神经网络的执行过程被分为 *N+2* 个 *Step*，两个 *Step* 之间用 *Synchronize* 事件进行分割，最终神经网络的执行时间是这 *N+2* 个 *Step* 的执行事件总和。

Step 1：性能模拟器会触发 *Load 1* 事件。*Load 1* 事件将第一个子运算对应的数据从 Off-chip Memory 加载到 Main Block 和 Sub Block 的 SRAM 中。此时由于片上缓存没有数据，Main Block 和 Sub Block 的 FU 无法进行计算。然后我们使用 *Synchronize* 事件进行同步。*Step 1* 的执行时间就是 *Load 1* 事件执行时间。

Step 2: 性能模拟器触发 *Load 2* 事件和 *Compute 1* 事件。其中 *Load 2* 事件将

第二个子运算对应的数据从 Off-chip Memory 加载到 Main Block 和 Sub Block 的 SRAM 中。此时由于片上缓存已经存储了第一个子运算对应的数据（经过 *Step 1* 的 *Load 1* 事件），所以模拟器会触发 *Compute 1* 事件，Main Block 和 Sub Block 的 FU 完成第一个子运算。值得注意的是，*Compute 1* 事件包括 *Matrix Compute 1* 事件，*Vector Compute 1* 事件和 *Scalar Compute 1* 事件这三个没有依赖关系的事件。最终 *Step 2* 的执行时间取决于 *Load 1*, *Matrix Compute 1*，*Vector Compute 1* 和 *Scalar Compute 1* 这四个事件的执行时间的最大值。在图 5.3的示例中，*Step 2* 的执行事件取决于执行最慢的 *Matrix Compute 1* 事件。为了表示方便，我们将 *Computen i* 事件执行时间定义为 *Matrix Compute i*，*Vector Compute i* 和 *Scalar Compute i* 这三个事件中执行时间的最大值。

Step 3: 性能模拟器触发 *Load 3* 事件，*Compute 2* 事件和 *Store 1* 事件。其中 *Load 3* 事件将第三个子运算对应的数据从 Off-chip Memory 加载到 Main Block 和 Sub Block 的 SRAM 中。*Compute 2* 事件开启 Main Block 和 Sub Block 的 FU 完成第二个子运算。此时，由于 Main Block 的 SRAM 中存储了第一个子运算对应的结果（经过 *Step 2* 的 *Compute 1* 事件），模拟器触发 *Store 1* 事件，将第一个子运算的运算结果存储到 Off-Chip Memory。最终 *Step 3* 的执行时间是 *Load 3*, *Compute 2* 和 *Store 1* 这三个事件执行时间的最大值。

.....

Step i+1：性能模拟器触发 *Load i+1* 事件，*Compute i* 事件和 *Store i-1* 事件，它们分别将第 *i+1* 个子运算对应的数据从 Off-chip Memory 加载到片上缓存，计算第 *i* 个子运算，将第 *i-1* 个数据块从片上缓存存储到 Off-chip Memory 中。*Step i+1* 的执行时间是 *Load i+1*, *Compute i* 和 *Store i-1* 这三个事件执行时间的最大值。

.....

Step N：性能模拟器触发 *Load N* 事件，*Compute N-1* 事件和 *Store N-2* 事件。它们分别将第 *N* 个子运算对应的数据从 Off-chip Memory 加载到片上缓存，计算第 *N-1* 个子运算，将第 *N-2* 个数据块从片上缓存存储到 Off-chip Memory 中。从 *Step 1* 到 *Step N*，性能模拟器将神经网络的 N 个子运算涉及的数据加载到片上缓存。

Step N+1：性能模拟器触发 *Compute N* 事件和 *Store N-1* 事件。它们分别计算第 *N* 个子运算，将第 *N-1* 个数据块从片上缓存存储到 Off-chip Memory。从 *Step 2* 到 *Step N+1*，性能模拟器完成了神经网络的 N 个子运算的计算任务。

Step N+2: 性能模拟器出发 *Store N* 事件将第 N 个子运算的计算结果从片上缓存存储到 Off-chip Memory。从 *Step 3* 到 *N+2*，性能模拟器将神经网络 N 个子运算的计算结果存储到 Off-chip Memory。至此，性能模拟器完成了神经网络的所有运算。

5.3　优化模拟器的性能和误差

我们为 Cambricon-S 设计两个性能模拟器，分别是周期精确的模拟器（记作 *simulator_a*）和经过优化的模拟器（记作 *Simulator_o*）。我们在七个 benchmark（如表 3.3所示）分别比较两种模拟器的性能和误差，我们以神经网络在周期精确模拟器上运行时间作为 baseline。两个性能模拟器均运行在英特尔志强 E5-2620 v2，CPU 工作频率为 2.1GHz，工艺为 22nm。

如图 5.4，我们比较了两种模拟器的性能和误差，我们将所有性能归一化和误差到周期精确模拟器上。实验显示，优化的模拟器的性能是周期精确模拟器 41.41 倍，但是误差小于 3%。实验结果，优化后的模拟器能够在误差允许范围内大大提高模拟性能的速度。

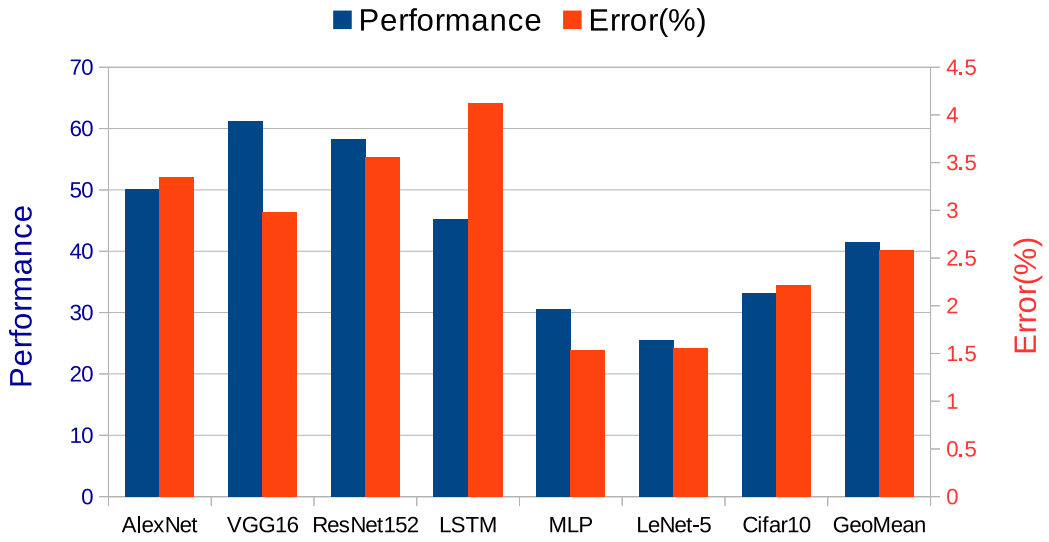


图 5.4　优化的模拟器与周期精确模拟器的对比

第 6 章　实验方法和实验结果

我们在七个 benchmark 上分别比较了 CPU，GPU，DianNao，Cambricon-X 和 Cambricon-S 的性能和能耗，我们发现 Cambricon-S 具有高性能和低能耗的特点。与最先进的稀疏的神经网络加速器 Cambricon-X 相比，Cambricon-S 能够获得 1.71 倍的加速比，同时减少 1.75 倍的能耗。在 65nm 工艺下，新型加速器面积和功耗仅仅为 $6.82mm^2$ 和 $821.19mW$。

6.1　实验方法

在本节中, 我们介绍 Cambricon-S 相关的实验方法。

我们使用 Synopsys 工具链中的 TMSC 65nm 库对加速器的 RTL 实现进行综合。我们使用 CACTI 6.0 预测 DRAM 访存能耗 [131]。我们使用基于 PrimeTime PX 的 VCD 波形文件评估能耗。最后我们使用基于事件触发的模拟器评估加速器的性能。

6.1.1　Baseline

我们将 Cambricon-S 与 CPU，GPU 和硬件加速器进行比较。

1. CPU

我们使用目前流行的深度学习框架 Caffe [58] 评估神经网络在 CPU 上的性能。CPU 的型号是拥有 6 核的英特尔志强 E5-2620 v2，工作频率为 $2.1GHz$，工艺为 $22nm$。同时，我们使用稀疏库（sparse-BLAS） [132] 来评估稀疏神经网络的性能。我们用 CPU-Caffe 和 CPU-Sparse 分别表示稠密网络和稀疏网络在 CPU 上的性能。

2. GPU

我们使用 Caffe 评估神经网络在 GPU 上的性能（用 GPU-Caffe 表示）。GPU 的型号是 Nvidia K20，拥有 $5GB$ GDDR5，峰值能够达到 $3.52TFlops$，工艺为 $28nm$。同时我们使用 cuBLAS 来实现神经网络算法（用 GPU-cuBLAS 表示）。最后, 我们使用先进的 cuSparse 库评估稀疏神经网络的性能（用 GPU-Sparse 表示）。

3. 硬件加速器

我们将新型加速器与目前最先进的神经网络加速器 DianNao 和 Cambricon-X 进行性能比较。DianNao 拥有很高的吞吐量，能够加速大多数的 CNN 和 DNN。Cambricon-X 是一个稀疏神经网络加速器，它能够利用稀疏特性提高性能同时降低能耗。我们选择 Cambricon-X 作为 baseline，主要是考虑到它能够最大程度利用稀疏特性，并且具有通用性。考虑到稀疏利用率，对比稠密神经网络，Cambricon-X 通过挖掘稀疏性能够获得 2.93 倍的加速比，而 Cnvlutin 和 SCNN 分别只能获得 1.37 倍和 2.7 倍的加速比。考虑到通用性，SCNN 在全连接层上的性能非常低，而 EIE 只能用来加速稀疏的全连接层。

6.1.2　Benchmark

如表 3.3所示，我们使用七个代表神经网络:AlexNet, VGG16, LeNet-5，MLP，Cifar10，ResNet152 和 LSTM 作为 benchmark。值得注意的是，我们表中参数都是经过粗粒度剪枝后的网络参数。

6.2　实验结果

6.2.1　硬件属性

在当前的加速器中，为了兼容剪枝块的大小，我们将 PE 的数量 T_n 以及每个 PE 内部乘法器数量 T_m 配置为 $T_n = T_m = 16$，同时考虑到神经网络神经元和权值的稀疏度，我们将 NSM 设计为 256 选 16 的结构，SSM 设计为 64 选 16 的结构，Encoder 设计为 64 选 16 的结构。新型加速器的布局特点，各模块的面积和功耗如表 6.1所示。加速器的总面积和总功耗分别是 $6.82mm^2$ 和 $821.19mW$，工作频率为 1GHz，吞吐量为 512GOP/s，片上 SRAM 共为 $54KB$。新型加速器的面积分别是 Cambricon-X（$6.38mm^2$）和 DianNao（$3.02mm^2$）的 1.07 倍和 2.26 倍，同时功耗比 Cambricon-X（$954mW$）低 $132.81mW$，比 DianNao（$485mW$）高 $336.19mW$。值得注意的是，我们当前加速器并不包含熵解码模块，因此不支持经过熵编码后的神经网络。

此外，Cambricon-S 中处理稀疏和量化模块（NSM, SSMs, Encoder 和 WDM）的面积和功耗分别是 $2.52mm^2$（占总面积的 36.95%）和 $210.26mW$（占总功耗的 25.60%），却能够获得比 Cambricon-X 高的 1.71 倍性能，同时降低 1.75 倍的能耗。值得注意的是，即使新增了突触选择模块，新型加速器的索引模块（即 NSM 和 SSM）的面积对比与 Cambricon-X 的 IM 模块减少了 2.11 倍（$0.94mm^2$ vs. $1.98mm^2$），功耗减少了 1.87 倍（$178.26mW$ vs. $332.62mW$）；同时新型加速器的

表 6.1　加速器详细属性

	Area(mm^2)	%	Power(mW)	%
Total	6.82	100.00%	821.19	100.00%
NBin	0.55	8.06	93.32	11.36
NBout	0.55	8.06	93.32	11.36
SIB	0.05	0.73	6.89	0.84
NIB	0.05	0.73	6.89	0.84
CP	0.16	2.35	75.06	9.14
NSM	0.69	10.12	121.46	14.79
Encoder	0.04	0.60	15.75	1.92
NFU	4.73	69.35	408.50	49.75
SB	1.05	22.20	151.91	37.19
SSM	0.25	5.29	56.80	13.90
WDM	1.54	32.56	16.25	3.98
PEFU	1.89	33.95	183.54	44.93

NSM 的面积和功耗分别是 Cambricon-X 中 IM 模块的 36.52% 和 34.85%，却实现了与 IM 模块相同的功能（即筛选神经元）。因此，我们的设计对比于 Cambricon-X 在面积，能耗上都有很大的提升。

6.2.2　性能

在表 3.3所列出的七个 benchmark 上，我们比较了 Cambricon-S，CPU，GPU，DianNao 和 Cambricon-X 的性能。在 CPU 和 GPU 上，我们同时评估稀疏神经网络和稠密神经网络的性能，即我们使用稀疏库（CPU-Sparse，GPU-Sparse）评估稀疏神经网络的性能，使用稠密库（CPU-Caffe，GPU-Caffe，GPU-cuBLAS）评估稠密神经网络的性能。为了与 CPU 和 GPU 公平比较，我们评估了加速器在稠密网络上的性能（ACC-dense）。同时为了与 Cambricon-X 和 DianNao 公平比较，我们评估了这三个加速器在稀疏神经网络上的性能。

在图 6.1，我们比较了 Cambricon-S，CPU，GPU，DianNao 和 Cambricon-X 的性能，同时，我们将所有性能的数据归一化到了 Cambricon-S 在稀疏网络上的性能。在稠密网络上，Cambricon-S 对比与 CPU-Caffe，GPU-Caffe 和 CPU-cuBLAS 分别能够获得 44.8 倍，5.8 倍和 5.1 倍的加速比。在稀疏网络上，Cambricon-S 对比与 CPU-Sparse 和 GPU-cuSparse 分别能够获得 331.1 倍和 19.3 倍的加速比。对

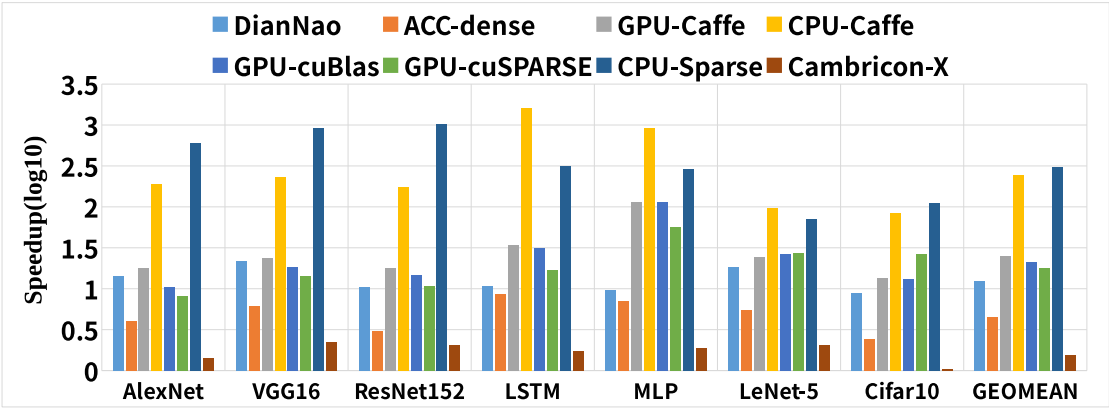


图 6.1　Cambricon-S 与 CPU，GPU，DianNao，Cambricon-X 的性能对比

比于 DianNao 和 Cambricon-X，Cambricon-S 分别能够获得 13.10 倍和 1.71 倍加速的加速比。实验结果充分显示了我们的加速器能够充分利用利用神经网络稀疏的特性，从而获得高加速比。值得注意的是，我们的加速器能够通过关闭/开启各个模块（NSM，SSM，WDM，Encoder 等），使得加速器能够处理稠密神经网络，稀疏神经网络（包括只利用权值稀疏，只利用神经元稀疏和同时利用神经元/权值稀疏三种情况），局部量化。

对比处理稠密网络，Cambricon-S 在处理稀疏神经网络时能够获得 4.32 倍的加速比，这部分的性能提升主要来自于三个方面。首先，NSM 可以充分利用突触稀疏（平均权值稀疏度为 87.99%）从而减少计算，最终实现 2.06 倍的加速比。第二，SSM 能够充分利用神经元稀疏（平均神经元稀疏度为 55.41%），从而减少神经网络所需的计算量，最终获得 1.44 倍的加速比。第三，粗粒度稀疏（78.99%）和局部量化（减少 2.76 倍突触数据）可以大大减少片外权值访问量，从而获得 1.46 倍的加速比

为了进一步探索 Cambricon-S 的性能，我们统计了 Cambricon-S 在卷积层和全连接层上的性能。如图 6.2所示，在卷积层上，对比于 CPU-Sparse，GPU-cuSparse 和 DianNao，我们的加速器分别获得了 283.31 倍，12.20 倍和 13.94 倍的性能提升。如图 6.3所示，在全连接层上，对比于 CPU-Sparse，GPU-cuSparse 和 DianNao，我们的加速器分别获得了 531.89 倍，79.05 倍和 13.56 倍的性能提升。值得注意的是，对比于 Cambricon-X，Cambricon-S 能够在卷积层和全连接层分别获得 1.66 倍和 2.15 倍的加速比。

在卷积层上的性能主要来源于 SSM 模块能够进一步挖掘神经元稀疏，因为卷积层是计算密集型的层，动态神经元稀疏可以大量减少神经网络中的计算量，例如在 AlexNet 网络的 *conv3* 层，稠密情况下需要 $52M$ 的 MAC 操作（Multiply–accumulate operation），而如果利用 45% 的动态神经元稀疏，仅仅需要 $29M$ 的 MAC 操作。

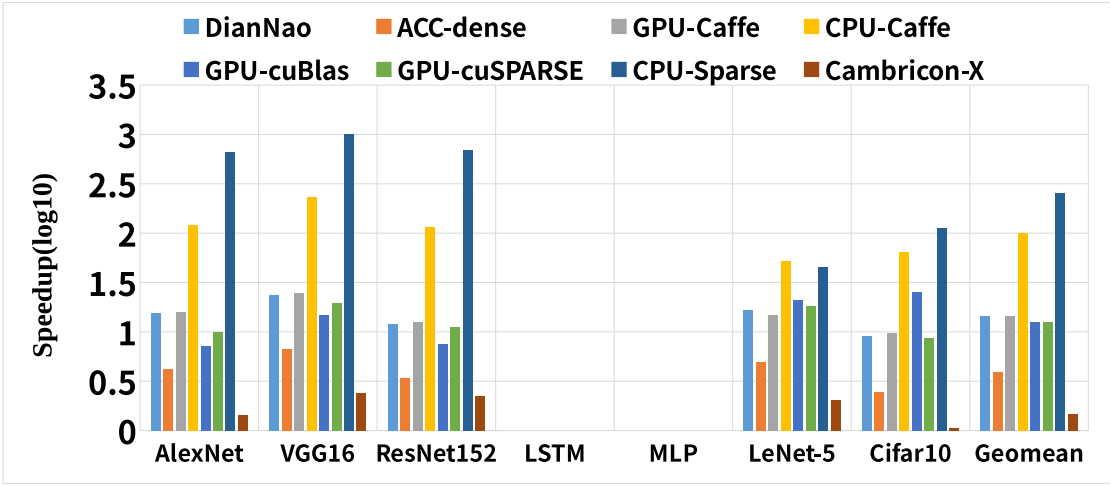


图 6.2　Cambricon-S 与 CPU，GPU，DianNao，Cambricon-X 在卷积层上的性能对比

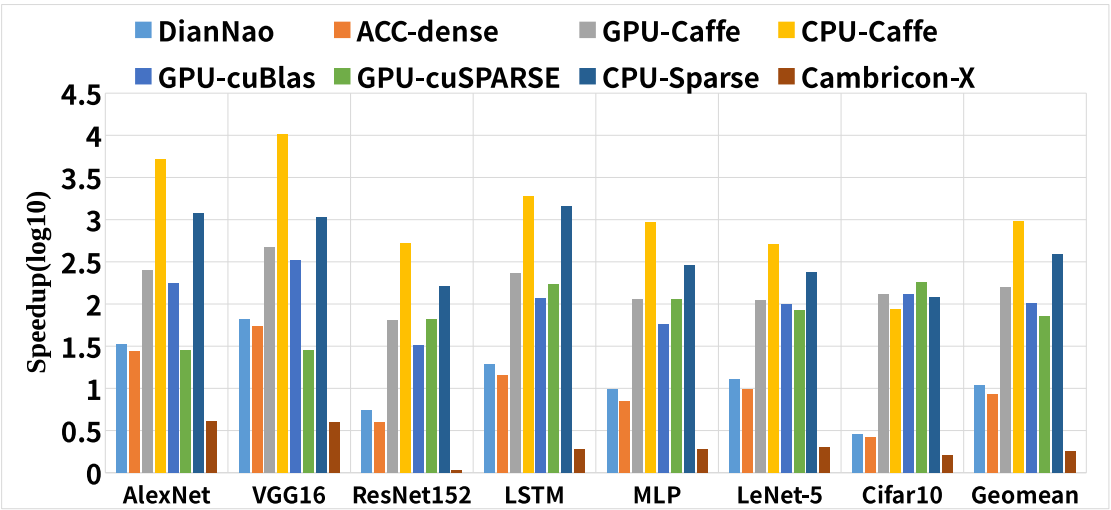


图 6.3　Cambricon-S 与 CPU，GPU，DianNao，Cambricon-X 在全连接层上的性能对比

在全连接层的性能提升主要来源于权值存储量的减少（局部量化）和索引存储量的减少（索引共享）。WDM 能够支持用户自定义比特长度的量化，从而减少突触的存储容量，获得 1.77 倍的加速比；粗粒度剪枝使得非零位置的索引信息能够被相邻多个输出神经元共享，因此能够额外获得 1.21 倍的加速比。

6.2.3　能耗

在七个 benchmark 上，我们比较了 Cambricon-S，GPU，DianNao 和 Cambricon-X 的能耗，其中我们包括了片外访存的能耗。如图 6.4所示，对比于 GPU，DianNao 和 Cambricon-X，Cambricon-S 能够分别减少 63.49 倍，11.72 倍和 1.75 倍的能耗。此外，我们我们观察到局部量化能够减少 1.24 倍的能耗，动态压缩神经元能够减少 1.28 倍的能耗。如果不考虑片外访存的能耗，Cambricon-S 分别比 GPU，DianNao 和 Cambricon-X 减少 1169.51 倍，12.30 倍和 1.75 倍能耗。以上实验数

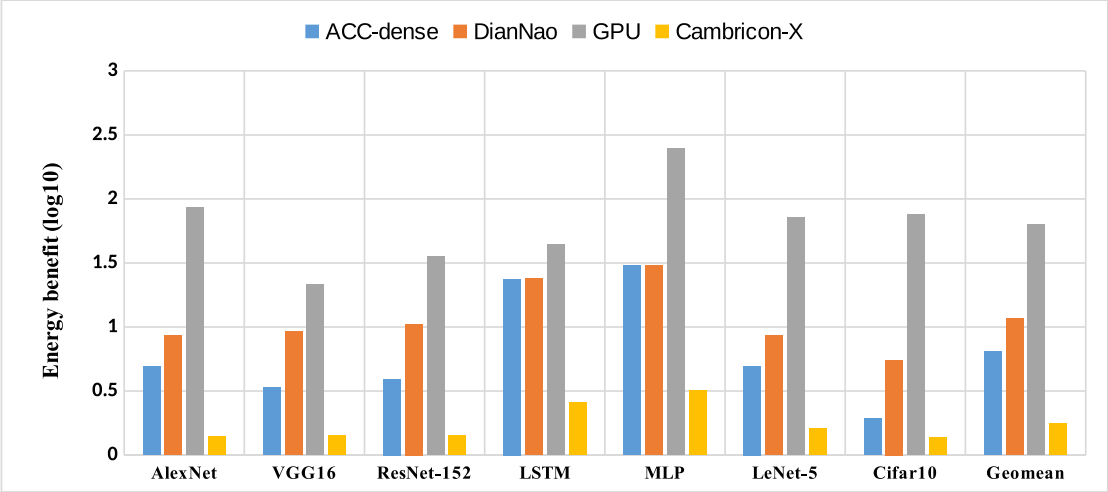


图 6.4　新型加速器与 GPU，DianNao，Cambricon-X 在能耗的对比

据证明我们的加速器能够使用很低的能耗完成神经网络的运算。

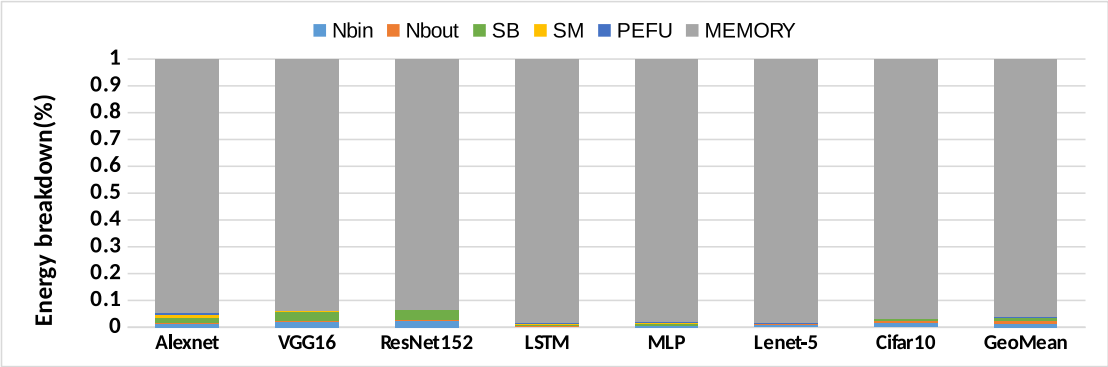


图 6.5　加速器在 benchmark 上的能耗分布（包括片外访存能耗）

我们分析了加速器在七个 benchmark 上的能耗分布。如图 6.5所示，我们可以观察到片外访存消耗了超过 90% 的总能量。在 LSTM 和 MLP 网络中，这个比例高达 98%，远高于其他神经网络，因此这两个网络是访存密集型的网络。实验结果显示，通过稀疏和量化可以显著减少片外访存的能耗。对比于稠密网络，稀疏网络能够减少 72.6% 的片外访存能耗。

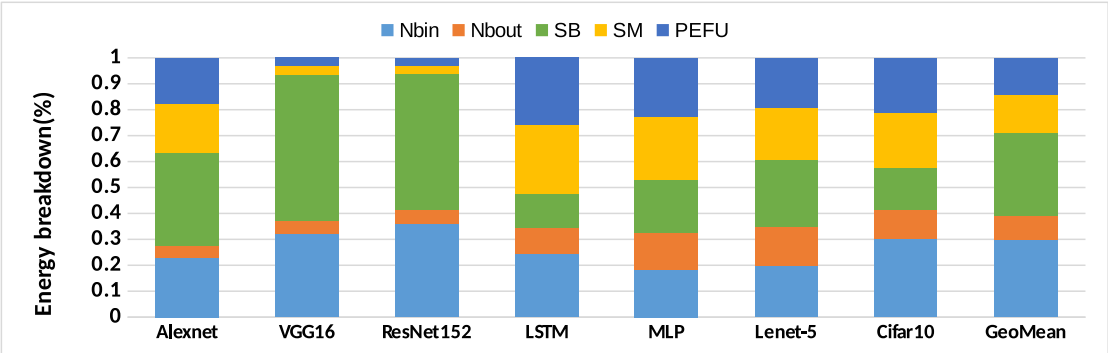


图 6.6　加速器在 benchmark 上的能耗分布（不包括片外访存能耗）

我们进一步分析了在不包括片外访存能耗情况下加速器的能耗分布。如图 6.6所示，片上缓存访问能耗（包括 NBin，NBout 和 SB）平均消耗约 70% 的总能量，这表明片上缓存访问能耗依然主导片上的能耗。对于 LSTM 和 MLP 这两个网络上，片上缓存能耗少于 60%，因为在这两个网络中权值不进行重用。对于深层的卷积网络，比如 VGG16 和 ResNet152，片上内存访问能源的比例超过 90%，因为在这两个网络中需要复杂的 loop tiling 和数据重用策略将大规模运算映射到加速器熵。值得注意的是，与 Cambricon-X 相比，局部量化能够减少 2.76 倍的权值存储量，从而使得 SB 减少 2.48 倍的能耗。

6.2.4　讨论

1. 熵编码和熵解码模块

目前 Cambricon-S 中并没有加入熵解码模块（entropy decoding module）来支持权值熵编码，主要是考虑到熵解码模块需要耗费非常大的面积和能耗，但是仅仅能够获得非常有限的性能提升。一个熵解码模块（entropy decoding module) 的面积为 $6.781 * 10^{-3} mm^2$，它能够在一个 cycle 解码出一个码字。由于熵编码是一种变长编码，因此对应的解码模块必须串行进行解码。即使我们可以将数据划分为许多并行的数据流，然后为每个数据流提供一个熵解码模块，这种方法将会引入巨大的面积和能耗开销。

考虑到每一个 SB 需要在一个 cycle 提供 $T_m \times 4$ 个数据，为了避免性能损失，我们必须为一个 SB 提供 $T_m \times 4$ 个熵解码模块，因此我们在加速其中总共需要集成 $T_n \times T_m \times 4$ 个熵解码模块。在 $T_m = T_n = 16$ 的配置下，我们总共需要 1024 个熵解码模块，这将引入额外 $6.94 mm^2$ 的面积和 $971.37 mW$ 的功耗，因此加速器的总面积和功耗分别是 $13.67 mm^2$ 和 $1769.92 mW$, 分别是原始设计的 2.03 倍和 2.22 倍。然而，新增熵解码的加速器在卷积层上几乎没有性能提升，在全连接层也只有 1.18 倍的性能提升，对比与额外的面积和功耗开销，这种性能提升是非常有限的。因此，我们在加速器中不加入熵解码模块。

2. 稀疏度与性能

我们研究了 Cambricon-S 对神经网络稀疏度的敏感性，即稀疏度对神经网络性能的影响。如图 6.7所示，我们分别探究了神经元稀疏度和粗粒度权值稀疏度对加速器性能的影响，更进一步的，我们从卷积层和全连接层两个角度进行探究。从图中的数据我们观察到了几个有趣的实验结果：

1) 考虑粗粒度权值稀疏，加速器能够在卷积层获得接近理想值的加速比（15.5 倍 vs. 16 倍）。为了兼容大多数网络的权值稀疏度（如表 3.2所示），我们将

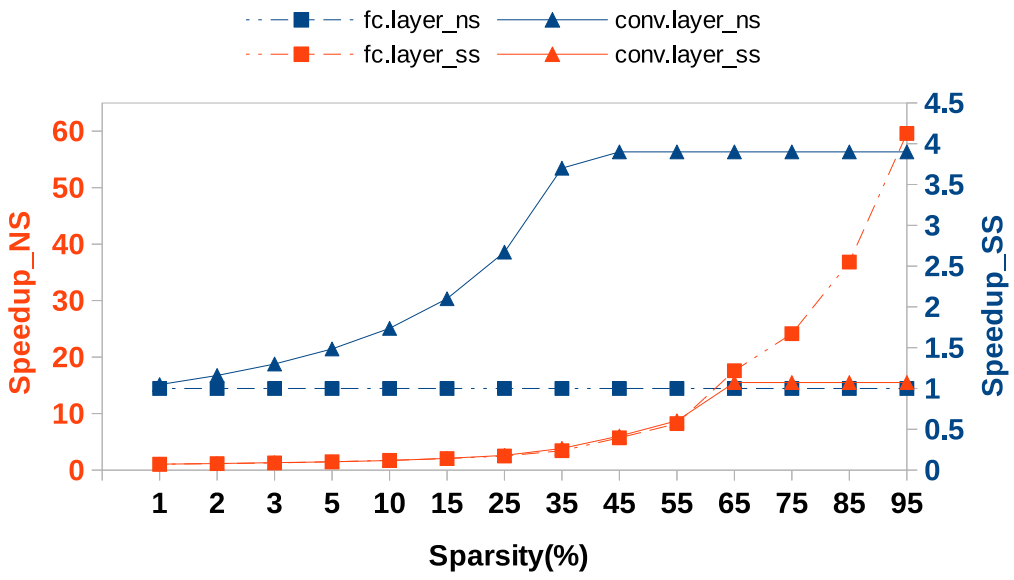


图 6.7　神经网络稀疏度对加速器性能的影响

NSM 设计成为 256 选 16 的结构，即 NSM 最多从 256 个输入神经元中筛选出 16 个需要进行计算的神经元，从而最多实现 16 倍的加速器。我们的加速器能够接近理想加速器的主要原因是我们使用 pingpong 的模式管理片上缓存，使得加速器的片外访存延迟能够被运算时间覆盖。值得注意的是，由于我们的加速器能够充分利用粗粒度稀疏实现负载均衡，因此在相同稀疏度情况下，Cambricon-S 能够获得比 Cambricon-X 更优的加速比。

2）考虑粗粒度权值稀疏，在全连接层中，Cambricon-S 能够很容易地在低稀疏度情况下（低于 5%）获得加速比，并且随着稀疏度的提高，加速比也会不断提高，在稀疏度为 99% 时能够获得 59.59 倍的加速比。主要原因是全连接层是一个访存密集的层，稀疏权值能够大大减少片外访存数据量，从而减少片外访存时间，减少总执行时间。

3）考虑神经元稀疏，Cambricon-S 在卷积层最高能够获得 3.9 倍的加速比，接近理想的 4 倍加速比。由于在大多数情况下，神经元的稠密度高于 25%（稀疏度低于 75%），我们将 SSM 设计成为 64 选 16 的结构，因此最多获得 4 倍的加速比。这个设计能够满足绝大多数神经网络的需求。

4）神经元稀疏并不能在全连接层带来性能的提升，因此在全连接层片外权值访存时间主导了总执行时间（超过 99%），而神经元稀疏并不能有效地减少内存访问。

上述的实验结果进一步验证了我们的加速器能够高效地利用神经网络中神经元稀疏和权值稀疏。

3. 减少的不规则度

减少稀疏神经网络的不规则度能够对加速器的设计，性能产生深远的影响。第一，减少不规则度能够简化加速器的设计。我们可以使用一个共享的 NSM 取代 T_n 个分布式的 NSM，从而节省 $10.35mm^2$ 的面积和 $1821.9mW$ 的功耗；同时我们可以用一个共享的 SIB 来代替 16 个独立的 SIB，从而节省 $15KB$ 的 SRAM 大小。第二，减少不规则度能够大大减少权值索引的规模（26.83 倍），减少片外访问权值，从而获得 1.06 倍的加速比，减少 1.11 倍的能耗。

4. 类似粗粒度稀疏的方法

最近有不少类似于粗粒度稀疏的研究，例如 SIMD-aware weight pruning [133]，synapse vector elimination [134]，channel reduction 和 filter reduction [135-136]。尽管这些技术可以减少稀疏神经网络的不规则度，从而简化加速器设计，但是通常会导致显著的精度损失 [137]。

Mao et al. [138]探究了一系列结构化稀疏的方法，包括细粒度稀疏（fine-grained sparsity），向量层次稀疏（vector-level sparsity），卷积核层次稀疏（kernel-level sparsity）和过滤器层次稀疏（filter-level sparsity），同时评估这些稀疏方法对神经网络不规则度和精度的影响。值得注意的是，我们提出的粗粒度剪枝方式并不会限制剪枝块的规模，因此是一种普遍性的剪枝方法。通过指定参数，粗粒度剪枝能够达到上述各种形式稀疏的效果。

5. 其他稀疏神经网络加速器

EIE [85] 能够加速稀疏神经网络的全连接层，它采用 CSC 的压缩方式压缩权值。为了能够完全消除片外访存，它使用非常大的片上缓存存储所有的突触。为了使 AlexNet 网络的全连接层权值能够完全存储在片上缓存，EIE 的总面积需要达到 $40.8mm^2$，是我们加速器的面积的 5.07 倍。同时，EIE 只支持固定比特的量化，而新型加速器中的 WDM 能够支持自定义比特的量化，从而权衡神经网络的压缩效果和精度。为了公平地与 EIE 比较性能，我们假设加速器的片上缓存足够大，能够存储全连接层的所有权值，因此我们将加速器的性能聚焦在计算时间。如表 6.2所示，相比于 EIE，新型加速器能够平均获得 1.65 倍的加速比。

我们的加速器与 Cambricon-X [82] 都是用了索引的方式利用神经网络的稀疏性质，但是 Cambricon-S 索引方式与 Cambricon-X 有三个方面不同。首先，Cambricon-S 包含一个共享的索引模块 (NSM)，它能够充分挖掘权值的粗粒度稀疏特性。由于粗粒度稀疏，加速器的各个 PE 共享相同的索引，因此 NSM 筛选出的神经元能够被 PE 共享，从而减少索引模块开销以及 NSM 与 PE 之间的

表 6.2　Cambricon-S 与 EIE 的性能比较 *(microsecond)*.

layer	AlexNet			VGG16			Geomean
	fc6	fc7	fc8	fc6	fc7	fc8	
EIE	30.30	12.20	9.90	34.40	8.70	7.50	–
ACC	18.43	8.19	5.13	25.23	4.12	5.09	–
Speedup	1.64×	1.49×	1.93×	1.36×	2.11×	1.49×	1.65×

带宽需求。第二，Cambricon-S 的 PE 中集成了一个本地的权值索引模块（SSM），从而充分利用神经元稀疏。虽然粗粒度稀疏能够减少稀疏神经网络的不规则性，但是稀疏的神经元依然存在不规则性，例如 ReLU 激励能够使得许多神经元的激励为 “0”，SSM 模块能够最小化这种不规则性的影响。因此新型索引模块能够充分利用神经元稀疏和权值稀疏，而 Cambricon-X 只能利用权值稀疏。第三，Cambricon-S 中集成了 Encoder 模块，动态压缩稀疏的神经元，从而减少片外神经元访问。实验显示，Encoder 模块能够进一步减少 1.28 倍的能耗；更具体地说，它能够在卷积层和全连接层分别减少 1.68 倍和 1.02 倍的能耗。第四，Cambricon-S 的每一个 PE 中集成了 WDM 模块用于解码量化后的权值，从而充分利用局部量化，进一步减少片外权值访问量。

SCNN [87] 能够挖掘权值稀疏和神经元稀疏，对比于稠密神经网络，它能够获得 2.7 倍的加速比，同时减少 2.3 倍的能耗。而我们加速器能够获得 4.32 倍的加速比，减少 6.53 倍的能耗，这充分显示了我们加速器高性能和低能耗的特点。

第 7 章　总结和展望

7.1　本文工作总结

人工神经网络近年来在学术界和工业界获得迅猛发展，目前神经网络已经成为众多领域，包括图像识别、物体检测、语音处理和自然语言处理等获得非常理想的效果。神经网络的功能不断增强的过程中，规模不断扩大，层数不断加深，因此神经网络需要庞大的存储资源，计算资源和能耗完成运算。因此研究者采用剪枝的方法将神经网络稀疏化，从而减少神经网络参数和计算量。但是稀疏神经网络的不规则性会使得 CPU，GPU 和大部分神经网络加速器无法利用稀疏性获得性能提升，尽管有一部分加速器能够支持稀疏特性，但是开销较大，而且效果并不理想。因此本文提出了一种软硬件结合的方法处理稀疏神经网络的不规则性，从算法和架构设计的角度提出多种优化策略，显著提升处理稀疏神经网络的效果。本文的主要创新点体现在一下几个方面：

1. 软硬件结合的方法处理稀疏神经网路的不规则性。本文首次提出采用软硬件结合的方法处理稀疏神经网络不规则性。在软件方面我们提出了一种新的神经网络剪枝方法，即粗粒度剪枝，对神经网络进行粗粒度稀疏，从而减少稀疏神经网络的不规则形；在硬件方面，我们提出了 Cambricon-S 来高效处理粗粒度稀疏和动态神经元稀疏，从而更加有效地利用稀疏性，获得性能的提升。

2. 基于局部收敛的神经网络压缩算法。本文通过大量的实验，观察到了局部收敛的现象，即神经网络的权值不是一种随机分布的情况，而是在训练过程中大的权值会聚集成簇。本文根据观察到的局部收敛现象，提出了一种新的神经网络压缩算法。压缩算法包括三个步骤，分别是粗粒度剪枝，局部量化和熵编码。粗粒度剪枝将神经网络的权值分为多个权值块，当一个权值块符合某个条件时将从网络拓扑中被完全剪除，粗粒度剪枝能够显著减少稀疏的神经网络的不规则度。局部量化将权值分为多个子区域，然后在每一个子区域内分别进行量化，从而降低表示权值的比特数，进一步压缩神经网络。之后，我们采用熵编码对神经网络进行无损压缩。新的神经网络压缩算法不仅能够降低稀疏神经网络的稀疏度，获得非常理想的压缩比，还能降低稀疏神经网络的不规则度。粗粒度剪枝可以将稀疏神经网络不规则性平均减少 20.13 倍。受益于粗粒度稀疏和局部量化，我们新的压缩方法可以将 AlexNet 和 VGG 分别压缩 79 倍和 98 倍，远远高于目前最先进的两个神经网络压缩算法 Deep Compression (35 倍/49 倍) 和 CNNPack (39 倍/46 倍)。

3. 高效处理压缩神经网络的神经网络加速器。本文设计并提出了首个能够

支持粗粒度稀疏神经网络的加速器微结构 Cambricon-S。该微结构的主要的特征是一个共享的神经元选择模块 NSM，用来挖掘粗粒度权值稀疏的特性，即神经元共享和索引信息共享。同时微结构还包括 SSM，Encoder 和 WDM，分别用来处理动态神经元稀疏，动态压缩神经元和解码经过局部量化的权值。新型加速器不仅能够处理普通的稠密神经网络，还能够通过打开/关闭稀疏处理模块支持多种稀疏/量化情况，包括神经元稀疏，粗粒度权值稀疏，神经元/权值同时稀疏情况，局部量化等。新型加速器能够非常高效的利用稀疏和量化，获得非常理想的性能和非常低的能耗。同时，为了减少用户的编程负担，我们为加速器设计了专用的基于库的编程模型。编程模型中集成了加速器专用的编译器，能够将 C++ 高级语言编译成为加速器可执行指令；值得注意的，编译器能够静态分析指令之间的依赖关系，提取出没有依赖关系的指令，使它们能够在加速器中同时进行发射。在 65nm 工艺下，加速器的总面积和总功耗分别是 $6.82mm^2$ 和 $821.19mW$，工作频率为 $1GHz$，吞吐量为 $512GOP/s$，片上 SRAM 共为 $54KB$。实验结果显示，对比目前最先进的稀疏神经网络加速器 Cambricon-X，新型加速器能够获得 1.71 倍的性能提升，同时降低 1.75 倍的能耗。

7.2　未来研究展望

本文提出了软硬件结合的方法处理稀疏神经网络的不规则性。在软件上，粗粒度稀疏减少了神经网络的不规则度，同时新的压缩算法能够深度压缩神经网络，取得非常理想的压缩比。在硬件上，新型加速器能够处理粗粒度稀疏神经网络，相比以前的稀疏神经网络加速器能够进一步提升性能，降低能耗。

未来神经网络处理器的发展主要会聚焦在两个方面，分别是嵌入式系统领域和云端处理领域。

高性能，低功耗的神经网络芯片。针对嵌入式设备中的神经网络应用，未来的工作可以从两个方面入手。一方面，在算法方面，我们可以通过裁剪网络，量化，低精度计算等方法，在精度损失允许的范围内，减少神经网络的参数数量和计算量，从而使得神经网络的规模能够部署到嵌入式设备中。另一个方面，我们需要设计特定的神经网络处理器，使其能够支持多种神经网络优化算法，如稀疏，量化等，从而获得高性能的同时降低能耗。

高吞吐量，低延迟的多核神经网络处理器。随着越来越多的神经网络算法部署到云端，云计算（cloud computing）必须能够保证神经网络运算的服务质量（Quality of Service，QoS）的同时，最小化资源消耗。针对神经网络应用的云服务，未来的工作可以从两个方面进行优化。一方面，在底层硬件方面，我们可以设计高吞吐量，低延迟的多核神经网络处理器，并将其部署到云端来代替传统

的 CPU 和 GPU，为神经网络应用提供服务。设计多核神经网络处理器会涉及到诸多设计选择（design choice）的问题，如单核处理器的框架的选择，片上网络（network on chip, NoC）的设计（包括核之间如何互联，核与存储器之间如何路由等），存储设计问题（如存储器层次划分，存储器容量，片上缓存选择 Cache 的形式还是 Stratch-pad 的形式等）。另一方面，在上层软件支撑方面，我们需要针对神经网络应用的特点和底层硬件的特点，设计针对云端神经网络应用的编程框架。编程框架中的一个核心问题就是调度问题，即如何将神经网络应用部署到底层硬件中，保证应用的 QoS 需求，同时最小化资源的消耗。

参 考 文 献

[1] MCCULLOCH W S, PITTS W. A logical calculus of the ideas immanent in nervous activity [J]. The bulletin of mathematical biophysics, 1943, 5(4): 115-133.

[2] HEBB D O. The organizations of behavior: a neuropsychological theory[M]. Lawrence Erlbaum, 1963.

[3] GERSTNER W, KISTLER W M. Mathematical formulations of hebbian learning[J]. Biological cybernetics, 2002, 87(5-6): 404-415.

[4] HODGKIN A L, HUXLEY A F. A quantitative description of membrane current and its application to conduction and excitation in nerve[J]. The Journal of physiology, 1952, 117 (4): 500-544.

[5] TAYLOR W K. Electrical simulation of some nervous system functional activities[J]. Information theory, 1956, 3: 314-328.

[6] ROSENBLATT F. The perceptron: a probabilistic model for information storage and organization in the brain.[J]. Psychological review, 1958, 65(6): 386.

[7] WIDROW B, HOFF M E. Adaptive switching circuits[R]. Stanford Univ Ca Stanford Electronics Labs, 1960.

[8] FITZHUGH R. Impulses and physiological states in theoretical models of nerve membrane [J]. Biophysical journal, 1961, 1(6): 445-466.

[9] MINSKY M L. Computation: finite and infinite machines[M]. Prentice-Hall, Inc., 1967.

[10] MINSKY M. Paper, s.(1969). perceptrons[Z]. MIT Press, Cambridge, 5.

[11] CAINIELLO E. Outline of a theory of thought-processes and thinking machines[J]. J. Theoretical Biology, 1961, 2: 2204-2235.

[12] NAGUMO J, SATO S. On a response characteristic of a mathematical neuron model[J]. Kybernetik, 1972, 10(3): 155-164.

[13] HOPFIELD J J. Neural networks and physical systems with emergent collective computational abilities[J]. Proceedings of the national academy of sciences, 1982, 79(8): 2554-2558.

[14] ZURADA J M, CLOETE I, VAN DER POEL E. Generalized hopfield networks for associative memories with multi-valued stable states[J]. Neurocomputing, 1996, 13(2-4): 135-149.

[15] HINDMARSH J L, ROSE R. A model of neuronal bursting using three coupled first order differential equations[J]. Proc. R. Soc. Lond. B, 1984, 221(1222): 87-102.

[16] ACKLEY D H, HINTON G E, SEJNOWSKI T J. A learning algorithm for boltzmann machines[J]. Cognitive science, 1985, 9(1): 147-169.

[17] KIRKPATRICK S, GELATT C D, VECCHI M P. Optimization by simulated annealing[J].

science, 1983, 220(4598): 671-680.

[18] ČERNỲ V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm[J]. Journal of optimization theory and applications, 1985, 45(1): 41-51.

[19] RUMELHART D E, HINTON G E, WILLIAMS R J. Learning representations by back-propagating errors[J]. nature, 1986, 323(6088): 533.

[20] MACKAY D J. A practical bayesian framework for backpropagation networks[J]. Neural computation, 1992, 4(3): 448-472.

[21] BISHOP C, BISHOP C M, et al. Neural networks for pattern recognition[M]. Oxford university press, 1995.

[22] WILLIAMS C K, RASMUSSEN C E. Gaussian processes for regression[C]//Advances in neural information processing systems. 1996: 514-520.

[23] VAPNIK V. Statistical learning theory. 1998: volume 3[M]. Wiley, New York, 1998.

[24] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C]//Advances in neural information processing systems. 2012: 1097-1105.

[25] SIMONYAN K, ZISSERMAN A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.

[26] REN S, HE K, GIRSHICK R, et al. Faster r-cnn: Towards real-time object detection with region proposal networks[C]//Advances in neural information processing systems. 2015: 91-99.

[27] HINTON G, DENG L, YU D, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups[J]. IEEE Signal processing magazine, 2012, 29(6): 82-97.

[28] AMODEI D, ANUBHAI R, BATTENBERG E, et al. Deep speech 2: End-to-end speech recognition in english and mandarin[J]. arXiv preprint arXiv:1512.02595, 2015.

[29] ZE H, SENIOR A, SCHUSTER M. Statistical parametric speech synthesis using deep neural networks[C]//Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013: 7962-7966.

[30] CONNEAU A, SCHWENK H, BARRAULT L, et al. Very deep convolutional networks for natural language processing[J]. arXiv preprint arXiv:1606.01781, 2016.

[31] MOYER C. How google’s alphago beat a go world champion[J]. The Atlantic, 2016, 28.

[32] DENG J, DONG W, SOCHER R, et al. Imagenet: A large-scale hierarchical image database [C]//Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. Ieee, 2009: 248-255.

[33] SZEGEDY C, LIU W, JIA Y, et al. Going deeper with convolutions[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 1-9.

[34] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 770-778.

[35] GIRSHICK R, DONAHUE J, DARRELL T, et al. Rich feature hierarchies for accurate object detection and semantic segmentation[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2014: 580-587.

[36] GIRSHICK R. Fast r-cnn[C]//Proceedings of the IEEE international conference on computer vision. 2015: 1440-1448.

[37] HE K, ZHANG X, REN S, et al. Spatial pyramid pooling in deep convolutional networks for visual recognition[C]//European conference on computer vision. Springer, 2014: 346-361.

[38] HE K, GKIOXARI G, DOLLÁR P, et al. Mask r-cnn[C]//Computer Vision (ICCV), 2017 IEEE International Conference on. IEEE, 2017: 2980-2988.

[39] REDMON J, DIVVALA S, GIRSHICK R, et al. You only look once: Unified, real-time object detection[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 779-788.

[40] GUPTA S, AGRAWAL A, GOPALAKRISHNAN K, et al. Deep learning with limited numerical precision[C]//International Conference on Machine Learning. 2015: 1737-1746.

[41] DETTMERS T. 8-bit approximations for parallelism in deep learning[J]. arXiv preprint arXiv:1511.04561, 2015.

[42] COURBARIAUX M, BENGIO Y, DAVID J P. Binaryconnect: Training deep neural networks with binary weights during propagations[C]//Advances in neural information processing systems. 2015: 3123-3131.

[43] RASTEGARI M, ORDONEZ V, REDMON J, et al. Xnor-net: Imagenet classification using binary convolutional neural networks[C]//European Conference on Computer Vision. Springer, 2016: 525-542.

[44] ZHOU A, YAO A, GUO Y, et al. Incremental network quantization: Towards lossless cnns with low-precision weights[J]. arXiv preprint arXiv:1702.03044, 2017.

[45] REED R. Pruning algorithms-a survey[J]. IEEE transactions on Neural Networks, 1993, 4 (5): 740-747.

[46] LECUN Y, DENKER J S, SOLLA S A, et al. Optimal brain damage.[C]//NIPs: volume 2. 1989: 598-605.

[47] MICHE Y, SORJAMAA A, BAS P, et al. Op-elm: optimally pruned extreme learning machine[J]. IEEE transactions on neural networks, 2010, 21(1): 158-162.

[48] HAN S, POOL J, TRAN J, et al. Learning both weights and connections for efficient neural network[C]//Advances in Neural Information Processing Systems. 2015: 1135-1143.

[49] GUO Y, YAO A, CHEN Y. Dynamic network surgery for efficient dnns[C]//Advances In Neural Information Processing Systems. 2016: 1379-1387.

[50] DENTON E L, ZAREMBA W, BRUNA J, et al. Exploiting linear structure within convolutional networks for efficient evaluation[C]//Advances in Neural Information Processing Systems. 2014: 1269-1277.

[51] LEBEDEV V, GANIN Y, RAKHUBA M, et al. Speeding-up convolutional neural networks using fine-tuned cp-decomposition[J]. arXiv preprint arXiv:1412.6553, 2014.

[52] CHAKRADHAR S, SANKARADAS M, JAKKULA V, et al. A dynamically configurable coprocessor for convolutional neural networks[C]//ACM SIGARCH Computer Architecture News: volume 38. ACM, 2010: 247-257.

[53] VANHOUCKE V, SENIOR A, MAO M Z. Improving the speed of neural networks on cpus [C]//Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop: volume 1. Citeseer, 2011: 4.

[54] FARABET C, POULET C, HAN J Y, et al. Cnp: An fpga-based processor for convolutional networks[C]//Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on. IEEE, 2009: 32-37.

[55] SCHERER D, SCHULZ H, BEHNKE S. Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors[C]//International conference on Artificial neural networks. Springer, 2010: 82-91.

[56] CIRESAN D C, MEIER U, MASCI J, et al. Flexible, high performance convolutional neural networks for image classification[C]//IJCAI Proceedings-International Joint Conference on Artificial Intelligence: volume 22. Barcelona, Spain, 2011: 1237.

[57] COATES A, HUVAL B, WANG T, et al. Deep learning with cots hpc systems[C]// Proceedings of The 30th International Conference on Machine Learning. 2013: 1337-1345.

[58] JIA Y, SHELHAMER E, DONAHUE J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014: 675-678.

[59] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: a system for large-scale machine learning.[C]//OSDI: volume 16. 2016: 265-283.

[60] CHEN T, LI M, LI Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[J]. arXiv preprint arXiv:1512.01274, 2015.

[61] DEAN J, CORRADO G, MONGA R, et al. Large scale distributed deep networks[C]// Advances in neural information processing systems. 2012: 1223-1231.

[62] OH K S, JUNG K. Gpu implementation of neural networks[J]. Pattern Recognition, 2004, 37(6): 1311-1314.

[63] LE Q V. Building high-level features using large scale unsupervised learning[C]//Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013: 8595-8598.

[64] FARABET C, MARTINI B, CORDA B, et al. Neuflow: A runtime reconfigurable dataflow processor for vision[C]//Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on. IEEE, 2011: 109-116.

[65] GOKHALE V, JIN J, DUNDAR A, et al. A 240 g-ops/s mobile coprocessor for deep neural networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. 2014: 682-687.

[66] ZHANG C, LI P, SUN G, et al. Optimizing fpga-based accelerator design for deep convolutional neural networks[C]//Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015: 161-170.

[67] SUDA N, CHANDRA V, DASIKA G, et al. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks[C]//Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2016: 16-25.

[68] QIU J, WANG J, YAO S, et al. Going deeper with embedded fpga platform for convolutional neural network[C]//Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2016: 26-35.

[69] RICE K L, TAHA T M, VUTSINAS C N. Scaling analysis of a neocortex inspired cognitive model on the cray xd1[J]. The Journal of Supercomputing, 2009, 47(1): 21-43.

[70] KIM S K, MCAFEE L C, MCMAHON P L, et al. A highly scalable restricted boltzmann machine fpga implementation[C]//Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on. IEEE, 2009: 367-372.

[71] LEE S Y, AGGARWAL J K. Parallel 2-d convolution on a mesh connected array processor [J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 1987(4): 590-594.

[72] STEARNS C C, LUTHI D A, RUETZ A, et al. A reconfigurable 64-tap transversal filter[C]// Custom Integrated Circuits Conference, 1988., Proceedings of the IEEE 1988. IEEE, 1988: 8-8.

[73] KAMP W, KUNEMUND R, SOLDNER H, et al. Programmable 2d linear filter for video applications[J]. IEEE Journal of Solid-State Circuits, 1990, 25(3): 735-740.

[74] HECHT V, RONNER K. An advanced programmable 2d-convolution chip for, real time image processing[C]//Circuits and Systems, 1991., IEEE International Sympoisum on. IEEE,

1991: 1897-1900.

[75] LEE J J, SONG G Y. Super-systolic array for 2d convolution[C]//TENCON 2006. 2006 IEEE Region 10 Conference. IEEE, 2006: 1-4.

[76] CHEN Y, CHEN T, XU Z, et al. Diannao family: energy-efficient hardware accelerators for machine learning[J]. Communications of the ACM, 2016, 59(11): 105-112.

[77] CHEN T, DU Z, SUN N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning[C/OL]//Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS). 2014: 269-284. http://dl.acm.org/citation.cfm?id=2541967.

[78] LIU D, CHEN T, LIU S, et al. Pudiannao: A polyvalent machine learning accelerator[C]// ACM SIGARCH Computer Architecture News: volume 43. ACM, 2015: 369-381.

[79] DU Z, FASTHUBER R, CHEN T, et al. Shidiannao: Shifting vision processing closer to the sensor[C]//ACM SIGARCH Computer Architecture News: volume 43. ACM, 2015: 92-104.

[80] CHEN Y H, EMER J, SZE V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks[C]//ACM SIGARCH Computer Architecture News: volume 44. IEEE Press, 2016: 367-379.

[81] JOUPPI N P, YOUNG C, PATIL N, et al. In-datacenter performance analysis of a tensor processing unit[C]//Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM, 2017: 1-12.

[82] ZHANG S, DU Z, ZHANG L, et al. Cambricon-x: An accelerator for sparse neural networks [C]//Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on. IEEE, 2016: 1-12.

[83] CHEN Y H, KRISHNA T, EMER J S, et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks[J]. IEEE Journal of Solid-State Circuits, 2017, 52(1): 127-138.

[84] ALBERICIO J, JUDD P, HETHERINGTON T, et al. Cnvlutin: Ineffectual-neuron-free deep neural network computing[C]//Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016: 1-13.

[85] HAN S, LIU X, MAO H, et al. Eie: efficient inference engine on compressed deep neural network[C]//Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016: 243-254.

[86] HAN S, KANG J, MAO H, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga[C]//Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2017: 75-84.

[87] ANGSHUMAN P, MINSOO R, ANURAG M, et al. Scnn: An accelerator for compressed-

sparse convolutional neural networks[J]. In 44th International Symposium on Computer Architecture, 2017.

[88] HAN S, MAO H, DALLY W J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding[J]. arXiv preprint arXiv:1510.00149, 2015.

[89] WANG Y, XU C, YOU S, et al. Cnnpack: Packing convolutional neural networks in the frequency domain[C]//Advances In Neural Information Processing Systems. 2016: 253-261.

[90] IOFFE S, SZEGEDY C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. arXiv preprint arXiv:1502.03167, 2015.

[91] BA J L, KIROS J R, HINTON G E. Layer normalization[J]. arXiv preprint arXiv:1607.06450, 2016.

[92] DMITRY U, ANDREA V, VICTOR L. Instance normalization: The missing ingredient for fast stylization[J]. arXiv preprint arXiv:1607.08022, 2016.

[93] WU Y, HE K. Group normalization[J]. arXiv preprint arXiv:1803.08494, 2018.

[94] GOODFELLOW I, BENGIO Y, COURVILLE A, et al. Deep learning: volume 1[M]. MIT press Cambridge, 2016.

[95] KÖSTER U, WEBB T, WANG X, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks[C]//Advances in Neural Information Processing Systems. 2017: 1742-1752.

[96] HU Q, WANG P, CHENG J. From hashing to cnns: Training binaryweight networks via hashing[J]. arXiv preprint arXiv:1802.02733, 2018.

[97] RUSSAKOVSKY O, DENG J, SU H, et al. Imagenet large scale visual recognition challenge [J]. International Journal of Computer Vision, 2015, 115(3): 211-252.

[98] LI F, LIU B. Ternary weight networks.(2016)[J]. arXiv preprint arXiv:1605.04711, 2016.

[99] ZHU C, HAN S, MAO H, et al. Trained ternary quantization[J]. arXiv preprint arXiv:1612.01064, 2016.

[100] WANG P, CHENG J. Fixed-point factorized networks[C]//Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on. IEEE, 2017: 3966-3974.

[101] HUBARA I, COURBARIAUX M, SOUDRY D, et al. Binarized neural networks[C]// Advances in neural information processing systems. 2016: 4107-4115.

[102] ZHOU S, WU Y, NI Z, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients[J]. arXiv preprint arXiv:1606.06160, 2016.

[103] CAI Z, HE X, SUN J, et al. Deep learning with low precision by half-wave gaussian quantization[J]. arXiv preprint arXiv:1702.00953, 2017.

[104] HE T, FAN Y, QIAN Y, et al. Reshaping deep neural network for fast decoding by node-

pruning[C]//Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. IEEE, 2014: 245-249.

[105] SRINIVAS S, BABU R V. Data-free parameter pruning for deep neural networks[J]. arXiv preprint arXiv:1507.06149, 2015.

[106] HU H, PENG R, TAI Y W, et al. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures[J]. arXiv preprint arXiv:1607.03250, 2016.

[107] MARIET Z, SRA S. Diversity networks[J]. arXiv preprint arXiv:1511.05077, 2015.

[108] JADERBERG M, VEDALDI A, ZISSERMAN A. Speeding up convolutional neural networks with low rank expansions[J]. arXiv preprint arXiv:1405.3866, 2014.

[109] CHEN Y, LUO T, LIU S, et al. Dadiannao: A machine-learning supercomputer[C]// Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014: 609-622.

[110] LIU S, DU Z, TAO J, et al. Cambricon: An instruction set architecture for neural networks [C]//ACM SIGARCH Computer Architecture News: volume 44. IEEE Press, 2016: 393-405.

[111] SRIVASTAVA N, HINTON G E, KRIZHEVSKY A, et al. Dropout: a simple way to prevent neural networks from overfitting.[J]. Journal of Machine Learning Research, 2014, 15(1): 1929-1958.

[112] HOLI J L, HWANG J N. Finite precision error analysis of neural network hardware implementations[J]. IEEE Transactions on Computers, 1993, 42(3): 281-290.

[113] VENKATARAMANI S, RANJAN A, ROY K, et al. Axnn: energy-efficient neuromorphic systems using approximate computing[C]//Proceedings of the 2014 international symposium on Low power electronics and design. ACM, 2014: 27-32.

[114] PILLAI P, SHIN K G. Real-time dynamic voltage scaling for low-power embedded operating systems[C]//ACM SIGOPS Operating Systems Review: volume 35. ACM, 2001: 89-102.

[115] OLSHAUSEN B A, FIELD D J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images[J]. Nature, 1996, 381(6583): 607.

[116] BOUREAU Y L, CUN Y L, et al. Sparse feature learning for deep belief networks[C]// Advances in neural information processing systems. 2008: 1185-1192.

[117] LEE H, EKANADHAM C, NG A Y. Sparse deep belief net model for visual area v2[C]// Advances in neural information processing systems. 2008: 873-880.

[118] LEE H, BATTLE A, RAINA R, et al. Efficient sparse coding algorithms[J]. Advances in neural information processing systems, 2007, 19: 801.

[119] HENNEAUX M, TEITELBOIM C. Quantization of gauge systems[M]. Princeton university press, 1992.

[120] MACKAY D J. Information theory, inference and learning algorithms[M]. Cambridge uni-

versity press, 2003.

[121] HUFFMAN D A. A method for the construction of minimum-redundancy codes[J]. Proceedings of the IRE, 1952, 40(9): 1098-1101.

[122] WITTEN I H, NEAL R M, CLEARY J G. Arithmetic coding for data compression[J]. Communications of the ACM, 1987, 30(6): 520-540.

[123] Coded representation of picture and audio information progressive bi-level image compression[C]//ISO/IEC International Standard 11544:ITU-T Rec.T.82. 1993.

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

[125] SRIVASTAVA N, HINTON G E, KRIZHEVSKY A, et al. Dropout : A Simple Way to Prevent Neural Networks from Overfitting[J]. Journal of Machine Learning Research (JMLR), 2014, 15: 1929-1958.

[126] KRIZHEVSKY A. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks[Z]. 2012.

[127] SAK H, SENIOR A W, BEAUFAYS F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling.[C]//Interspeech. 2014: 338-342.

[128] VOLDER J E. The cordic trigonometric computing technique[J]. Electronic Computers Ire Transactions on, 1959, EC-8(3): 330-334.

[129] WALTHER S. A unified algorithm for elementary functions[C]//May 18-20, 1971, Spring Joint Computer Conference. 1971: 379-385.

[130] TEMAM O. A defect-tolerant accelerator for emerging high-performance applications[J]. Acm Sigarch Computer Architecture News, 2012, 40(3): 356-367.

[131] MURALIMANOHAR N, BALASUBRAMONIAN R, JOUPPI N. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0[C]//Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2007: 3-14.

[132] DUFF I S, HEROUX M A, POZO R. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum[J]. ACM Transactions on Mathematical Software (TOMS), 2002, 28(2): 239-267.

[133] YU J, LUKEFAHR A, PALFRAMAN D, et al. Scalpel: Customizing dnn pruning to the underlying hardware parallelism[C]//Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM, 2017: 548-560.

[134] HILL P, JAIN A, HILL M, et al. Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission[C]//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2017: 786-799.

[135] WEN W, WU C, WANG Y, et al. Learning structured sparsity in deep neural networks[C]// Advances in Neural Information Processing Systems. 2016: 2074-2082.

[136] LEBEDEV V, LEMPITSKY V. Fast convnets using group-wise brain damage[C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 2554-2564.

[137] LI H, KADAV A, DURDANOVIC I, et al. Pruning filters for efficient convnets[J]. arXiv preprint arXiv:1608.08710, 2016.

[138] MAO H, HAN S, POOL J, et al. Exploring the regularity of sparse structure in convolutional neural networks[J]. arXiv preprint arXiv:1705.08922, 2017.

致　谢

不知不觉博士生涯就要结束了，在这五年半的时间里，我在许多城市，包括合肥，苏州，北京，上海等进行学习，我遇到了很多帮助我的老师和同学，在这里，我向他们表达最真诚的谢意。

首先要感谢我的导师周学海教授，前两年半的博士生涯，是周老师一步一步带着我进行体系结构的学习，带我进入了计算机体系结构的大门，让我领略了体系结构的魅力。周老师让不管在生活上还是在研究上都充满了大智慧，特别是在研究上，能够在 high-level 的角度一针见血地点出问题的本质。周老师在 2016 年年初将我推荐到计算所陈老师的国重实验室（智能处理器研究中心）进行深入学习，使我能够接触最前沿的科学研究，非常感谢周老师当时的决定。

其次，我要感谢计算所的陈云霁研究员。陈老师学识渊博，平易近人，能够处处为学生着想，为学生创造一个非常良好的学习和工程的环境。我还记得当时和陈老师一起通宵写课题，一起爬山，打桌游，吃火锅的时光。在北京计算所的一年时间里，我学到了许多知识，包括工程方面，如前端设计，前端验证，驱动编写等；还包括科研方面，包括神经网络算法知识，性能分析，加速器架构设计等。

同时我要感谢杜子东老师。在 2017 年，我加入了杜老师的前瞻组，并在接下来的一年半时间里在上海寒武纪分部专心进行科研。杜老师领我进入了科研的大门，帮我培养学术上的思维习惯，教我如何选择 idea，组织文章结构，呈现硬件架构，呈现实验结果，进行性能分析，能耗分析等，从而完成一篇高质量的论文。杜老师为人师表，在科研上孜孜不倦，认真负责，在杜老师的帮助下，最终让我达到了毕业的要求。

另外，我要感谢嵌入式系统实验室的李曦老师，陈香兰老师，王超老师对我的指导。同时还要感谢寒武纪陈天石老师，刘少礼老师，王在老师，喻歆老师，郭琦老师，刘道福老师，罗韬老师，支天老师和软件所的李玲老师等对我生活上，工程上和学术上的帮助。

同时，感谢嵌入式系统实验室的师兄师弟们，特别感谢周金红师兄，余奇，马翔，万波，李俊，陈航，赵勇对我的帮助。感谢在寒武纪和我一起工作的同学们：兰慧盈，周聖元，刘雨辰，韩栋，李震，张士锦等对我的照顾，跟大家一起学习，做工程，做研究真的学到了很多。

最后，感谢我的父母和亲人，成长就是最好的报答。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. **Xuda Zhou**, Zidong Du, Shijin Zhang, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, Yunji Chen: Addressing Sparsity in Deep Neural Networks. TCAD

2. **Xuda Zhou**, Zidong Du, Qi Guo, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, Yunji Chen: Cambricon-S, Addressing Irregularity in Sparse Neural Networks: A Cooperative Software/Hardware Approach. MICRO 2018.

3. xxxx

专利