中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

深度学习处理器研究

作者姓名：罗韬

指导教师：陈云霁 研究员

中国科学院计算技术研究所

学位类别：工学博士

学科专业：计算机系统结构

研究所：中国科学院计算技术研究所

2015 年 10 月

Research on Deep Learning Processors

By

Luo Tao

A Dissertation/Thesis Submitted to
The University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Doctor of Computer Science

Institute of Computing Technology, Chinese Academy of Sciences

Oct, 2015

声　　明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其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任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：罗韬　　日期：2015.11.24

论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：罗韬　导师签名：[illegible]　日期：2015.11.24

摘　要

深度学习是一类多层大规模的人工神经网络方法的统称，目前已经被广泛地应用到云服务器和智能终端的广告推荐、语音识别、图像识别等核心任务上。由于大数据时代的到来，互联网每天都会产生海量的数据需要进行深度学习处理。但通用 CPU 与 GPU 的深度学习处理速度太慢，能耗极高。例如 2012 年谷歌大脑在识别猫脸的深度学习模型中甚至需要使用 1.6 万个 CPU 核进行训练。为了解决深度学习实用化“卡脖子”的速度问题，业界迫切需要面向深度学习的新型处理器芯片。

本文的目标是设计高性能、低能耗的深度学习专用处理器，并设计由多个深度学习处理器组成的硬件平台以进一步提升深度学习处理速度。为实现这个目标，本文从深度学习处理器多核结构、深度学习处理器多芯片互联结构和深度学习处理器片间光互联设计三个方面进行了研究。

• **多核深度学习处理器设计** 在本章中，我们设计了一个支持多种深度学习算法（如分类层、卷积层、池化层、LRN 层等）的多核深度学习处理器-DaDianNao。在设计多核结构的过程中，我们采用了无访存设计的设计思想来解决访存瓶颈问题，设计了基于 H 树的片上多核互联结构来解决高内部带宽带来的物理连线拥堵问题。模拟实验表明，在 ST 28 纳米工艺下，DaDianNao 的面积为 67.73 平方毫米。对于本文选用的测试集，其平均性能为 Nvidia 的一款通用计算 GPU K20 的 21.38 倍，平均能耗相比 Nvidia K20 降低了 330.56 倍。

• **深度学习处理器多芯片互联设计** 多个 DaDianNao 芯片可以通过互联提供较大的片上存储空间，将所有的神经网络参数存储在片上缓存中，绕过访存墙的限制。并且多个芯片可以提供多组运算单元从而提升运算性能。我们搭建了模拟器来评估多个 DaDianNao 组成的多芯片系统。实验结果表明，对于本文选用的测试集，64 个 DaDianNao 组成的多芯片系统平均性能为 Nvidia 的一款通用计算 GPU K20 的 450.65 倍，平均能耗相比 Nvidia K20 降低了 150.31 倍。

•**深度学习处理器片间光互联设计** 自从 1984 年 Goodman 等首先提出集成电路光互联的概念以来，光互联作为一个有效解决电互联潜在问题的办法备受关注。由于在深度学习处理

器多芯片结构中，片间数据传输是性能的瓶颈，因此为了使深度学习处理器多芯片系统的性能得到进一步提升，我们对如何在芯片间使用光互联和使用光互联带来的性能提升、功耗下降进行了研究。实验结果表示：64 芯片的 DaDianNao 多芯片光互联系统的性能可以达到 Nvidia K20 的 743.57 倍，而能耗降低了 213.44 倍。

关键词：深度学习；处理器；ASIC；体系结构；访存优化；多芯片系统

Research on Deep Learning Processor

Luo Tao
Directed By Professor Chen Yunji

Abstract

Many companies are deploying services, either for consumers or industry, which are largely based on machine-learning algorithms for sophisticated processing of large amounts of data. The state-of-the-art and most popular such machine-learning algorithms are Convolutional and Deep Neural Networks (CNNs and DNNs), which are known to be both computationally and memory intensive. A number of neural network accelerators have been recently proposed which can offer high computational capacity/areacost ratio, but which remain hampered by memory accesses.

However, unlike the memory wall faced by processors on general-purpose workloads, the CNNs and DNNs memory footprint, while large, is not beyond the capability of the onchip storage of a multi-chip system. This property, combined with the CNN/DNN algorithmic characteristics, can lead to high internal bandwidth and low external communications, which can in turn enable high-degree parallelism at a reasonable area cost.

In this paper, we propose three novel techniques related to design a custom multi-chip deep-learning architecture.

• **High performance multi-core deep-learning processor.** Observed that the main performance limitation of the single-core processor is is the memory bandwidth requirements of two important

layer types: convolutional layers with private kernels (used in DNNs) and classifier layers used in both CNNs and DNNs. In order to reduce memory access, we accomodate all parameters in the neuron network on chip. Which means we need to create an asymmetric architecture of which the chip footprint is massively biased towards storage rather than computations. Base on that, we proposed a multi-core processor design that can accommodate CNNs and DNNs. Compared to Nvidia K20, it is 21.38x faster, and it can reduce the total energy by 330.56x.

• **Multi-chip architecture to accomodate large-scale neuron network.** There is an emerging trend that researchers use very large scale neuron networks to raise the recognition score, it's a big challenge to hardware performance.However, unlike the memory wall faced by processors on general-purpose workloads, the CNNs and DNNs memory footprint, while large, is not beyond the capability of the onchip storage of a multi-chip system. We investigated the interconnection procotol and layer mapping method of a multi-chip deep-learning architecture. A 64-chip system is possible to achieve a speedup of 450.65x over Nvidia K20, and reduce the energy by 150.31x.

• **Silicon photonics in Multi-chip architecture** Since Goodman first proposed silicon photonics in the year 1984, it has been an effective way to solve potential problems of the original inter-chip communication. In our multi-chip system, inter-chip communication always become the bottleneck of performance. So we investigated on how to apply silicon photonics on our multi-chip architecture. After silicon photonics is applied, 64-chip system is possible to achieve a speedup of 743.57x over Nvidia K20, and reduce the energy by 213.44x.

Keywords: Deep Learning; Processor; ASIC; Micro Architecture; multi-chip system

目　　录

图目录

表目录

第一章 绪论

自 2006 年 Geoffrey Hinton 在 Science 上发表关于深度置信网络的论文[1]后，深度学习的概念被提出，机器学习的研究进入了一个新的时代。深度学习作为机器学习算法中最前沿的分支，在从移动端到云端服务的一系列场景中都得到了应用。例如苹果公司的 iPhoto 和谷歌公司的 Picasa 中的人脸识别、Siri 和 Google Now 中的语音识别、以及一系列广告预测投放[2]、机器人[3]和药物产业[4]等等。

近年来业界对深度学习应用的处理速度和性能功耗比也提出了越来越高的要求。一方面，由于大数据时代的到来，互联网公司每天都可以产生海量的数据用于深度神经网络的训练，但由于通用 CPU 与 GPU 对深度学习算法的处理速度都不理想，工业界经常出现由于处理速度不足导致无法充分利用现有训练数据的情况。比如科大讯飞的语料库获取速度是其语音识别模型训练速度的 5 倍，2012 年谷歌大脑在识别猫脸的模型中甚至使用了 1.6 万个 CPU 进行训练。

另一方面，随着深度学习算法的不断演化，深度神经网络模型的规模和复杂度也在不断增加。2012 年谷歌大脑使用了一个具有 10 亿个可训练参数的深度神经网络模型，而 2013 年就将规模扩大到了 110 亿个可训练参数。2014 年百度大脑则使用了一个具有 200 亿个可训练参数的深度神经网络模型。虽然使用更大规模的模型可以提高识别的精度，但也会随之带来训练和识别（即前馈运算）两方面的复杂度，从而影响神经网络识别的实时性。而很多深度神经网络应用比如语音识别，广告推荐，手写识别等都由于其应用场景而需要一定的实时性。比如广告推荐的实时性会影响用户的上网体验，手写识别更要在尽量短的时间内完成，因此服务器端的硬件需要足够高的性能来支持深度学习识别的实时性。而对于移动端硬件（如手机），则由于需要考虑电池的续航能力，其性能功耗比更是至关重要。

目前，业界一般使用通用 CPU/GPU 来进行深度学习算法处理，但是由于 CPU/GPU 需要耗费大量面积用以支持其通用性，处理深度学习算法的速度和能效并不理想。并且由于芯片制造工艺技术发展趋缓，频率难以显著提高，CPU/GPU 在可预见的将来很难取得深度学习处理速度质的飞跃。即使采用多 CPU/GPU 芯片系统来解决深度学习处理的速度问题，在增加性能的同时，整个系统的功耗也会随之增加，从而增加了成本。因此有必要为深度学习专门设计一款处理器以应对大数据时代对深度学习处理提出的性能与能耗问题。

在设计深度学习处理器的过程中，我们主要需要考虑以下几个问题。

首先由于深度学习的应用范围广阔，不同的深度学习应用又需要使用不同的深度学习算法进行处理，因此我们设计的深度学习处理器需要对多种深度学习算法提供支持，也就是说需要考虑其灵活性。

同时，由于深度学习处理器是为深度学习处理设计的专用硬件，所以需要具有明显高于通用 CPU/GPU 处理器的性能。要说明的是，因为深度学习处理器的应用场景非常很多样化，所以也要对多种不同规模的深度神经网络模型提供支持。因此我们不仅要对多种深度学习算法，还要对多种不同规模的深度神经网络模型进行优化，以取得总体较高的效率。

此外，因为深度学习处理器不仅可以应用在云服务器上，也应当可以引用在对能耗有严格限制的移动终端上。所以能耗也是至关重要的。

本文针对前面几个问题分别进行了研究。首先，本文在解决了访存带宽不足和片上连线拥塞的问题后，设计了多核通用深度学习处理器 DaDianNao。在 ST（SGS-THOMSON Microelectronics，意法半导体）28 纳米工艺下面积为 67.73 平方毫米的 DaDianNao 的性能可以达到 Nvidia K20 的 21.38 倍，而能耗降低了 333.56 倍。其次，为了适应深度神经网规模逐渐变大的发展趋势并且进一步提高性能，本文提出了深度学习处理器的多核互联结构，根据模拟实验的结果，64 芯片的 DaDianNao 多芯片系统的性能可以达到 Nvidia K20 的 450.65 倍，而能耗降低了 150.31 倍。最后，为了使深度学习处理器多芯片系统的性能得到进一步提升，我们对如何在芯片间使用光互联和使用光互联带来的性能提升、功耗降低进行了研究。实验结果表示：64 芯片的 DaDianNao 多芯片光互联系统的性能可以达到 Nvidia K20 的 743.57 倍，而能耗降低了 213.44 倍。

本章将介绍本文的研究背景。具体来说会从深度学习的发展，应用和业界使用的深度学习加速方式几个方面进行论述。

1.1 深度学习及其应用

深度学习是机器学习的一个分支，是近年来机器学习领域的一个重大突破，有着广泛的应用前景[85][86][87]。它通过多层次的学习得到对于原始数据的不同抽象层度的表示，进而提高分类和预测等任务的准确性。随着 Google 公司公开谷歌大脑计划，业界对于深度学习的研究热情越来越高，并且已经在图像识别[5]、语音识别[6][7]、自然语言处理[8]等领域取得了突破性的进展。

1.1.1 深度学习的实用价值

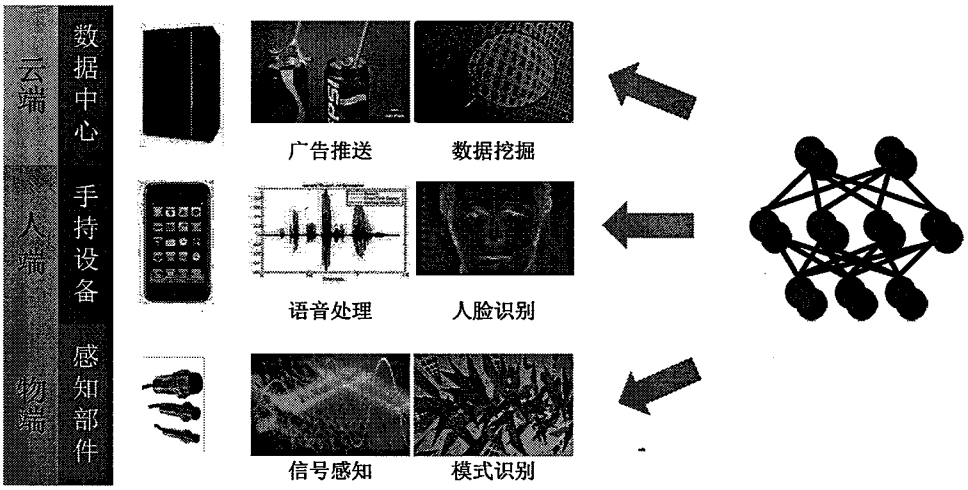


图 1.1 深度学习的应用

作为机器学习算法中最前沿的分支之一，深度学习在业界取得了广泛的应用。如图 1.1 所示，深度学习的应用覆盖了在从服务器端到移动端的一系列认知任务。包括数据中心中的广告推送和数据挖掘、手持设备中的语音处理和人脸识别、感知部件中的信号感知和模式识别等等。不仅如此，随着深度学习研究的不断深入和互联网，大数据等技术的发展，越来越多的深度学习应用将会继续涌现。

深度学习之所以能取得如此广泛的应用，是因为其可以在机器学习广泛应用的领域中取得比传统的机器学习算法更高的识别率。例如卷积神经网络将图像识别领域中 Top5 的识别率从 74%提高到了 85%[12]；深度神经网络将语音识别领域的识别率提高了 30%左右。

近年来，业界各大主要公司都投入了大量人力物力，试图在深度学习领域中取得成就。2012 年，谷歌公司的 Andrew Ng 和 Jeff Dean 带领团队了开始谷歌大脑项目，他们搭建了由 16000 个 CPU 处理器核组成的并行计算平台。谷歌大脑在图像识别和语音识别领域均获得了突破性的进展[13]，其最成功的案例是其通过自动对 YouTube 上选取的视频进行分析，将图像自动聚类以后可以在没有外界帮助下自动对猫脸进行识别。

同样在 2012 年，微软公司推出了一套同声传译系统，Rick Rashid 对其进行了演示[10]。这套同声传译系统可以将他的英文发言实时转换成音色与他相近的中文语音，深度学习则是其中的关键技术。

到了2013年，百度深度学习研究院（IDL: Institute of Deep Learning）成立了，LDL将深度学习用于包括图像识别，语音识别和广告点击率预估等在内的多个产品。

1.1.2 深度学习应用对性能的需求

近年来业界对深度学习应用的处理速度和性能功耗比也提出了越来越高的要求。一方面，由于大数据时代的到来，互联网公司每天都可以产生海量的数据用于深度学习的训练[105][106]，但由于通用CPU与GPU对深度学习算法的处理速度都不理想，工业界经常出现由于处理速度不足导致无法充分利用现有训练数据的情况。比如科大讯飞的语料库获取速度是其语音识别模型训练速度的5倍，2012年谷歌大脑在识别猫脸的模型中甚至使用了1.6万个CPU进行训练。

另一方面，随着深度学习算法的不断演化，深度学习模型的规模和复杂度也在不断增加。2012年谷歌大脑使用了一个具有10亿个可训练参数的深度学习模型，而2013年就将规模扩大到了110亿个可训练参数。2014年百度大脑则使用了一个具有200亿个可训练参数的深度学习模型。虽然使用更大规模的模型可以提高识别的精度，但也会随之带来训练和识别（即前馈运算）两方面的复杂度，从而影响神经网络识别的实时性。而现有的很多深度学习应用比如语音识别，广告推荐，手写识别等都由于其应用场景而需要一定的实时性。比如广告推荐的实时性会影响用户的上网体验，而手写识别更要在尽量短的时间内完成，因此服务器端的硬件需要足够高的性能来支持深度学习识别的实时性。而对于移动端硬件（如手机），则由于需要考虑电池的续航能力，其性能功耗比更是至关重要。

1.2 深度学习算法的传统运算平台

关于如何高性能地处理深度学习应用，学术界和工业界从多种方面进行了探索和研究。这些研究主要可以分成两个方面：一方面是从算法角度出发，通过优化深度学习算法与神经网络结构来找到执行时间与预测精度的平衡点，从而提升深度学习算法的执行效率；另一方面是从运算平台角度出发，通过在高性能的硬件平台运行深度学习应用来获得较高的性能。本文工作属于第二个方面的研究。

业界用于加速深度学习应用的硬件平台大概可以分为两类，CPU/GPU和专用硬件。

1.2.1 CPU/GPU 加速平台

使用 CPU/GPU 来进行深度学习模型的训练和预测是一种常用的解决方案，开源社区有多种较为成熟的工具可以支持 CPU/GPU 的编程。例如 Caffe 可以在 CPU 和 GPU 上实现深度学习算法[11][14]；Cuda-convnet 可以提供在 CPU 及 GPU 上的采用反向传播算法的深度卷积神经网络实现，并且支持多 GPU 上的数据并行和模型并行训练[15][16]；Kaldi 用于语音识别，同样可以用在 CPU 和 GPU 上[17][18]；Theano 是一套用于深度学习的 python 库，可以用在 GPU 上 [19][20]。

业界公司一般会使用规模较大的深度学习模型，而且对性能敏感，因此会搭建 CPU/GPU 集群来进行加速。

比较著名的 CPU 集群框架是 Google 的 DistBelief 系统，集群内使用上万个 CPU 核来训练包含十亿个可训练参数的深度学习模型[21]。

由于 GPU 不仅具有通用性，而且有比 CPU 更高的性能，业界有很多使用由多块 GPU 搭建的硬件平台进行深度学习处理的案例。例如 Facebook 使用 4 块 NVIDIA Titan GPU 搭建了硬件平台，这套硬件平台对 ImageNet 的 1000 分类网络进行训练[22]只需要几天。百度搭建了支持数据并行和模型并行的 Paddle 多机 GPU 训练平台，通过 Parameter Server 协调多个机器之间的训练并将数据分布到不同机器进行处理[23]。Google 的 COTS HPC 系统可以在使用 3 台 GPU 服务器并行计算的情况下在数天内完成对包含 10 亿个可训练参数的深度学习的训练[24]。

然而，CPU/GPU 对于深度学习的处理依然是低效的。总的来说这是因为 CPU/GPU 在运算单元方面和存储层次的设计上主要考虑了通用性，没有对深度学习算法进行专门的优化导致的。

运算单元方面，CPU/GPU 不仅可以支持加减乘除和与或非这些基本运算，也可以实现各种复杂函数的运算。虽然这样确实可以实现所有的深度学习算法，但是 CPU/GPU 中的这种通用运算结构的性能与深度学习算法需要的大规模运算相比差距还是很大。另外，CPU/GPU 每完成一次运算以后要将之前的运算结果写回寄存器堆，这样会带来性能/能耗代价，并且还导致下次运算又要再次把之前运算的结果从寄存器堆读出然后发送给运算器，又再一次带来了性能/能耗代价。

存储层次方面，CPU/GPU 一般使用片外内存和多级片上缓存的结构。由于 CPU/GPU 的设计主要考虑到了通用性，因此片上缓存的容量不大。比如 Nvidia 的高端通用计算 GPU K20x 的片上缓存总容量不到 3MB，而 Intel 的高端服务器 CPU E7-8880L 的片上缓存总容

量也不到 50MB。现有的深度学习模型的规模通常会达到几百 MB，大规模神经网络的规模甚至会达到几十 GB，因此很难完全存储在 CPU/GPU 的片上缓存中。这使得 CPU/GPU 在处理深度学习应用时需要频繁访问片外内存，造成很大的性能/能耗开销。根据 Nvidia 的实验，从片外搬运数据到片上缓存耗费时间和能量是对同样位宽的数据进行乘法所耗费时间和能量的 10 倍以上。因此如果在运算的过程中需要频繁访存，那么大部分的时间和能量将耗费在访存操作上。不仅如此，CPU/GPU 的片上缓存替换策略也不是为深度学习算法专门设计的，因此会经常地把需要频繁访问的数据换出到片外，导致下次使用的时候需要再次访存，又一次耗费了时间和能量。

1.2.2 深度学习的专用硬件加速方式

因此，逐渐有公司和研究院所开始了对深度学习专用硬件的研究，如 Chakradhar 等人提出利用 FPGA 实现卷积神经网络来处理识别任务；Temam 在注重瑕疵容忍（defect-tolerant）的基础上提出了多层的网络计算；Qadeer 等人提出了针对类卷积计算的可编程，高性能的加速器；Farabet 等人提出了专门处理卷积神经网络的专用处理器（Application Specific Integrated Circuit，ASIC）；Kim 等人则提出了面向其他神经网络算法的加速器。

然而，现有的研究都存在一些不足。一方面是研究平台的限制。现有的研究中大部分是基于 FPGA 平台的，而面对大规模深度学习应用的处理，FPGA 平台的性能和性能功耗比都不令人满意，因此需要在专用集成电路平台上开展对于深度学习处理器的研究。另一方面，现有的研究大部分都只对于少数几种深度学习算法进行加速，缺乏对深度学习算法的共性包括访存特征，公共计算引子的研究，即缺乏通用性。

1.3 本文主要贡献

本文的目标是设计支持多种深度学习算法的深度学习处理器，以达到比传统的深度学习硬件平台性能更高，能耗更低的的效果。并且设计多芯片系统以支持大规模深度学习的运算。为了实现这个目标，本文在进行了三个方面的研究：单核深度学习处理器、多核深度学习处理器、深度学习处理器多芯片互联结构。

• **多核深度学习处理器设计** 为了进一步提升性能，我们在继承前述单核深度学习处理器设计思想的基础上设计了一个支持多种深度学习算法（如分类层、卷积层、池化层、LRN层等）的多核深度学习处理器-DaDianNao。在设计多核结构的过程中，我们采用了无访存设计的设计思想来解决访存瓶颈问题，设计了基于H树的片上多核互联结构来解决高内部带宽带来的物理连线拥堵问题。模拟实验表明，在ST 28纳米工艺下，DaDianNao的面积为67.73平方毫米。对于本文选用的测试集，其平均性能为Nvidia的一款通用计算GPU K20的21.38倍，平均能耗相比Nvidia K20降低了330.56倍。

• **深度学习处理器多芯片互联设计** 多个DaDianNao芯片可以通过互联提供较大的片上存储空间，将所有的神经网络参数存储在片上缓存中，绕过访存墙的限制。并且多个芯片可以提供多组运算单元从而提升运算性能。我们搭建了模拟器来评估多个DaDianNao组成的多芯片系统。实验结果表明，对于本文选用的测试集，64个DaDianNao组成的多芯片系统平均性能为Nvidia的一款通用计算GPU K20的450.65倍，平均能耗相比Nvidia K20降低了150.31倍。

•**深度学习处理器片间光互联设计** 自从1984年Goodman等首先提出集成电路光互联的概念以来，光互联作为一个有效解决电互联潜在问题的办法备受关注。由于在深度学习处理器多芯片结构中，片间数据传输是性能的瓶颈，因此为了使深度学习处理器多芯片系统的性能得到进一步提升，我们对如何在芯片间使用光互联和使用光互联带来的性能提升、功耗下降进行了研究。实验结果表示：64芯片的DaDianNao多芯片光互联系统的性能可以达到Nvidia K20的743.57倍，而能耗降低了213.44倍。

1.4 本文组织结构

本文共包括七章，具体组织如下：

在第一章中，也即本章中，我们对深度学习处理器研究进行了概述。

在第二章中，我们介绍了深度学习处理器的相关工作。第二章首先介绍了深度学习的总体研究现状对比了浅层学习和深度学习的异同，介绍了具体的算法；并且关注了深度学习硬件加速器的的研究现状，着重介绍了H. Esmaeilzadeh等人提出的通用近似计算神经

加速器和 Olivier Temam 等人提出的瑕疵可容忍神经网络加速器，并分析了这些设计的优点和不足；最后介绍了陈天石等人提出的单核深度学习处理器 DianNao。

在第三章中，我们介绍了本文使用的测试集以及性能仿真与后端设计所采用的实验平台。包括作为性能基准的 CPU/GPU 性能仿真平台、用于进行单芯片性能仿真的 VCS 仿真平台、用于多芯片互联性能仿真的 Booksim 仿真平台和用于后端设计的逻辑综合工具 DC（Synopsys Design Compiler）、布局布线工具 ICC（Synopsys IC Compiler）、时序功耗分析工具 PT（Synopsys Primetime）。

在第四章中，我们为了进一步提升性能，同时继承了前述单核深度学习处理器的设计思想，设计了一个支持多种机器学习算法（分类层，卷积层，池化层，LRN 层等）的多核深度学习处理器-DaDianNao。在设计多核结构的过程中，我们设计了无访存设计的设计思想来解决访存瓶颈问题；设计了基于 H 树的片上多核互联结构来解决高内部带宽带来的物理连线拥堵问题。最后我们使用 Verilog 实现了本章介绍的单核深度学习处理器，使用后端设计平台对其进行了逻辑综合和布局布线。模拟实验结果显示，使用 ST 28 纳米工艺库进行布局布线后，DaDianNao 的面积为 67.73 平方毫米。对于本文选用的测试集，其平均性能为 Nvidia 的一款高端通用计算 GPU K20 的 21.38 倍，平均能耗相比 Nvidia K20 降低了 330.56 倍。

在第五章中，由于 DaDianNao 采用了无访存设计的设计思想，需要将所有参数都存储于片上，同时由于多个芯片可以提供多组运算单元以升运算性能，所以为了支持大规模深度学习模型的训练和预测，我们使用多个 DaDianNao 芯片搭建了并行化框架。仿真结果显示，与单芯片系统相比，多芯片系统的性能有大幅度的提升。模拟实验结果表明，对于本文选用的测试集，64 芯片的 DaDianNao 多芯片系统的平均性能为 Nvidia 的一款高端通用计算 GPU K20 的 450.65 倍，平均能耗相比 Nvidia K20 降低了 150.31 倍。

在第六章中，由于在深度学习处理器多芯片结构中，片间数据传输是性能的瓶颈，因此为了使深度学习处理器多芯片系统的性能得到进一步提升，我们对如何在芯片间使用光互联和使用光互联带来的性能提升、功耗下降进行了研究。实验结果表示：64 芯片的 DaDianNao 多芯片光互联系统的性能可以达到 Nvidia K20 的 743.57 倍，而能耗降低了 213.44 倍。

在第七章中，也就是最后一章中，我们对本文的工作进行了总结，并探讨了未来研究的方向。

第二章 相关工作

2.1 深度学习算法

2.1.1 浅层学习与深度学习

深度学习是一类多层大规模的人工神经网络方法的统称[25]。而在深度学习被提出之前，机器学习经过了很长一段时间的浅层学习时期[26]。

浅层学习(shallow learning)

反向传播[27]对于机器学习的发展起了很大的促进作用。人工神经网络可以通过反向传播算法在训练过程中自动调节参数，以获得更好的识别效果。虽然在这一时期中，神经网络称为作多层感知器[28]，但实际上它是只包含一个隐层的浅层模型。在这种浅层学习的发展过程中，还出现如最大熵方法[29](如逻辑回归)、逻辑回归 Boosting[30]、支持向量机[31]等的浅层模型。而浅层学习也广泛地应用在了多种领域中，如内容推荐系统，点击率估计，垃圾邮件过滤[32]等等。

深度学习(deep learning)

自 2006 年 Geoffrey Hinton 在 Science 上发表关于深度置信网络的论文后[33]，深度学习的概念被提出，机器学习的研究进入了一个新的时代。深度学习作为机器学习算法中最前沿的分支，在从移动端到云端服务的一系列场景中都得到了应用。

2010 年，美国国防部对纽约大学和斯坦福大学共同参与的[34]的深度学习研究项目额提供了支持。Hubel 和 Wiesel 由于在视觉神经系统方面的贡献获得了诺贝尔医学奖[35]。2012 年，医药企业在药品活动中使用了深度学习，取得了很好的效果[36]。2012 年，由 Andrew NG 领导的谷歌大脑项目使用深度学习识别了猫的形象。[37]。

2.1.2 人工神经网络

单个神经元

首先，单个神经元的计算方法如图 2.1 所示：

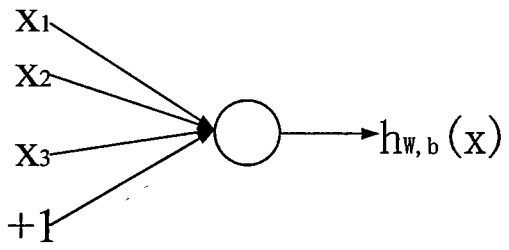


图 2.1 单个神经元结构

这个神经元的输入为 x1， x2， x3。用于计算这个神经元输出的公式是：

$$h_{W,b}(x) = f(W^T x) = f\left(\sum_{i=1}^{3} W_i x_i + b\right)$$

我们可以使用 sigmoid 作为激活函数：

$$f(z) = \frac{1}{1 + e^{-z}}$$

除了 sigmoid 外，双曲正切函数 tanh 也可以作为激活函数：

$$f(z) = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

图 2.2 和图 2.3 分别为 sigmoid 函数和 tanh 函数的曲线图。

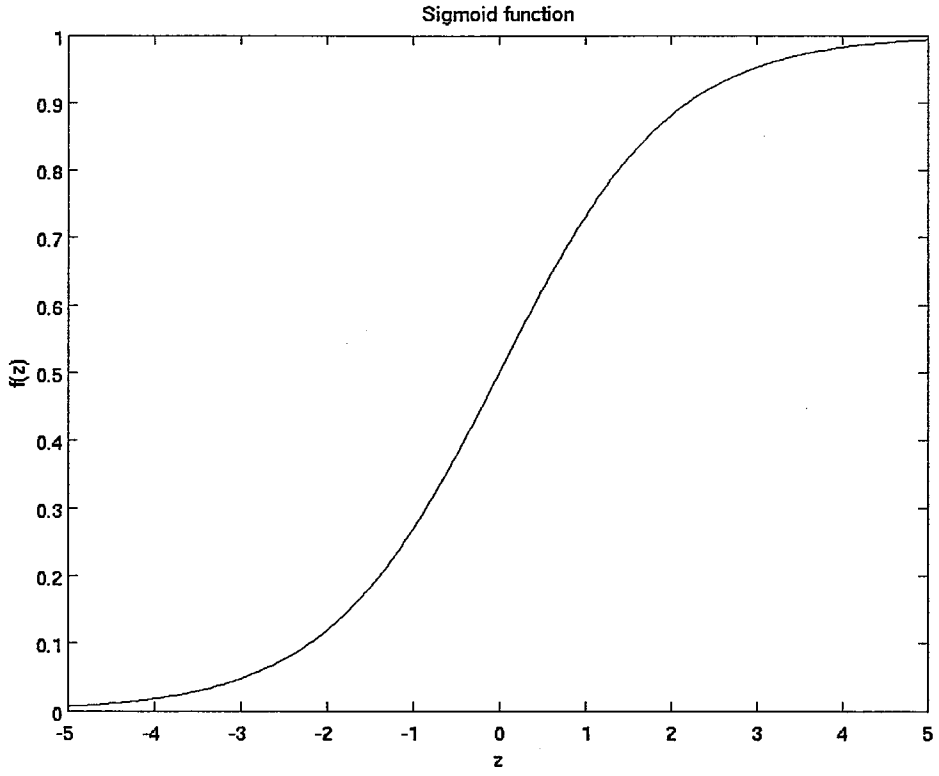


图 2.2 sigmoid 函数曲线图

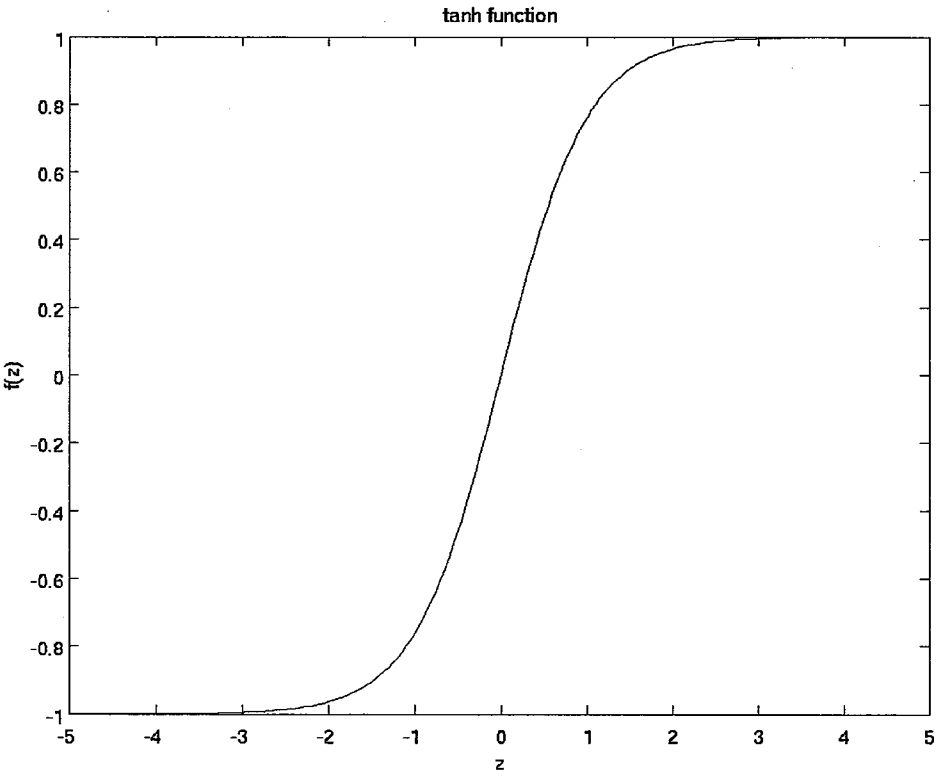


图 2.3 tanh 函数曲线图

神经网络模型

图 2.4 就是一个基本的神经网络：

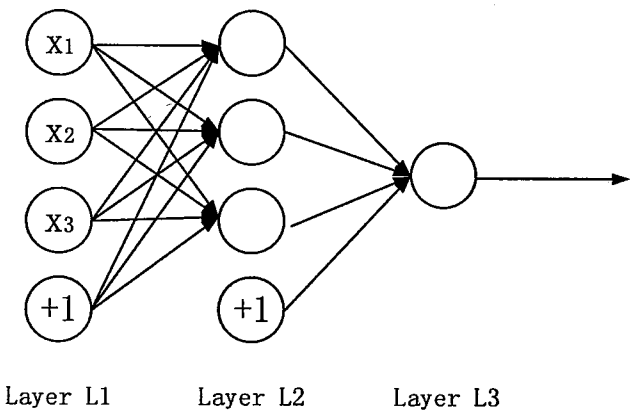


图 2.4 神经网络结构

图 2.4 为一个输入层包含 3 个神经元、隐藏层包含 3 个神经元，输出层包含 1 个神经元的神经网络。

我们使用S_l表示第 l 层的神经元数量，n_l表示神经网络的层数，将神经网络中的第 l 层记为L_l，则这个神经网络包含有参数$(W,b) = \left(W^{(1)},b^{(1)},W^{(2)},b^{(2)}\right)$。我们使用（不包含偏置节点）。同时，我们使用$a_i^{(l)}$表示第 l 层中第 i 节点的输出值。

对于图 2.4 中给出的神经网络，前馈运算的公式如下：

$$a_1^{(2)} = f\left(W_{11}^{(1)}x_1 + W_{11}^{(1)}x_1 + W_{11}^{(1)}x_1 + b_1^{(1)}\right)$$

$$a_2^{(2)} = f\left(W_{21}^{(1)}x_1 + W_{21}^{(1)}x_1 + W_{21}^{(1)}x_1 + b_2^{(1)}\right)$$

$$a_3^{(2)} = f\left(W_{31}^{(1)}x_1 + W_{31}^{(1)}x_1 + W_{31}^{(1)}x_1 + b_3^{(1)}\right)$$

$$h_{W,b}(x) = a_1^{(3)} = f\left(W_{11}^{(2)}a_1^{(2)} + W_{11}^{(1)}a_2^{(2)} + W_{11}^{(1)}a_3^{(2)} + b_1^{(2)}\right)$$

如果我们使用$z_i^{(l)}$来表示第 l 层中第 i 个神经元的值，用向量表示激活函数 f（x）。那么上面的公式可以表示为：

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$h_{W,b}(x) = z^{(3)} = f(z^{(3)})$$

如果使用$a^{(1)} = x$表示输入层的激活值，那么在$a^{(l)}$给定以后，计算第 l+1 层的激活值$a^{(l+1)}$的公式为：

$$a^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$

$$z^{(l+1)} = f(z^{(l+1)})$$

上面介绍的神经网络结构中只有一层隐藏层，属于浅层学习。在深度学习中，我们会使用含有多层隐藏层的神经网络，如图 2.5。

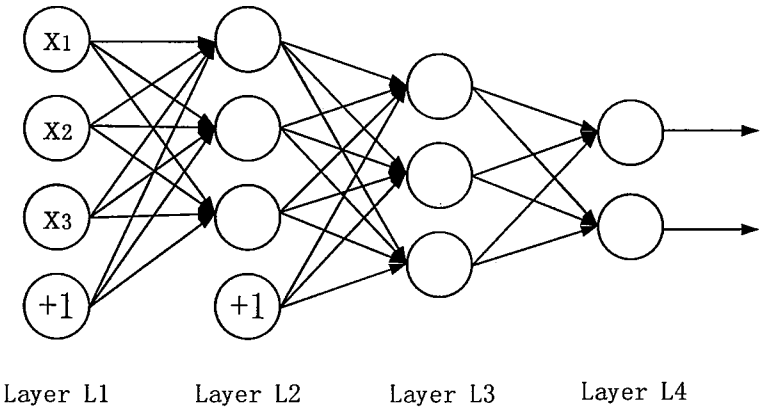


图 2.5 多层神经网络结构

反向传播算法

对于带标签的含有 m 个样例的样本集$\{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\ldots,(x^{(m)},y^{(m)})\}$，我们使用梯度下降法进行反向传播[39]，对于单个样本(x，y)它的代价函数如下：

$$J(W,b;x,y)=\frac{1}{2}\left\|h_{W,b}(x)-y\right\|^2$$

上式是一个方差代价函数，而整体代价函数可以被定义为：

$$J(W,b;x,y)=\left[\frac{1}{m}\sum_{i=1}^{m}J\left(W,b;x^{(i)},y^{(i)}\right)\right]+\frac{\lambda}{2}\sum_{l=1}^{nl-1}\sum_{i=1}^{sl}\sum_{j=1}^{sl+1}\left(W_{ji}^{(l)}\right)^2$$

$$=\left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right)-y^{(i)}\right\|^2\right]+\frac{\lambda}{2}\sum_{l=1}^{nl-1}\sum_{i=1}^{sl}\sum_{j=1}^{sl+1}\left(W_{ji}^{(l)}\right)^2$$

上面公式中的后一项是为了减小权重幅度，防止产生过拟合现象[40]使用的权重衰减项。在反向训练的过程中，我们首先将参数随机初始化，而不是设为 0[41]。然后，我们使用梯度下降法对参数 W 和 b 进行调节，梯度下降法使用的公式如下：

$$W_{ij}^{(l)}=W_{ij}^{(l)}-\alpha\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}$$

$$b_i^{(l)}=b_i^{(l)}-\alpha\frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

$\alpha\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}}$ 和$\alpha\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}}$是单个训练样本(x, y)对应的代价函数$J(W,b;x,y)$的偏导数。我们可以根据这两个偏导数求解出整体样本代价函数$J(W,b)$的偏导数：

$$\alpha\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}=\left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial J\left(W,b;x^{(i)},y^{(i)}\right)}{\partial W_{ij}^{(l)}}\right]+\lambda W_{ij}^{(l)}$$

$$\alpha\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J\left(W,b;\mathrm{x}^{(i)},\mathrm{y}^{(i)}\right)}{\partial b_i^{(l)}}$$

下面是反向传导算法过程的细节：

首先反向传播前需要进行前馈运算，得出L_2，L_3，...，L_{nl}各层节点的激活值。

对于神经网络的输出层，计算输出和标签之间的残差：

$$\delta_i^{(nl)} = -\left(y_i - a_i^{(nl)}\right)\cdot f'\left(z_i^{(nl)}\right)$$

L_{nl-1}，L_{nl-2}，...，L_2，各层的残差计算方式如下：

$$\delta_i^{(l)} = \left(\sum_{j=1}^{sl+1} W_{ji}^{(l)}\,\delta_j^{(l+1)}\right)\cdot f'\left(z_i^{(l)}\right)$$

最后，偏导数的计算公式可表示为：

$$\frac{\partial J(W,b;\mathrm{x},\mathrm{y})}{\partial W_{ij}^{(l)}} = a_j^{(l)}\delta_i^{(l+1)}$$

$$\frac{\partial J(W,b;\mathrm{x},\mathrm{y})}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

总结一下，反向传播的过程如下：

1. 反向传播前需要进行前馈运算，得出L_2，L_3，...，L_{nl}各层节点的激活值。
2. 对于神经网络的输出层，计算输出和标签之间的残差：

$$\delta^{(nl)} = -\left(y - a^{(nl)}\right)\cdot f'\left(z^{(nl)}\right)$$

3. 对L_{nl-1}，L_{nl-2}，...，L_2，等各层，计算每个节点的残差：

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T\delta^{(l+1)}\right)\cdot f'\left(z^{(l)}\right)$$

4. 最后，计算最终的偏导数值：

$$\nabla_{\mathrm{W}(t)}J(W,b;\mathrm{x},\mathrm{y}) = \delta^{(l+1)}\left(a^{(l)}\right)^T$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}$$

2.2 深度学习处理器研究现状

从上世纪90年代以来，人们就开始尝试使用专用硬件的方式来对神经网络的运算进行加速，人们对神经网络专用硬件的探索也经历了曲折的过程[42][43]。在研究的早期，由于以下几个原因，人们对神经网络专用硬件的研究热情一度不是很高：1.人工神经网络的效果在当时不如其他机器学习算法比如支持向量机更好；2.当时科学计算占据了高性能需求应用的主流，所以人工神经网络的需求得不到关注；3.当时人们对如果神经网络的研究主要局限于浅层神经网络模型，因此对于硬件资源的需求也不是很迫切。

然而随着深度学习的出现，这三点因素在当下都已经发生了变化，首先深度神经网络（DNN）和卷积神经网络（CNN）在很多领域的识别率超过了传统机器学习算法，并且深度学习算法已经在很多领域中取得了广泛的应用，此外深度学习算法无论是对计算能力还是存储能力都有很高的要求。

因此，逐渐有公司和研究机构开始了对深度学习专用硬件的研究。Yeh 等人就提出了使用FPGA 设计专用加速器来加速k-近邻算法（k-NN）[60][61]，而Manolakos 等人则在2010 年设计了一个用于加速k-近邻算法的IP 核[62]。Hussain 等人为k-Means 算法设计了一个专用的 FPGA 加速器[63]，并将专用加速器和 CPU 以及 GPU 进行了比较[64]；Maruyama 等人则为实时的k-平均聚类算法(k-Means) 应用设计了一个FPGA专用加速器[65][66]。Meng Hongying 等人则为朴素贝叶斯算法（Naive Bayes，NB）在图像识别上的应用提出了一个 FPGA 专用加速器[67]。而对于最流行的机器学习方法－支持向量机和神经网络，则有更多的专用加速器被提出。在支持向量机方面，Cadambi[68] 和Markos[69] 等人分别提出了两种基于 FPGA 的支持向量机加速器，Kyrkou 则提出了能够用于实时物体识别的支持向量机加速器[70]。在神经网络方面，则有更多的工作被提出[71][72][73]，具体包括，Farabet 等人在2009 年以及2010 年分别研究了卷积神经网络加速器核心的实现[74] 以及大规模的卷积神经网络的在 FPGA 上的实现[75]，Farabet 还研究了卷积神经网络在机器视觉的具体应用[76]和专门处理卷积神经网络的专用处理器（Application Specific Integrated Circuit ，ASIC）[51]。Chakradhar 等人提出利用FPGA实现卷积神经网络来处理识别任务[44]；IBM近年提出的认知芯片，是一个具备高能效的例子[45]，以及另一个认知计算芯片，是用硬件进行神经网络生物

仿生的例子[46]；Temam 在注重瑕疵容忍（defect-tolerant）的基础上提出了多层的网络计算[47]；Qadeer 等人提出了针对类卷积计算的加速器，具有可编程、较高性能、相对低能耗的特点[48]；Farabet 等人提出了；Kim 等人则提出了面向其他神经网络算法的加速器[49]，等等。下面我们从现有的深度学习专用硬件中选取 2 种典型的例子，对他们的工作进行介绍和分析。

2.2.1 近似计算神经加速器

H. Esmaeilzadeh 等人基于鹦鹉变换（Parrot transformation）算法提出了一种可以高性能低能耗地对代码执行过程进行加速的架构[50]。本架构的核心思想是学习代码中会产生近似结果的部分的行为，同时设计神经处理单元 NPU（neural processing unit），再根据学习的结果对 NPU 编程，达到加速程序执行并降低能耗的目的。总的来说，这套架构包括三个主要的部分：

（1）人为标注：通过人工方式，确定程序中满足以下条件的程序片段：1.可以用近似计算代替且不会丧失整个应用的可靠性；2.是程序中关键的、不可避免的部分。

（2）编译器处理：将被标注代码的功能转换为神经网络计算，转换过程包括确定代码行为、收集训练数据并确定要使用的神经网络的规模和拓扑结构、训练神经网络参数、生成对应指令四个步骤。

（3）NPU 执行：NPU（neural processing unit）即神经处理单元，是与 CPU 紧耦合的片上模块，类似于协处理器。在程序运行时，CPU 首先将程序编译成包含神经网络运算的指令，然后输出至 NPU，NPU 计算完成后再把输出结果输出至 NPU。

对应以上的三个步骤，这种策略的效果主要取决于代码段标注是否合适，编译器对神经网络运算的转换是否合理，以及 NPU 的性能和和能耗三点。以下我们重点介绍 NPU 相关的部分。

加速器顶层结构

NPU 是一个延迟可变、通过优先队列与 CPU 核心紧耦合的加速器，功能上类似一个协处理器。如图 2.6，CPU-NPU 之间的接口由三个优先队列组成，一个用来发送、接收配置信息，一个用来发送输入数据，最后一个用来接收 NPU 的输出数据。对应的访问队列的指令集由 4 条指令构成。

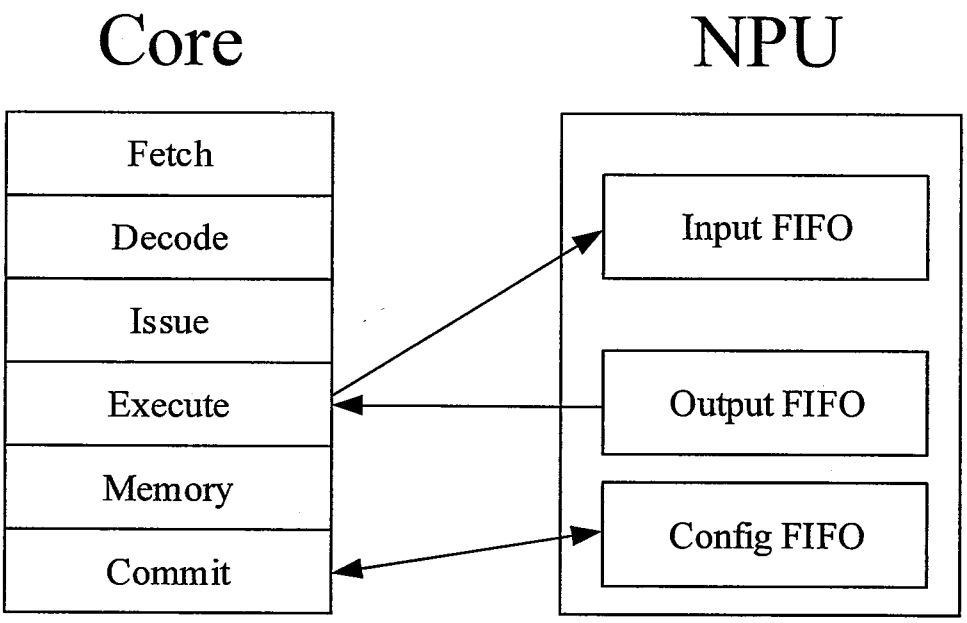


图 2.6 NPU 与 Core 通讯方式

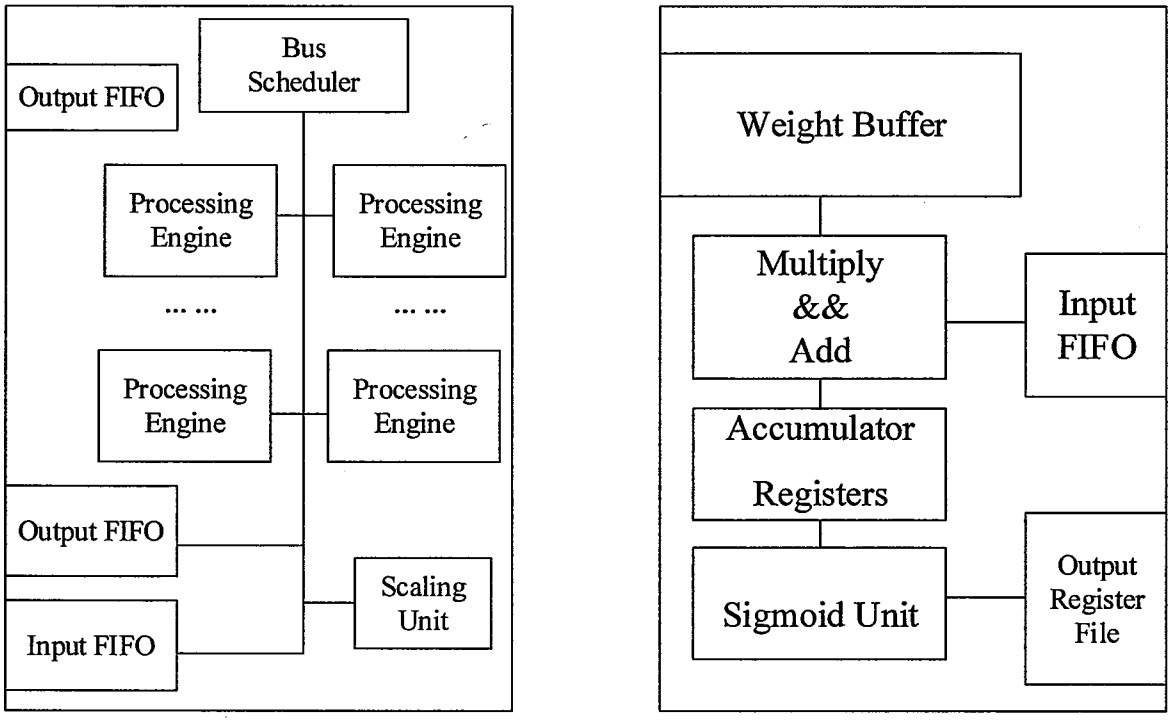
为了配置 NPU，程序通过一系列入队列指令向 NPU 发送配置参数。配置参数包括输入输出的数据量、网络的拓扑结构、神经元和权值数据等。在上下文切换的过程中，操作系统使用出队列指令将 NPU 的配置输出并保存。为了调用 NPU，程序通过重复地执行入队列指令来发送配置信息，配置神经网络。当所有的输入配置信息都进入队列后，NPU 开始根据这些配置信息进行计算并将相应结果放置输出队列中。最后程序再通过出队列指令从输出 FIFO 中取出结果。

最后，指令的执行顺序需要得到一定的保证，从而输入输出队列的指针也需要得到相应的维护。

NPU 的结构

鹦鹉变换为不同的代码段生成了不同拓扑结构的神经网络，因此 NPU 也需要是可配置的，这样才能对不同拓扑结构的神经网络的执行过程进行加速。如图 2.7(a)，NPU 包含 8 个独立的处理引擎 PE(processing engines)，和一个规模变换电路(scaling unit)。规模变换电路可以根据 NPU 的配置信息决定是否使用比例因子来放大或缩小输入输出数据的规模。所有 PE 都是可配置的，配置信息包含在 NPU 的配置信息中。神经网络里的每个神经元都被指定到一个 PE 上执行，神经网络的不同拓扑结构会对 PE 执行时间、总线

访问和队列访问的配置方式产生影响。NPU 把从总线接收到的配置信息存储在循环的配置缓存中，如图 2.7(a)。每条配置信息都控制着总线把一组数据从 PE 或输入 FIFO 中发送到目的 PE 或输出 FIFO 中。



(a) 8-PE NPU　　(b) Processing Engine

图 2.7 可重配置的 8-PE NPU

图 2.7(b)示意了单个 PE 内部的结构，每个 PE 执行所有与指定给它的神经元相关的计算。神经网络的权值存储在一个权值缓存中。当 PE 从总线接收到输入神经元时，它把接收到的值存储在输入 FIFO 中；当 PE 从总线接收到权值时，它把接收到的值存储在输入权值的缓存中。PE 顺序执行收到的指令，这样可以保证输入神经元的到达顺序和权值在缓存中存储的顺序是对应的，因此可以按照输入数据进入输入 FIFO 的顺序来执行向量点乘、加法树、激活函数等操作。

NPU 的配置可以简述为以下过程：在代码生成阶段，编译器生成包含执行神经网络计算的配置信息。接着静态 NPU 的配置算法对神经网络的输入进行排序，这个顺序决定了 CPU 发送输入数据给 NPU 的顺序以及 PE 执行向量点乘、加法树、激活函数的顺序。然

后以以下步骤顺序处理神经网络的每一层：

a) 给每个神经元指定一个 PE。

b) 根据每层的输入数据的顺序，指定向量点乘、加法树、激活函数的顺序。

c) 指定每层输出的顺序。

d) 根据之前的操作顺序，生成总线调度方式。

最后一层神经网络的调度顺序决定了程序用出队列命令接收 NPU 输出结果的顺序。

实验结果

图 2.8 示意了在给定的标准测试集上，使用 8-PE NPU+CPU 运行测试代码相较于仅使用 CPU 运行测试代码的加速比。图中同样示意了理想情况下的加速比，即 NPU 的计算时间为 0 时的加速比。

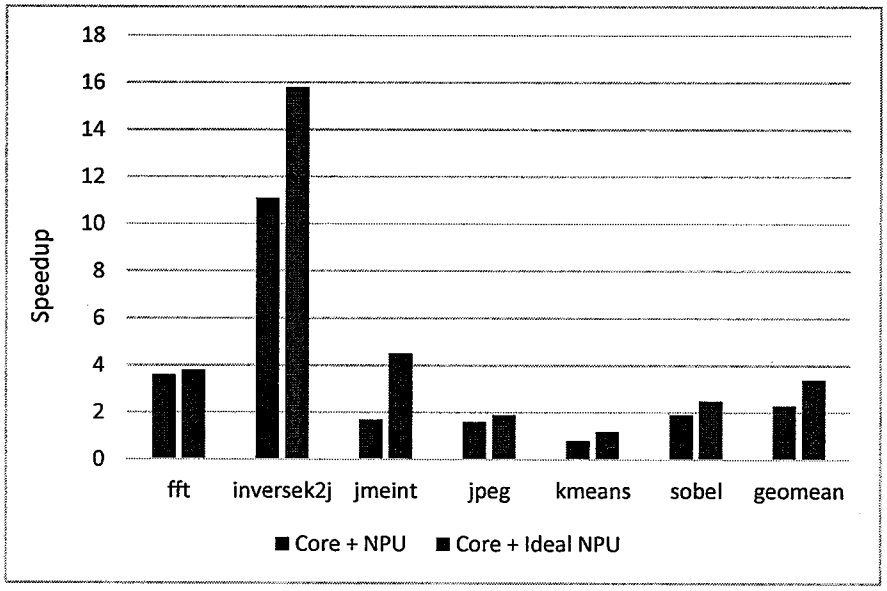


图 2.8 8-PE NPU 在不同测试集上对于 CPU 的加速比

图 2.8 示意了上述过程对应的能耗降低率。同样的，图中也示意了当 NPU 能耗为 0 时的最理想能耗降低率。

优点和不足

由之前的阐述可知，该解决方案的应用范围比较广泛，凡是涉及到对计算结果有一定误差容忍的程序，均可以尝试用此方案来处理，常见的应用场景例如科学计算、图像处理、信号处理、机器人控制等。此外，与使用 CPU 的情况相比，该方案也获得了不错

的加速比和能耗降低率。

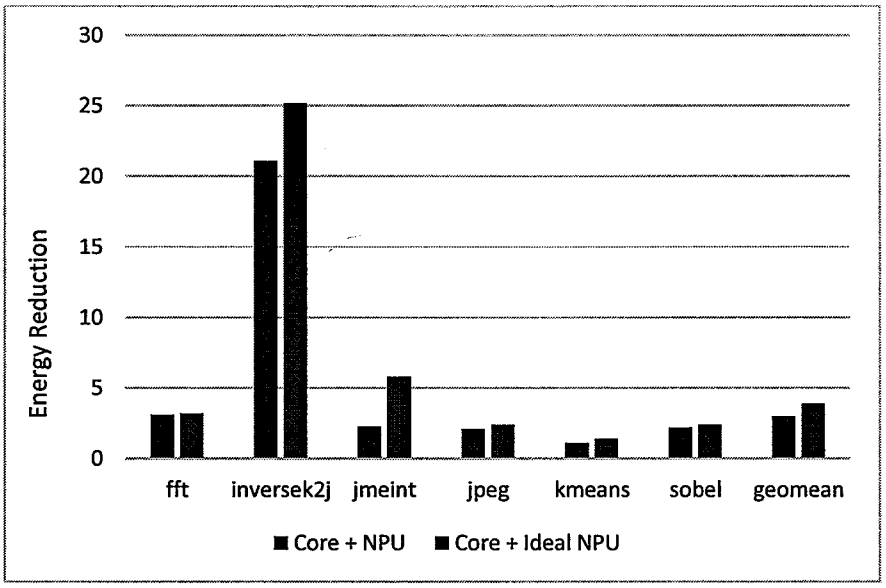


图 2.9 8-PE NPU 在不同测试集上对于 CPU 的能耗降低率

但这套方案同时也存在一些缺陷。首先在性能上，由于这里只设计了协处理器来对深度学习算法进行加速，因此无论是在性能上还是在能耗降低率上都没有足够好的效果。另外，其在功能方面也有一些不足，如编译器在生成输入数据和神经网络结构的过程可能会存在时间开销过大的问题、NPU 只支持 MLP 计算是否会导致在某些领域出现表达能力不足或者计算量被大量浪费的问题（例如以卷积、池化来代替全连接的 mlp 会提高预测的速度）、以及如何适配更广阔的应用场景、如何避免人为标注代码段的错误、如何提高计算准确度的问题等等。

2.2.2 瑕疵可容忍神经网络加速器

Olivier Temam 注意到了一定程度的瑕疵（defect）并不会对人工神经网络的结果造成太大的影响，并由此提出了面向瑕疵容忍和能耗降低的硬件加速器。

基于空间扩展的神经网络运算方式

很多已有的神经网络实现都是基于时分(time-multiplexed)的，如图 2.10。即使用硬件实现的神经元和突触存储在分散的存储器中，在每一个时钟节拍内，一层中的部分

神经元通过取出对应权值的方式映射到硬件网络上。隐藏层的输出神经元被存回至下来作为下一层的神经元输入。

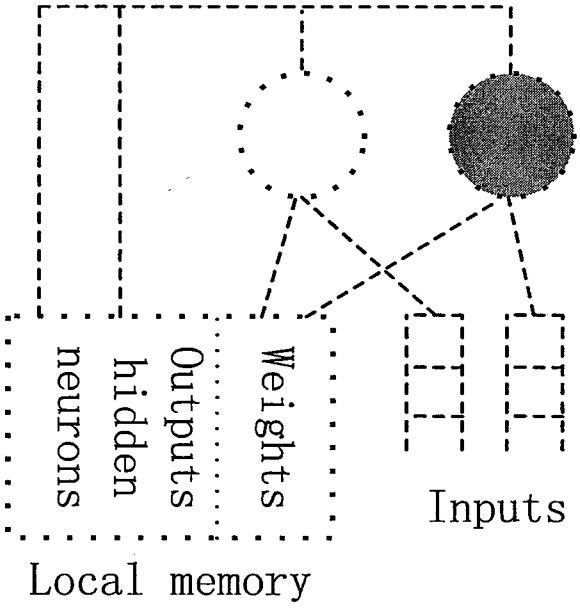


图 2.10 分散存储突触权值

空间扩展的神经网络的结构和操作与理想化的神经网络非常类似：所有神经元都被映射到硬件上，权值依照神经元的分布被存储在独立分布的存储器中。在这种实现方式下，除权值的访问延迟小之外，整体的带宽也非常高，不会受到单一存储器结构的影响。同时，去中心化的存储意味着权值与神经元的空间距离比较近，这也会降低能耗。

在基于时分的神经网络实现中，相当一部分的逻辑都被用到了时分过程本身。时分过程包括地址解码、将权值传送至操作器、传回结果，这些控制逻辑上的任何一个晶体管出错都会导致整个加速器不能工作。

而在空间扩展的神经网络实现中，由于使用了去中心化的存储器，因此只在输入输出上有控制逻辑，读操作是不需要解码的（虽然写操作依然需要解码逻辑）。这样若干个神经元或权值上的错误通常不会对结果产生较大的影响。

瑕疵可容忍神经网络加速器的硬件结构

为了提高流水线的性能，在输入/输出方面，该加速器为 DMA 中的每个输入配备了 2

路输入缓存，同样对个每个输出也都配备了 2 路缓存。加速器与 DMA 使用简单的 2 信号握手协议（ready/accept）来通信。

数据精度方面，其使用了 16 位定点数的设计，其中小数部分为 10 位，整数部分为 6 位。作者认为尽管很多 ANN 的软件实现都依赖于浮点数计算，但通常都是不需要使用这么高的精度的，甚至仅有 8 位的定点运算也可以在很多情况下得到与 32 位定点运算相似的精度结果。

关于激活函数的实现，由于激活函数 sigmoid 是非线性的计算，直接使用逻辑门来实现激活函数的话会有很大的硬件开销，该加速器使用了以查找表为基础的分段线性近似来实现激活函数。即首先根据输入数据的值确定输入数据所属的的分段，然后根据分段号查找这个段的系数(ai, bi)，最后根据系数(ai, bi)计算出变换结果（等于输入数据 × ai + bi)。

当硬件规模与空间扩展神经网络所需的硬件规模不匹配的时候，该加速器依然采用了时间扩展神经网络的方法实现。首先将所有神经网络中的神经元看做是属于一个足够大的神经层，这样就需要直接把输出层的输入数据传送进去（或者直接接受隐藏层的输出）。实现的方式是增加和隐层神经元数量相等的输入线路，并将这些新增的输入电路直接连接到隐藏层和输出层之间的内部总线上。这样这些输入线路就可以直接把输入数据传输给输出层神经元。同时，隐层神经元的输出也可以通过额外线路直接连接到加速器的输出上。这样，所有片上的神经元都能作为单一层的一部分被使用。使用时间扩展神经网络的方法实现时，处理一个包含 N 倍于硬件神经元数量的神经网络需要 N 倍于硬件神经网络满负荷处理一次输入数据的延迟时间。

实验结果

在 90nm 工艺下，瑕疵可容忍神经网络加速器的面积为 9.02mm^2，功耗为 4.7W。其满负荷处理一次输入的时间是 14.92ns，耗能 70.16nJ。如果和 Intel Stealey 的性能对比，以其最高频率 800MHz 来考虑，计算同样的神经层则需要 68388nJ 的能耗以及 19680 周期，即 24600ns 的时间。

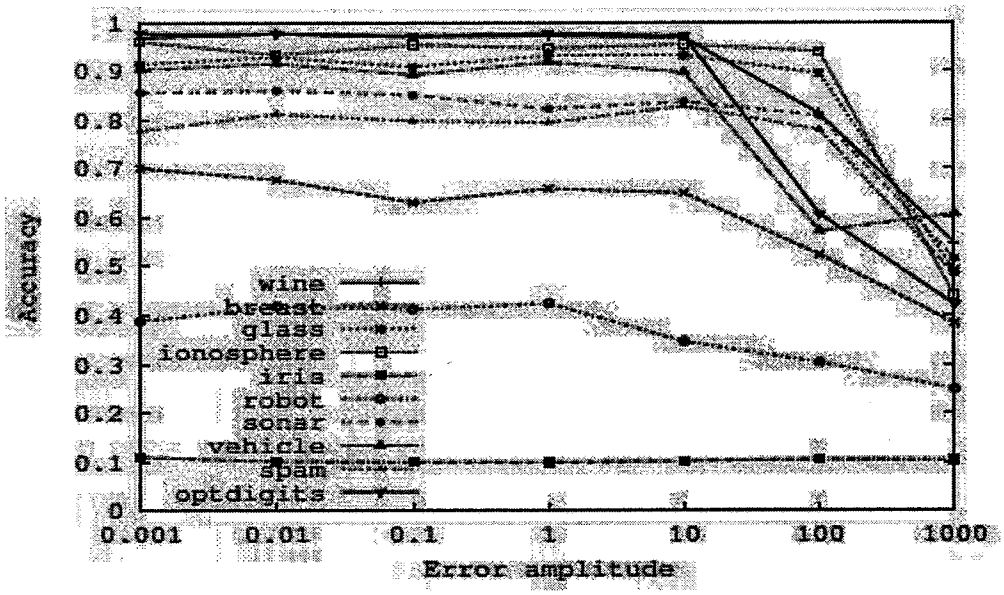


图 2.11 瑕疵可容忍神经网络加速器的误差幅度

另一个值得关注的是计算结果的精度问题，图 2.11 示意了加速器在输出层的误差幅度，可以看到 iris 和 robot 两者的精度比较低。并且其他应用的精度虽然相对高一些，但都不是很理想。

优点和不足

该加速器适用于对结果有一定误差容忍的神经网络计算，然而应用在有误差出现的情况下精度会急剧地下降，如 iris、robot 等，并且在其他情况下其效果也不是很理想。因此，精度不足是其比较突出的一个问题。此外，当硬件规模与空间扩展神经网络所需的硬件规模不匹配的时候，该加速器依然需要采用时间扩展神经网络的方法实现，也导致其适用范围比较窄。

2.3 单核深度学习处理器

陈天石等研究者提出了一种单核心的深度学习处理器 DianNao。他们对几种常见的深度学习算法进行了数据局部性分析和访存优化，优化了深度学习算法的访存瓶颈，并且进行了片上缓存，运算器，指令集几个方面的设计。

模拟实验结果显示，使用 TSMC 65 纳米工艺库进行布局布线后，DianNao 的面积为 3.023 平方毫米，频率为 0.99GHz。其平均性能可以达到主频为 2GHz 的 128bit-SIMD CPU 的 117.87 倍，平均能耗则降低了 21.08 倍。

2.3.1 DianNao 访存优化

深度学习中含有大量的神经元和权值参数，所以其运算中含有大量的访存操作，而这些访存操作造成了深度学习应用的性能瓶颈。下面介绍 DianNao 中使用的分块运算（tiling）访存优化方式。

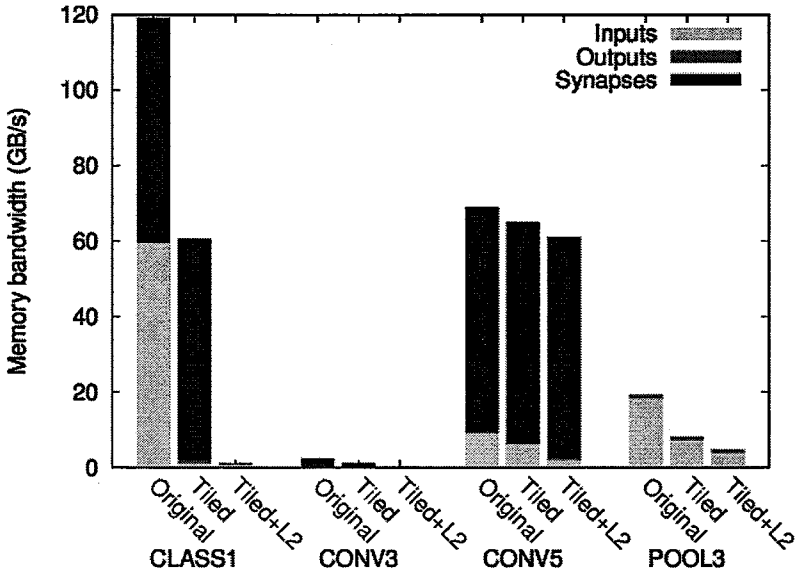


图 2.12 Diannao 访存优化性能分析

分类层访存优化

在分类层中，输入神经元和输出神经元都以一维的形式排列，并且所有输入神经元和所有输出神经元之间通过权值全相连，因此权值是二维的。分类层算法的伪代码见图 2.13，使用分块运算方式优化访存后，分类层的运算方式见图 2.14。

```
for (int n = 0; n < Nn; n += 1) {
  for (int i = 0; i < Ni; i += 1)
    sum[n] += synapse[n][i] * neuron[i];
  neuron[n] = sigmoid(sum[n]);
}
```

图 2.13 分类层原始运算方式

在图 2.13 与图 2.14 中，Ni 表示本层输入神经元的个数、 Nn 表示本层输出神经元的个数、 Tn 和 Ti 表示运算单元每个时钟周期能计算的输出神经元个数以及输入神经元个数、 sum[n]表示第 n 个输出神经元的中间结果，即部分和、 neuron[n]表示第 n 个输出神经元的值。

```
for (int nnn = 0; nnn < Nn; nnn += Tnn) { // tiling for output neurons
  for (int iii = 0; iii < Ni; iii += Tii) { // tiling for input neurons
    for (int nn = nnn; nn < nnn + Tnn; nn += Tn) {
      for (int n = nn; n < nn + Tn; n++)
        sum[n] = 0;
      for (int ii = iii; ii < iii + Tii; ii += Ti)
        // -- Original code --
        for (int n = nn; n < nn + Tn; n++)
          for (int i = ii; i < ii + Ti; i++)
            sum[n] += synapse[n][i] * neuron[i];
    for (int n = nn; n < nn + Tn; n++)
      neuron[n] = sigmoid(sum[n]);
} } }
```

图 2.14 分类层分块运算方式

在不进行访存优化的情况下（见图 2.13），进行一次规模为 Ni × Nn 的分类层运算总共需要进行的访存次数为 Ni × Nn（读输入神经元）＋ Ni × Nn（读权值）+ Nn（写输出神经元），这需要很大的访存带宽。比如测试集中的 CLASS1 需求的总访存带宽高达 120GB/s（见图 2.12 中 CLASS1-Original）。下面介绍 DianNao 中分类层分块算法的设计思路。

对于输入/输出神经元的复用性

对于每个输出神经元的计算都需要用到所有的输入神经元，如果能将所有的输入神经元都存储在片上缓存中，那么由读取输入神经元引起的访存就只需要 Ni 次，相对于原来的 Ni × Nn 次访存会有大幅度的减少。但在实际的应用中，神经网络中输入神经元的个数可能会多达几千甚至上万个，很难全部存储在片上缓存中。

因此只能在片上缓存一部分输入神经元，即可以根据片上缓存的大小将每层的输入神经元分块，使得每块都可以全部存储在片上缓存中。这里假设其中每块包含 Tii 个输入神经元。

使用分块存储的方式进行访存优化之后，分类层运算的顺序是：首先从内存中读入第一块中的 Tii 个输入神经元并计算所有输出神经元的部分和，接下来读入第二个分块中的所有输入神经元，计算所有输出神经元的部分和并与之前算出的部分和相加，以此类推。当所有的输入块都从内存中读入并运算后，输出神经元的最终和计算完成。最后，对输出神经元的最终和做非线性变换以后就能得到本层的最终结果。

上面对输入神经元进行了访存优化，而对于输出神经元的访存优化与上面类似。：根据片上缓存的大小将每层的输出神经元分成大小为 Tnn 的块，在计算出第一个 Tnn 输出神经元块后，再计算第二个 Tnn 输出块，以此类推直到计算出所有的输出神经元。

总的来说，一级缓存容量的限制导致了对于输入神经元的分块运算，对部分和的复用导致了对于输出神经元的分块运算。实验结果表明，对输入/输出神经元进行分块运算可以显著降低访存带宽。在未使用分块运算前，输入/输出神经元的总访存带宽是 60GB/s；使用分块运算后，总访存带宽不超过 2GB/s（见图 2.12 中 CLASS1-Tiled）。

对于权值的复用性

在分类层中，因为权值通常都是各不相同的，并且每个权值只会被用于一次计算，所以权值没有被复用的情况。实验结果也表明对分类层进行分块运算前后权值的访存带宽几乎没有变化(见图 2.12 中 CLASS1-Tiled)。

但是虽然分类层没有层内的权值复用，但多个分类层之间却可能存在权值复用。如果二级 Cache 足够大，能够存下所有的权值，那么就能利用权值在层间的复用性，在多次前馈运算或多次训练中复用权值。根据实验结果，可以看到权值访存带宽有了显著的下降(图 2.12 中 CLASS1-Tiled+L2)。

卷积层

与分类层相比，卷积层有几个明显的特征：1. 卷积层的输入和输出都是三维的（由多个二维的 feature map 组成）;2. 卷积层含有滑动窗口;3. 卷积层中的卷积核（权值）根据共享情况可分为私有卷积核和公共卷积核两种类型。虽然卷积层和分类层的数据维度不同，但运算操作是类似的。由于数据维度的增加，卷积层的复用性和分块运算方式会比分类层复杂。卷积层算法的伪代码见图 2.15，使用分块运算方式优化访存后，卷积层的运算方式见图 2.16。

在图 2.15 与图 2.16 中，Nxin 表示输入 feature map 在 x 轴上的神经元个数、Nyin 表示输入 feature map 在 y 轴上的神经元个数、 Ni 表示输入 feature map 的个数、 Nn 表示输出 feature map 的个数、 Kx 表示滑动窗口在 x 轴上的神经元个数、 Ky 表示滑动窗口在 y 轴上的神经元个数、 sx 表示滑动窗口在 x 轴上的滑动步长、 sy 表示滑动窗口在 y 轴上的滑动步长、输出 feature map 在 x 轴上的神经元个数 Nxout 由 Nxin， Kx 和 sx 共同决定、输出 feature map 在 y 轴上的神经元个数 Nyout 由 Nyin、Ky 和 sy 共同决定、 neuron[yout][xout][n]表示位于第 n 个 feature map 中坐标为（xout，yout）的神经元。下面介绍 DianNao 中卷积层分块算法的设计思路。

输入神经元

卷积层的滑动窗口是三维的，大小为 Kx × Ky × Ni。每个输出像素点由滑动窗口与一个同样大小的 kernel 进行卷积运算得到。卷积层对输入数据的复用有两种方式：第一种复用方式是由滑动窗口之间的重叠引起的，第二种复用方式是由属于同一输出像素点的不同输出 feature map 之间共用同一个滑动窗口引起的。

对于第一种复用，只要满足 sx < Kx 或者 sy < Ky，两个连续的滑动窗口内就会有数据重叠。更准确的说，对于某个窗口，它之后的连续$\frac{K_x \times K_y}{s_x \times s_y}$个窗口都会和它有数据重叠（重叠的数据量会越来越少）。同时，这个滑动窗口会被输出像素点的每个 feature map 复用。每个输出像素点一共有 Nn 个 feature map，所以一个滑动窗口会被$\frac{K_x \times K_y}{s_x \times s_y}$× Nn 个输出神经元复用。第二种复用与分类层对输入神经元的复用方式类似。

```
for (int yout = 0; yout < Nyin; yout += 1)
  for (int xout = 0; xout < Nxin; xout += 1){
    for (int n = 0; n < Nn; n++)
      sum[n] = 0;
    for (int ky = 0; y < Ky; y += 1)
      for (int kx = 0; x < Kx; x += 1)
        for (int i = 0; i < Ni; i += 1)
          // version with shared kernels
          sum[n] += synapse[ky][kx][n][i]
                    * neuron[ky + sy*yout][kx + sx*xout][i];
          // version with private kernels
          sum[n] += synapse[yout][xout][ky][kx][n][i]
                    * neuron[ky + sy*yout][kx + sx*xout][i];
    for (int n = 0; n < Nn; n++)
      neuron[yout][xout][n] = non_linear_transform(sum[n]);
}
```

图 2.155 卷积层原始运算方式

```
for (int yy = 0; yy < Nyin; yy += Ty) {
  for (int xx = 0; xx < Nxin; xx += Tx) {
    for (int nnn = 0; nnn < Nn; nnn += Tnn) {
      // -- Original code -- (excluding nn, ii loops)
      int yout = 0;
      for (int y = yy; y < yy + Ty; y += sy) { // tiling for y;
        int xout = 0;
        for (int x = xx; x < xx + Tx; x += sx) { // tiling for x;
          for (int nn = nnn; nn < nnn + Tnn; nn += Tn) {
            for (int n = nn; n < nn + Tn; n++)
              sum[n] = 0;
            // sliding window;
            for (int ky = 0; ky < Ky; ky++)
              for (int kx = 0; kx < Kx; kx++)
                for (int ii = 0; ii < Ni; ii += Ti)
                  for (int n = nn; n < nn + Tn; n++)
                    for (int i = ii; i < ii + Ti; i++)
                      // version with shared kernels
                      sum[n] += synapse[ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
                      // version with private kernels
                      sum[n] += synapse[yout][xout][ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
            for (int n = nn; n < nn + Tn; n++)
              neuron[yout][xout][n] = non_linear_transform(sum[n]);
          } xout++; } yout++;
} } } }
```

图 2.16 卷积层分块运算方式

输入神经元的分块方式与分类层类似，一般来说卷积层中的输入神经元（共有 Nxin × Nyin × Ni 个）不能全部存储在片上缓存中，所以需要对输入神经元进行分块。首先将 Nxin 分成大小为 Tx 的块，将 Nyin 分成大小为 Ty 的块，这样每块输入的大小就是 Tx × Ty × Ni。对于这样一块输入神经元包含若干个滑动窗口，其中的每个滑动窗口可以通过计算得到一个输出像素点的最终和。

由于 Kx 和 Ky 最多在 10 个左右，Ni 一般会在几个到几百个之间，所以一般来说滑动窗口可以完全存储在片上缓存中，不需要对 Ni 进行分块。当然，当 Ni 非常大时，会超出片上缓存的容量，此时需要对 Ni 进行分块（与分类层类似，将 Ni 分成大小为 Tii 的块，用 iii 来索引这些 Tii 块，由于运算单元每拍能计算 Ti 个输入，所以每个 Tii 块又被分成大小为 Ti 的块，用 ii 来索引这些 Ti 块）。

输出神经元

如果片上存储的容量足够，可以将一个输入滑动窗口完全存储在片上的话，那么每次都可以计算出一个输出神经元的最终和，所以每个输出神经元都因此的输出神经元不需要复用。

如果不能将一个输入滑动窗口完全存储在片上，那么与分类层的复用情况类似，可以将一个输出像素点的所有 feature map 根据片上缓存的大小分成大小为 Tnn 的块，在计算出第一个 Tnn 输出 feature map 块后，再计算第二个 Tnn 输出块，以此类推，直到计算出一个输出像素点的所有 feature map。这时，所有的输入块将被从内存读入 Nn/Tnn 次，每个输入块被复用 Tnn 次。

权值

对于公共 kernel 的卷积层，卷积核会被所有的输出像素点共用。所以当卷积核可以完全存储在片上时，可以被复用 Nxout × Nyout 次。

对于私有 kernel 的卷积层，权值各不相同并且没有复用，与分类层的情况类似。但和分类层不同的是，对于分类层来说，如果设计一个足够大的二级缓存，将权值全部存入二级缓存中，就能利用权值在层间的复用性。但对于私有 kernel 的卷积层来说，即使输入与输出之间的稀疏连接可以显著地减少私有 kernel 卷积层中的权值数量，但是对于大规模的卷积层，权值的数量依然可以达到上亿个，远远超出了二级缓存的容量，因此很难使用二级缓存增加权值的复用性。

实验结果中，由于 CONV3 是公共卷积核的卷积层，输入神经元、输出神经元、权值需要的带宽相对都不大，其中权值需要的带宽最大。而使用一级缓存进行优化后，由于一级缓存可以容纳全部的输入输出神经元和一个滑动窗口所需的部分权值，因此输入神经元和输出神经元可以得到全部复用，所需带宽下降显著。权值可以在部分运算中得到复用，所需带宽也有所下降。使用二级缓存优化后，由于二级缓存可以容纳全部的输入输出神经元和一个滑动窗口所需的所有权值，因此输入输出神经元和权值可以在全部运算中得到复用，所需带宽均下降显著。

而对于 CONV5，由于其为私有卷积核的卷积层，权值需要的带宽很大，输入神经元，输出神经元需要的带宽较小。而使用一级缓存进行优化后，由于一级缓存可以容纳部分输入输出神经元和部分权值，因此输入神经元和输出神经元可以得到部分复用，所需带宽下降显著，而权值得不到复用，所需带宽不变。使用二级缓存优化后，由于二级缓存可以容纳全部的输入输出神经元和部分权值，因此输入输出神经元可以在全部运算中得到复用，所需带宽均下降显著，权值依然得不到复用，所需带宽不变。

池化层

和卷积层类似，池化层的输入和输出都是三维的数据，包括多个二维的 feature map。与卷积层不同的是，池化层没有卷积核，即没有权值，另外，卷积层的滑动窗口是三维的，而池化层的滑动窗口是二维的，即在池化层中，每个输出神经元都是由同一个输入 feature map 中 Kx × Ky（滑动窗口大小）个输入神经元计算得来的。因此，在池化层中，输入 feature map 的个数等于输出 feature map 的个数。池化层算法的伪代码见图 2.17，使用分块运算方式优化访存后，池化层的运算方式见图 2.18。

```
for (int yout = 0; yout < Nyin; yout += 1)
  for (int xout = 0; xout < Nxin; xout += 1){
    for (int n = 0; n < Nn; n++)
      value[n] = 0;
    for (int ky = 0; y < Ky; y += 1)
      for (int kx = 0; x < Kx; x += 1)
        for (int i = 0; i < Ni; i += 1)
          // version with average pooling;
          value[i] += neuron[ky + y][kx + x][i];
          // version with max pooling;
          value[i] = max(value[i], neuron[ky + y][kx + x][i]);
    for (int n = 0; n < Nn; n++)
      neuron[yout][xout][n] = non_linear_transform(sum[n]);
}
```

图 2.17 池化层原始运算方式

```
for (int yy = 0; yy < Nyin; yy += Ty) {
  for (int xx = 0; xx < Nxin; xx += Tx) {
    for (int iii = 0; iii < Ni; iii += Tii)
      // -- Original code -- (excluding ii loop)
      int yout = 0;
      for (int y = yy; y < yy + Ty; y += sy) {
        int xout = 0;
        for (int x = xx; x < xx + Tx; x += sx) {
          for (int ii = iii; ii < iii + Tii; ii += Ti)
            for (int i = ii; i < ii + Ti; i++)
              value[i] = 0;
            for (int ky = 0; ky < Ky; ky++)
              for (int kx = 0; kx < Kx; kx++)
                for (int i = ii; i < ii + Ti; i++)
                  // version with average pooling;
                  value[i] += neuron[ky + y][kx + x][i];
                  // version with max pooling;
                  value[i] = max(value[i], neuron[ky + y][kx + x][i]);
        } } } }
        // for average pooling;
        neuron[xout][yout][i] = value[i] / (Kx * Ky);
        xout++; } yout++;
} } }
```

图 2.18 池化层分块运算方式

图 2.17 与图 2.18 中，Nxin, Nyin, Ni, Nn, Kx, Ky, sx, sy, neuron[yout][xout][n] 这些参数的含义与卷积层中对应的参数含义相同。

关于数据复用性，在池化层中，数据只有一种复用情况，也就是窗口滑动引起的输入重叠，这与前面的卷积层情况类似。唯一不同的是卷积层的窗口是三维的，而池化层的窗口是二维的。使用分块运算进行访存优化后的结果也与卷积层类似，但是由于这层的复用比较少，因此相比于卷积层，优化以后输入和输出的访存带宽总和要更高，分块算法带来的效果也没有那么好。

2.3.2 运算单元设计

由于 DianNao 需要支持包括分类层，卷积层，池化层在内的多种深度学习算法，所以需要设计运算单元来高效地对这些算法中的运算提供支持。

算术运算与流水线

DianNao 支持的几种深度学习算法中的运算均可分解为两到三个阶段：

卷积层和分类层的运算可以分解为三个阶段。第一阶段将输入神经元与对应的权值进行向量点乘；第二个阶段将第一个阶段向量点乘的结果累加；第三个阶段对第二个阶段的累加结果做非线性变换（激活函数）。非线性变换可以选择不同的函数，如 sigmoid(x)或 tan(x)。

池化层的运算可以分解两个阶段。第一个阶段根据不同的池化层种类，在几个输入神经元之间做求平均运算或者求最大值运算。第二个阶段与分类层同样是非线性变换，可以选用 sigmoid 或者 tan。

可以看到，每层算法的分解方式可以统一起来，使用同一个运算部件（NFU）完成。如果将第 n 个阶段的运算表示为 NFU—n，那么 NFU 需要以下基本运算：NFU—1 做乘法运算（对应卷积层和分类层），NFU—2 做加法运算（对应卷积层，分类层和求均值的池化层），移位运算（对应求均值的池化层）和选择运算（对应求最值的池化层），NFU—3 做非线性变换（对应卷积层，分类层和池化层）。对应三个阶段的运算，DianNao 使用了三级流水线。

为了提高 NFU 的性能，DianNao 将 NFU 的每个阶段都并行化，让它每拍可以计算多个数。在 4.2 节算法分析中，最内层循环是 i 和 n（分类层和卷积层算法中的 i，n 循环以及池化层算法中的 i 循环），即 NFU 每拍都可以完成对 Tn 个输出神经元的计算。这样，NFU—1 每拍能计算 Ti 个输入神经元 × Tn 个权值，需要 Ti × Tn 个乘法器。NFU—2 每拍能计算 Tn 个数的部分和相加，需要 Tn-1 个加法器，每个加法器有 Ti 个输入，为了并行化加法运算，每个加法器实际上是一棵加法树，每棵加法树共有 Ti—1 个加法器。对于池化层来说，NFU—2 还需要包含一个可以处理 Tn 个输入的移位器（用于做除法），和一个可以处理 Tn 个输入的选择器（用于求最值）。NFU—3 每拍能计算 Tn 个数的非线性变换，需要 Tn 个非线性变换单元。

NFU-3 功能实现

NFU—3 用于处理非线性变换，也就是激活函数（sigmoid 或者 tan）。因为直接使用逻辑门来实现激活函数的硬件开销很大，所以使用分段线性插值法实现，见图 2.19。

分段线性插值法的实现过程如下：首先根据输入数据的值确定输入数据所属的的分段；然后根据分段号查找这个段的系数(ai, bi)；最后根据系数(ai, bi)计算出变换结

果（等于输入数据 × ai + bi），见图 2.12。在硬件实现方面，需要使用两个 16 选 1 的多路选择器，一块 RAM，一个乘法器和一个加法器。其中，多路选择器用来选定分段的两个边界。16 个分段系数(ai,bi)存储在 RAM 中，通过改变 RAM 中的段系数(ai,bi)对不同的函数进行差值，可以改变 NFU—3 的功能，使得它可以对 sigmoid，tanh 和其他一些非线性函数进行支持。

数据的精度

关于数据的精度，DianNao 使用 16 位定点表示算术操作数，其中 6 位用来表示整数部分，10 位用来表示小数部分。根据先前文献中的结论，即使使用更短的位数（比如 8 位）来表示操作数，对神经网络识别精度的影响也不大。因为深度学习中数据的变化范围不太大，所以使用定点表示是足够的。32bit 浮点运算与 16bit 定点运算识别准确率对比见图 2.20。

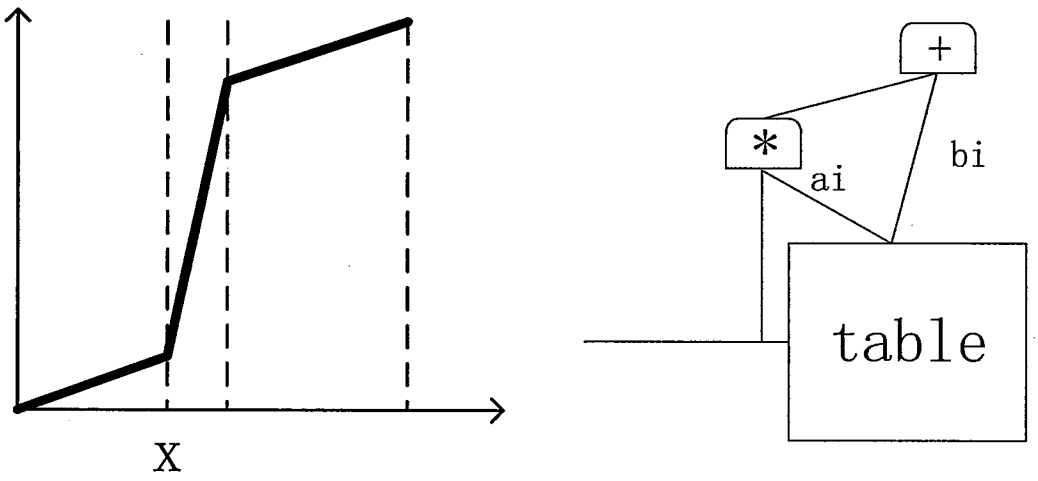


图 2.19 NFU 非线性变化实现

2.3.3 片上缓存设计

本节讨论深度学习处理器片上缓存的设计。为了对 4.2 节中讨论的数据复用性进行支持，我们为深度学习算法中输入神经元、权值、输出神经元三类数据分别设计了片上缓存并使用不同的 DMA 进行控制。

分散存储

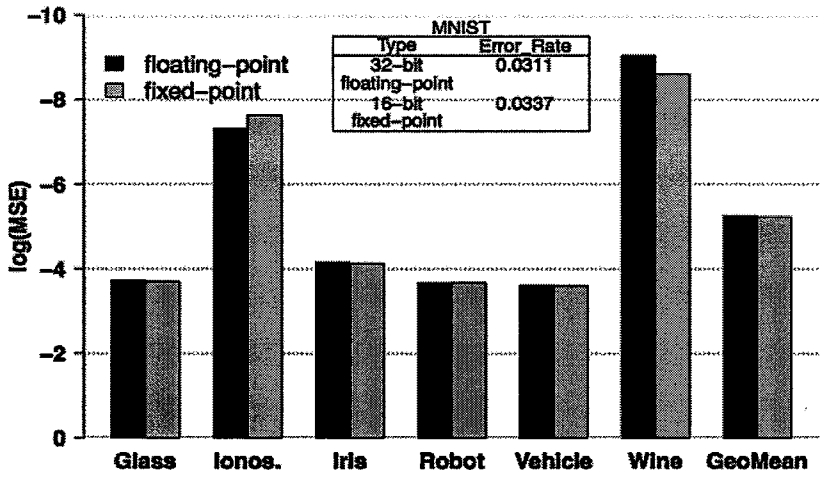


图 2.20 32bit 浮点运算与 16bit 定点运算识别准确率对比

DianNao 使用手动管理的缓存做片上缓存。这样可以很好地支持分块算法，得到比自动管理的 Cache 更高的性能。

深度学习算法中的数据一共可以分成三类，分别是：输入神经元，权值，输出神经元。DianNao 将这三类数据分别存储于不同的片上缓存，而不是在同一块片上缓存中（像 Cache 和内存那样）。DianNao 中，存储输入神经元的片上缓存叫作 NBin(Neuron Buffer in)，存储权值的片上缓存叫做 SB (Synapse Buffer)，存储输出神经元的片上缓存叫做 NBout(Neuron Buffer out)。

这种分散存储的优点首先是可以合理利用访存带宽。以访问中的读操作为例，如果只有一块缓存的话，也只有一种读宽度（类似于 Cache 行大小），那么无论将这个读宽度设置成多少，都不能达到最优，因为深度学习算法中不同类型数据的读宽度不一样，一种读宽度不能满足多类数据。

假设缓存器的读宽度与突触权值一致，也就是说每行宽度为 Tn × Tn × 2 字节，那么从 Tn × Tn × 2 字节的一行里读取读 Tn × 2 字节的数据时，会存在显著的能耗损失。因为真正需要的数据只有 Tn × 2 字节，而一共读取了 Tn × Tn × 2 字节，这种能耗损失会随着 Tn 的变大而变大。而另一方面如果假设读宽度与输入神经元一致，也就是说行大小为 Tn × 2 字节的话，那么从 Tn × 2 字节的一行里读取读 Tn × Tn × 2 字节的数据时，需要读 Tn 次，花费的时间太长。

而使用分布式存储时不会发生上面的问题。如果一个缓存器只存储一类数据，那么

可以将读宽度设置成相应的大小，因为运算器每时钟周期能计算 Tn × Ti 的数据，这样可以将 NBin 的宽度设置为 Ti × 2 字节，NBout 的宽度可以设置成 Tn × 2 字节，SB 的宽度设置成 in × Tn × 2 字节。分布式存储对每次读请求都可以提供合适的数据位宽，从而节约能耗，也不会增加访存时间。

分布式存储的第二个优点是可以避免处理访存冲突带来的开销。考虑这样一种情况，Cache 行为了支持高速读写突触权值，行大小需要设置成 Tn × Tn × 2 字节，同时需要控制 Cache 的尺寸，能满足这二者的唯一的方法就是使用高度关联。因此在解决访存冲突方面，只能使用多路组相连的 Cache。然而，在一个 n 路组相连的 Cache 中，高速读是通过预先猜测性地并行读取所有的 n 路数据实现的，这样做不仅会影响性能，并且会使能耗迅速增长。

所以，用一个多路组相连的 Cache 作为深度学习处理器的片上缓存将是一种能耗很大的解决方案。而分布式存储以及对数据复用性的先验知识可以使设计者预知可能会发生的数据冲突并以手动（根据算法设计硬件或编写指令）的方式解决，不需要因为 Cache 中复杂的算法耗费大量的能量。

DMA 控制

DianNao 使用 DMA 控制片上缓存的读写操作。由于片上缓存结构是分散存储的，所以相应地 DianNao 中有多个 DMA 分别进行控制，即对每块缓存都有一个 DMA。其中控制 NBin 和 SB 的 DMA 用来控制从内存载入数据到片上缓存，控制 NBout 的 DMA 用来控制从片上缓存读取数据并写回内存。

深度学习处理器对于 DMA 的控制使用微指令实现。为缓存 DMA 收到的微指令，每个 DMA 都有一个单独的指令队列。微指令由控制器发送至 DMA，然后被暂存在指令队列中。芯片工作时，DMA 从指令队列中取出微指令并执行，这样使得 NFU 与不同片上缓存之间的交互以及 NFU 运算之间可以以异步的形式进行，提高了效率。

利用输入数据的局部性

根据 4.2 节中的分析，可以将所有层的输入神经元分解成若干块，以便其中的一块能放到 NBin 中。这些输入块会在连续的多次计算中重复使用。当运算所需的数据不在 NBin 中时，需要通过相关的 DMA 从内存读取。当数据在 NBin 中时，相关的 DMA 不需要访存，可以直接复用以前的数据。由于片上缓存的行为完全由 DMA 控制，为了给 DMA 提供关于指令中所需的数据是否在 NBin 中的信息，指令中有一个标志位用来表示是否复

用。如果这一位数据的值为 1，表示需要复用以前的数据，不需要访问内存；如果这一位数据的值为0，则表示需要从内存读取。

在实际运算中，当将一个大小为Tii 的数据块读入到NBin 后，DianNao 会逐次计算这个大小为Tii 的数据块相对于 Tnn/Tn 个 Tn 输出块的部分和，像这样，一个大小为Tii 的数据块会被重复使用 Tnn/Tn 次。对 NBin 的循环利用是借助于一个寄存器来记录当前数据的位置（类似指针）实现的，这种方法类似于软件中的实现。

对于权值，只有公共 kernel 的卷积层才会有复用，其它情况均没有复用。SB 对突触权值的复用性支持与 NBin 类似。

利用输出数据的局部性

对于分类层和卷积层，一种可能的运算方式是：先使用 NBin 中每块输入神经元经过连续运算后得到 Tn 个输出神经元的部分和，然后这块输入神经元又用来计算另一个 Tn 大小的输出神经元块。不过这种运算次序会带来两个问题。

第一个问题，假设一个 Tii 输入块由 3 个 Ti 块组成，一个 Tnn 输出块由 2 个 Tn 块组成。如果当前时刻 NBin 中存在一个 Tii 输入块，下面要计算这个 Tii 块相对于某个 Tnn 输出块的部分和，则一共需要 6 次运算，具体的运算次序见图 2.21。

当 Ti^1相对于 Tn^2的部分和（记为 P_1）的计算完成后，接下来要计算 Ti^2相对于 Tn^1的部分和（记为 P_2），当完成 P_2的运算时，需要将 P_2与 P_1相加，然后继续计算 Ti^3相对于 Tn^1的部分和（记为 P_3），之后又要将之前 P_2与 P_1的和与 P_3相加。这样，一个 Tii 相对于 Tn－1 的部分和就全部完成了。接下来按照相同的次序计算 Tii 相对于 Tn－2 的部分和。完成之后，一个 Tii 相对于一个 Tnn 的部分和也全部完成了。再之后会从内存读入另一个 Tii 块存入 NBin，计算这个新 Tii 块相对于 Tnn－1 的部分和，当读取并计算所有的 Tii 块后，这个 Tnn 输出块就得到了它的最终和。

在对这一系列运算的处理中，有两个问题需要考虑。第一个问题是当 P_1计算完成后，应该在哪里存储。由于当 P_2计算完成后，需要和 P_1相加，如果之前将 P_1移出 NFU 流水线，等需要的时候再读进来，将造成时间和能耗损失。另外对 P_1这类的部分和使用非常频繁，而且经常会重复使用同一个部分和。所以为了解决这个问题，NFU－2 使用一组寄存器来暂存 P_1这类的部分和。

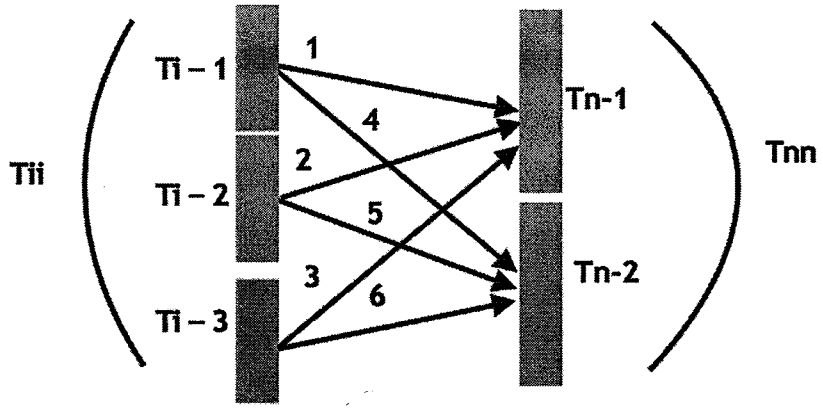


图 2.21 分类层运算次序

第二个类似的问题是当第一个 Tn 块的部分和计算完成后，应该在哪里存储。观察到当整个 Tnn 块的部分和计算完成后，会从内存读取一个新的 Tii 块，计算这个新 Tii 块相对于 Tn^1 的部分和，完成这步后，新 Tii 块相对于 Tn^1 的部分和和原有的 Tii 块相对于 Tn^1 的部分和相加，类似于第一个问题中提到的 P_1 与 P_2 相加。

第二个问题中的运算与第一个问题中运算之间的差别是第二个问题中的两次计算之间不是连续的，因为中间会间隔对 Tn^2 的运算。但是 P_1 与 P_2 之间没有其它运算，因此可以将原有的 Tii 块相对于 Tn^1 的部分和转存到 NBout 中，这样既不需要增加多余的存储单元，也不会影响性能，因为对原有的 Tii 块相对于 Tn^1 的部分和的使用不是连续的，当需要它时，可以通过 DMA 预加载。

虽然前面说到 NBout 的功能是用来存储将要写回到内存的输出神经元，但是使用 NBout 存储部分和也不会与 NBout 原有的功能之间发生访存冲突。因为实际上，只要不是所有的输入神经元都参与了某块输出的部分和计算，那么这块输出就没有完成计算，也就是没有生成最终和，所以这时控制 NBout 的 DMA 不需要向 NBout 写入数据，NBout 是空闲的。因此，可以把 NBout 当成临时的缓存来使用，将原有的 Tii 块相对于 Tn^1 的部分和转存到 NBout 中。

当然，为了使这些部分和能完全存入 NBout，需要对输出进行分块（每块大小为 Tnn，伪代码中的 nnn 变量用来索引这些 Tnn 块）。因此，NBout 不仅连接着 NFU—3 和内存，也连接着 NFU—2，NFU—2 中专用寄存器的数据可以存入 NBout，NBout 中的行也可以加载到 NFU—2 中的专用寄存器。

2.3.4 指令集设计

DianNao 的控制是由控制器 CP(Control Processor)发送指令到 DMA 和 NFU 完成的。如图 2.22，深度学习处理器的指令包含五个指令域：CP，SB，NBin，NBout 和 NFU。CP 从内存中读取指令，并将读取到的指令存储在 CP 中的 SRAM，再读取 CP 中的 SRAM 并发送相应的指令域到 NBin，SB，NBout，NFU 来控制它们运行。DMA 和 NFU 各自的指令域会存储在指令 FIFO 中，顺序执行。不同功能部件中指令的协同可以通过不同部件之间的握手协议实现。

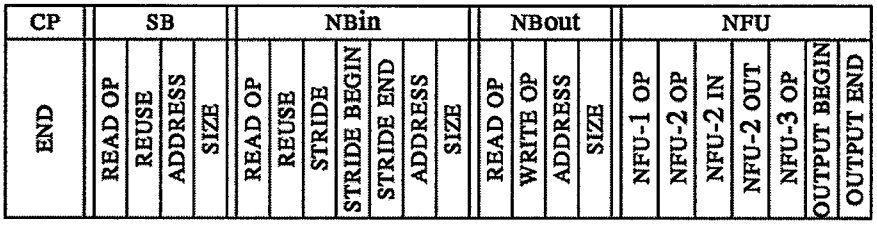


图 2.22DianNao 指令格式

深度学习的每一层都是通过一组指令完成的，每条指令可以完成卷积层和分类层伪代码中的 ii，i，n 循环操作，或者池化层伪代码中的 ii，i 循环操作，即一个 Tii 输入块相对于一个 Tn 输出块的运算量。由于 DianNao 只对几种深度学习算法提供支持，所以没有专门开发一个编译器，而是实现了一个指令生成器来通过神经网络的参数（神经元数量，滑动窗口大小等）生成每层的指令。

2.3.5 总体结构

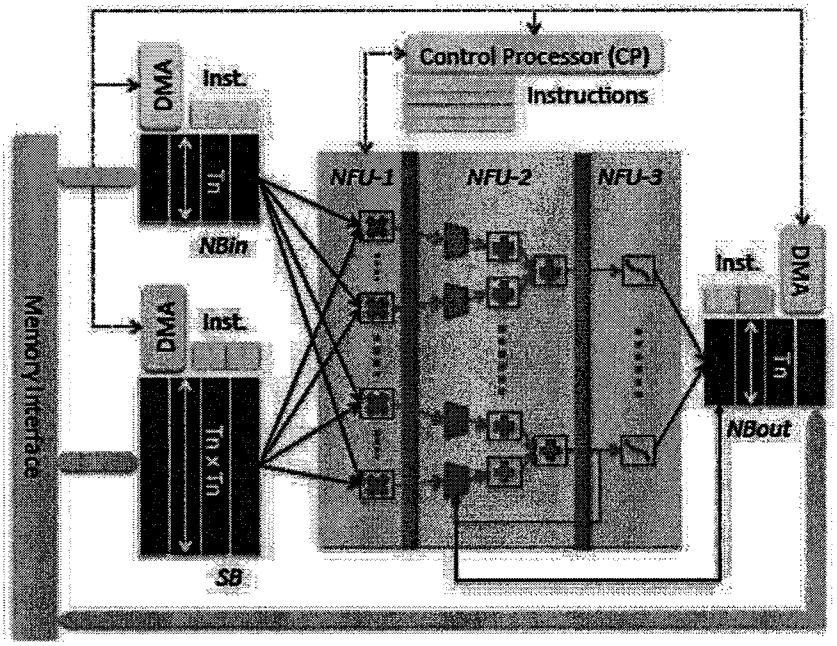


图 2.23 DianNao 结构图

DianNao 的结构图如图 2.23，其中主要包括运算部件、存储部件和控制部件三大部分。运算部件 NFU 由三级流水组成，第一级流水 NFU—1 包含 Tn × Ti 个乘法器，第二级流水 NFU—2 包含 Tn 棵加法树（每棵加法树由 Ti—1 个加法器组成）、一个 Tn 输入的移位器、一个 Tn 输入的选择器，第三级流水 NFU—3 包含 Tn 个乘法器和 Tn 个加法器。存储部分包三个数据缓存器（NBin，NBout，SB），每个缓存器都有一个数据队列和一个 DMA，数据队列和 DMA 用来控制数据的传输。控制部件 CP（Control Processor）负责整个结构的控制。

芯片工作方式如下：

1 芯片外部配置 CP，芯片开始工作。

2 CP 根据配置从内存读取指令至片上缓存，并开始发送指令至 NBin，NBout，SB，NFU。

3 NBin，SB 根据指令从内存读取输入神经元/权值至片上缓存，并将输入神经元/权值从片上缓存输出至 NFU。

4 NFU 根据指令对输入神经元/权值进行运算，并将结果从 NFU-2 或 NFU-3 输出至 NBout。

5 NBout 根据指令，接收 NFU 的输出存于片上缓存，并将最终结果存回内存。

2.3.6 DianNao 实验结果

芯片参数

DianNao 中使用了规模为 16 × 16 的运算单元（每拍处理 16 个输入神经元对应 16 个输出神经元）。所以 NFU-1 包含 256 个 16 位乘法器（用于分类层和卷积层）；NFU-2 包含由 15 个加法器组成的加法树（用于分类层，卷积层和使用平均值的池化层），还有 16 个移位器和比较器（用于池化层）；NFU-3 包括 16 个 16 位乘法器和 16 个加法器（用于分类层，卷积层和池化层）。

在运行分类层和卷积层时，NFU-1 和 NFU-2 每拍都会启动，因此每拍会进行 256 + 16 × 15 = 496 次定点运算，在 0.98GHz 的频率下相当于 452 GOP/s。在每层结束时，NFU-3 会启动，同时 NFU-1 和 NFU-2 也同时会启动，这时芯片的性能会在短时间内达到峰值，峰值性能是每拍处理 496 + 2 × 16 = 528 次定点运算（482 GOP/s）。

在 TSMC 65 纳米 GP 工艺库下，DianNao 的面积和功耗信息见表 2.1，其频率为 0.98GHz。当前时序的关键路径在从 NBin 和 SB 向 NFU 发送数据的逻辑上。片上缓存(NBin + NBout + SB + CP 中的指令 ram）的总容量为 44KB（CP 中 RAM 大小为 8KB）。占用面积和功耗的最多的部分为片上缓存(NBin/NBout/SB)，占用的百分比分别为 56%和 60%；NFU 也占用了大量的面积功耗资源，分别为 28%和 27%。

性能

图 2.24 为 DianNao 相对于 SIMD 模拟器的加速比。由于在 DianNao 中使用了 128bit 的 SIMD 模拟器，所以每拍可以进行 8 次 16 位运算。DianNao 每拍可以处理 496 个 16 位运算，即运算性能是 SIMD 模拟器的 62 倍。而实验结果显示单核深度学习处理器相对 SIMD 模拟器的加速比为 117.87 倍，相当于运算能力比例的两倍。这是由 SIMD 模拟器中使用的 Cache 的自动换入换出算法不能有效地对深度学习参数进行复用造成的。

组成部分	面积(um^2)	面积百分比	功耗(mw)	功耗百分比
整个加速器	3,023,077	100%	485	100%
Combinational	608,842	20.14%	89	18.41%
Memory	1,158,000	38.31%	177	36.59%
Registers	375,882	12.43%	86	17.84%
Clock network	68,721	2.27%	132	27.16%
Filler cell	811,632	26.85%		
SB	1,153,814	38.17%	105	22.65%
NBin	427,992	14.16%	91	19.76%
NBout	433,906	14.35%	92	19.97%
NFU	846,563	28.00%	132	27.22%
CP	141,809	5.69%	31	6.39%
AXIMUX	9,767	0.32%	8	2.65%
Other	9,226	0.31%	26	5.36%

表 2.1 DianNao 面积功耗信息

能耗

图 2.25 显示了 DianNao 与使用 SIMD 的 CPU 之间的能耗比。虽然能耗比平均达到了 21.08 倍，但与之前对于通用处理器和专用加速器性能的研究相比，还是少了一个数量级。比如 Hameed 等人的研究显示能耗比可以达到 500 倍[92]，而 Temam 等人的研究显示能耗比可以高达 974 倍[93]。这要是由于访存带来的能耗太高（前面的两组研究中没有考虑访存的能耗）。虽然像前面的两组研究一样，DianNao 中运算器和片上缓存的能耗都得到了显著的降低，但由于访存能耗明显的高于运算器和片上缓存的能耗，因此整个芯片的能耗受制于 Amdahl 法则，只能通过减少访存能耗来实现。DianNao 的能耗组成如图 2.26。

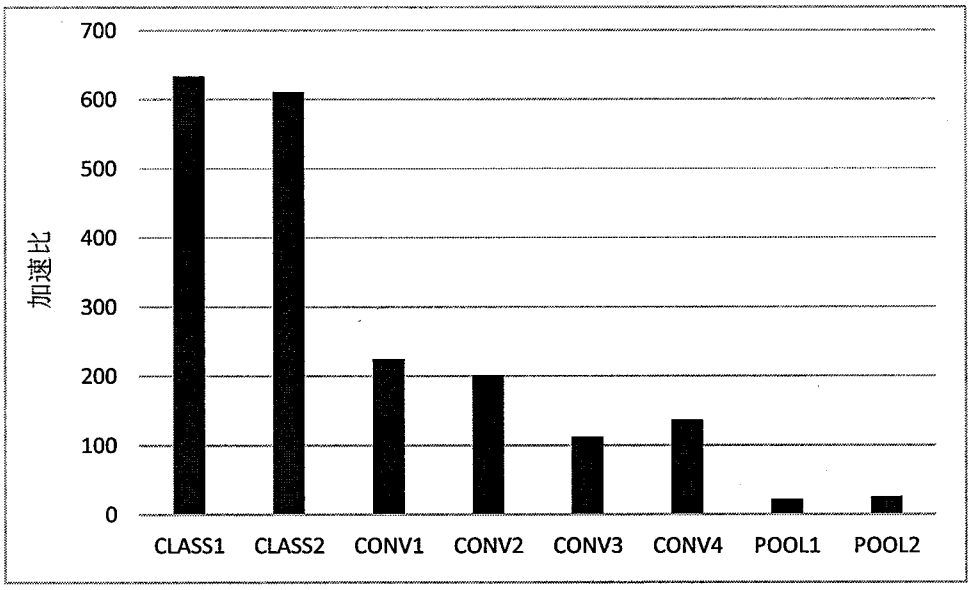


图 2.24 DianNao 与 128bit-SIMD CPU 性能比较

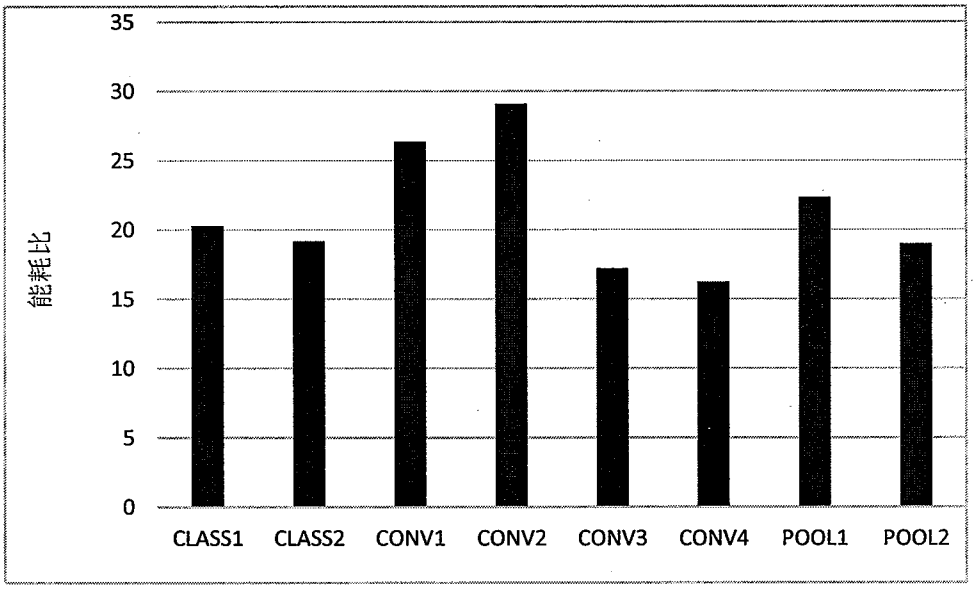


图 2.25 DianNao 与 128bit-SIMD CPU 能耗比较

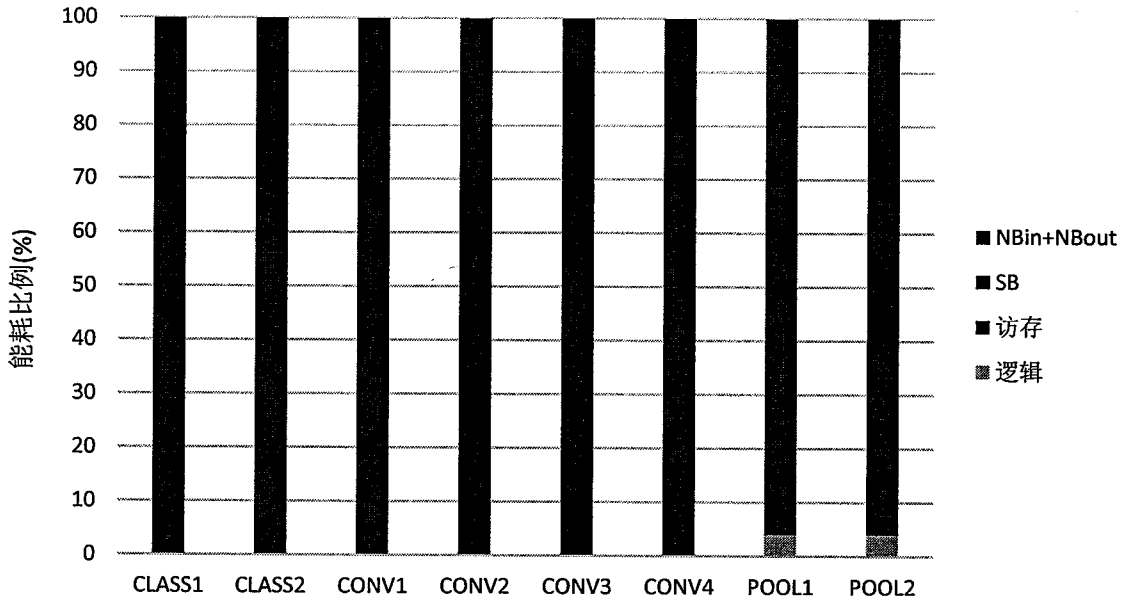


图 2.26 DianNao 能耗组成

第三章 实验平台及环境

本章主要介绍性能仿真与后端设计所采用的实验平台及环境。包括作为性能基准的CPU/GPU 性能仿真平台、用于进行单芯片性能仿真的 VCS 仿真平台、用于多芯片互联性能仿真的Booksim仿真平台和用于后端设计的逻辑综合工具DC(Synopsys Design Compiler)、布局布线工具 ICC（Synopsys IC Compiler）、流片时序功耗分析工具 PT（Synopsys Primetime）。本章组织形式如下：首先介绍本文选用的基准测试集，接下来介绍用于性能仿真的平台，最后介绍后端设计平台。

3.1 基准测试集

本文采用一组常见的神经网络模型作为基准测试集。其中包括 1 个完整的神经网络和10 个单层神经网络。

其中完整的神经网络来自于 2012 ImageNet 竞赛的冠军，是一个用于自然图像中多实体识别的神经网络。在我们用到的单层神经网络测试集中，CLASS1 用于实体识别和语音识别[52]、CLASS2, CONV1, POOL2, LRN1, LRN2 来自于上述的完整神经网络基准测试集、CONV2和 POOL1 来自于街景扫描算法（识别街景中的楼房，车辆等）[53]、CONV3 来自于 YouTube中的人脸识别算法（google）[54]、CONV4 来自于 YouTube 中的实体识别算法[55]，是目前为止实际使用到的最大规模的单层神经网络。

对于单层神经网络的具体参数，表 3-1 给出了描述；对于完整神经网络的具体参数，表 3-2 给出了描述。

3.2 CPU/GPU 性能仿真平台

Cuda-convnet 基于 C++/CUDA 编写，采用反向传播算法的深度卷积神经网络实现。2012年 cuda-convnet 发布，可支持单个 GPU 上的训练，基于其训练的深度卷积神经网络模型

在 ImageNet LSVRC-2012 对图像按 1000 个类目分类，取得 Top 5 分类 15%错误率的结果；2014 年发布的版本可以支持多 GPU 上的数据并行和模型并行训练。

为了体现 GPU 与使用 SIMD 优化后 CPU 之间的性能差别，我们将它与 CUDA Convnet 的 C++版本在使用 SIMD 优化后的 Intel CPU 上的性能做了对比，结果见图 3-1。

对于 C++版本的 CUDA Convnet，我们也对使用 SIMD 的版本和未使用 SIMD（未开启 SIMD 编译）的版本做了对比，确认了使用 SIMD 的版本平均提速 4.07 倍，因此编译器可以有效地支持 SIMD。

Layer	Nx	Ny	Kx	Ky	Ni	No	Synapse
CLASS1					2560	2560	12.5MB
CLASS2					4096	4096	32MB
CONV1	256	256	11	11	256	384	22.96MB
CONV2	500	375	9	9	32	48	0.24MB
CONV3	200	200	18	18	8	8	1.29GB
CONV4	200	200	20	20	3	18	1.32GB
POOL1	492	367	2	2	12	12	0.24MB
POOL2	256	256	2	2	256	256	
LRN1	55	55			96	96	
LRN2	27	27	2	2	256	256	

表 3.1 本文使用的单层神经网络基准测试集

	Layer	Nx	Ny	Kx	Ky	Ni	No
1	CONV	224	224	11	11	3	96
2	LRN	55	55			96	96
3	POOL	55	55	3	3	96	96
4	CONV	27	27	5	5	96	256
5	LRN	27	27			256	256
6	POOL	27	27	3	3	256	256
7	CONV	13	13	3	3	256	384
8	CONV	13	13	3	3	384	384
9	CONV	13	13	3	3	384	256
10	CLASS					9216	4096
11	CLASS					4096	4096
12	CLASS					4096	1000

表 3.2 本文使用的完整神经网络基准测试集

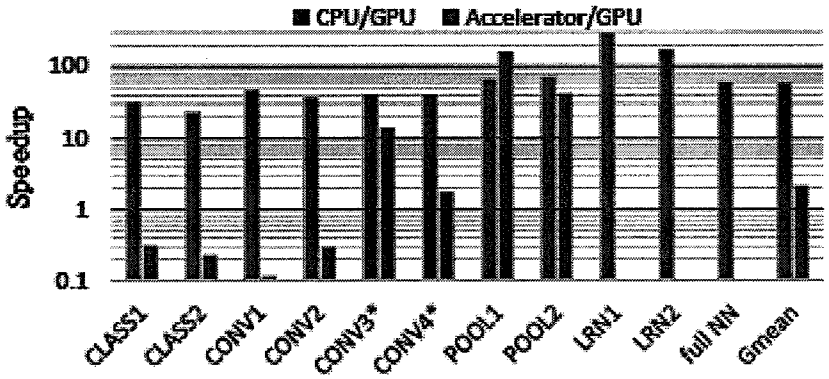


图 3.1 GPU 相对于 CPU（SIMD）的加速比

3.3 芯片性能仿真平台

3.3.1 单芯片性能仿真平台

VCS

VCS[77]是一种 Verilog 编译器，支持 OVI 标准的 Verilog HDL 语言，SDF 和 PLI。

在单芯片性能仿真中，本文使用 verilog 实现了深度学习处理器的单芯片结构，并使用 VCS 对其进行了编译和仿真。同时，我们根据深度学习处理器的指令集使用 c++编写了对应的指令生成器，用来根据指定的神经网络结构生成指令。

对于每种基准测试集，我们首先使用指令生成器生成指令，然后将生成的指令和随机生成的输入神经元/权值作为输入，开始使用 VCS 仿真并记录拍数；最后指令执行完毕，结束仿真后，记录总使用拍数。

3.3.2 多芯片互联性能仿真平台

Booksim

本文多芯片互联部分的实验采用 booksim[56]作为模拟器。Booksim 模拟器是由斯坦福 大学的 Bill Dally 教授所领导的 CVA 小组开发的，其最开始是为 Dally 和 Bowles 撰写的书籍《Principle and Practices of Interconnect Networks》设计的配套模拟器。后来 CVA 小组成员又逐渐增加了许多功能。Booksim 模拟器由于其代码易读性、模块性和书籍配套性被大量的片上网络研究人员采用。

我们在 Booksim 中使用 Orion2.0[57]作为能耗模型。

3.4 物理设计仿真平台

在本文中，我们对本文设计的芯片进行了物理设计来评估其频率，面积和功耗。本节首先介绍本文中物理设计使用的工具，然后介绍了本文使用的物理设计流程。

3.4.1 后端设计工具

Design Compiler [78]为 Synopsys 公司推出的逻辑综合工具，可以根据约束和设计描述，对特定的工艺库将设计描述综合成门级电路。本文使用 Design Compiler 作为后端设计中的逻辑综合工具。

IC Compiler [79]是 Synopsys 推出用于布局布线的后端工具。他将物理综合的过程并入整个布局布线过程，得到了很好的效果。本文使用 IC Compiler 作为后端设计中的布局布线工具。

StarRC[80]是 Synopsys 公司推出的提取寄生参数的工具。它可以为定制数字电路、SoC、内存 IC 和数模混合电路的设计提供高性能，硅准确的寄生参数提取，并可为包括 16nm 在内的芯片设计工艺提供物理信息建模。本文使用 StarRC 作为后端设计中的提取寄生参数工具。

PrimeTime [81]是 Synopsys 公司推出的时序检查工具。作为静态时序分析工具，PrimeTime 可以为设计提供以下的设计检查和时序分析：建立和保持时间的检查(setup and hold checks) 、时钟门的检查(clock-gating checks)、组合反馈回路(combinational feedback loops) 、时钟脉冲宽度的检查、unclocked registers、未约束的时序端点(unconstrained timing endpoints) 和基于设计规则的检查，包括对最大传输时间、最大扇出和最大电容的检查等。本文使用 PrimeTime 作为后端设计中的静态时序检查工具。

3.4.2 后端设计流程

图 3-2 描述了本文后端设计的流程：

1. 使用 Design Compiler 读入时序约束文件和标准单元库后，对 RTL 代码进行逻辑综合，将综合后的门级网表输出至 ICC。

2. 使用 ICC 进行读入时序约束文件，标准单元库，工艺文件后，读入 Design Compiler 输出的网表文件。然后对设计进行布局规划、布局、时钟树综合、布线后，将 Milkyway lib (Synopsys 专用的一种物理设计文件存储格式) 输出至 PrimeTime 和 StarRC。

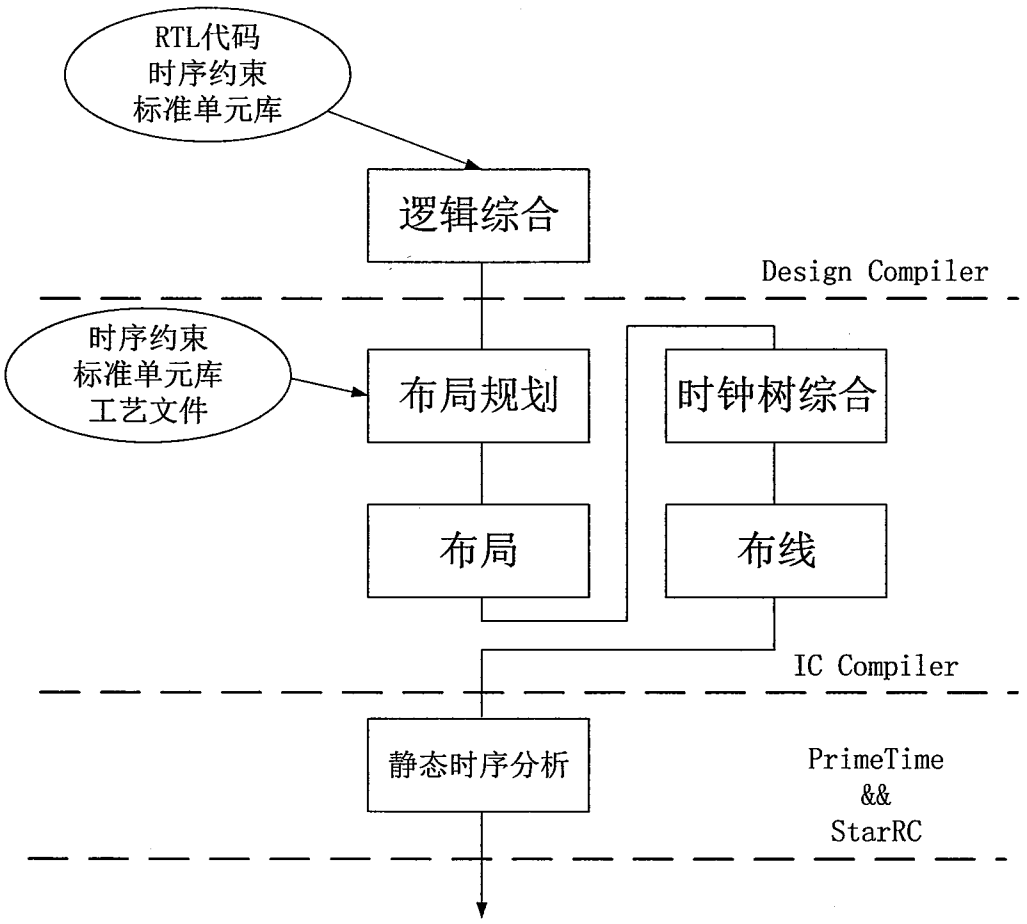


图 3.2 后端设计流程

3. 使用 StarRC 抽取 RC 信息后，使用 PrimeTime 读入 IC Compiler 输出的 Milkyway lib，并使用报告时序及功耗信息。

4. 关于后端设计实验数据：面积使用 ICC 布局布线后的 die area 面积（面积利用率为 65%）、时序为 PT 报告的结果、功耗为将翻转率设置为 100%后，PT 报告的结果。

第四章 深度学习处理器多核结构设计

上一章中，我们设计了一种单核深度学习处理器，在很低的能耗下获得了和与Nvidia 高端通用计算图形处理器 K20[101]类似的性能。本章在上一章的基础上扩大芯片的规模，设计一款高性能的多核深度学习处理器。

本章首先介绍了相关工作的进展，并结合具体场景和问题对本章的工作进行了分析。接着探讨了在进一步提升深度学习芯片性能的过程中遇到的难点，并指出了解决思路和解决方案。最后我们结合以上的分析设计了一款多核深度学习处理器 DaDianNao，并对其进行了逻辑实现和物理设计。

模拟实验结果显示，在 ST28 纳米工艺下，DaDianNao 的面积为 67.73 平方毫米。对于本文选用的测试集，其平均性能为 Nvidia K20 的 21.38 倍，平均能耗相比 Nvidia K20 降低了 330.56 倍。

4.1 相关工作

虽然芯片的制作工艺和体系结构在不断的发展，但近十几年来 CPU 单核性能的提升速度却在逐渐变慢。图 4.1 为康奈尔大学提供的 CPU 性能发展趋势图。可以看到，从 2005 年以来，CPU 单线程性能的提升明显减缓（多核性能却一直在提升）。

单核性能难以提升有两方面的原因。一方面，芯片主频难以继续提升。由于 $Power=nCV^2$，其中电压 V 又与频率正相关，因此功耗的提升会更快，从而降低了芯片的性能功耗比。这样除了会浪费能源和降低移动设备的续航能力外，功耗变高带来的散热问题也会影响到芯片的封装。例如 Intel 在发表奔腾 4 时宣布 NetBurst 架构可以运行在 10GHz，然而 NetBurst 架构在 3.8GHz 时就遇到了高功耗问题导致无法散热，导致 Intel 放弃了继续提升单核频率，转向使用多核技术提升处理器性能。

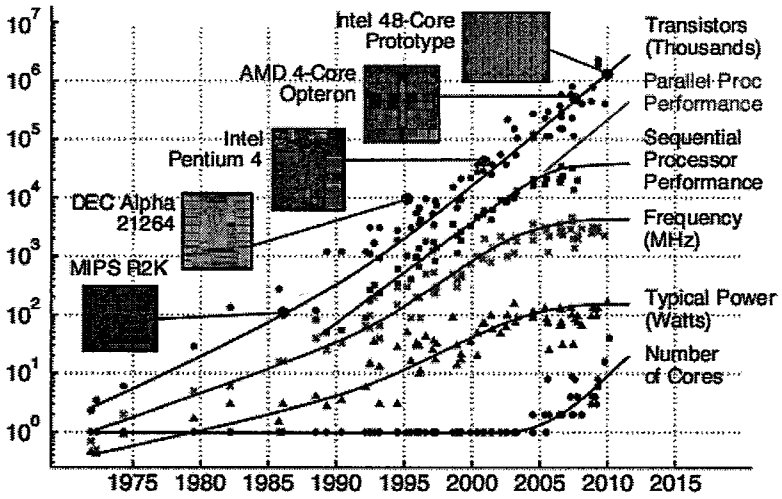


图 4.1 CPU 性能发展趋势

另一方面，虽然可以从体系结构角度上使用超标量，乱序执行，优化 Cache 替换策略等方式对单核通用处理器性能进行优化。但事实上这些技术经过多年的发展以后已经比较成熟，很难继续取得突破性的进展。

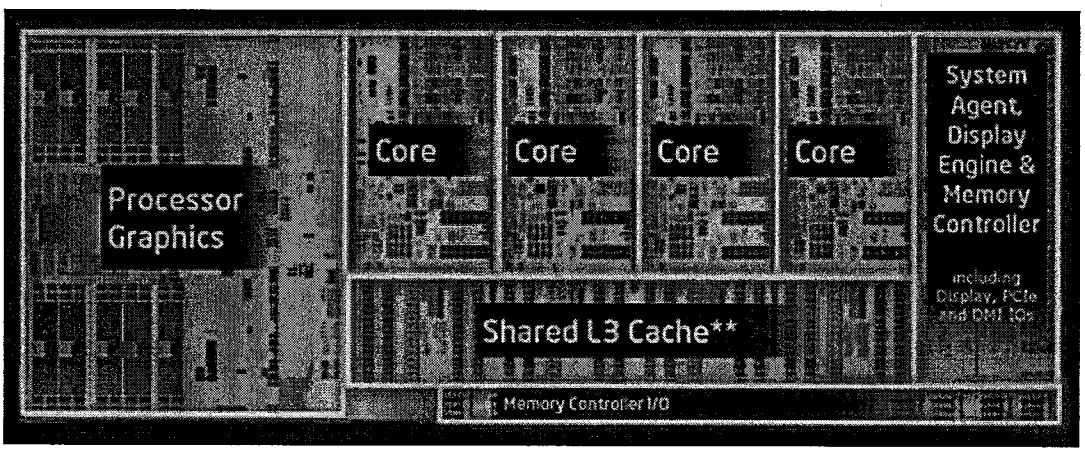


图 4.2 Intel Core i7-4770 处理器结构

因此，自 2005 年以来，Intel，AMD 等公司都停止了继续提升单核处理器性能的做法，开始设计和生产多核通用处理器。至今为止，英特尔（Intel）公司 2012 年推出了 50 核 Phi 众核处理器[102]，AMD 公司 2011 年推出了 16 核的 Opteron 6200 处理器[103]，ARM 公司于 2012 年推出了 64 核的 Centip3De 处理器[104]。图 4.2 展示的是

Intel Core i7 系列中的 Core i7-4770 处理器，其核心部分由 4 个共享三级 Cache 的处理器核组成。

相比于单核处理器，多核处理器可以并行处理多条指令，很适合通用处理器的多任务应用场景，即同时要处理操作系统，图像，音频等多种请求。因此虽然多核处理器具有需要并行编程支持，对于单个进程的处理速度提升不明显等不足，但已经成为了处理器发展的一种主流趋势。

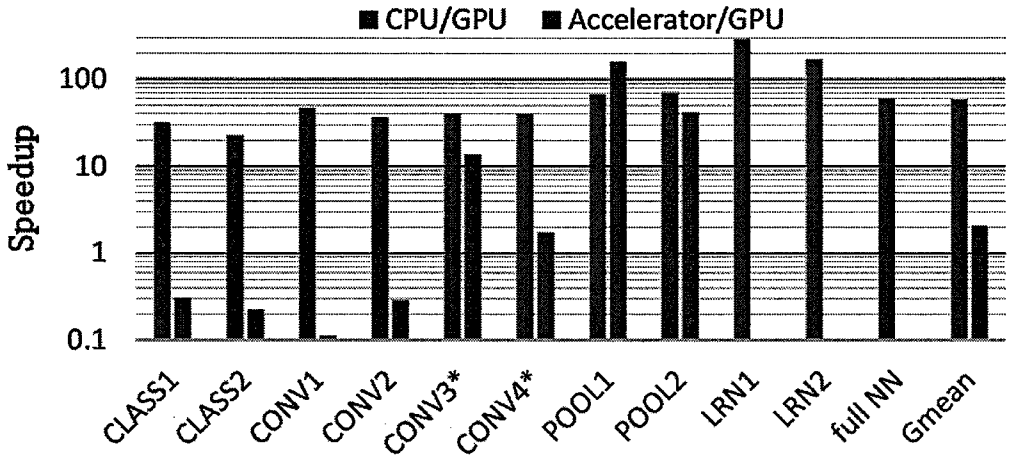


图 4.3 GPU CPU 和单核深度学习处理器性能比较

4.2 概述

虽然我们在上一章中尽可能地发掘了定制芯片的潜能，但由于单核深度学习处理器的规模比较小（65nm 工艺下面积为 3 平方毫米），导致它相对于 GPU 没有性能上的优势。单核深度学习处理器，通用处理器与 Nvidia K20 对于本文选用的测试集的性能见图 4.3。可见对于不同的神经网络算法，单核深度学习处理器的平均性能只有 gpu 的 0.47 倍。在本章中，为了提高性能，我们扩大了单核深度学习处理器的规模，设计了多核深度学习处理器 DaDianNao。

DaDianNao 总体结构

如图 4.4，DaDianNao 的核心部分由一系列被称作“tile”的核心组成，其中包括 1

个位于芯片中央的 central tile 和若干个 leaf tile(DaDianNao 中为 16 个)。所有 tile 之间通过 H 树连接。每个 leaf tile 接受相同的输入神经元作为输入，并且负责不同的输出神经元。输入神经元通过 H 树从 central tile 广播至 leaf tile，并且 leaf tile 中的输出神经元通过 H 树传回 central tile。除此之外，DaDianNao 周围有 4 块用于片间通信的高速 IO 模块，

Central tile 中含有用于片间通信的路由模块和 2 块 eDRAM，一块 eDRAM 用作输入神经元的片上缓存，另外一块用作输出神经元的偏上缓存。整个芯片中的所有输入神经元和输出神经元都缓存在 central tile 中。

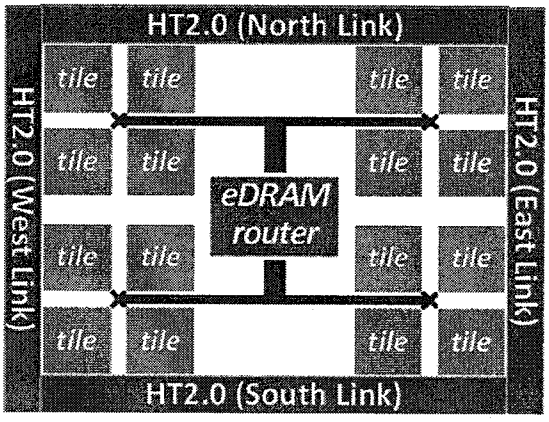


图 4.4 DaDianNao 总体结构

Leaf tile 中包含运算单元 NFU 和权值的片上缓存。由于 DaDianNao 由 16 个 leaf tile 组成，总的运算能力为 64x64，所以每个 leaf tile 负责计算 4 个输出神经元，也就是说 leaf tile 中 NFU 的运算能力为 64x4。另外，为了提高访问 eDRAM 的效率，我们将 eDRAM 分成了 4 个 bank，芯片工作时会交替地访问这些 bank（central tile 中也使用了相同的处理方式）。

在优化单核深度学习处理器性能的过程中，也就是在设计 DaDianNao 体系结构的过程中，我们主要解决了两个问题。第一个是由于芯片的访存带宽不足引起的性能瓶颈，第二个是芯片的内部带宽过大引起的拥堵问题。

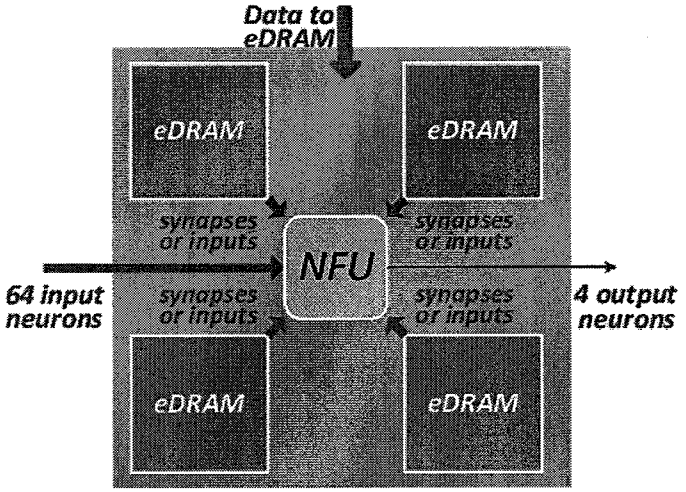


图 4.5 leaf tile 结构

4.3 针对高访存带宽的设计

4.3.1 无访存设计

第四章中我们分析了单核深度学习处理器的性能。可以看出，单核深度学习处理器在处理分类层和私有 kernel 的卷积层时性能较低。经过分析，我们发现原因是这两种深度学习算法对权值的总需求量很大，并且这些权值无法被复用，导致每次都需要从内存中读取权值，形成了访存瓶颈。

对于一个可以每拍处理 16 个输入神经元对应 16 个输出神经元的运算的 NFU，在主频为 0.98GHz 的情况下，需求的访存带宽为 467.30 GB/s。与此对应的是，NVIDIA K20M GPU 的内存接口为 320bit，在 2.6 GHz 主频下可以每半个时钟周期完成一次操作，总带宽为 208 GB/s；而 DDR4-3200 的带宽只有 25.6GB/s。

可见即使对于单核深度学习处理器，如果 NFU 满负荷运转，其需要的访存带宽要比内存带宽高一个数量级，并且也要高于显存的带宽。因此在不解决访存瓶颈问题的情况下，提高芯片规模对提升性能的作用有限。而且对于如此大的带宽差距，使用提高访存带宽的方式解决问题已经不现实。

我们采用了提高数据复用性的办法解决问题。由于导致访存瓶颈的原因是过多的权值参数，因此如果能将权值全部存储在片上缓存中，那么虽然在一次运算过程中这些权

值只能被用到一次，但是无论是在前馈运算，还是在反向训练中，我们可以在多次运算之间复用这些权值。

对于前馈运算，在运算开始时，先将所有权值从内存读入片上缓存，这是唯一一次有关权值的访存操作。后面每次进行前馈运算时，都只需要读入输入神经元，在运算完成后再将输出神经元存回内存即可。由于输入/输出神经元的数据量相比于权值数据量要小得多（大概是权值的 1/2 次方），所以这样访存不会成为瓶颈。

比如对于上文中的例子，NFU 每拍处理 16 个输入神经元对应 16 个输出神经元的运算，主频为 0.98GHz。如果使用一个 5 层的神经网络进行识别（实际情况下常常多于 5 层），那么需求的访存带宽为 5.84GB/s，大概是 DDR4-3200 带宽的五分之一，可见访存不再成为瓶颈。

对于反向训练，在运算开始时，先将所有权值从内存导入片上缓存。后面每次进行反向训练时，都只需要读入样本的输入和标签，在训练完成后将权值写回内存即可。虽然反向训练中不仅读入样本的输入，还需要读入标签，但对于每组输入/输出神经元，都需要进行前向和反向运算两个步骤，实际运算量多于前馈运算运算量的两倍，所以与前馈运算类似，访存依然不会成为瓶颈。

由于除了在运算开始时需要将神经网络参数从内存导入片上，结束时需要将神经网络参数从片上缓存存回内存之外，在整个运算过程中不需要访存，所以我们称这种设计为无访存设计。在无访存设计中，所有的神经网络参数都均得到了复用，此时访存带宽将不再成为性能瓶颈。

4.3.2 片上缓存设计

由于我们采用了无访存设计的设计思想，所以需要把所有的神经网络参数存储在片上。第三章表 3.3 中列出了本文使用的测试集的规模，可以看到每层神经网络的规模从小于 1MB 到 1GB 左右不等，大多数是几十 MB，因此我们的片上缓存也至少要达到几十 MB 的量级。但是与此对应的是，Intel i7-4770 处理器的片上缓存容量只有 8MB，由此可见深度学习处理器对片上缓存容量的需求很大。为了实现如此大的片上缓存，我们采用了三方面的手段。

1.使用先进工艺

制约片上缓存容量最主要的因素是芯片面积。而对于同样的设计，占用的面积与芯片制程的平方成反比。所以使用更先进的工艺是增加片上缓存容量最直接的办法。但由

于现阶段工业界最先进的 16 纳米工艺实现困难，所以本文选择在 32 纳米工艺下对 DaDianNao 进行设计。

2.为片上缓存分配更大比例的面积

根据之前的分析，在片上缓存中存储所有权值是 DaDianNao 性能的保障。如果不能做到这一点的话，会导致权值失去复用性，访存带宽再次成为性能瓶颈。因此我们要为片上缓存分配尽量大的芯片面积。在后面的实验结果中可以看到，DaDianNao 中片上缓存占用了将近一半的面积。

3.使用 eDRAM 替代 SRAM

eDRAM(embedded DRAM,增强动态随机存取存储器),是一种可以嵌入在 ASIC 上 DRAM。相比于用于内存中的 DRAM，eDRAM 单位容量占用的面积较大，但性能更高；相比于 SRAM，eDRAM 性能较低，但单位容量占用的面积较小。

虽然使用 SRAM 作片上缓存也是合适的,但对于 DaDianNao 所需要的大规模存储,SRAM 没有足够的密度，占用芯片面积较大。比如在 28nm 工艺下，一个 10MB 的 SRAM 存储器需要 20.73 平方毫米的面积[88]，而一个容量相同的 eDRAM 存储器，却只需要 7.27 平方毫米[89]，存储密度高于前者 2.85 倍。同时，eDRAM 的能耗也显著低于 SRAM[90]。所以本文选用了 eDRAM 作为神经网络参数的片上缓存。

然而 eDRAM 有三个显著的缺点：比 SRAM 有更高的延迟、有损读取、定期刷新[91]。这些缺点使得 eDRAM 的性能较低，相比于 SRAM，eDRAM 不能保证每拍都可以被读写。为了弥补 eDRAM 的不足，使得 NFU 每拍都可以得到输入，我们把 eDRAM 分成 4 个 bank,并将权值交叉排布在四块片上缓存中，保证每 4 个相邻的权值存储在不同的 eDRAM 中，这样芯片工作时会交替地访问这些 bank。

4.4 针对高内部带宽的设计

4.4.1 连线拥堵问题

在上一节中，我们解决了访存带宽不足的问题，因此 NFU 的规模不再会受到访存速度的限制。我们可以通过增加 NFU 的规模提升 NFU 的运算能力，从而是提高整个芯片的性能。DaDianNao 中，我们将 NFU 的性能提升为单核深度学习处理器的 16 倍。也就是说我们的 NFU 从每拍处理 16 个输入神经元对应 16 个输出输出神经元的运算提升为每拍可以处理 64 个输入神经元对应 64 个输出输出神经元的运算。

我们在ST28nm LP工艺下对以上的设计进行了布局布线，结果如图4.6。

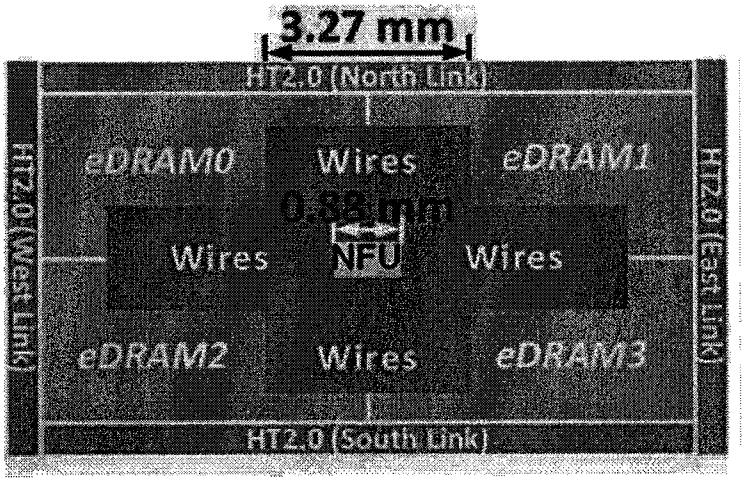


图 4.6 单NFU深度学习处理器版图示意图

可以看到，NFU占用的面积很小，只有0.78mm2（0.88mm×0.88mm）。但工艺要求片上的金属线间需要有0.2um的线间距，而且水平和竖直方向上分别只有4层金属。另外，对于运算能力为64x64的NFU，连接NFU和eDRAM的线宽为65536bit。

根据计算，这些导线宽度为65536*0.2/4 = 3.2768mm（参照图4.6）。导线占用的总面积为4×3.2768×3.2768−0.88×0.88 = 42.18mm2，这几乎和全部eDRAM、NFU、I/O接口占用面积的总和一样多。这显然是对面积资源的浪费，并且随着NFU规模的继续扩大，连线面积占整个芯片面积的比例会越来越大。

4.4.2 连线拥堵解决方案

我们可以通过分析算法特性找到这个问题的解决办法。由于在深度学习算法中，权值有两个特点：

1. 具有局部性。也就是说对于不同的输入神经元/输出神经元对，参与运算的权值都不同。所以我们不需要将存储权值的片上缓存放置在空间上接近的位置，可以将权值在芯片上分散存储。

2. 占用大部分的片上带宽。由于对一个运算能力为Ti*Tn的深度学习处理器，输入神经元、输出神经元、权值需要的连线宽度分别为Ti*16、Tn*16、Ti*Tn*16。可见权值和运算器之间的数据通路占用了绝大部分的片上带宽，如果解决了这部分的连线拥堵问

题，也就解决了整个芯片的连线拥堵问题。

所以我们可以以分散存储权值为基础，设计多核结构。首先将存储权值的片上缓存分为若干部份，分散分布在芯片上，然后将NFU也相应地分割，和对应的存储权值的片上缓存摆放在一起。这样就可以将存储权值的片上缓存与NFU之间的数据通路分成若干小部分，解决了拥堵问题。

这样引出了另一个问题：输入/输出神经元是不具有局部性的，如何在多核之间共享这些信息。显然在每个核中都保存一份副本的做法是不合适的，这样不光会造成面积上的浪费，并且维持多份存储副本间的一致性也需要开销。观察到输入输出神经元占用的带宽较低，我们可以采用类似总线的设计来向这些核广播输入神经元。而为了充分用总线带宽，同一时刻所有核心需要的输入神经元最好是相同的。

根据以上的分析，连线拥堵问题的解决方案如下：

1. 采用多核结构设计芯片，每个核由分散摆放的权值片上缓存与 NFU 组成，叫做 leaf tile。所有的运算任务和权值的片上缓存都由 leaf tile 完成。

2. 由于NFU随着leaf tile被分到了多个核中，所以需要在leaf tile之间划分运算。运算的划分方式为不同的leaf tile负责计算不同的输出神经元。

3. 由于要将输入神经元广播至各个leaf tile，同时要接收leaf tile传回的输出神经元，所以将存储输入神经元和输出神经元的片上缓存放置在芯片中央。放置在芯片中央的两块缓存与用于片间通信的路由模块组成central tile。

4. 使用H树连接central tile与所有的leaf tile。central tile每拍通过H树向所有 leaf tile 广播相同的输入神经元，leaf tile 通过相同的输入神经元计算不同的输出神经元，并传回central tile。容易看到，由于每一拍都有central tile到leaf tile，leaf tile到central tile两个方向的数据要通过H树，所以H树的位宽为NFU输入的位宽加上NFU输出的位宽。在DaDianNao中，H树的位宽为(64+64)*16=2048bit。

可见，在实际运行过程中，我们可以将多核结构的DaDianNao在逻辑上看成与单核深度学习处理器等价的结构进行控制。即central tile中的两块片上缓存相当于单核深度学习处理器中的NBin和NBout，所有leaf tile中存储权值的缓存的总和相当于单核深度学习处理器中的SB，所有leaf tile中的NFU的总和相当于单核深度学习处理器中的NFU。

如前所述，我们可以使用与单核结构类似的指令集和算法映射方式来对 Dadianno 进行控制，也就是把DaDianNao看成一个单核结构来控制。当然，如果未来需要支持新算法或者有性能上的需求，需要更灵活的控制方式，也可以将其看做一个分布式的多核

系统来进行控制。

4.5 DaDianNao

根据以上分析，我们设计的多核深度学习处理器 DaDianNao 中包含 16 个 leaf tile 和一个 central tile。

Leaf tile 中包含 4 块 1024 行 x4096 位的 eDRAM。因此 leaf tile 中的 eDRAM 总容量为 4×1024×4096 = 2MB。Central tile 中包含 2 块 4096 行 x4096 位的 eDRAM。因此，central tile 中的 eDRAM 总容量为 2×4096×4096 = 4MB。这样，DaDianNao 中总的 eDRAM 容量是 16×2 + 4 = 36MB。

为了避免电路和时间同步异步传输的开销，我们决定在芯片的核心部分使用与 eDRAM 使用相同的时钟频率，即 606MHz。因此对于 16 位的操作，芯片的峰值性能为 16*（288+288）*606=5.58 TeraOps/s。对于 32 位的操作，芯片的峰值性能为 16*（144+72）*606=2.09 TeraOps/s。

表 4.1 列出了 DaDianNao 的部分参数。

Parameters	Settings	Parameters	Settings
Frequency	606MHz	tile eDRAM latency	~3 cycles
# of tiles	16	central eDRAM size	4MB
# of 16-bit multipliers/tile	256+32	central eDRAM latency	~10 cycles
# of 16-bit adders/tile	256+32	Link bandwidth	6.4x4GB/s
tile eDRAM size/tile	2MB	Link latency	80ns

表 4.1 DaDianNao 芯片参数

4.6 实验结果

我们使用 Verilog 对 DaDianNao 进行了实现，并且使用 ST（意法半导体）28nm LP（Low Power）工艺库进行了逻辑综合和布局布线。本节首先介绍布局布线后 DaDianNao 的物理参数，包括面积和功耗，然后对其性能和能耗进行了分析。

4.6.1 芯片参数

布局布线后，DaDianNao 的版图见图 4.7。面积和功耗信息见表表 4.2。

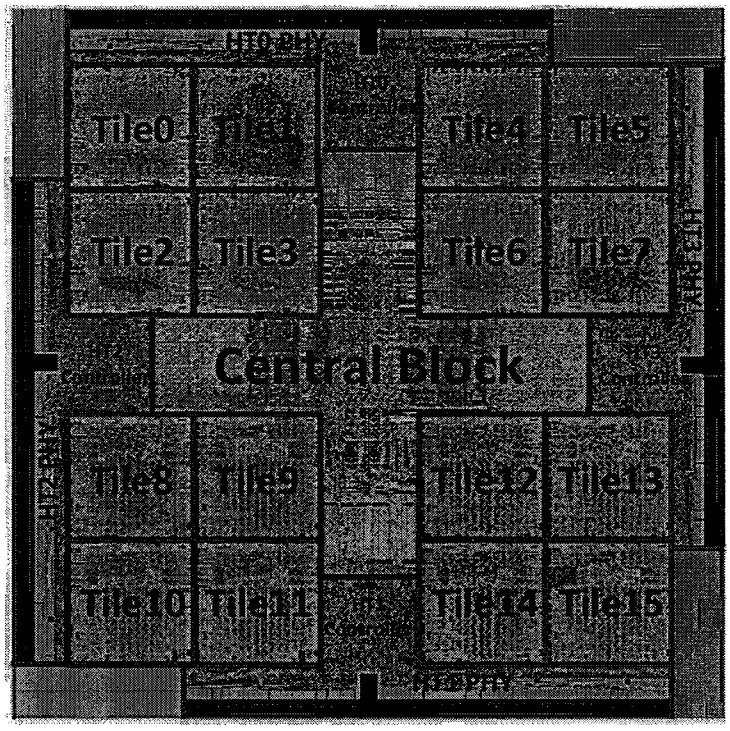


图 4.7 DaDianNao 版图

面积方面，DaDianNao 的总面积为 67.73 平方毫米。关于不同模块之间的面积分布，16 个 leaf tile 占芯片总面积的 44.53%，4 个 HT 共占 26.02%，central tile 占 11.66%，H 树占 8.97%。关于不同组成部分之间的面积分布，片上缓存占芯片总面积的 47.55%，而组合逻辑和寄存器只分别占了 5.88%和 4.94%。

功耗方面，DaDianNao 的峰值功耗为 15.97W（在翻转率为 100%的情况下），大概相当于显卡的 5%到 10%。关于不同模块之间的功耗分布，leaf tile 和 central tile 共占总功耗 的三分之一左右(38.53%)， 4 个 HT 共占一半(50.14%)。关于不同组成部分之间的功耗分布，片上缓存占总功耗的 38.30%，组合逻辑和寄存器分别占 37.97%和 19.25%。

4.6.2 性能

图 4.8 显示了 DaDianNao 相对于 Nvidia K20 的加速比。其中，由于 CONV1 需要的片上缓存太大，需要 4 个以上的 DaDianNao 芯片处理，我们将在下一章中讨论它的性能。

虽然 CONV1 是使用公共卷积核的卷基层，但是它的输入 feature map 有 256 个，输出 feature map 有 384 个，滑动窗口大小为 11*11。所以权值需要的总容量是 256*384*11*11=11,894,784，即 22.69MB。输入神经元需要的总容量是 256*256*256*2 = 32MB，输出神经元需要的总容量是 246*246*384*2=44.32MB。所以 CONV1 一共需要的片上婚车容量为 22.69 + 32 + 44.32 = 99.01MB，大于 DaDianNao 的片上缓存 36MB。同样，CONV3, CONV4, fullNN 的性能都将在下一章中讨论。

Component/Block	Area (μm^2)	(%)	Power (W)	(%)
WHOLE CHIP	67,732,900		15.97	
Central Block	7,898,081	(11.66%)	1.80	(11.27%)
Tiles	30,161,968	(44.53%)	6.15	(38.53%)
HTs	17,620,440	(26.02%)	8.01	(50.14%)
Wires	6,078,608	(8.97%)	0.01	(0.06%)
Other	5,973,803	(8.82%)		
Combinational	3,979,345	(5.88%)	6.06	(37.97%)
Memory	32207390	(47.55%)	6.12	(38.30%)
Registers	3,348,677	(4.94%)	3.07	(19.25%)
Clock network	586323	(0.87%)	0.71	(4.48%)
Filler cell	27,611,165	(40.76%)		

表 4.2 DaDianNao 面积功耗信息

对于测试集中的神经网络，DaDianNao 相对于 GPU 的平均性能提升了 21.38 倍。性能提升的第一个原因是更多的运算器，DaDianNao 中有 9216 个运算器 （大多数是乘法器和加法器），而 K20 中有 2496 个。第二个原因是由于使用了无访存设计，DaDianNao 可以为片上的运算器提供足够的带宽。

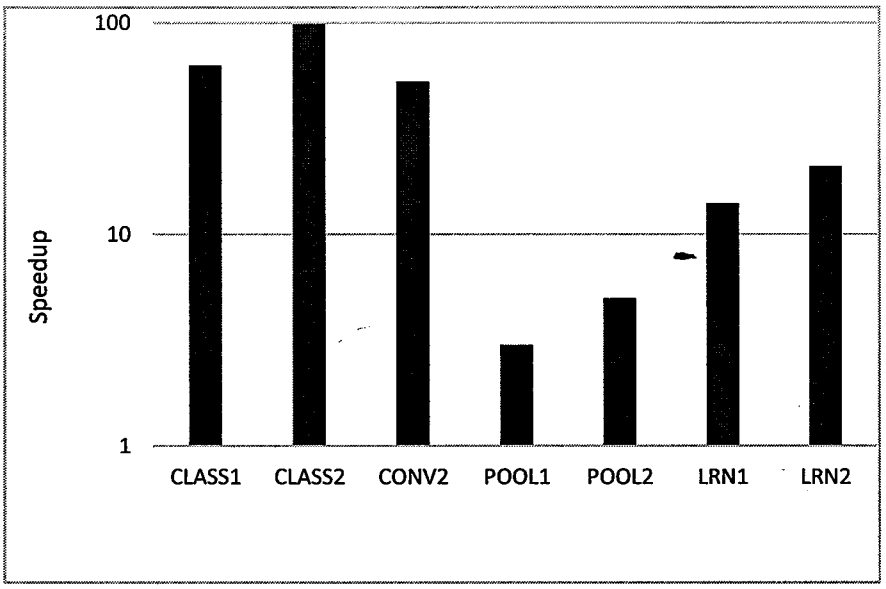


图 4.8 DaDianNao 与 GPU 性能比较

4.6.3 能耗

图 4.9 显示了 DaDianNao 相对于 Nvidia K20 的能耗降低。可以看到对于测试集中的神经网络，DaDianNao 相对于 GPU 的能耗降低提升了 330.56 倍。

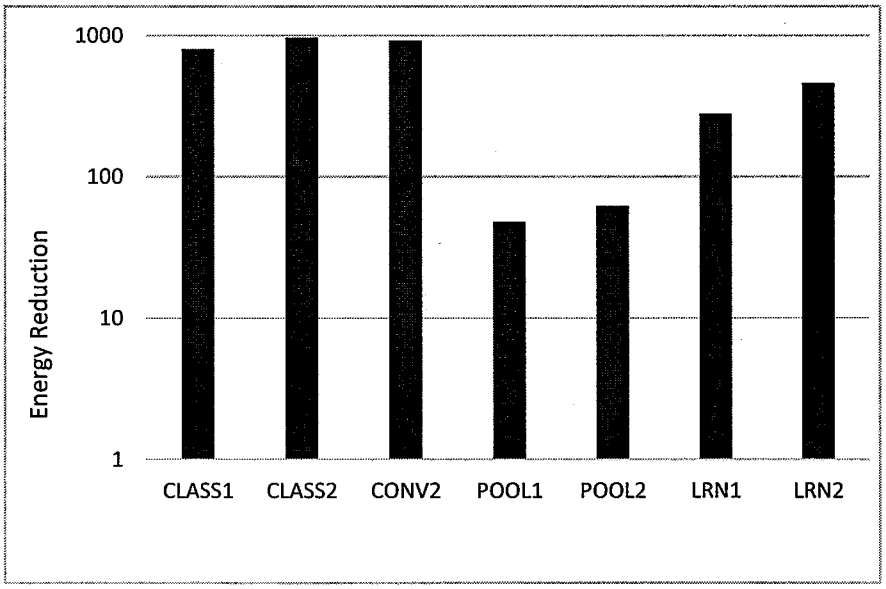


图 4.9 DaDianNao 与 GPU 能耗比较

4.7 总结

本章首先介绍了相关工作的进展，并结合具体场景和问题对本章的工作进行了分析。接着探讨了在进一步提升深度学习芯片性能的过程中遇到的难点，并指出了解决思路和解决方案。最后我们结合以上的分析设计了一款多核深度学习处理器 DaDianNao，并对其进行了逻辑实现和物理设计。

实验结果显示，在 ST28 纳米工艺下，DaDianNao 的面积为 67.73 平方毫米。对于本文选用的测试集，其平均性能为 Nvidia K20 的 21.38 倍，平均能耗相比 Nvidia K20 降低了 330.56 倍。

第五章 深度学习处理器多芯片互联结构设计

随着深度学习的不断发展，使用大规模深度学习进行训练与预测逐渐成为了一种趋势，近几年谷歌，百度，腾讯等公司均搭建了具有数百亿个可训练参数的深度学习模型并各自实现了用于训练和预测的并行化框架。

对于这种大规模的深度学习模型的训练和预测，我们需要使用多个 DaDianNao 芯片搭建成的并行化框架来支持。这主要有两方面的原因。首先，由于 DaDianNao 采用了无访存设计的设计思想，需要将所有参数都存储于片上，而在现有技术下在片上存储数百亿个参数显然是不可能的。第二方面的原因是使用多个芯片处理深度学习算法也可以使得我们可以使用多组运算器同时运算，从而提升效率。因此在对单个深度学习处理器性能进行优化的同时，我们还需要同时对多处理器互联方式进行探索和研究，以适应大规模神经网络的要求。

本章首先介绍了使用多芯片并行化框架处理大规模深度学习应用的研究现状，接下来对不同的深度学习算法介绍了 DaDianNao 的多芯片互联结构设计，最后我们对本章片间互联结构的性能和能耗进行了仿真。

5.1 引言

医学研究显示，人脑中含有大概一千亿个神经元细胞和一百万亿突触。而由于计算机运算能力和存储能力的的制约，业界用于工程实践和科学研究的机器学习算法中参数的规模远远小于这个数字。近些年来，随着计算机性能的不断提升和深度学习的提出，一些公司和研究机构开始试图使用大规模深度学习作为机器学习模型进行训练和预测，并取得了一些成效。

2012 年 6 月，谷歌推出了谷歌大脑,使用由 16000 个通用处理器核心组成的并行计算平台训练一个具有 10 亿个可训练参数的深度学习模型,可以从一千万张图片中找出带有猫的照片。一年后，百度成立了百度研究院及下属的深度学习研究所（LDL）。2014

年百度深度学习研究院将谷歌大脑的主要设计者 Andrew NG 招致麾下，并推出了百度大脑，使用由 GPU 搭建的并行计算平台训练一个具有 200 亿个可训练参数的深度学习模型。

如果使用 DaDianNao 来处理如此大规模的深度学习模型，也同样需要使用很多块 DaDianNao 芯片搭建成的平台实现。这首先是因为在处理规模很大的神经网络时，会出现单个芯片的片上缓存无法存储神经网络中所有参数的情况。而 DaDianNao 采用了在片上存储整个神经网络中的所有参数的无访存设计思想，因此需要使用多处理器系统进行处理，将整个神经网络存储在多个芯片的片上缓存中。例如 2012 年的谷歌大脑使用了有十亿个可训练参数的深度学习模型。即使假设存储空间可以被理想地利用，十亿个可训练参数也需要$(10^9 \times 16)/(1024^3 \times 8) = 1.86$GB 的容量，而 DaDianNao 的片上缓存容量为 36MB，因此至少需要$(1.86 \times 1024) / 36 = 52$ 个 DaDianNao 组成的多芯片系统才能提供足够的片上缓存。

其次，虽然DaDianNao的性能高于单块CPU/GPU，但是如果仅仅只使用单块DaDianNao芯片来处理大规模神经网络的话，其能提供的计算资源与由大量 CPU/GPU 组成的多芯片系统相比肯定没有优势。

因此，对于这种大规模的神经网络，我们有必要使用由多个 DaDianNao 芯片组成的多芯片系统对其进行训练和预测。本章将从片间互联拓扑结构和通信机制两方面出发分别对 DaDianNao 支持的几种深度学习算法介绍相应的片间互联策略。

5.2 概述

由于多芯片系统中片间数据传输速率低的特点，整个系统的效率会受到片间传输的制约。而随着整个系统中芯片数量的增加，片间传输数据量也会增加，因此芯片数量对系统效率带来的提升是具有边界效应的。事实上，我们发现在芯片数量很多的情况下，增加芯片的数量甚至会使得整体性能下降。因此在设计 DaDianNao 多芯片互联结构时我们不仅需要考虑系统在当前规模下的效率，同时也要考虑系统的可扩展性，即要使得整个系统的效率尽量与芯片数量之间保持线性关系，使之在处理更大规模的深度学习时依旧可以具有较高的效率。

根据上面的设计目标，我们在设计 DaDianNao 多芯片互联结构时，主要的目标是降低片间传输的数据量。与片间传输数据量相关的问题主要有两个，一个是深度学习中三类参数的数据规模与局部性情况，另一个是如何在多芯片之间分配神经网络的运算任务。

5.2.1 深度学习参数的规模与局部性

神经网络的参数可以分为三种类型：输入神经元，输出神经元，权值。下面我们对于每种深度学习算法中三种类型神经网络参数的数据规模和数据的局部性特征进行分析。

在分类层中，由于对于每一个输出神经元的计算都需要所有输入神经元参与运算，并且一层的输出神经元是下一层的输入神经元。因此在当前层计算完毕后，需要在片间传输这一层的输出神经元，即输入/输出神经元不具有局部性。而由于在对任意一个输出神经元的计算中，参与运算的权值都与其他运算中参与运算的权值不同，所以权值具有局部性。关于数据规模，由于输入神经元与输出神经元之间通过权值全连接，所以权值的数据规模远远大于输入/输出神经元的数据规模。

在公共卷积核的卷积层中，由于对于一个输出像素点的计算需要多个滑动窗口参与运算，并且与不同输出像素点相关的滑动窗口之间有重叠，所以输入/输出神经元不具有局部性。而由于卷积核公有，所以权值不具有局部性。关于数据规模，由于卷积核公有，即每层神经网络只包含一组权值，所以权值的数据规模较小。而由于输入/输出神经元的规模不会影响权值的规模，因此有些应用的深度学习结构中公共卷积核的卷积层会含有大量的输入/输出神经元。

在私有卷积核的卷积层中，由于对于一个输出像素点的计算需要多个滑动窗口参与运算，并且与不同输出像素点相关的滑动窗口之间有重叠，所以输入/输出神经元不具有局部性。而由于卷积核私有，所以权值具有局部性。关于数据规模，由于卷积核为私有，所以与分类层的情况类似，权值的数据量大于输入/输出神经元的数据量。

在池化层中，由于与不同输出像素点相关的滑动窗口之间没有重叠，所以输入/输出神经元具有局部性。与公共卷积核的卷积层类似，输入/输出神经元的的数据量取决于具体算法。池化层中无权值参数。

对于 LRN 层，与池化层类似，输入/输出神经元具有局部性，且输入/输出神经元的的数据量取决于具体算法。LRN 层中没有权值参数。

5.2.2 片间任务划分

按照神经网络的结构划分，神经网络可以分为两类。分类层为一维神经网络，卷积层，池化层，LRN 层为三维神经网络。

对于一维神经网络，最直接的片间任务划分方式是将输出神经元分块，在不同的芯片中计算不同分块的输出神经元。但是这种情况下由于每个芯片都需要计算对应输出神经元分块的最终和，因此需要获得所有的输入神经元参与运算，也就是说在当前层计算完毕后我们需要在芯片之间广播本层的输出神经元来为下一层的运算做准备。这种片间任务划分方式下片间传输数据量较大，如果当前层有 Ni 个输入神经元，使用 n 个芯片进行计算，则片间传输的信息量为(n - 1) × Ni。

另外一种分配方式是将输入神经元和输出神经元分别分块，在不同芯片中计算不同的输入神经元 - 输出神经元对。在这种片间任务分配方式下，对于输入神经元，只需要在一部分芯片间广播一部分输入神经元即可（在另一部分芯片间广播另一部分输入神经元）。对于输出神经元，同样在一部分芯片间广播一部分输出神经元即可，广播的同时将部分和相加，最后得到最终和。在这种片间任务划分方式下，片间传输数据量较小，如果当前层有 Ni 个输入神经元，使用 n 个芯片进行计算，则片间传输的信息量为 $2 \times \sqrt{n} \times Ni$。

三维神经网络一共包括卷积层，池化层，LRN 层三种算法。卷积层中，由于滑动窗口之间有重叠，所以在当前层计算完毕后我们需要在芯片之间广播滑动窗口之间重叠的部分来为下一层的运算做准备。所以我们可以在不同的芯片之间分配计算不同输出像素点的任务，并且将计算相邻像素点的芯片也相邻摆放，这样每个芯片就只需要与和它相邻芯片进行通信。关于权值的局部性，对于公共卷积核的卷积层我们可以在每个芯片中存储同一份公有的卷积核，因此卷积核不需要片间传输。在私有卷积核的卷积层中，每个芯片中的卷积核都应该是不同的，因此卷积核同样也不需要在片间传输。

在池化层和 LRN 层中，滑动窗口之间不会重叠，并且不存在权值。所以可以在当前层计算完毕立即开始进行下一层的运算，不需要进行片间传输。

5.3 分类层多芯片互联结构设计

5.3.1 分类层片间互联拓扑结构

本节对于不同的应用场景分别使用了环和 Torus 两种结构作为片间互联的拓扑结构。其中，基于环的片间互联结构实现起来较为简单，且在芯片数量不大时效率也比较高。但随着芯片数量的增加，全系统的效率的提升会由于片间传输量的增加而明显减缓。基于 torus 的片间互联结构在片间任务划分与片间传输的实现上比较复杂，但是扩展性

较好，全系统的效率会随着芯片数量增加有较明显的提升。因此对于规模不大的神经网络，可以采用基于环的路由进行片间路由，而对于较大规模的神经网络，则适合使用基于 torus 结构的路由。下面分别对两种结构进行介绍。

环形结构

如图 5.1 所示，环形结构是一种简单直观的拓扑结构[99]。本文使用的环形拓扑结构是一个双向环，即网络中每两个相邻的节点可以同时给对方传输数据。在环中顺时针和逆时针方向的数据传输可以同时进行。

在双向环结构中，如果假设现有节点数量为 n，相邻两节点间片间通信带宽为 K，则在整个网络中，总的可利用的带宽为 n×K×2，距离最远的两芯片间距离为 n/2 跳。

环形结构可以在基于 Mesh 结构的物理连接中使用，图 5.2 展示了一种在 Mesh 拓扑中使用环形拓扑的方式（当然，方式不是唯一的）。由于无论使用何种方式在 Mesh 结构中找到一个环形结构进行片间通信对片间互联的性能没有影响，并且易于实现，所以本文中不进行讨论。

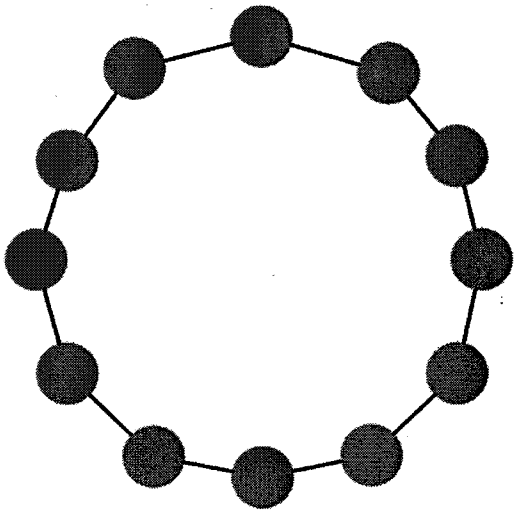


图 5.1 环形拓扑结构

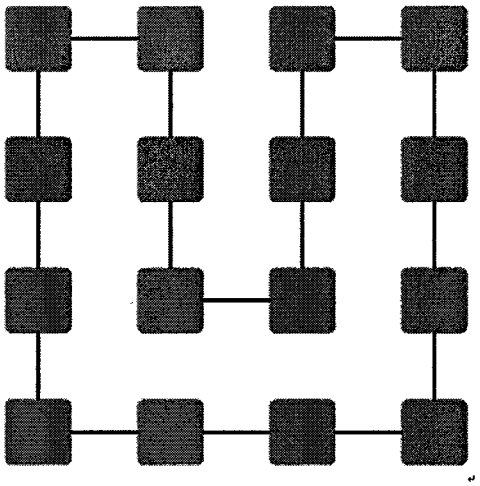


图 5.2 在 Mesh 结构中使用环形结构

Torus 结构

如图 5.3 所示，2D-torus 结构[98]可以看成是对 Mesh 结构[94][95][96][97]的一种扩展，即在边界的节点上增加了一条长的环路。因此，网络中的所有节点度为 4。2D Torus 拓扑在物理形式上与 Mesh 结构相似，由于其中存在很多的环路，所以节点之间的最远距离小于Mesh结构，但是在路由算法和路由仲裁方面都要复杂。又由于2D-torus 拓扑的每个路由节点的度都为 4，连接情况相同，所以扩展性也要比 Mesh 结构高。与之前类似，本节中的 2D Torus 结构中，网络中每两个相邻的节点可以同时给对方传输数据。

在 2D Torus 结构中，假设现有节点数量为 n，相邻两节点间片间通信带宽为 K，则总的可利用的带宽为 n×K×4，最远节点间距$\sqrt{n}X2$跳。

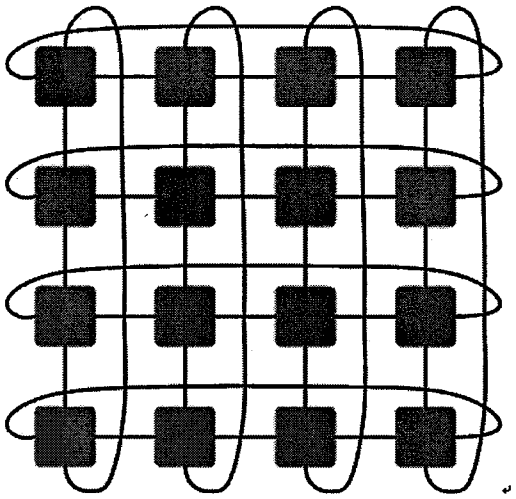


图 5.3 2D Torus 拓扑结构

5.3.2 分类层通信机制

本节分别介绍在环形拓扑结构和 Torus 拓扑结构下，分类层片间通信的通信机制。在对两种拓扑结构的讨论中，我们均假设芯片总数为 m×n（m 行 n 列），当前分类层的输入，输出神经元数量分别为 Ni, Nn。

环形结构

在片间任务分配方面，我们将输出神经元按照芯片数量分成 n 个分块，使用不同的芯片计算不同分块的输出。在这种任务分配方式下，片间通信方式如下：

1. 在计算开始时，假设所有输入神经元及相关权值已经按照片间任务分配的方式预先存储在每个芯片中。

2. 每个芯片使用自身片上缓存中的输入神经元和权值计算其负责部分的输出（直接计算出输出神经元的值）。

3. 由于当前层的输出为下一层的输入，所以为了准备下一层的输入数据，需要将本层的输出神经元广播到每个芯片。当广播完毕后，本层结束。

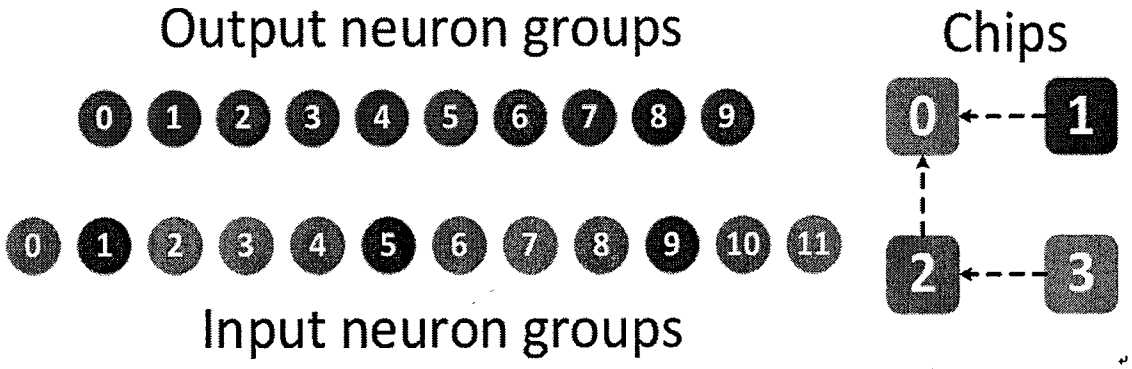


图 5.4 环状结构运算方式示意图

举例来说，假设共有 4 个芯片参与运算，输入规模为 12 组，输出为 10 组，当前计算的是第 8 组输出数据。则结点 2（编号均从 0 开始）需要预先存储第 2、6、10 组输入，随后计算出由这三组输入汇总得出的部分和，并在接到结点 3 传来的结果并与自身部分和汇总后，再传给结点 0。同理，结点 1 直接把部分和结果传给目的结点 0。图 5.4 示意了这一计算过程的数据分割、结果传输方式，需要指出的是，结点 3 的部分和结果，既可以传给结点 2，也可以传给结点 1，具体方向由人为规定。

2D Torus

在输入数据分配及部分和计算的过程中，每一行结点可看做一个独立的环形结构，这意味着所有结点同时计算 m 组输出，且每一列结点共享相同的输入组。当完成部分和的计算后，每行的目的节点为坐落在对角线上的结点，并且此目的结点对于计算不同的输出组保持不变。

当所有对角线上的结点接收并汇总出最终的部分和、计算出激活值（即输出值）后，再将每列看做一个独立的环。各个对角线结点把激活值广播到自身所在列的所有结点上，这个激活值将作为下一层神经网络的输入参与进一步的计算。以拓扑中最下行和最右列的两个环为例，图 5.5 和图 5.6 分别示意了在一个简单情况下处理的两个阶段。

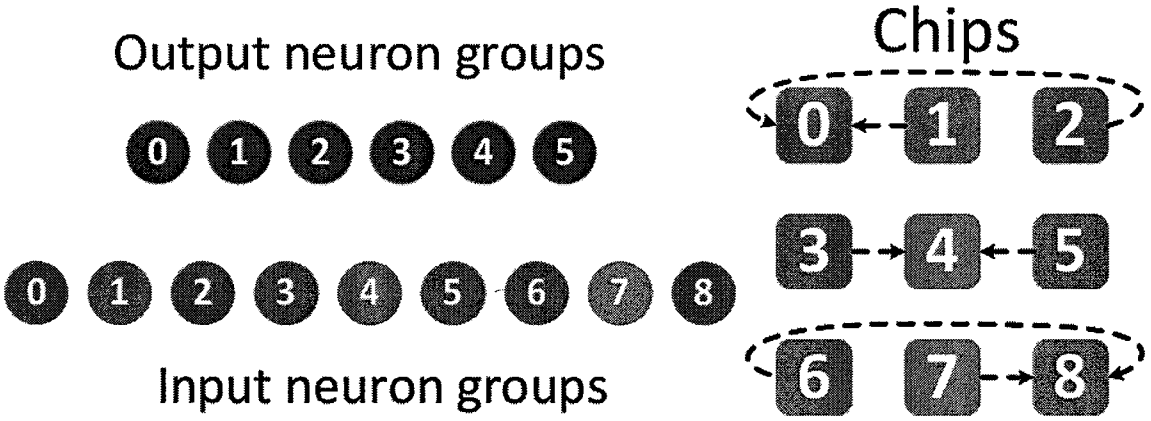


图 5.5 Turos 结构计算分类层的第一阶段

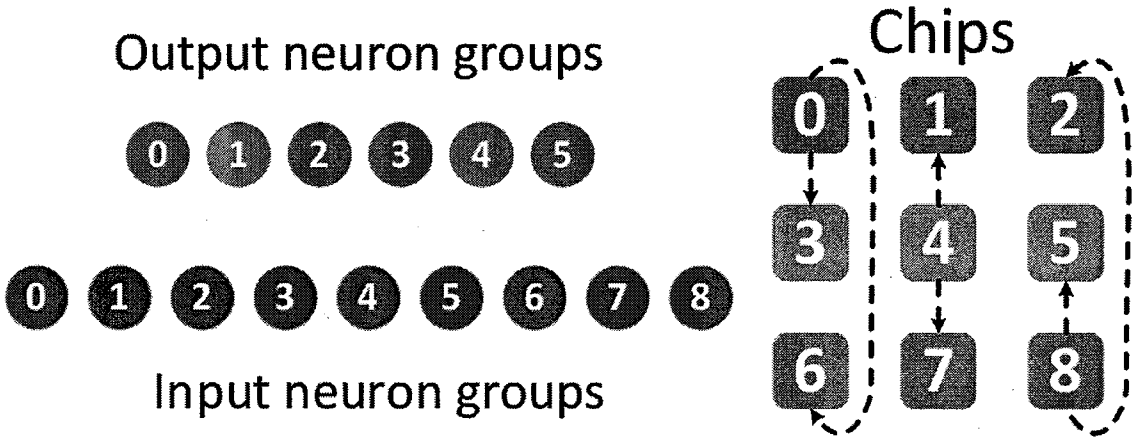


图 5.6 Turos 结构计算分类层的第二阶段

5.4 卷积层，池化层，Pooling 层多芯片互联结构设计

通常情况下，LRN 与 POOL 的正向与反向计算过程中所需的输入数据全部都能被存储在单个芯片的片上存储器中，输出数据也可以直接存储在片上存储器中，整个计算过程几乎不需要片间通信，所有芯片均可独立完成。因此，LRN/POOL 的计算能力对于芯片数量有非常强的近似于正比的扩展性，且几乎不会受到拓扑方式的影响。

5.4.1 片间互联拓扑结构

卷积的多片协同计算涉及到输入/输出 feature map 的分割和滑动窗口在 feature map 子区域边界情况的处理两个问题。输入/输出特征映射的分割，只需要按照单个芯片的存储能力来划分即可。而对于滑动窗口在单个芯片内的特征映射子区域边界附近的情况（参照图 5.7），则需要与相邻节点进行少量的通信。即把需要的部分输入数据传输到相邻节点上。

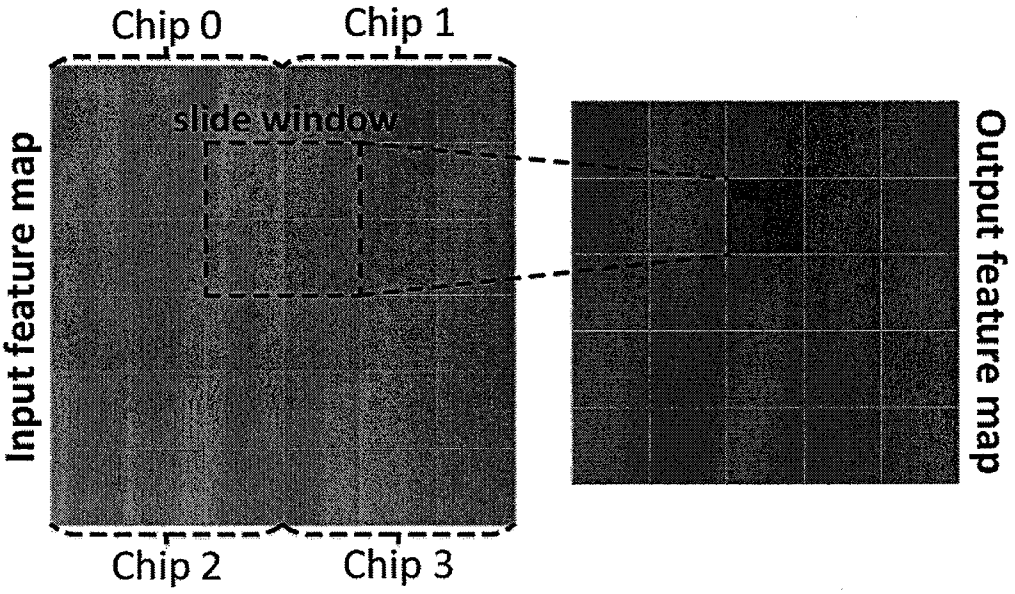


图 5.7 卷积数据分割与边界情况示意图

5.4.2 卷积层通信机制

卷积的多片协同计算涉及到输入/输出特征映射（feature map）的分割和滑动窗口（slide window）在特征映射子区域边界情况的处理两个问题。输入/输出特征映射的分割，只需要按照单个芯片的存储能力来划分即可。而对于滑动窗口在单个芯片内的特征映射子区域边界附近的情况（参照图 5.7），则需要与相邻节点进行少量的通信。这种通信只是简单地把需要的部分输入数据传输到相邻节点上，因此不再特别描述。

5.4.3 池化层/LRN 层通信机制

通常情况下，LRN 与 POOL 的正向与反向计算过程中所需的输入数据全部都能被存储在单个芯片的片上存储器中，输出数据也可以直接存储在片上存储器中，整个计算过程

几乎不需要片间通信，所有芯片均可独立完成。因此，LRN/POOL 的计算能力对于芯片数量有非常强的近似于正比的扩展性，且几乎不会受到拓扑方式的影响。

5.5 实验结果

5.5.1 性能

分类层

容易知道，不管采用何种输入输出数据结构、拓扑结构的划分，在单个芯片计算能力固定的前提下，完成特定计算任务所需的最小计算量是不变的。比如对于一个 128*128 的 MLP 层来说，需要 64*64 计算能力的芯片运算至少 4 次，这个下界无法再降低。因此对于给定芯片能力和数量前提下的总体性能的分析，重点在于通信方式、数据分配上。

由之前的叙述可知，在环形结构下，所有结点在完成一次输出组的计算后，用于传输部分和以产生输出结果这一过程的最大跳数为 O(n/2)，并能产生一组输出数据。而 orus 结构下，横向汇总部分和与纵向分发新的输入数据均需要跳数为 O((1/2)*n^(1/2))，因此整个汇总-分发过程的最大跳数为 O(n^(1/2))，并能产生 n^(1/2)组输出数据。对比这两种分析可发现，所有芯片在计算上花费的总时间是一致的，但 Torus 结构平均每产生一组输出数据，需要的传输跳数 O(n^(1/2)/n^(1/2))=O(1)，大大优于 Ring 结构的 O(n/2)，这进一步降低了 MLP 计算过程中的通信瓶颈。

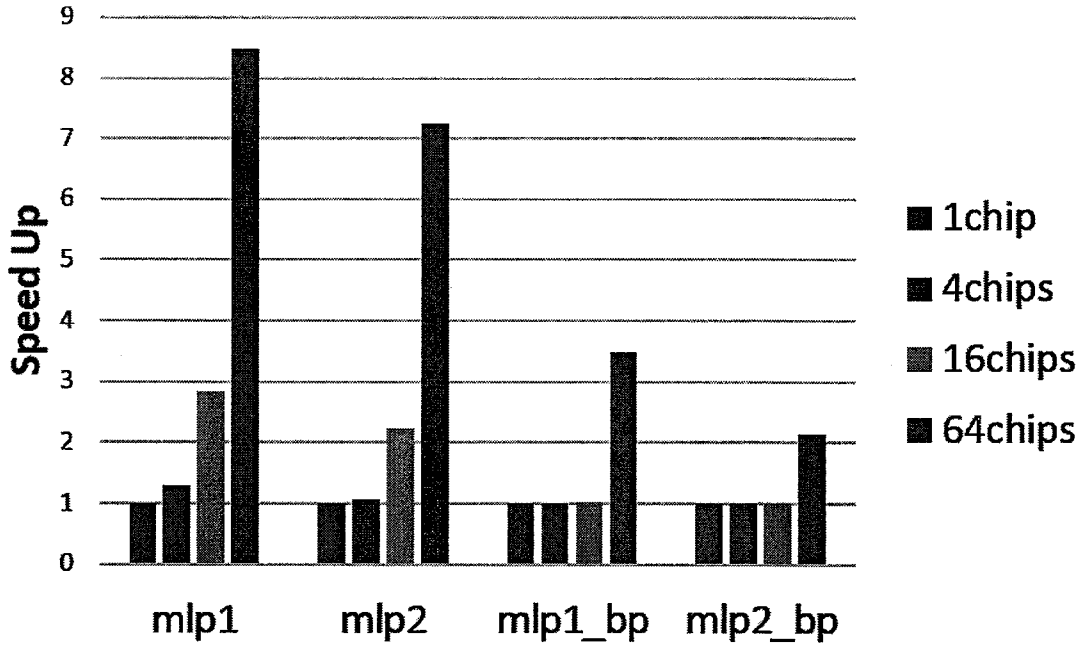


图 5.8 分类层 Torus 与 Ring 结构性能比较

图 5.8 示意了以相同互联方式处理不同规模的数据时，采用 Ring 结构和 Torus 结构的整体实现在 MLP 上的性能差异。需要指出的是，在反向过程中，由于计算完 β 后还需要大量的片内计算来完成权值的更新，因此较大的计算密度掩盖了数据传输方面的性能提升，只有在参与结点较多（图中为 64）时，才开始体现出 Torus 的优势，或者说是 Ring 结构的劣势开始更加充分地显现出来。

卷积层

卷积层的运算虽然需要部分的通信，但瓶颈重点体现在单片的计算性能上。故不同的互联方式对整体的性能影响不大，这里只给出采用了 HT 连接方式的卷积层相较于传统 GPU 的性能提升比，如图 5.9。

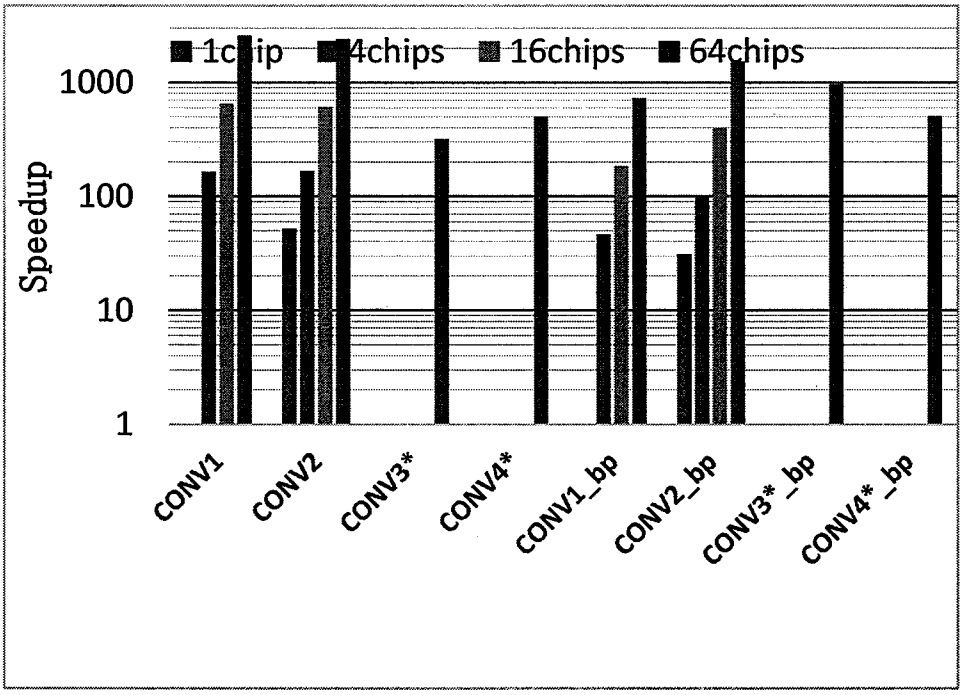


图 5.9 卷积层多芯片互联性能

池化层/LRN

如前所述，LRN/POOL 层所需输入数据的规模较小、可完全在片内运行、不需片间通信的特点决定了总体的运算性能只与参与运算的结点数量相关，且接近正比关系。因此这里只给出两种算法在不同规模下相较于传统 GPU 的性能提升，如图 5.10。

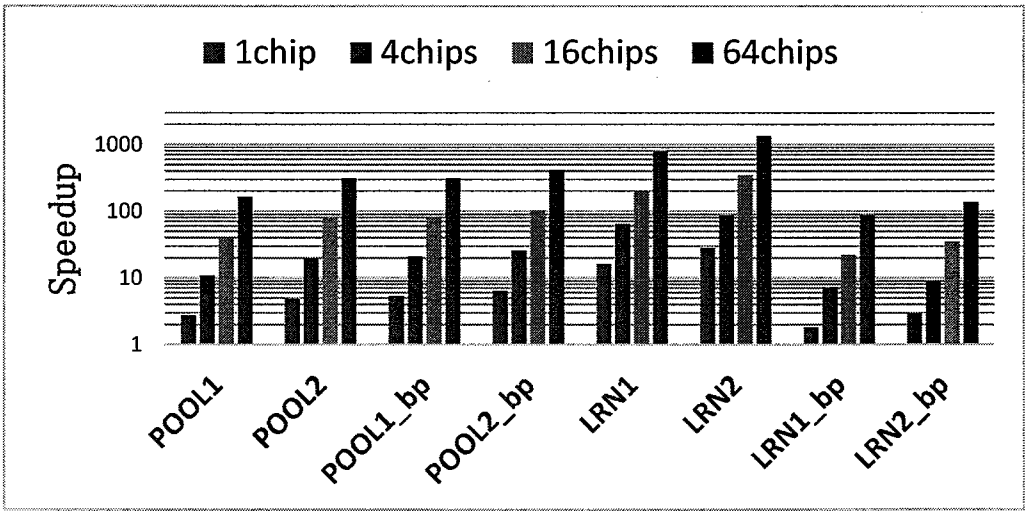


图 5.10 池化层/LRN 层多芯片互联性能

从以上三组的讨论中不难看出，在多片互联领域，对通信密集型的 MLP，应更多地关注拓扑结构和互联方式所带来的瓶颈，而对于计算密集型的 CONV/LRN/POOL，则应重点关注单片计算能力的提升。只有这样，才能更容易地突破整体性能的瓶颈，如图 5.11 示意了不同神经网络层的通信和计算时间在总体运行时间中占用的比例，图中数据有力地说明了应对不同种类的神经层，应如何选择优化的重点以得到最大化的提升效果。特别地，由于通信和计算往往是在同时进行的，因此两者的比例总和可以超过 100%，并且由于 LRN/POOL 没有通信，图中的统计结果省略了这二者；Gmean 代表各个神经层的通信、计算占比的几何平均值。

图 5.12 给出了 full NN 以及各种测试样例层的算数平均加速比。

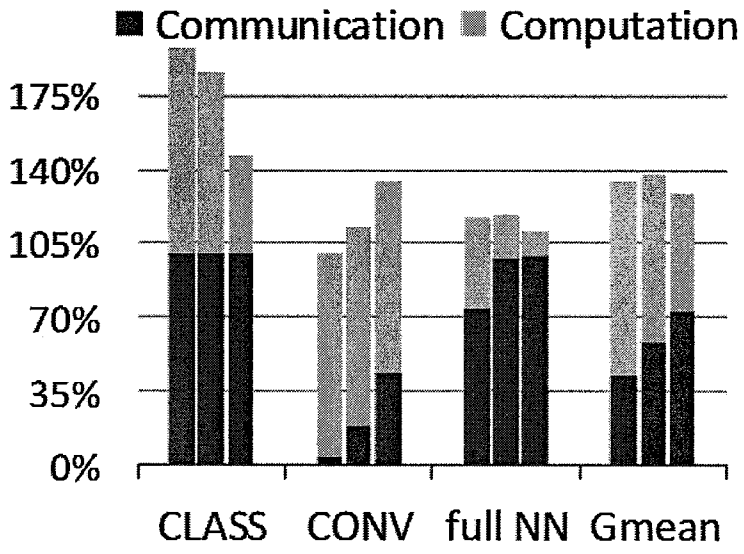


图 5.11 通信与计算时间占比

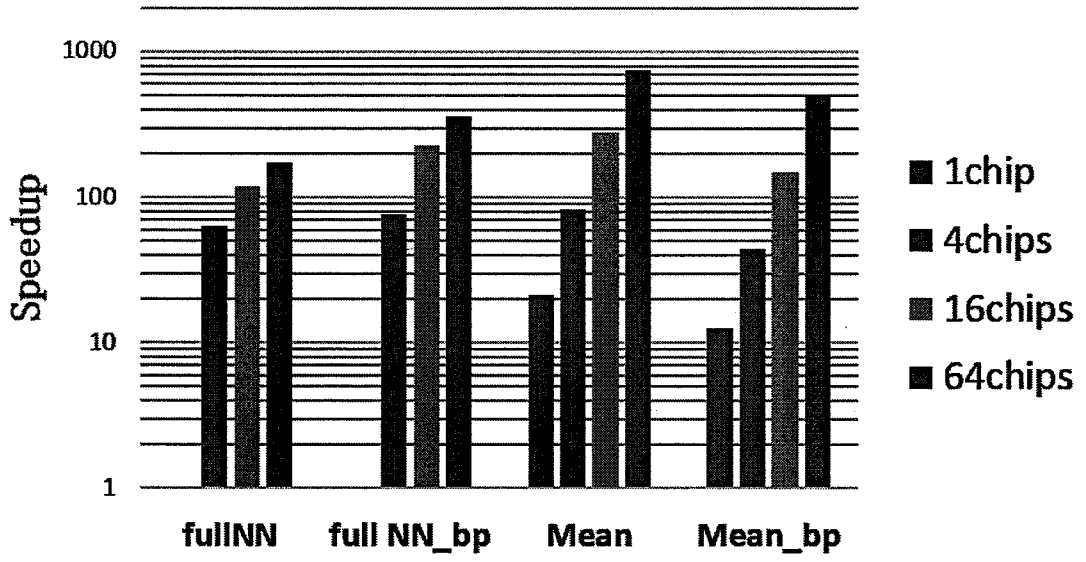


图 5.12 full NN 与总体平均性能

5.6.2 能耗

分类层

由运作方式一节的叙述可知，对于 Ring 结构处理 mlp 的通信阶段来说，除了接收（目的）结点外，环上其他每个结点均向目的方向发送了一次部分和，共 n-1 次数据传输。忽略掉较小的常数，所有结点共需要 O(n)次的数据传输来产生一组输出值。但对于 Torus 来说，横向汇总阶段每行作为一个 Ring 需要 O(n1/2)次传输，n1/2 行共计 O(n)次传输，同理分析，纵向分发数据阶段，共需 O(2n)次传输。考虑到 Torus 结构每次通信过程可产生 n1/2 个输出数据，平均每个输出数据只消耗 O(2n1/2)次片间通信。显而易见，Torus 结构用在通信方面的能耗将比 Ring 结构大大降低。

图 5.13 示意了采用 HT 互联方式的两种拓扑结构在 4、16、64 结点上处理不同规模 mlp 的能耗占比的算数平均值。图中的数据可以强力地印证之前的分析，即 Torus 结构下的通信能耗占比大大的降低了。图 5.14 示意了能耗方面对于 GPU 的缩减比例。

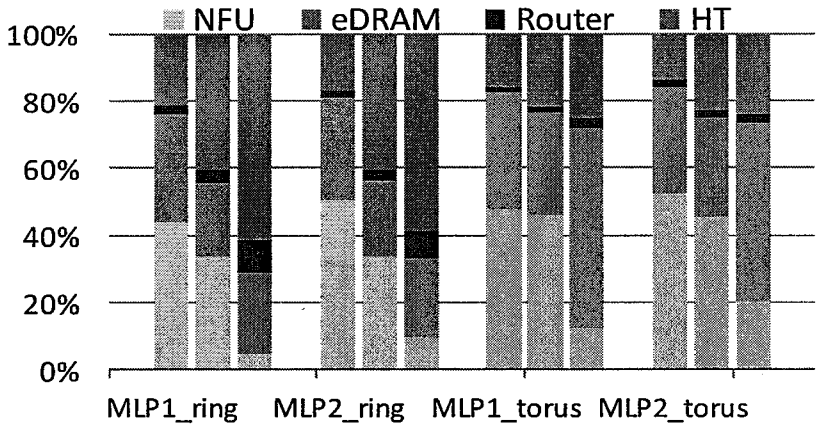


图 5.13 各组成部件能耗比

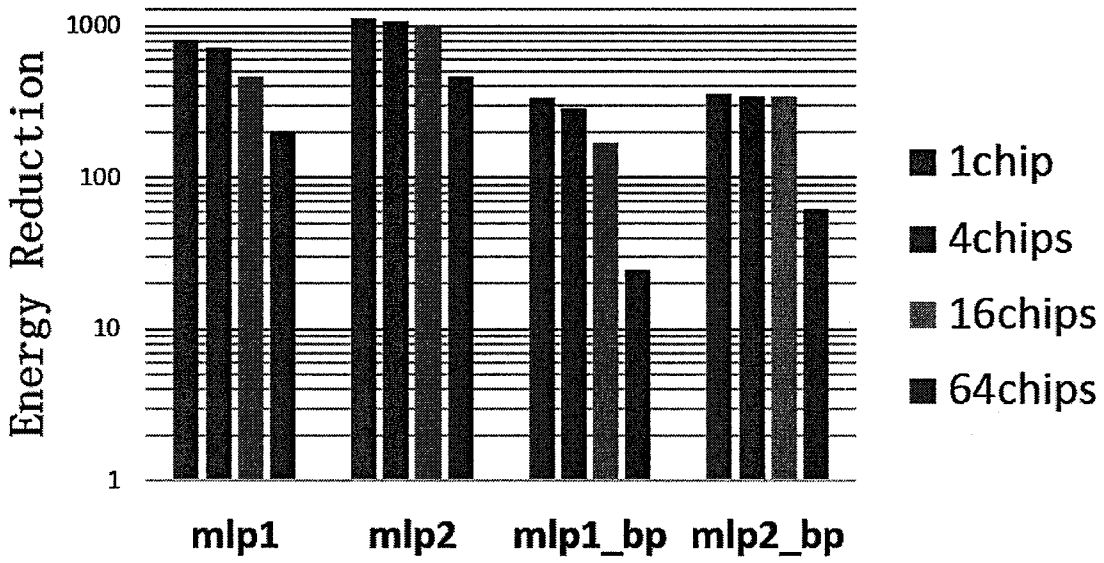


图 5.14 分类层前馈运算多芯片互联能耗

卷积层/池化层/LRN

卷积层 池化层 LRN 层前馈运算和反向训练的多芯片互联能耗见图 5.15、图 5.16。

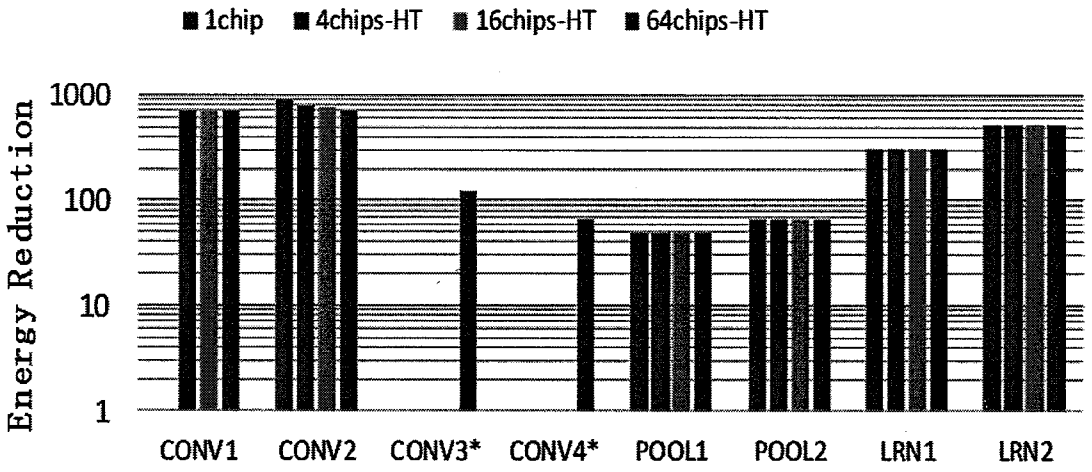


图 5.15 卷积层 池化层 LRN 层前馈运算多芯片互联能耗

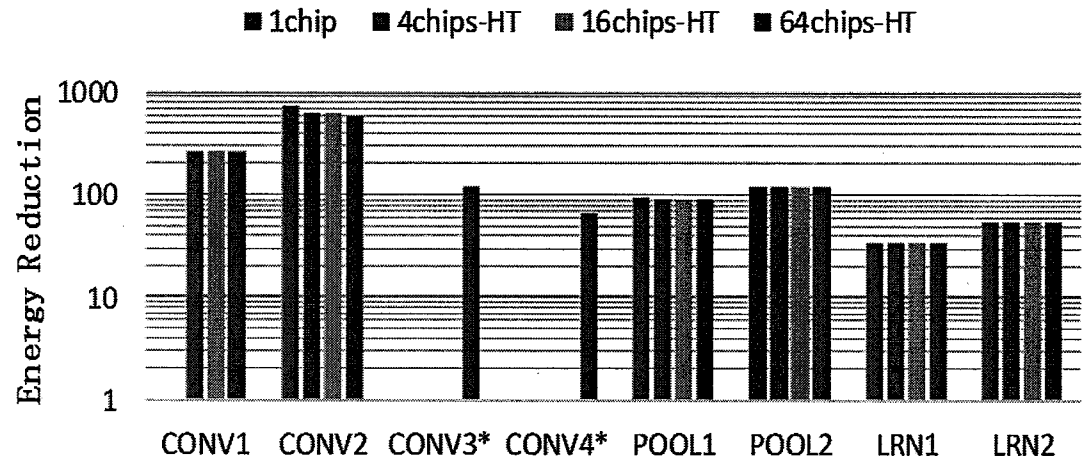


图 5.16 卷积层 池化层 LRN 层反向训练多芯片互联能耗

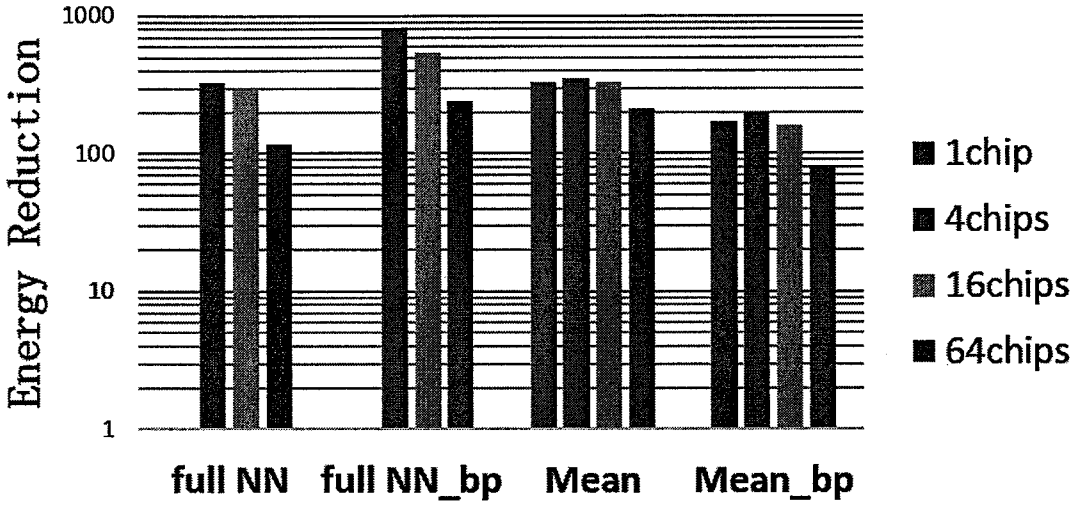


图 5.17 full NN 与算数平均能耗缩减比

图 5.17 给出了各种测试样例层的算数平均情况以及 full NN 的能耗缩减比。

5.7 总结

本章介绍了多芯片互联的相关背景，阐述了多芯片协同处理神经网络的思路和方法，并分析了不同方法下的相关性能和能耗。当指定单个芯片的性能及计算方式、计算原理时，多芯片互联的性能在物理上主要受到拓扑结构及互联方式的影响，选择更加合理、贴合需求的拓扑，更加高速低功耗的互联将会给总体性能带来极大的提升；而在算法层面上，则主要受到输入/输出数据在不同结点上的划分、充分利用拓扑结构合理传输数据以及边界等特殊情况的处理这三方面的影响。不同的处理方式，将会导致性能和能耗上较大的差异，因此对于所使用的处理方式需要特别仔细的分析。

第六章 深度学习处理器片间光互联设计

自从1984年Goodman等首先提出集成电路光互联的概念以来，光互联作为一个有效解决电互联潜在问题的办法备受关注。由于在深度学习处理器多芯片结构中片间数据传输是性能的瓶颈，因此如果能使用光互联降低片间互联的延时，增加片间互联的带宽，那么深度学习处理器多芯片系统的性能将会进一步提升。

6.1 现有工作

光互联主要有两种形式：波导光互联和自由空间光互联。波导光互联可提供高密度的互连通道，易于对准，适用于芯片内或芯片间层次上的互联。但是其本身损耗比较严重，而且集成度低。自由空间光互联可以使互联密度接近光的衍射极限，不存在信道对带宽的限制，易于实现重构互联，适用于芯片间和MCM之间层次上的互联。不过，自由空间光互联的对准问题有待解决。

光互联的主要优势在于大带宽、低串扰噪声、低驱动电源、系统和长距离互联时有良好的时钟同步性能及设计简化等。另外，光互联用于芯片互联不需要物理上的新突破。虽然光互联有着很多优点，但是像集成电路一样广泛的实用化还要面临很多技术上的挑战，如on-chip或off-chip的短距离光互联要求功耗低、反应时间短、物理尺寸小而且能和主流的硅基电路工艺兼容。因此，很多的研究工作在这一领域展开。

1. 基于性价比的驱动，开展互联光学部件（光发射器、光传输器、光接收器件等）的制造工艺和现有成熟的CMOS工艺兼容性、新的材料和器件结构的研究；

2. 从系统的整体性能考虑，兼顾电互连和光互连两者的优点，通过对互连网络的合理分配来发挥两者的潜能（带宽、功耗、时钟分配、时钟分布等）；

3. 适应当今集成电路设计的需要，提出和简历更精确的光互联器件的模型（激光器、PIN和APD）、模拟仿真软件、综合工具和建立可重复使用的光电IP核等。

6.2 光互联的实现

我们采用以多微环为基础的方式来实现片间的光互连。如图 6.1 所示，整个点对点硅-光连接系统包括以下几个组成部分：母版、硅衬底、处理器、光学模、光纤、激光源。其中，母版用于搭载所有的部件，硅衬底为处理器和光学模的互连提供载体，光学模用来处理电信号和光信号的互相转换，光纤用来连接不同的光学模，激光源为光学模提供多种波长的光源，分为片上和片外两种类型。目前，限于片上激光源的面积和功耗问题，比较常用的还是片外激光源。

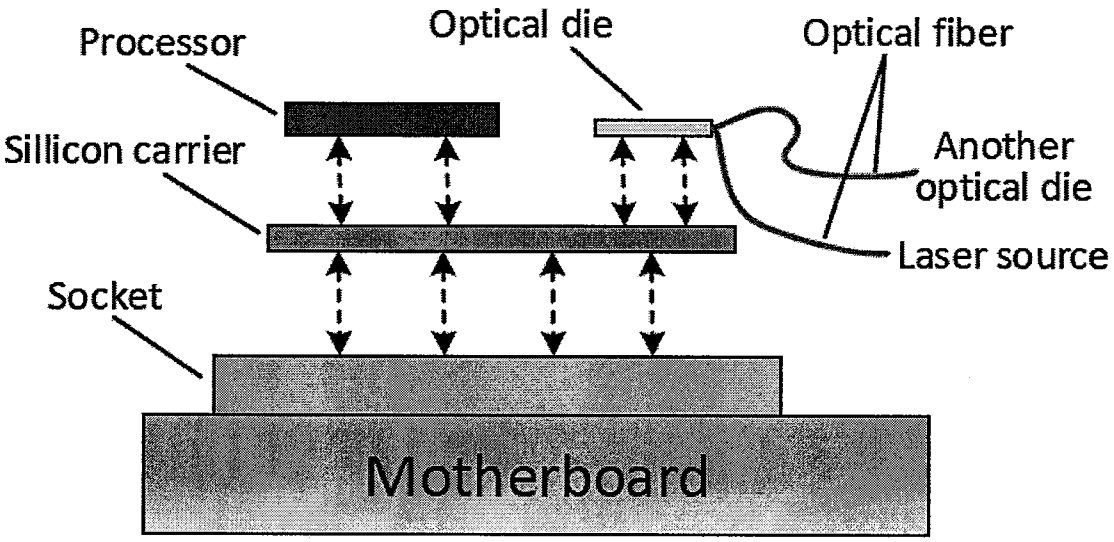


图 6.1 光互连系统示意图

图 6.2 示意了用于处理光电信号转换的光学模内部的微环谐振器结构。微环谐振器是光互连系统中的重要基础器件，是由光波导制作而成的环形谐振腔。光学模包含发送端和接收端两部分：发送端由微环调制器阵列构成，接收端则由两阶段的微环分离阵列构成。电光信号的整个传输与转换过程可简单表述为：调制——传输——筛选——解调制。

在微环调制器阵列中，高速的电信号在微环附近被调制成在波导中传输的连续波长的光信号。调制器的调制波长由微环的温度、半径等参数决定，为了使其谐振波长达到特定值，需要将电荷注入到谐振腔来改变其光吸收系数与折射率。一个波导上的多个微环可以组成一个波分复用调制器阵列，该阵列可以同时操作一系列的波长信道。

在信号分离阵列中，微环则被用来从共享的波导中提取出不同的波长信号，而后被提取的光信号被光电探测器转换为模拟电信号输出。微环在滴口处提取出共振信号来，而其他非共振信号则继续在波导中传播。

为了与共振频率保持一致，所有的微环是以逐渐增加的半径来交错排列的。由于单个微环可能不足以抑制其他波长的信号，接收端可以采用多个微环级联的方式来提升抑制能力。在接收和发送端，使用锗-光探测器来检测高速信号并稳定热量。激光源是片外的分离单元，可以独立配置。

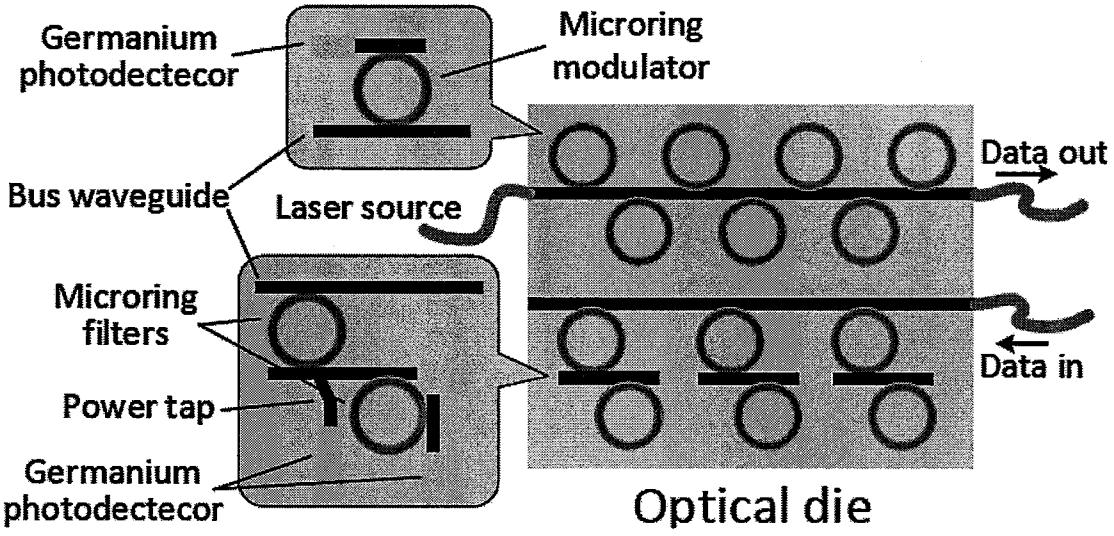


图 6.2 光学模示意图

处理器与光学模的连接，一般采用直接的导线连接。在 FPGA 上，一根导线的传输速率可达到 10Gb/s，实际的互连可以达到更高的性能。因此在实际连接时，可根据带宽和频率的需要来决定互连导线的条数。此外，考虑到缩减导线数量、节省走线资源，还可以选择使用串并转换模块来处理信号通信。

6.3 实验结果

性能

由于 SiPh（silicon photonic）可达到 225TB/s 的单向带宽和 0.04ns 的通信延迟，而 HT 只有 6.4GB/s 的单向带宽和 80ns 的通信延迟，使用 SiPh 以后，多芯片系统的性能

可以得到很大提升。图 6.3 和图 6.4 分别显示了在前馈运算和反向训练中，使用光互联的深度学习处理器多芯片系统相对于 GPU 的性能。

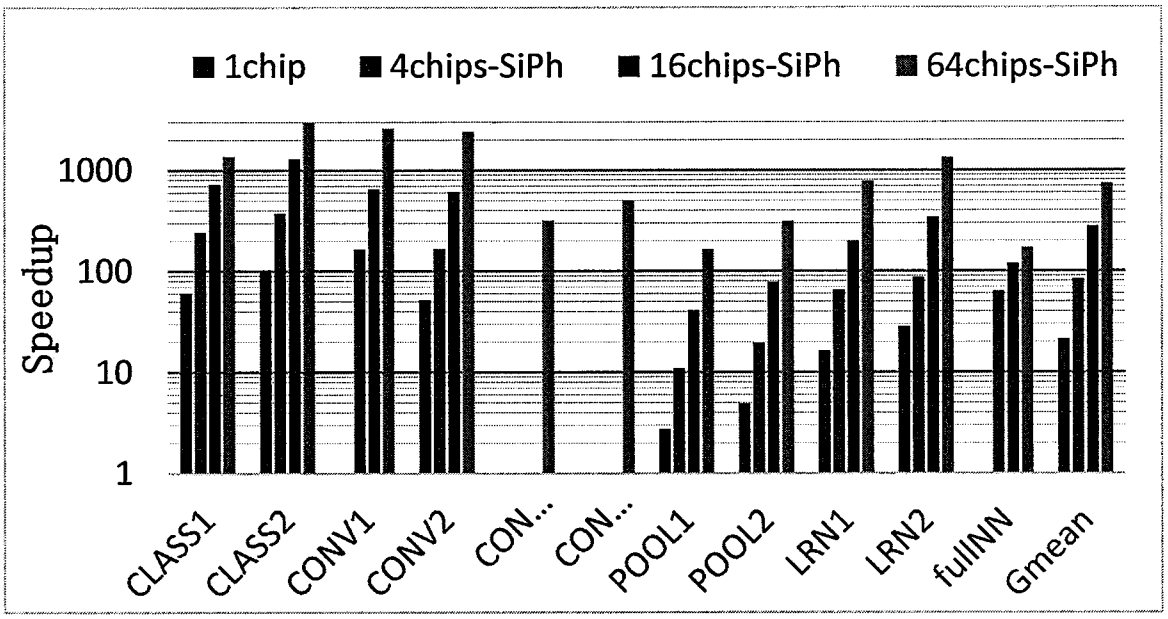


图 6.3 多芯片系统与 GPU 前馈运算性能对比

可以看到，对于前馈运算，在使用光互联的深度学习处理器多芯片系统中，单芯片，4 芯片，16 芯片，64 芯片的平均性能是 GPU 的 21.38、83.00、277.41、743.57 倍。

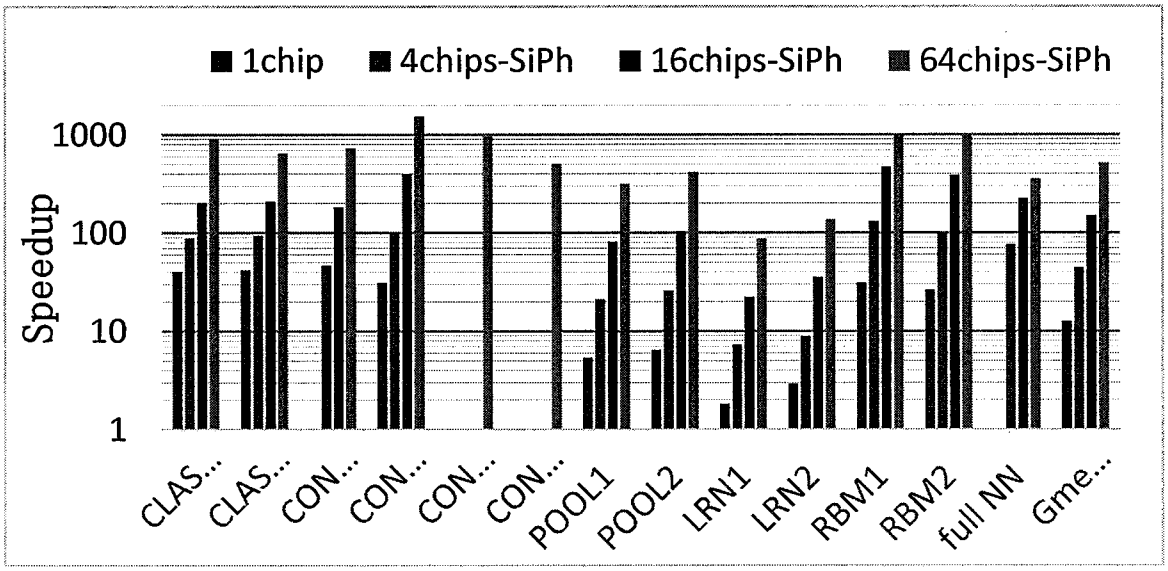


图 6.4 多芯片系统与 GPU 反向训练性能对比

可以看到，对于反向训练，在使用光互联的深度学习处理器多芯片系统中，单芯片，4 芯片，16 芯片，64 芯片的平均性能是 GPU 的 12.62、44.53、150.73、522.07 倍。

能耗

由于 SiPh 的性能高于电互联（HT），因此片间传输使用的时间比较短。并且 SiPh 的功耗同时也低于 HT。因此，在使用 SiPh 以后，多芯片系统的能耗也可以得到降低。图 6.5 和图 6.6 分别显示了在前馈运算和反向训练中，使用光互联的深度学习处理器多芯片系统相对于 GPU 的能耗。

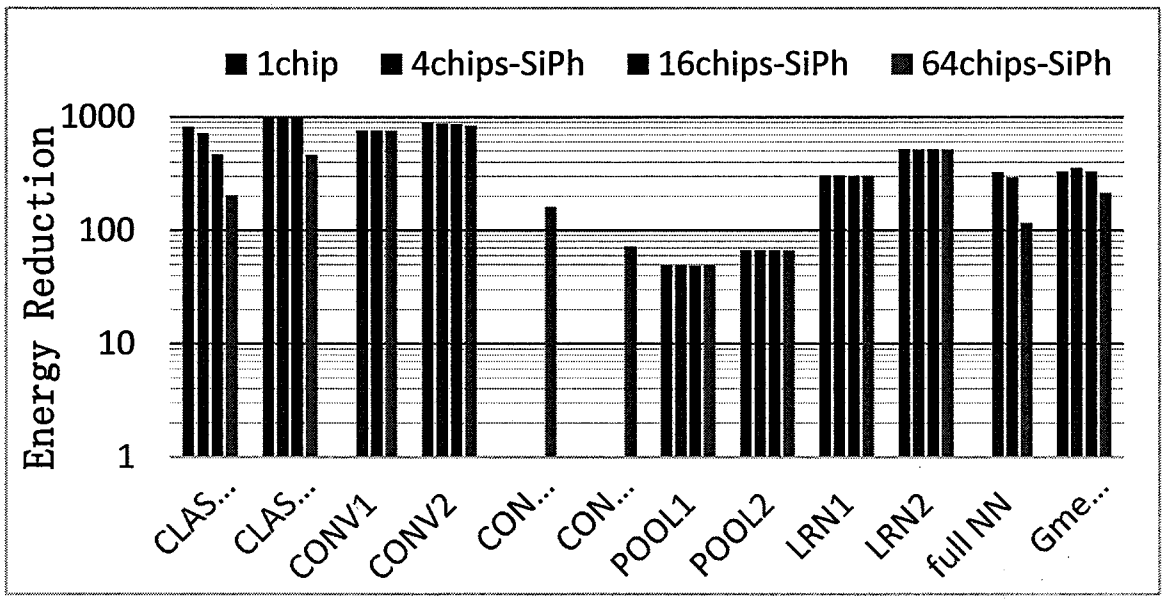


图 6.5 多芯片系统与 GPU 前馈运算能耗对比

可以看到，对于前馈运算，在使用光互联的深度学习处理器多芯片系统中，单芯片，4 芯片，16 芯片，64 芯片的平均性能是 GPU 的 330.56、352.88、331.25、213.44 倍。

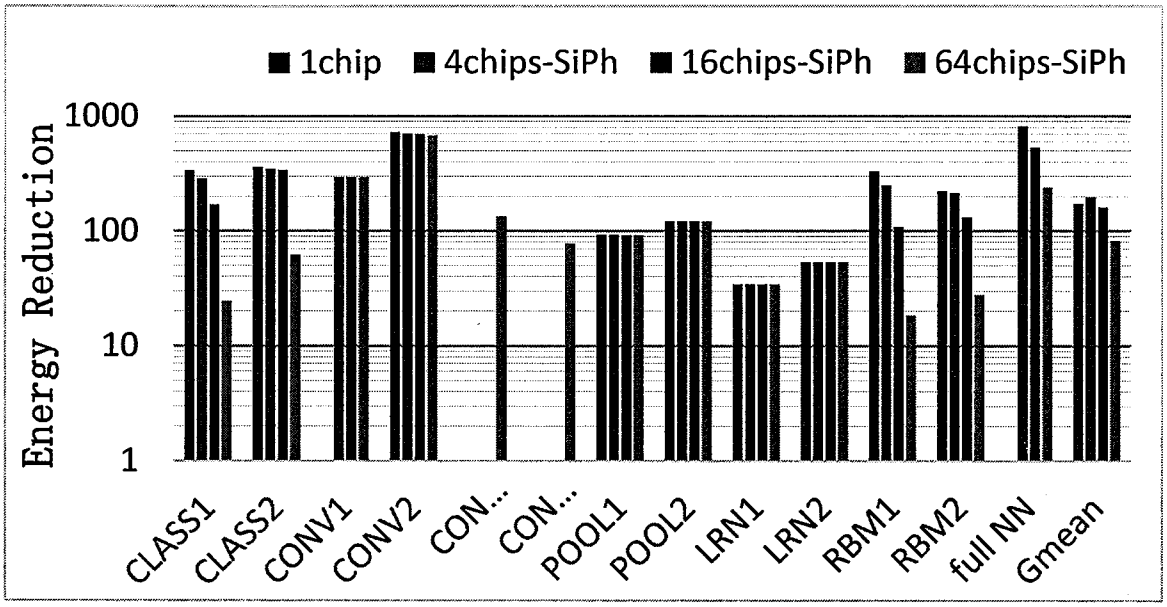


图 6.6 多芯片系统与 GPU 反向训练能耗对比

可以看到，对于反向训练，在使用光互联的深度学习处理器多芯片系统中，单芯片，4 芯片，16 芯片，64 芯片的平均性能是 GPU 的 172.39、198.46、161.13、82.34 倍。

第七章 总结与展望

深度学习是一类多层大规模的人工神经网络方法的统称，目前已经被广泛地应用到云服务器和智能终端的广告推荐、语音识别、图像识别等核心任务上。由于大数据时代的到来，互联网每天都会产生海量的数据需要进行深度学习处理。但通用 CPU 与 GPU 的深度学习处理速度太慢，能耗极高。例如 2012 年谷歌大脑在识别猫脸的深度学习模型中甚至需要使用 1.6 万个 CPU 核进行训练。为了解决深度学习实用化“卡脖子”的速度问题，业界迫切需要面向深度学习的新型处理器芯片。

本文的目标是设计高性能、低能耗的深度学习专用处理器，并设计由多个深度学习处理器组成的硬件平台以进一步提升深度学习处理速度。为实现这个目标，深度学习处理器多核结构，深度学习处理器多芯片互联结构和深度学习处理器片间光互联三个方面进行了研究。

• **多核深度学习处理器设计** 为了进一步提升性能，我们在继承前述单核深度学习处理器设计思想的基础上设计了一个支持多种深度学习算法（如分类层、卷积层、池化层、LRN 层等）的多核深度学习处理器-DaDianNao。在设计多核结构的过程中，我们采用了无访存设计的设计思想来解决访存瓶颈问题，设计了基于 H 树的片上多核互联结构来解决高内部带宽带来的物理连线拥堵问题。模拟实验表明，在 ST 28 纳米工艺下，DaDianNao 的面积为 67.73 平方毫米。对于本文选用的测试集，其平均性能为 Nvidia 的一款通用计算 GPU K20 的 21.38 倍，平均能耗相比 Nvidia K20 降低了 330.56 倍。

• **深度学习处理器多芯片互联设计** 多个 DaDianNao 芯片可以通过互联提供较大的片上存储空间，将所有的神经网络参数存储在片上缓存中，绕过访存墙的限制。并且多个芯片可以提供多组运算单元从而提升运算性能。我们搭建了模拟器来评估多个 DaDianNao 组成的多芯片系统。实验结果表明，对于本文选用的测试集，64 个 DaDianNao 组成的多芯片系统平均性能为 Nvidia 的一款通用计算 GPU K20 的 450.65 倍，平均能耗相比 Nvidia K20 降低了 150.31 倍。

• **深度学习处理器片间光互联设计** 自从1984年Goodman等首先提出集成电路光互联的概念以来，光互联作为一个有效解决电互联潜在问题的办法备受关注。由于在深度学习处理器多芯片结构中，片间数据传输是性能的瓶颈，因此为了使深度学习处理器多芯片系统的性能得到进一步提升，我们对如何在芯片间使用光互联和使用光互联带来的性能提升、功耗下降进行了研究。实验结果表示：64芯片的DaDianNao多芯片光互联系统的性能可以达到Nvidia K20的743.57倍，而能耗降低了213.44倍。

在未来，我们会继续深度学习处理器进行改进。一方面我们将继续改进深度学习处理器核心结构，让本深度学习处理器以更高的性能支持更多的深度学习算法。另一方面，我们将在继续对多芯片互联结构进行优化，以带来性能的提升。