密级:_____

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

硕士学位论文

深度学习的加速器研究

作者姓名：王 佳

指导教师：胡伟武 研究员 陈云霁 研究员

中国科学院计算技术研究所

学位类别：工学硕士

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

2014 年 5 月

Research of Accelerators for Deep Learning

By

Wang Jia

A Dissertation Submitted to
Graduate University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Master of Engineering in Computer Architecture

Institute of Computing Technology, Chinese Academy of Sciences
May, 2014

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：王佳　　日期：2014年4月28日

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：王佳　导师签名：　日期：2014年4月28日

摘 要

近年来异构加速器凭借其优秀的性能功耗比成为了目前体系结构研究的主流方向。同时随着深度学习的兴起，深度学习神经网络的研究也重新回到了机器学习领域的潮头。因此，如何在加速器上高效地实现神经网络处理系统受到了学术界和工业界广泛的关注。

本文从深度学习领域最常见的卷积神经网络出发，抽象出三种典型的神经网络层次，结合算法特点，在目前最常用的基准平台与加速器平台上实现了这些网络层次。主要的贡献包括以下三个方面：

1. 以卷积神经网络和深度神经网络为出发点，抽象并剥离出三种最常见的神经网络层次。并结合不同平台的架构特点，包括 SIMD 体系架构的 Intel SSE 指令集，GPU 加速器的 CUDA 编程环境以及我们自己实现的专用神经网络加速器，对这三种神经网络层算法进行分块化、发掘数据复用性等优化，对算法进行重定制和实现。
2. 在 10 个测试程序上对三种平台进行了神经网络处理的实验。结果表明，专用神经网络加速器在性能上相比于 SIMD 基准平台有平均 117.87 倍的提升，在功耗利用率上有平均 21.08 倍的提升，而面积则仅相当于 Ivy Bridge 架构的 1.87%。相比于 GPU，专用神经网络加速器在性能上有平均 0.22 倍的提升，而面积上则仅有 GPU C2070 片上面积的 0.56%。
3. 通过对实验分析发现，对于神经网络算法，GPU 架构的主要性能瓶颈在于 PCIe 带宽对于数据传输的限制；SIMD 架构的主要性能瓶颈在于并行化程度不够高。专用神经网络加速器则设计 DMA 利用数据复用性优化数据存取流程，设计不完全流水利用计算独立性实现高并发处理，这两点都从体系架构上越过了上述性能瓶颈。

关键词： 深度学习，加速器，数据复用，性能功耗

Abstract

According to its excellent power efficiency ratio, heterogeneous accelerator has been in the main research stream. Meanwhile, with the heat of deep learning, research of deep learning networks has been back into popularity in machine learning. Thus how to efficiently implement neural networks on accelerator has been the main issue in both industrial and academical area.

Here in this research, I generate three typical neural network models from modern convolution neural network, a classical neural network among deep learning models, and implement them in a SIMD baseline and two common accelerators. The main contributions are as follows:

Starting from the architecture of convolution neural network and deep neural network, I generate three typical neural network layers. Combining architectural features of a baseline platform and two different accelerators, including Intel's SSE instruction set with the architecture of SIMD, NVIDIA's CUDA programming environment with the architecture of GPU and our own neural network accelerator, I customize and implement the algorithms with the utilization of tiling and data reuse.

After implementing 10 benchmark programs of neural network processing on 3 different architectures, the performance of our neural network accelerator shows 117.87x faster than the SIMD baseline on average and 21.08x higher in power efficiency on average, with only 1.87% area of Ivy Bridge. Our accelerator shows 0.22x faster than the GPU architecture on average, with only 0.56% area of GPU footprint on chip.

I discover that the bottle neck of GPU performance is the PCIe bandwidth which limits the data transfering. I also find that the bottle neck of SIMD baseline is the less parallelism of the hardware. Nevertheless, our neural network accelerator is equipped with a customized DMA optimizing the data transfer processing utilizing data reuse. Plus, the accelerator comprises multi neurons with staggered pipeline which can deal with more parallelism. These two features of our accelerator have remedied the above defects.

Key words: Deep Learning, Accelerators, Data Reuse, Efficiency and Power

目 录

摘 要 I

Abstract III

目 录 V

图目录 VII

表目录 IX

第一章 绪论 1

1.1 研究背景与意义 1

1.2 相关工作和研究目标 1

1.2.1 相关工作 1

1.2.2 研究目标 2

1.2.3 研究内容 3

1.3 主要贡献 3

1.4 论文结构 4

第二章 深度学习神经网络 5

2.1 人工神经网络概述 5

2.1.1 人工神经网络发展简史 5

2.2 形式化描述 6

2.2.1 神经网络学习过程 7

2.3 Perceptron 描述及学习过程 8

2.4 多层次神经元 10

2.5 卷积神经网络及划分 16

2.5.1 分类层计算模型分析 19

2.5.2 卷积层计算模型分析 20

2.5.3 归并层计算模型分析 22

2.5.4 计算位数对精度的影响 23

第三章 神经网络的 SIMD 实现 25

3.1 SIMD 介绍 25
3.2 SSE 编程环境 26
3.3 SIMD 实现方法 28
3.3.1 Sigmoid 函数的处理 29
第四章 神经网络的 GPU 实现 31
4.1 GPU 架构 31
4.1.1 GPU 并行化特点 33
4.2 GPU 编程模型 34
4.3 卷积神经网络 CUDA 编程模型 35
4.3.1 分类层 CUDA 实现 35
4.3.2 卷积层 CUDA 实现 37
4.3.3 归并层 CUDA 实现 40
第五章 专用神经网络加速器实现 43
5.1 专用神经网络加速器架构 43
5.2 神经网络的硬件实现 46
5.3 控制及实现 48
5.3.1 控制芯片 48
5.3.2 网络层实现代码 48
第六章 实验与结果分析 51
6.1 Benchmark 51
6.2 实验平台 52
6.3 结果分析 53
6.3.1 神经网络加速器与 SIMD 基准的结果对照 53
6.3.2 神经网络加速器与 GPU 加速器的结果对照 55
第七章 结论及未来工作 57
7.1 本文工作及结论 57
7.2 未来研究方向 57
参考文献 59
致 谢 i
作者简介 iii

图目录

图 2.1 神经网络发展鱼骨图 6
图 2.2 Sigmoid 函数曲线 7
图 2.3 Perceptron 神经元模型 9
图 2.4 BP 网络参数示图 13
图 2.5 多隐藏层 BP 神经网络传播示意图 15
图 2.6 分块对带宽影响结果图 18
图 2.7 分类层划分示意图 19
图 2.8 卷积层划分示意图 20
图 2.9 归并层划分示意图 23
图 2.10 UCI 数据集合 32 位与 16 位操作结果准确性比较 23
图 3.1 SSE 编程环境 27
图 3.2 SSE 指令集数据结构 27
图 3.3 典型 16 位定点数 SIMD 指令形式 28
图 3.4 步进 sigmoid 线性拟合函数图像 30
图 4.1 分类层基本算法框架 36
图 4.2 分类层的 GPU 实现流程图 37
图 4.3 卷积层基本算法框架 38
图 4.4 卷积层 GPU 实现流程图（1） 39
图 4.5 卷积层 GPU 实现流程图（2） 40
图 4.6 归并层 GPU 实现流程图 41
图 5.1 专用神经网络加速器架构图 44
图 5.2 神经网络加速器实现 sigmoid 的硬件电路 45

图 5.3 SRAM 读写功耗与 port 宽度关系 46
图 5.4 NBin 读入数据转置示意图 47
图 5.5 CP 控制代码各位置含义 48
图 5.6 神经网络加速器分类层部分控制代码 49
图 6.1 神经网络加速器相比于 SIMD 实现的性能加速比示意图 54
图 6.2 神经网络加速器相比于 SIMD 实现的功耗加速比柱状图 55
图 6.3 神经网络加速器相比于 GPU 实现的性能加速比结果 56

表目录

表 2.1 BP 神经网络参数表 13
表 2.2 Cache 模拟器简单参数列表 18
表 2.3 MNIST 32 位与 16 位操作结果准确性比较 23
表 3.1 步进 sigmoid 线性拟合系数表 30
表 6.1 测试用 Benchmark 51
表 6.2 C2070 参数列表 52
表 6.3 神经网络加速器相比于 SIMD 实现的性能加速比结果 54
表 6.4 神经网络加速器相比于 SIMD 实现的功耗加速比结果 54
表 6.5 神经网络加速器相比于 GPU 实现的性能加速比结果 55

第一章 绪论

1.1 研究背景与意义

相比于传统的同构多核处理器，集成了通用核和加速器的异构多核处理器能提供优秀的性能功耗比，因而成为了目前体系结构领域的一大研究热点，对异构多核架构来说，加速器的设计至关重要。理想的加速器应该能够覆盖广大的应用面，又要能显著地提升性能功耗比（相比于通用核）。

同时，一直以来机器学习作为计算机科学一个非常重要且火热的研究方向，在学术界和工业界都有着非常丰富的工作。近年来随着深度学习的火热兴起，神经网络，这个一度成为机器学习边缘领域的方向也重新得到了重视，工业界利用神经网络的大型应用也是不胜枚举[1-4]。但是现有神经网络算法多是在软件层面进行优化，在传统通用 CPU 上进行搭建，高效的神经网络实现方式成为大家的迫切需求，而加速器作为通用 CPU 的补充与取代，以其优秀的功耗表现引起了大家的注意。但同时要注意到，深度学习应用范围广泛，神经网络模型繁多，曾经出现的很多机器学习加速器都是针对特定应用的[3, 5]，远没有达到大范围适用的地步。

我们刚刚设计并发表了专用的神经网络加速器[6]，能够适用于较大范围的神经网络应用，而 CPU 的 SIMD 因为其通用性作为很多神经网络实现方式的比较基准[7]， GPU 加速器作为最常见的加速方式也是很多学者和工程师在实现神经网络的常用选择[8-10]。因此本文选取在众多神经网络当中脱颖而出的卷积神经网络（CNN）[11]和深度神经网络（DNN）[12]，它们被证明在很多应用当中都有着出色的表现[13]，在现有架构和新型的神经网络加速器上进行实现和优化，对神经网络在各种加速器与基准平台上的表现进行对比，期望得到深度学习算法与神经网络模型在加速器平台上表现的一般规律，并对未来神经网络加速器的发展提供有益的指导。

1.2 相关工作和研究目标

1.2.1 相关工作

在现代处理器设计要求当中，对能量约束的要求越来越多[14, 15]，很多高性能的体系结构都转向了异构多核处理器的框架，这些框架都是传统 CPU 核心和加速器的组合。加速器可能是为了具体一种任务而设计的专用处理器核心也可能是一些不太复杂的 ASIC 电路，例如 H.264 加速器[16]，或者更具灵活性的适用范围更广的处理器[17, 18]，例如

QsCores[19]或者面向图像处理的加速器[20]。

在过去的二十年当中，很多的硬件实现的神经元和神经网络层出不穷[21]，但是这些硬件的研究对象更多地是上千个神经元相连接的生物神经网络，而其研究内容并没有真正的涉及到体系结构领域[22-24]，也没有考虑到计算和存储的复杂度及高效率。实际上，这些实现当中只有极少数的真正应用到计算任务当中 [25-27]。

最近，随着工业界当中大规模神经网络应用的回归，机器学习领域的发展也是如火如荼。虽然软件算法方向不断地发展，但是硬件制约一直是机器学习和神经网络领域的一大瓶颈，专用的神经网络加速器作为传统 CPU 的补充开始进入人们的视线，并且由于神经网络对错误的不敏感[28, 29]，成为了大家在追求低功耗与高性能时候的首选[30, 31]。

面对大规模的机器学习任务，尤其是规模达到上千层，并且每一层都含有大规模神经元和突触的深度学习网络来说，这些程序对于存储能力以及访存速度的敏感程度已经可以和其对性能的追求相媲美。我们希望通过我们的分析和实验能够证实良好的存储能力和访存速度是这类深度学习应用达到良好性能和功耗的关键，对存储问题和数据摆放的深入考量能够对加速器设计及结果产生非常深远的影响。

近年来不仅是在体系结构领域，在神经网络和机器学习领域对于硬件设计的研究也层出不穷，他们期望调研专用的硬件设计来加速神经网络处理过程，特别是加速实时应用。Neuflow[32]正是计算机视觉领域内的 CNN 专用前向加速器，可以达到不错的速度与功耗标准，它根据卷积层和归并层的滑动窗口的特性重新组织了计算部件和寄存器的硬件位置。但它同时也忽略了从输入特征阵列和输出特征阵列的额外复用特性，其对存储的考量也仅仅达到了 DMA 层面，而没有专用的片上存储。虽然 DMA 能够实现很精巧的数据读写方式，但是相比于专用的片上存储，DMA 对性能的提升仍然有限。另外一个和 Neuflow 性能相仿的较复杂的架构是由 Kim 等人[33]提出的，他使用了 128 个 SIMD 处理器，每一个处理器都带有 16 个预先编译好的可执行文件库，但是这样的设计非常的复杂，并且只能支持单一的计算机视觉神经模型。Maashri 等人[34]实现了另外的神经网络模型，使用一系列定制的加速器以及专用的片上网络来模拟和利用数据的复用特性，处理特定的计算机视觉问题。Chakradhar 等人[35]实现了一个可配置的硬件电路用来解决 CNN 问题，但是该加速器主要考虑的是如何将 CNN 神经网络给映射到电路当中，提升带宽的使用效率。

1.2.2 研究目标

本论文希望通过对现有主流深度神经网络（以 CNN 和 DNN 为主）的计算流程的分析，挖掘算法当中的数据复用性和计算并行性。并且对现有主流的并行架构 SIMD 架构、GPU 加速器架构以及我们最新研究的专用神经网络加速器的架构分析，以及具体实验结果的对照来说明在性能和功耗比较标准下深度神经网络的较优选择。

1.2.3 研究内容

本文从常见神经网络的结构入手，对现有深度神经网络的结构进行了深入分析，随后在现有最常见的几种并行体系架构入手，分别对其结构进行分析与实验，实现了深度神经网络在多种体系结构（其中包括两种加速器架构）中的正向传播过程，并且对实验结果进行了分析与对比，发掘不同并行架构下实现神经网络的性能与瓶颈。具体内容包括：

1. 从最常见的人工神经元模型入手对多层次神经元进行理论分析，其次通过对现有业界应用最广泛的卷积神经网络和深度神经网络的模型入手，对一般深度学习神经网络结构进行分离和剖析，划分出三种典型的神经网络层次，并且对每一种神经网络层的数据复用性进行了探讨。
2. 对现有 SIMD、GPU 加速器以及专用神经网络加速器的架构及编程模型进行了分析，结合平台特征以及神经网络算法特点对各种神经网络层次在各个加速器上的实现方式进行了详细的构建。
3. 对各种加速器上的神经网络实现结果进行了分析与比较，指出了神经网络实现的性能瓶颈，对下一步专用神经网络加速器的发展提供了指导。

1.3 主要贡献

本文的主要贡献在于：

1. 以卷积神经网络和深度神经网络为出发点，抽象并剥离出三种最常见的神经网络层次。并结合不同平台的架构特点，包括 SIMD 体系架构的 Intel SSE 指令集，GPU 加速器的 CUDA 编程环境以及我们自己实现的专用神经网络加速器，对这三种神经网络层算法进行分块化、发掘数据复用性等优化，对算法进行重定制和实现。
2. 在 10 个测试程序上对三种平台进行了神经网络处理的实验。结果表明，专用神经网络加速器在性能上相比于 SIMD 基准平台有平均 117.87 倍的提升，在功耗利用率上有平均 21.08 倍的提升，而面积则仅相当于 Ivy Bridge 架构的 1.87%。相比于 GPU，专用神经网络加速器在性能上有平均 0.22 倍的提升，而面积上则仅有 GPU C2070 片上面积的 0.56%。
3. 通过对实验分析发现，对于神经网络算法，GPU 架构的主要性能瓶颈在于 PCIe 带宽对于数据传输的限制；SIMD 架构的主要性能瓶颈在于并行化程度不够高。专用神经网络加速器则设计 DMA 利用数据复用性优化数据存取流程，设计不完全流水利用计算独立性实现高并发处理，这两点都从体系架构上越过了上述性能瓶颈。

1.4 论文结构

本论文的章节具体组织如下：

第一章为绪论，首先介绍了论文的选题背景和研究意义，其次对现有神经网络加速器的相关研究工作进行概述，最后对本文的研究内容进行框架性地介绍并对本文的主要贡献进行一般性总结。

第二章为深度学习神经网络，从最简单的神经元模型出发，逐步介绍至目前最常用的卷积神经网络和深度神经网络，并且抽象出三种神经网络层次，并对每一种网络层的数据复用性进行详细分析。

第三章为 SIMD 指令集实现，作为加速器实现的比较基准，介绍 SIMD 体系结构并且对 SSE 编程环境的分析，以分类层为代表分析神经网络的 SSE 指令集实现。

第四章是神经网络的 GPU 加速器实现，首先对 GPU 的 SIMT 架构与 SSE 的 SIMD 架构的并行化特点进行对比，然后根据 GPU 加速器的编程模型分别对分类层、卷积层和归并层的 GPU 加速器实现流程进行详细规划与分析。

第五章是专用神经网络加速器的实现，对我们实现的专用神经网络加速器的体系架构进行详细分析，并且以分类层的控制代码为例说明各神经网络层在专用神经网络加速器上的实现流程。

第六章是实验与结果分析，对基准测试用例在各个加速器平台上的实现结果进行分析与比较，指出现有神经网络加速器实现的瓶颈。

第七章是结论及下一步计划，对现有工作进行总结，并且提出了下一步神经网络加速器的改进和发展。

第二章 深度学习神经网络

本章首先对人工神经网络的发展简史进行概述，然后从最简单的神经元出发逐步介绍到本次实验所用到的深度学习神经网络的理论基础，其后是本章的主要部分，对深度学习神经网络模型切分成三类常用的神经网络层次，然后在对每个神经网络层次已经分块化处理的基础上对数据复用性进行分析。

2.1 人工神经网络概述

在计算机科学以及相关的计算科学当中，神经网络模型是从生物的中央神经系统当中启发并且抽象出来的，现在常用于模式识别以及机器学习领域。动物的神经系统是大量的神经元或者神经细胞相连而成，通过末梢神经元的刺激信号不断传递进行联系；而类似的，人工神经网络一般是由大量的结点互联而成，从一定的输入数据开始，每个结点都会得到相应的输入数据产生相应的输出数据作为下一个结点的输入，通过这样的方式将信息不断的传递到每一个结点，直到获得输出。同样的，动物的神经系统在学习的时候会不断调整神经元之间的突触连接[36]，人工神经网络也是要调整结点之间的连接权重等。

人工神经网络当中一个比较常见的例子就是手写识别[37]，这也是模式识别的一个具体应用场景：每张手写图片的每一个像素的灰度值作为一个 8bit 值或者 1bit 值都可以作为神经网络的输入结点的输入值，而输入值想要激活神经网络当中的其他结点，是通过结点之间连接权值以及结点内部的作用函数的共同作用的。处在最后的输出结点会被激活，通过输出信息反应这张原始图片是哪个数字或者字母的手写图片。

2.1.1 人工神经网络发展简史

神经网络的研究看起来是最近若干年才兴起的研究课题，但是实际上其发展历史绵长且坎坷。

早在 1943 年时候 Warren McCulloch 与 Walter Pits[38]就根据他们自己对当时生物神经网络发展现状的理解发明了人工神经元，他们的神经元设计非常的朴素，处理的信号都是有着固定阈值的二元信号。在他们二人研究成果的基础上，来自 IBM 的 Farley 与 Clark 联合来自 McGill 大学的生物神经网络专家继续了人工神经网络的研究，这也奠定了以后神经网络的两条发展方向：生理研究与人工智能研究。前者强调不断探索人类大脑的信号处理方式，从而抽象并修复人工神经网络模型，而后者则强调从人工智能的应用场景出发调整人工神经网络模型。

同时在理论上，Donald Hebb 根据生物神经元的变动规律提出了人工神经网络的神经元学习法则，现在称之为 Hebbian 学习法则，是现在常用的最典型的无监督学习法则。而 Frank Rosenblatt 则提出了神经元（perceptron）的概念。

在 1969 年，Marvin Minsky 与 Seymour Papert 发表了神经网络领域著名的论文[39]，使得神经网络相关的研究都停下了脚步，论文中提出神经网络的发展局限主要在于单层神经网络的模型能力有限，甚至连基本的异或电路都不能模拟，另外当时的计算能力完全不足以支撑大型的神经网络的长时间执行。

虽然不久之后 Werbos 提出了反馈学习算法就解决了上面提到的第一个问题，并且在二十世纪八十年代中期兴起的并行分布式处理也可以用来模拟神经网络模型，但是在实际研究以及应用的领域，更为简单的支持向量机（Support Vector Machines，SVM）以及其他线性分类器的使用则更为广泛。

直到 2000 年之后深度神经网络算法的兴起，神经网络的研究才重新步入了大众的视线。Geoff Hinton[40]以及多伦多大学的研究小组发明的各种反馈神经网络的变形算法都可以被用来训练深度网络当中的高度非线性化神经系统。现在常见的深度前馈网络，例如卷积神经网络一般结构都是在由卷积层、归并层和分类层共同组成。Ciresan[1,2]等人曾经利用 GPU 加速器实现的深度神经网络获得过两项模式识别方面的比赛的冠军，这足以证明神经网络在现阶段的发展与广泛的应用。

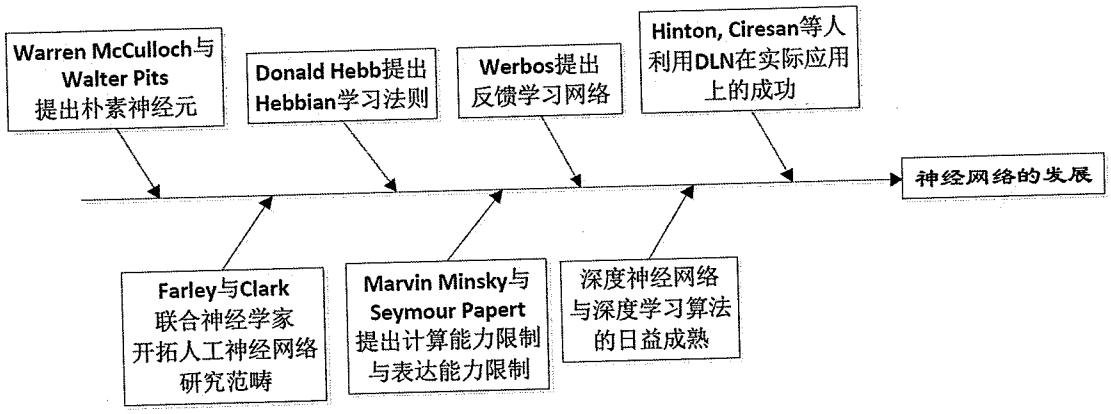

图 2.1 神经网络发展鱼骨图

2.2 形式化描述

人工神经网络当中的网络主要是指系统当中不同层次的神经元之间的连接，以典型的三层神经网络为例，第一层仅有输入神经元，通过将输入的信号值通过突触，也就是神经元之间的连接传递给第二层上的神经元，然后通过更多的突触传递给第三层也就是输出层的神经元。更复杂的网络可能会包含更多层次的神经元，也可能会有更多的输入神经元和输出神经元，突触当中存储的数值称之为权值用来对传输的数据进行影响和修正。

一个神经网络是由三组数据确定下来的：突触的权重，权值更新的速率以及每一个神经元的激活函数。其中权值的学习速率是指权值调整时候的敏感度，激活函数是每个神经元内部的处理函数，用来将该神经元接收到的输入数据转换为输出数据。

数学上一般用一个神经网络映射规则 f(x)来表示一个确定的人工神经网络，一个广泛使用的神经网络映射规则表示形式如下：

$$f(x)=K(\sum_i \omega_i g_i(x))$$

其中 $g_i(x)$ 是神经网络的每一个输入，ω_i 则是该输入所对应的权值。函数 K 是预先设定的函数（即上文所提到的激活函数），一般使用 sigmoid 函数或者双曲正切函数，主要目的是为了将变量的范围缩小到 0 到 1 之间或者-1 到 1 之间。

以 Sigmoid 函数为例，其函数表述及曲线如图 2.2所示：

$$K(t)=\frac{1}{1+e^{-t}}$$

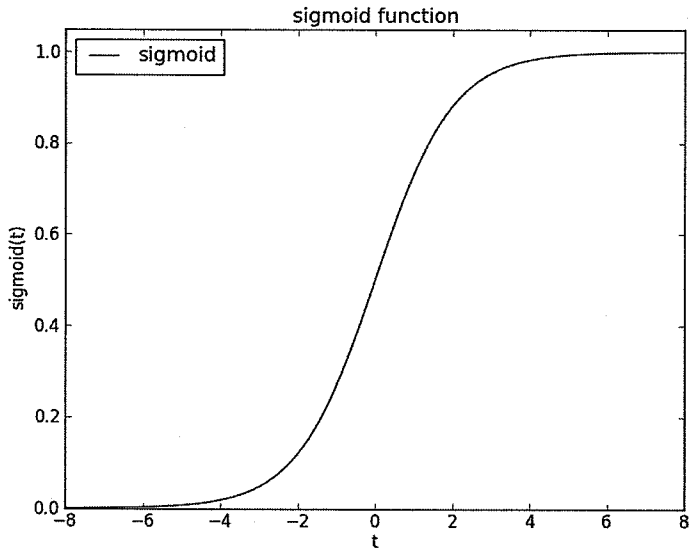

图 2.2 Sigmoid 函数曲线

2.2.1 神经网络学习过程

神经网络学习的过程简单来说是指在给定待解决问题的前提下，我们需要找到一组合适的函数集合 $f^* \in F$ 来解决该问题，该组解可能不是最优，但要在某方面达到预先设定的标准。

在学习的过程当中往往会有一定的代价函数 C: F→R 来衡量特定的函数集合 f 在学习过程中的表现，当然任何可能的解都不会比最优解的代价来的低，即 C(f)≥C(f*)。在学习的过程中代价函数是一个非常重要的概念，它可以告诉我们当前的解与所求的最优解还有多大的差距。神经网络的学习的过程就是通过搜索最合适的函数集合使得其代价

值最小。

有两种常见的抽象学习任务：有监督学习、无监督学习。

有监督学习当中，我们会被预先给出一组数对，(x, y)，$x \in X$，$y \in Y$，我们希望神经网络能够实现这样的映射f: X → Y，来满足之前给出的数对。或者说我们希望找到的神经网络能够表达的映射和实际情况下X与Y之间的映射的差距尽可能的小。这种情况下，我们普遍使用的代价函数一般是均方差函数，而我们的判断标准则是希望当前神经网络的输出结果 f(x)与预先给定的输出结果 y 的均方差尽可能的小。多层神经元的模型一般用来实现这样的神经网络，而梯度下降方法则是人们用来减少误差的常见方式，使用这两者的结合来实现神经网络的反馈过程，修改神经网络各神经元之间连接的权重。

一般模式识别问题（或者称为分类问题）和回归问题（或者称为函数拟合问题）都会使用到有监督学习。由于先验知识（已经给定的数对）的存在，有监督学习有时候又被称为有教师学习，用来在给定的数对的基础上不断找到更精确的映射关系。

无监督学习的训练集合没有人为的标定，也就是说，无监督学习只有输入以及特定的限定规则。虽然在无监督学习当中网络的调整不受到外界的干预，但是需要有额外的组织规则来负责网络内部的调整，可以认为网络所输出的各个模式是相互关联的，如果特定输入引发了特定模式的出现，那么激发该种模式的网络内部特性都会得到增强，反之网络内部特性就会得到抑制。无监督学习中用的神经网络模型是自组织图（self-organized map）等，无监督学习一般适用于聚类问题以及统计分布估计问题，这些问题往往更倾向于输入数据在这些神经网络下面的后验信息。

还有其他类型的学习策略如强化学习、混合学习等，都是能够在实际当中应用到的学习策略。但是主流的连接主义的学习方法主要还是以上两种策略。

神经网络最为成功的地方就是能够从已经观察到的数据建立起所研究问题的模型，而不需要知道所研究问题的本质特征或者因果关系。但是神经网络在使用的时候并不是那么简单，一般首先要根据要解决的问题以及数据规模数据类型来选择合适的网络模型，并且选择合适的学习算法，不同的学习算法在特定的数据应用和前提假设的情况之下的表现大相径庭，并且在未知的数据上面调整算法以及参数需要非常大量的尝试和非常丰富的经验。

下面从两种最为常见的神经网络入手来介绍神经网络具体的使用。

2.3 Perceptron 描述及学习过程

神经网络有监督分类学习当中最常见和最基本的模型是 perceptron 模型，该模型能够得到如此广泛的应用一来是由于它结构简单，二来是因为对于需要精确推理的问题，perceptron 模型能够提供足够的健壮性[41, 42]。Perceptron 模型的一般结构如图 2.3所示。

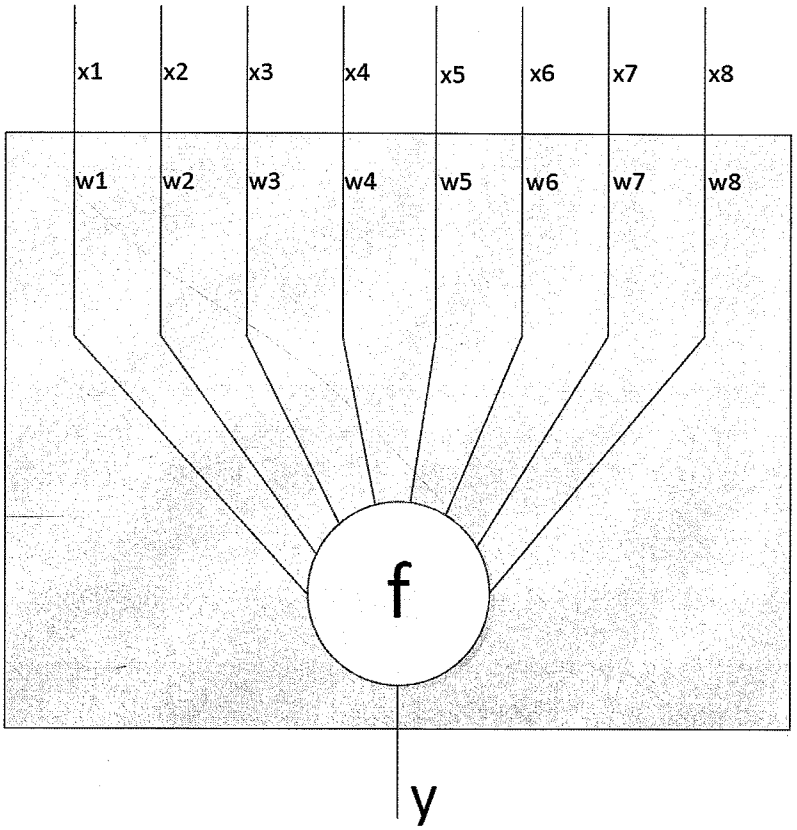

图 2.3 Perceptron 神经元模型

Perceptron 是一个二进制分类器可以将输入数据向量$\vec{x}$映射为单一的二进制输出值$f(\vec{x})$

$$f(\vec{x})=\begin{cases}1 & \vec{x}\cdot\vec{w}+b\geq 0\\ 0 & \text{否则}\end{cases}$$

其中$\vec{w}$是实际权值的向量表示，上述公式当中的点乘可以得到一个乘积之和，而 b 则是偏置（bias），这是预先设定好的常量，与输入数据没有关系。

$f(\vec{x})$的结果是将$\vec{x}$分类为正类或者反类，如果 b 是一个负值，那么点积就需要是一个较大的正值，且其绝对值应超过|b|才能使最终分类结果为正类。因此来说，b 实际上是一个决策门限。

而在复杂的人工神经网络当中，perceptron 模型是构成神经网络每个结点的基本模型，有时候 perceptron 模型也被称作单层 perceptron。作为线性分类器，单层 perceptron 是最简单的前馈神经网络。

在介绍单层 perceptron 的训练之前首先来明确一些概念：

- y=f($\vec{z}$)表示对输入向量$\vec{z}$的输出结果

- b 是决策门限，在我们介绍的例子当中设为 0
- D={($\vec{x_1}$,d_1), …, ($\vec{x_s}$,d_s)}是 s 个训练样本的集合，其中$\vec{x_s}$是一个 n 维向量，而 d_s 则是对于特定输入的期望输出
- $x_{j,i}$表示的是第 j 次训练样本的第 i 个结点的值，并且假设$x_{j,0}$=1
- w_i代表了权值向量的第 i 个值，会与输入向量的第 i 个输入结点相乘，并且由于预先已经设定了$x_{j,0}$=1，因此w_0反而变成了决策门限。
- 我们可以采用$w_i(t)$来表示 t 时刻的第 i 个权重值，可以表示权重随着时间的变化，或者称之为随着训练样本的变化。

在学习的过程当中还会引入学习速率的概念，它表示着权重调整的速率，用α来表示，一般取值是 0<α<1。

因此一般的 perceptron 的使用过程可以表示为：

1. 初始化决策门限与权值向量，一般选择为 0 或者接近 0 的小随机数。
2. 对于训练样本集合 D 中的第 j 个样本($\vec{x_j}$,d_j)，首先可以计算出实际的输出是：

$$y_j(t) = f[\overrightarrow{w(t)} * \vec{x_j}] = f[w_0(t) + w_1(t)x_{j,1} + w_2(t)x_{j,2} + ... + w_n(t)x_{j,n}]$$

然后更新权值：

$$w_i(t+1) = w_i(t) + a(d_j - y_j(t))x_{j,i}$$，对于 i 有 0≤ i ≤ n

3. 对于离线训练来说，第二步停止的前提是迭代的方差$\frac{1}{s}\sum_{j}^{s}[d_j - y_j(t)]$，作为判定的标准，小于一定的门限。或者迭代的次数已经达到了预先设定的次数上限了。

但是 perceptron 的表现能力又是极其有限的，它的训练过程在输入向量不是线性可分的情况下永远无法停止，这也就意味着它对于非线性分类是无能为力的，一个最具代表性的例子就是对异或问题的训练[39]。

由于异或问题是线性不可分的，因此简单的 perceptron 模型并不能够描述并解决足够多的问题。

2.4 多层次神经元

多层次神经元模型是在单层次神经元的基础上发展而来，具有更为丰富的描述能力，可以对非线性分类问题进行描述和操作。其基本的训练过程与单层 perceptron 也是非常相似，前馈的过程也是一致的，唯一的不同在反馈传播的过程。

多层神经元模型采用了反向传播（backpropagation）的反馈策略来调整权值，通过这种方法，输入数据可以被反复的输入到神经网络中进行计算，并对神经网络进行调整，每一次计算得到的实际输出都会与期望的输出进行比较得到误差，根据这一误差神经网络当中的权值可以被调整，使得下一次同样的输入数据通过神经网络计算得到的结果具有更小的误差。这一不断重复的过程就称之为训练的过程。

从具体的问题与神经网络的结构出发可以按照如下步骤来对多层神经元模型或者多层神经网络进行描述：

如果多层神经网络当中的每一个神经元都仅含有线性激活函数的话，那么多层的神经网络就都可以简化为两层的神经网络了。而多层神经网络之所以内容这么丰富，表述能力这么强大，一个关键的原因在于其每个神经元都有非线性的激活函数，这些激活函数相当于人类大脑中的神经元的作用，在数学上来要求，这些人为规定的非线性激活函数需要能够归一化且可导。一般我们会选择诸如下面的 sigmoid 函数来作为激活函数：

$$f(x)=\tanh x=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \quad \text{或者} \quad f(x)=\frac{1}{1+e^{-x}}$$

这两个函数一般都会作用在权重向量与输入向量的点积和之上，然后作为输出。其他还有一些比较特别的激活函数适用于特定类型的神经网络[43]。

算法 2.1 神经网络反馈传播算法

输入：神经网络训练集合

输出：调整后的神经网络模型

初始化：神经网络权值随机初始化

```
1. repeat
2.     repeat
3.         选择训练集合的一个样本
4.         输入到现有神经网络模型
5.         计算输出并且计算其与理想输出的误差
6.         根据梯度下降原则调整权值
7.     until 训练集合当中所有的样本都已经被使用到
8. until 全局误差小于预设上限或者循环次数达到上限
```

一般多层神经网络至少是指三层，输入层、输出层和至少一层隐含层，当然这些层都是由含有非线性激活函数的结点构成的。每一层次中的每个结点都通过权值 $w_{i,j}$ 来和下一层次当中的结点进行互联。

多层神经网络的一个重要特点就是通过反馈传播进行训练和学习。而反馈传播算法的关键就在于定义全局使用的神经网络能量来衡量神经网络的误差，反馈传播的目的就

是尽量减少这一网络能量。一般来说我们定义能量函数如下：

$$E_{TOTAL}=\sum_{p}E_{p}$$

其中

$$E_{p}=\frac{1}{2}\sum_{i=1}^{N_o}(d_i^o-y_i^o)^2$$

能量实际上是下面的参数：

$$E=f(\vec{X},\vec{W})$$

其中权值$\vec{W}$是变量，而输入数据$\vec{X}$则是定值。

每次输入样本输入我们都能得到一个网络能量的值，实际上也就是当前神经网络模型的输出与理想输出的方差，我们的目标就是通过修改权值和阈值门限让这一值尽可能减小。因此我们选取了梯度下降算法来降低调整权重降低误差。

$$\Delta w_{jk}=-\eta\frac{\partial E}{\partial w_{jk}}$$

如上面公式所示，我们希望权值的变化方向是和梯度（梯度本身是有方向的向量，这里展示的是每一个方向上的偏导）的方向是相反的，其中η是学习速率，表示权值调整的速度。

在标准的神经网络一般结构的参数如表 2.1所示。

对于能量函数，有：

$$\frac{\partial E}{\partial w_{jk}}=\frac{\partial E}{\partial s_k}\frac{\partial s_k}{\partial w_{jk}}$$

其中可以用

$$\sigma_k=-\frac{\partial E}{\partial s_k}\text{ 以及 }y_j=\frac{\partial s_k}{\partial w_{jk}}$$

$$（其中 s_k=\sum_{j}w_{jk}y_j\ ）$$

来做替代，得到

$$\Delta w_{jk}=-\eta\sigma_k y_j$$

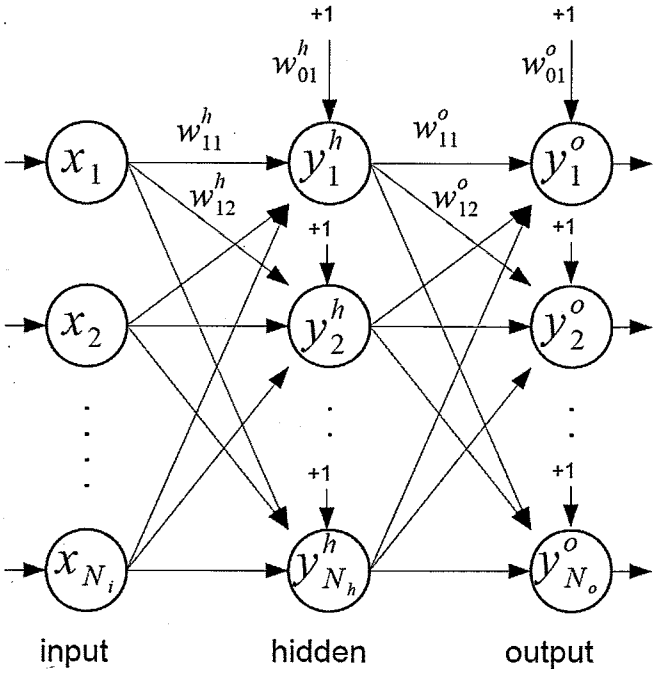

图 2.4　BP 网络参数示图

表 2.1　BP 神经网络参数表

w_{jk}	神经元 j 和神经元 k 之间的权值
w_{0k}^m	m 层中神经元 k 的阈值
w_{jk}^m	m-1 层与 m 层之间连接的权值
s_k^m	m 层中神经元 k 的内部加权和
y_k^m	m 层中神经元 k 的输出
x_k	第 k 个输入
N_i, N_h, N_o	输入层、隐藏层、输出层的规模

进一步分析，在神经网络的输出层有：

$$\sigma_k^o = -\frac{\partial E}{\partial s_k^o}$$

带入 E 的定义 $E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$，有

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

前半部分$\frac{\partial E}{\partial y_k^o}=-(d_k^o-y_k^o)$，后半部分记做$\frac{\partial y_k^o}{\partial s_k^o}=S'(s_k^o)$，为激活函数的微分，带入输入层的权值调整公式有：

$$\Delta w_{jk}^o=\eta\sigma_k^o y_j^h=\eta(d_k-y_k^o)S'(s_k^o)y_j^h$$

而对于隐藏层的权值来说

$$\sigma_k^h=-\frac{\partial E}{\partial s_k^h}$$

后面一部分可以进行展开

$$\frac{\partial E}{\partial s_k^h}=\frac{\partial E}{\partial y_k^h}\frac{\partial y_k^h}{\partial s_k^h}$$

后半部分和前面的输出层是类似的，都是激活函数的微分：$\frac{\partial y_k^h}{\partial s_k^h}=S'(s_k^h)$，前半部分则可以展开为：

$$\frac{\partial E}{\partial y_k^h}=\sum_{l=1}^{N_o}\frac{\partial E}{\partial s_l^o}\frac{\partial s_l^o}{\partial y_k^h}=\sum_{l=1}^{N_o}\frac{\partial E}{\partial s_l^o}\frac{\partial}{\partial y_k^h}(\sum_{i=1}^{N_h}w_{il}^2 y_i^h)$$

根据前面推导的结果$\sigma_k^o=-\frac{\partial E}{\partial s_k^o}$，可以将上式化简为：

$$\frac{\partial E}{\partial y_k^h}=\sigma_l^o w_{kl}^o$$

带入原来的式子有：

$$\frac{\partial E}{\partial s_k^h}=\frac{\partial E}{\partial y_k^h}\frac{\partial y_k^h}{\partial s_k^h}=-(\sum_{l=1}^{N_o}\sigma_l^o w_{kl}^o)S'(s_k^h)$$

$$\sigma_k^h=-\frac{\partial E}{\partial s_k^h}=(\sum_{l=1}^{N_o}\sigma_l^o w_{kl}^o)S'(s_k^h)$$

这样就可以得到隐藏层的权值调整为：

$$\Delta w_{jk}^h=\eta\sigma_k^h x_j=\eta(\sum_{l=1}^{N_o}\sigma_l^o w_{kl}^o)S'(s_k^h)x_j$$

关键的问题都在于激活函数需要是可微的，而我们选取的激活函数，无论是 sigmoid 函数，还是双曲正切函数都是连续且可分的。拿 sigmoid 函数为例子：

$$S'(s_k)=(\frac{1}{1+e^{\gamma s_k}})'=\frac{\gamma}{1+e^{\gamma s_k}}\frac{e^{\gamma s_k}}{1+e^{\gamma s_k}}=\gamma y_k(1-y_k)$$

而对于深度学习当中经常会出现的多隐藏层的神经网络来说，权值调整的策略也是一样的，都是通过后面权值来影响前面的权值

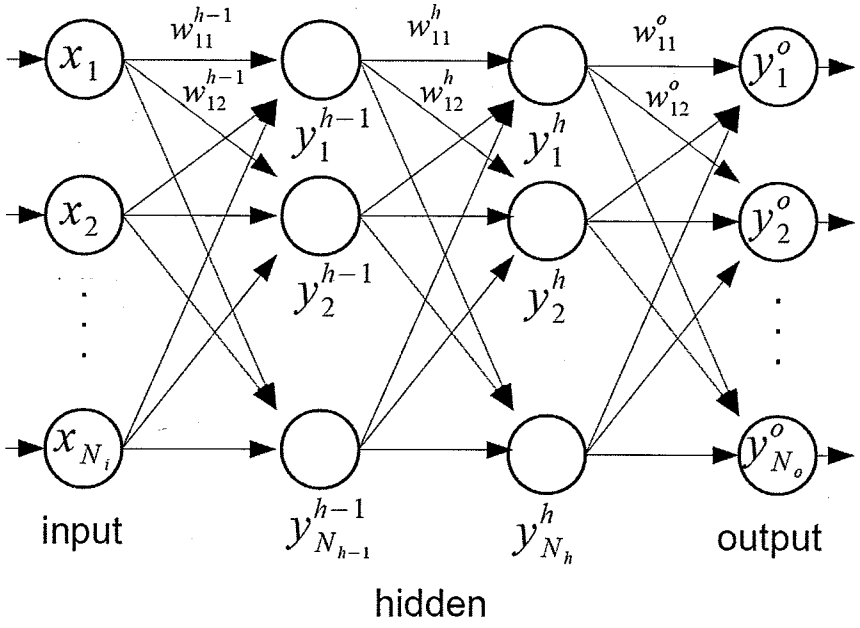

图 2.5 多隐藏层 BP 神经网络传播示意图

如图 2.5所示，h-1 层的权值是由 h 层的σ_k^h来决定的。

在采用 sigmoid 作为激活函数的情况下，输出层权值调整为：

$$\Delta w_{jk}^o=\eta\gamma y_k^o(1-y_k^o)(d_k-y_k^o)y_j^h$$

隐藏层的权值调整为：

$$\Delta w_{jk}^m=\eta\sigma_k^m y_j^{m-1}=\eta\gamma y_k^m(1-y_k^m)(\sum_{l=1}^{N_{m+1}}\sigma_k^{m+1}w_{kl}^{m+1})y_j^{m-1}$$

可以得到权值调整规则为：

$$w_{jk}(t+1)=w_{jk}(t)+\Delta w_{jk}(t)$$

BP 规则非常的流行和经典，现在很多实际实用的神经网络都采用了 BP 规则作为学习策略。并且从上面的 BP 规则的分析来看，反馈过程实际上和正向传播过程的计算方式非常相似，其核心计算仍然是加权和的形式，虽然本次论文主要研究神经网络正向传播在各种体系架构上的实现，但是对于以后扩展到反向传播过程也具有一定的借鉴意义。

2.5 卷积神经网络及划分

深度学习领域最为常见的就是以卷积神经网络（Convolution Neural Network，CNN）为代表的一系列常用神经网络，它是受到生物学上的启发，从多层次神经元（Multi Layer Perceptron，MLP）变形而来的一种神经网络。通过对猫视觉皮层的研究观察[44]，发现视觉皮层的神经细胞的排列有着极为复杂的规则：这些细胞有一部分作为视觉输入称之为接受域，它们通过推进式地激活这一区域的全部细胞来覆盖整个视觉范围，这样的处理方式能够揭示诸如此类的自然形象的空间关联。

卷积神经网络发展的背景也很有趣，正如前面提到的在2006年，随着深度学习网络（Deep Learning Network，DLN）的发展以及计算机视觉领域[45]的兴起，伴随着诸多应用一起进入大家视线的是一系列的神经网络模型，其中最有代表性的就是卷积神经网络以及深度神经网络（Deep Neural Network，DNN）。这两种神经网络模型在结构上有非常高的相似程度，只是在其实现形式以及卷积层的操作上略有差异[11,46]，因此本文主要以卷积神经网络为深度学习领域的代表神经网络模型，来研究深度学习的加速器实现。

在本篇论文当中我们着重探讨的是神经网络的正向传播过程而不是反向训练过程，这一选择是基于技术和市场的综合考虑之后做出的。虽然有很多人都认为从技术上来看，在线的训练和学习过程是非常重要且必须的；但是实际上，对于很多工业上的应用来说，线下训练就已经足够了，神经网络模型在线下用数据集进行大量的训练，在调整好权值阈值等参数之后作为黑盒子成品交递给用户，用户在所需要的场景之中直接使用神经网络的正向传播功能根据输入就可以得到合适的输出，而无需关心黑盒子内部的权值选择与阈值设置。例如，对于手写数字的识别，对于车牌号的识别，对于面部识别和物体识别等应用场景当中，充足的线下训练有非常充裕的训练时间，甚至可以定期在后台进行线下重复训练对模型进行微调，最后展现给使用者的只是训练好的神经网络模型的黑盒；而如果选择进行线上训练，那就意味着非常短的训练时间，根据现有计算机以及加速器的计算能力，这样的训练是非常不充分的，并且训练得到的神经网络模型的准确率也不会理想。统观今日之工业界与学术界，希望加速训练过程的机器学习专家和工程师只是代表了市场的一小部分，面向终端用户的市场则是一大块蛋糕，终端用户需要的是高效率的前向传播网络。有趣的是，最近一些关注硬件加速器的机器学习专家[32]也做出了同样的选择。另外还有一点值得注意，从计算上来看，正如前面分析的那样，训练过程，尤其是反馈传播过程的计算形式与前馈过程非常相似，在对前向传播过程进行充分研究的基础上，后续关注于反向传播过程的实现也不是什么难事。

尽管卷积神经网络和深度神经网络在实现当中有着各种各样的形式与变化，但是它们都有一些共性可以梳理出来：这些实现都是由大量的网络层次堆叠起来的；这些网络层次是串行执行的，网络之间可以被认为是相互独立的；每一个网络层都有自己的输入

特征阵列（input feature maps），权值阵列和输出特征阵列（output feature maps）等。总体来看，不同的网络层大体可以依据计算特征分为三种：卷积层（convolution layer），归并层（pooling layer，又称为二次抽样层 subsampling layer）以及处在模型最顶端的分类层（classifier layer）。

下面对深度学习神经网络架构的三种不同类型的神经网络层次进行简单说明：

➢ 分类层

卷积层和归并层一般都是在卷积网络架构的底层，而在这一架构的顶层往往会搭建一个分类器（classifier），这样的分类器可以是一个简单的线性分类器或者是多层神经元（一般是两层）。与卷积层类似，分类层也需要在神经元的输出之后加上一个非线性函数进行归一化处理，例如 $f(x)=\frac{1}{1+e^{-x}}$，但是和卷积层不同的地方是，分类层对所有的特征数值阵列一视同仁，在分类层当中也就没有了特征数值阵列这一概念了。

➢ 卷积层

卷积层的作用是将前一层的数据应用到滤波器上，考虑到输入数据是一副图像的情况下，卷积层的作用就是将一个 K_x*K_y 大小的窗口作用到一个同样大小的数据块上。最核心的数据是输入层和输出层之间的权值。

一般在输入层中，往往会包含多个特征数据阵列（feature map），而输出层的特征点的计算往往是将多个输入特征数据阵列的同一位置的窗口数据分别与不同权值阵列的窗口进行卷积操作，其核心是一个三维操作，其最内层规模为 $K_x*K_y*N_i$，其中 N_i 就是输入特征数据阵列的个数。在这种情况之下，与全连接多层神经网络不同，这里的连接一般情况之下是稀疏的，即对于每一个输出特征点来说，并不是每一个输入数据都被使用到。卷积层还有一个比较重要的特征是两个连续的窗口之间是有交叠的，如同下面卷积层伪代码当中的 x，y 循环的 s_x，s_y 的步骤。

在有些情况（主要是在卷积神经网络）之下，输入层的不同特征数据特征阵列都使用同一组核特征阵列（kernel）作为权值阵列，对于不同输入点的权重数值是完全一样的，相当于对于整个输入的特征数据阵列来说核特征阵列是共享的，称之为共享核（shared kernel）。而在深度神经网络之中，核特征阵列在不同的输入特征阵列之间是各自独立私有的，称之为私有核（private kernel）。

➢ 归并层

归并层所起到的作用是把相邻输入数据的信息给集合起来，仍然拿图像问题来举例子，归并层主要起到的作用就是输入数据的每一个窗口之内的最显著的特征给保留下来。这样归并带来的一个非常重要的额外作用就是大大降低了输入数据的数据规模和维度。要注意到每一个特征数据阵列都是被归并层独立作用到的，也就是说和卷积层的三维操作不同，归并层在数据上没有交叠，采用的是二维操作。归并层所采用的保留特征的方

式也是多种多样，有对窗口内所有数据取平均值的，有选取最大值的。而在归并层之后是否添加非线性函数也是可选的。

加速大规模神经网络的关键在于如何处理潜在的高内存传输，在下面这一段文字当中本文将就不同神经网络层的数据复用性进行具体的分析。对于本段当中的带宽分析，我们使用了一个连接在理想计算框架下的 cache 模拟器上进行的，这个理想计算框架在每一个 cycle 当中可以计算带有 T_i 个权值的 T_n 个神经元。Cache 模拟器是仿照 Intel Core i7 设计的，其参数如表 2.2所示。

表 2.2 Cache 模拟器简单参数列表

	L1	L2（可选）
大小	32KB	2MB
块大小	64-byte	64-byte
路数	8	8

和 core i7 的设置不同，这里的 cache 模拟器有足够多的端口来提供神经元需要的的 T_n*4 字节的输出数据写出以及 T_n*T_i*4 字节大小的权值数据读入，在实验当中我们选取 T_n=16，T_i=16，虽然在实际应用当中，T_n 与 T_i 都会选择较大的数据。并且这样的 cache 也是很难设计的，但是对我们的研究工作来说，T_n 与 T_i 的选取具有参考价值。

为了说明分块和数据复用对带宽的影响，我们将该 cache 模拟器作用到 4 个 benchmark 程序当中（CLASS1，CONV3，CONV5，POOL3），这些测试程序将在第 6 章当中进行具体说明。

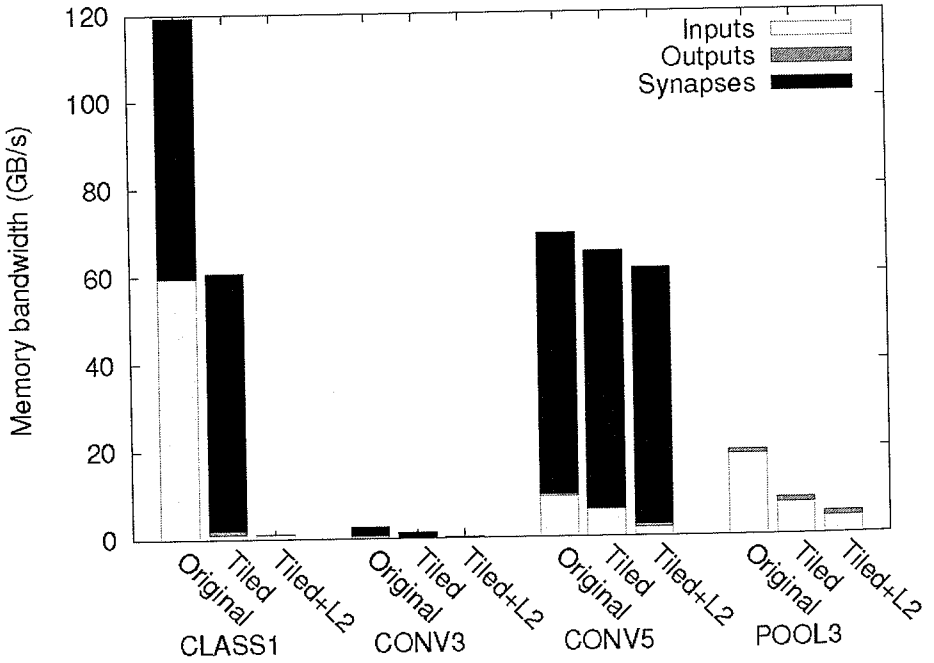

图 2.6 分块对带宽影响结果图

2.5.1 分类层计算模型分析

算法 2.2 分类层伪代码

```
1.  for nnn=0, nnn<Nn, nnn+=Tnn do:
2.    for iii=0, iii<Ni, iii+=Tii, do:
3.      for nn=nnn, nn<nnn+Tnn, nn+=Tn, do:
4.        for n=nn, n<nn+Tn, n++, do:
5.          sum[n]=0
6.        for ii=iii, ii<iii+Tii, ii+=Ti, do: \\以下为核心代码段
7.          for n=nn, n<nn+Tn, n++, do:
8.            for i=ii, i<ii+Ti, i++, do:
9.              sum[n]+=synapse[n][i]*neuron[i]
10.       for n=nn; n<nn+Tn, n++, do:
11.         neuron[n]=sigmoid(sum[n])
```

ii 层循环和 nn 层循环反映的是每个神经元都带有 Ti 个突触权值的 Tn 个神经元的计算框架，因此从上面代码当中可以看出总的内存数据迁移数量是 Ti*Tn+Ti*Tn+Tn 个（载入输入数据+载入突触权值数据+写入输出数据），对于 benchmark 当中的 CLASS1 来说，对应的内存带宽有 120GB/s，是相当巨大的传输量。

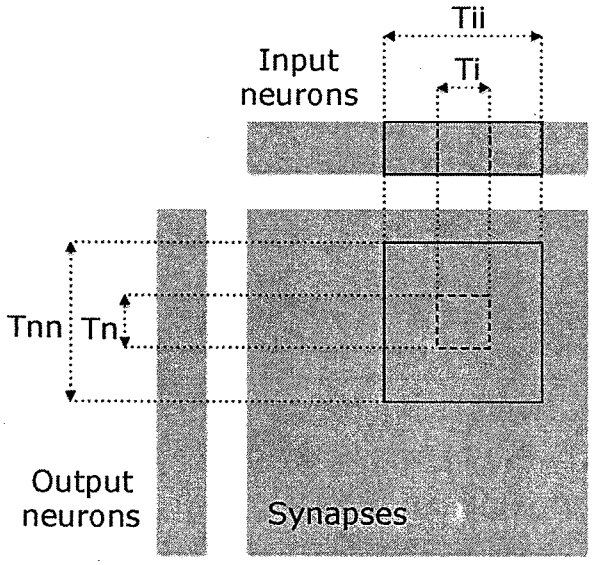

图 2.7 分类层划分示意图

再次分析上面原始代码以及图 2.7，可以看到对于各自的输出神经元来说，每一个输入神经元的数据都会被重复使用，可以考虑到将输入神经元的数据放到 L1 cache 当中。但是一般情况之下，输入神经元的个数可能从几十个到几十万个不等，因此很可能会出现 L1 cache 当中存放不下的情况。因此我们对输入神经元 ii 进行了划分，每次划分的规模是 Ti，这种方式带来的一个缺点是在提升了输入神经元数据复用性的同时，增加了加权和在复用时候的访存距离，也就是说对于每个 sum[n]进行计算的时候，可能会牵涉到

在不同划分当中进行取值，额外增加了缓存与内存之间的数据传输的开销。因此我们也考虑了对 nnn 循环进行划分，划分规模是 Tnn，专门只对部分和进行操作，消除了加权和与输入权值之间的额外距离。这种划分方式能够极大的减少输入神经元的带宽需求，输出神经元的带宽限制得到了放松，这时候带宽的主要限制就放到了突触权值上面。

在神经元模型当中，所有的权值都是独立的，因此神经层内部是没有数据复用的。但是另一方面来看，突触权值在神经层次之间是有复用的，这就成了我们下手的着眼点。例如：对于每一组新输入神经网络的数据之间，突触权值是存在复用的。因此我们一般都会选择一个足够大的 L2 cache 来存放突触权值信息来利用这种复用。对于使用私有核的深度神经网络来说，其突触权值的数量能达到上亿个，这种想法也是不够现实的；但是对于使用共享核的深度神经网络和卷积神经网络模型来说，总共的突触数量是在百万量级，一个足够大的 L2 cache 还是能够容纳的了这么大规模的数据的。如图 2.6所示的 CLASS1-Tiled+L2 的数据，在我们考虑了这么多的数据复用性并且采取了 L1 cache 和 L2 cache 的帮助之下，我们对于内存带宽的要求实际上已经降到了非常低的水平。

2.5.2 卷积层计算模型分析

我们考虑一个二维的卷积神经网络层，与分类层的不同点在于，卷积层的输入和输出是多个特征阵列数据，并且拥有 K_x*K_y 的核阵列。

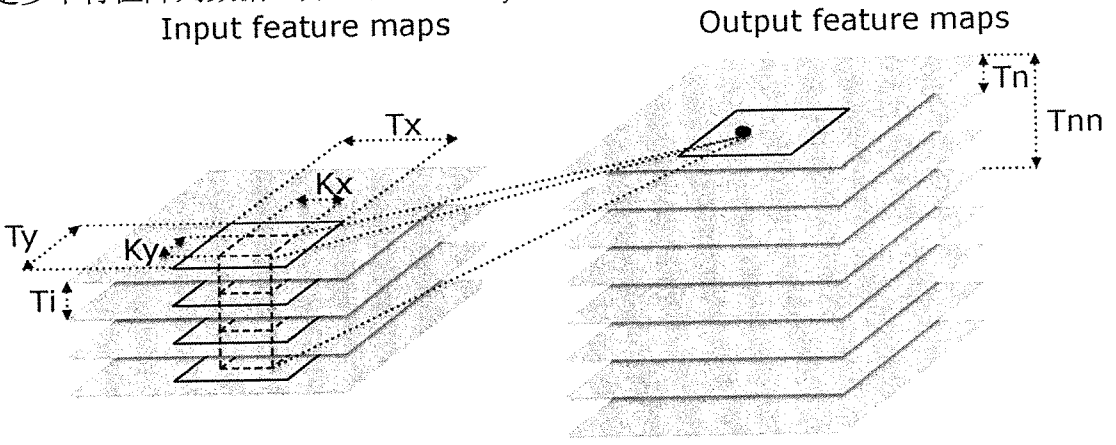

图 2.8 卷积层划分示意图

对于输入输出有两个可以重复使用的机会：滑动窗口可以扫描输入层数据造成的滑动窗口间数据交叉的复用，以及 N_n 个输出特征阵列之间的数据复用（每个特征阵列的计算都要触及 N_i 个权值特征阵列）。前者涉及到的复用次数最多为 $\frac{K_x * K_y}{s_x * s_y}$，后者涉及到的数据复用次数为 N_n。针对面前一种情况，正如伪代码当中描述的一样，我们使用了 Tx 与 Ty 的划分，但是对于后一种情况，我们没有采取任何措施，其原因在于一般 K_x 与 K_y 大小一般在 20 以下，而 N_i 的大小在数十左右，因此 $K_x*K_y*N_i$ 的核阵列权值数据都是能够在 L1 cache 当中放置的。如果这三个参数的值比较大的时候，我们也有补救办法，可

以对输入的特征阵列进行划分，正如分类层当中引入的对 ii 的划分 iii 一样。

算法 2.3 卷积层伪代码

```
1.  for yy=0, yy<Nyin, yy+=Ty, do:
2.    for xx=0, xx<Nxin, xx+=Tx, do:
3.      for nnn=0, nnn<Tn, nnn+=Tnn, do: //以下为核心代码段
4.        yout=0
5.        for y=yy, y<yy+Ty, y+=sy, do: //为 y 分块
6.          xout=0
7.          for x=xx, x<xx+Tx, x+=sx, do: //为 x 分块
8.            for nn=nnn, nn<nnn+Tnn, nn+=Tn, do:
9.              for n=nn, n<nn+Tn, n++, do:
10.               sum[n]=0
11.             for ky=0, ky<Ky, ky++, do: //滑动窗口
12.               for kx=0, kx<Kx, kx++, do:
13.                 for ii=0, ii<Ni, ii+=Ti, do:
14.                   for n=nn, n<nn+Tn, n++, do:
15.                     for i=ii, i<ii+Ti, i++, do:
16.                       //共享核版本
17.                       sum[n]+=synapse[ky][kx][n][i]*neuron[ky+y][kx+x][i]
18.                       //私有核版本
19.                       sum[n]+=synapse[yout][xout][ky][kx][n][i]
                                    *neuron[ky+y][kx+x][i]
20.             for n=nn, n<nn+Tn, n++, do:
21.               neuron[yout][xout][n]=non_linear_transform(sum[n])
22.         xout++
23.       yout++
```

对于使用共享核的卷积层来说，所有的 xout，yout 输出特征阵列都会使用同样的核阵列权值参数，这本身占用的内存带宽就非常的低了，正如图 2.6当中 CONV3 列示意。但是鉴于整个共享核的大小达到了 Kx*Ky*Ni*No，这超出了 L1 cache 的容量，因此我们仍然需要对No进行划分(划分规模为Tnn)，使得共享核的大小减小到Kx*Ky*Ni*Tnn，这样就可以进一步减少卷积层整体的内存带宽需求。

对于使用私有核的卷积层来说，所有的突触权值都是独有的，不存在复用的可能性，其内存带宽需求如图 2.6中的 CONV5 所示，与分类层的行为非常的相似，然后对于分类层来说，如果 L2 cache 足够大的话，在层次之间仍然存在着复用的可能性。而对于卷积层，虽然在卷积层当中，有（sx，sy）这一步骤的存在以及稀疏的输入数据与输出数据之间的关联，从网络结构上大大减少了私有核潜在的突触权值的数量，但是必须的突触权值数量依然有上亿字节的规模，远远超过了 L2 cache 的容量，造成的超高的内存带

宽也是无法避免的。

值得注意的是，在学术圈依然有大量关于私有核与共享核的辩论[13, 47]，在机器学习领域关于私有核取代共享核的倾向也不是非常明确。就加速器领域的实际测试来看内，选用私有核还是选用共享核在统一架构下有着截然不同的表现，这一点也给选用私有核还是共享核时需要额外考虑的。

2.5.3 归并层计算模型分析

与卷积层不同，归并层的输入特征阵列的数量与输出层的特征阵列的数量是相同的，更重要的是，卷积层中没有私有核与共享核的概念，也就不需要存储这额外的突触权值。每一个输出特征阵列的数据完全是由 Kx*Ky 个输入特征阵列的二维数据窗口来决定的，这就造成了数据的复用性只能在滑动窗口当中才能体现（而不是由输出特征阵列和滑动窗口共同决定的）。通过上面的分析也能看到，归并层当中可以利用的数据复用较少（基本只存在于滑动窗口的边缘部分），相比于卷积层，归并层的输入神经元要求的内存带宽要相对较小，而对 Tx，Ty 进行划分操作带来的提升效果也相对较小，正如图 2.6所示。

算法 2.4 归并层伪代码

```
1.  for yy=0, yy<Nyin, yy+Ty, do:
2.    for xx=0, xx<Nxin, xx+Tx, do:
3.      for iii=0, iii<Ni, iii+Tii, do: //以下是核心代码段
4.        yout=0
5.        for y=yy, y<yy+Ty, y+=sy, do:
6.          xout=0
7.            for x=xx, x<xx+Tx, x+=sx, do:
8.              for ii=iii, ii<iii+Tii, ii+=Ti, do:
9.                for i=ii, i<ii+Ti, i++, do:
10.                 value[i]=0
11.               for ky=0, ky<Ky, ky++, do:
12.                 for kx=0, kx<Kx, kx++, do:
13.                   for i=ii, i<ii+Ti, i++, do:
14.                     //平均归并
15.                     value[i]+=neuron[ky+y][kx+x]
16.                     //最大归并
17.                     value[i]=max(value[i], neuron[ky+y][kx+x])
18.         neuron[xout][yout][i]=value[i]/(Kx*Ky)
19.         xout++
20.       yout++
```

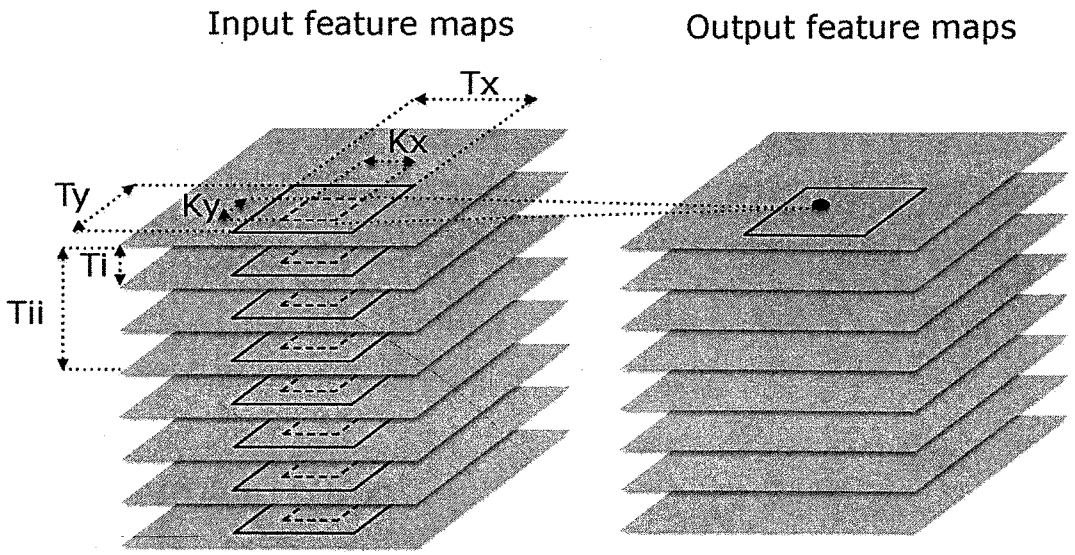

图 2.9 归并层划分示意图

2.5.4 计算位数对精度的影响

表 2.3　MNIST 32 位与 16 位操作结果准确性比较

类型	错误率
32 位浮点操作符	0.0311
16 位定点操作符	0.0337

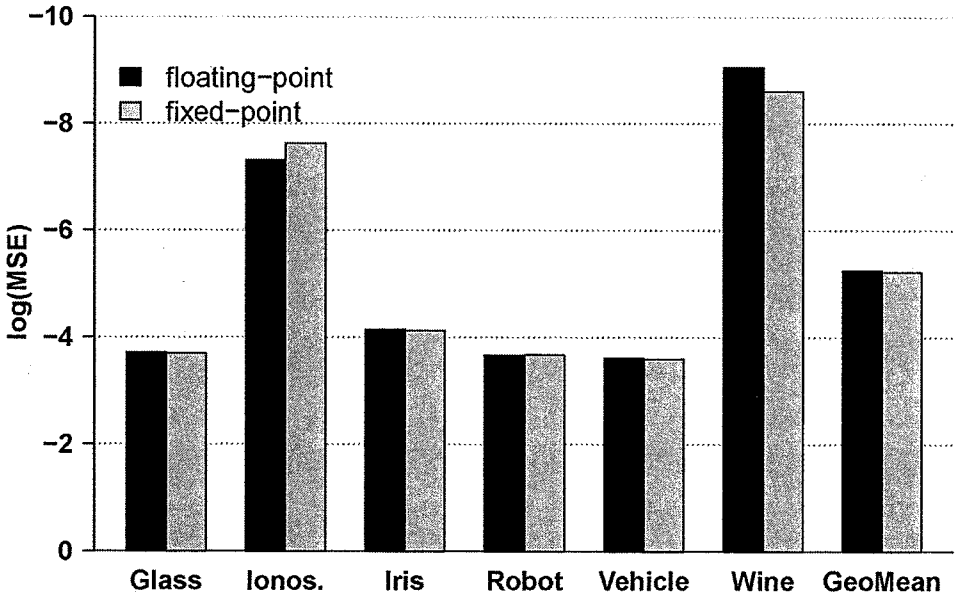

图 2.10　UCI 数据集合 32 位与 16 位操作结果准确性比较

我们选择16位的定点算术指令替代了32位的浮点算术指令来作为整体的评价标准，这也是由于最后要比较的神经网络加速器的选择决定的。虽然看起来很不可思议，但是很多文献[48-50]当中都有充分的证据表明即便是位数很小的操作指令（8 位甚至更少）对于神经网络的准确度的影响也是非常有限的。为了进一步验证和揭示这一事实，我们训

练并且测试了加州大学欧文分校的机器学习算法集合，其结果如图 2.10所示。另外我们还训练了标准的 MNIST 机器学习测试程序[11]（手写数字识别程序），在同时采用了 16 位和 32 位操作指令（在测试当中使用了交叉验证方法），结果如表 2.3所示。对于定点运算符来说，我们定义其前六位为整数部分，后十位为浮点部分。以上实验的结果都说明了我们采用 16 位运算符对于神经网络精确性带来的影响微乎其微。

第三章 神经网络的 SIMD 实现

上面具体介绍了各种神经网络层次的算法及分块过程，下面就是考量主流的体系架构，以及上面设计的神经网络层次的核心算法在不同加速平台上的实现。作为最为经典和最为常见的 SIMD 结构，也一直以来成为实现神经网络这样的并行算法的主流平台之一[7, 51]，也是我们评价其它加速器时候最常用的比较基准。

本章首先对 SIMD 架构以及其编程环境进行介绍，然后以分类层为代表介绍了神经网络层次在 SIMD 架构上的实现。

3.1 SIMD 介绍

单指令流多数据流（Single Instruction Multi Data）是针对计算机系统结构进行分类的费林分类法（Flynn's taxonomy）的一个种类，SIMD 描述的计算机体系结构拥有多个处理部件，可以同时对多个数据进行同一种操作，这样的结构能够实现数据流并行：有并行的数据处理，但是每次只能执行 1 条指令。一般 SIMD 结构在数字图像或者是视频音频的处理问题当中非常常用，而神经网络算法也需要对大量数据施加相同操作，这也是我们经常选用 SIMD 作为神经网络层次实现的评价基准的重要原因。而 SIMD 架构作为传统 CPU 架构在面对诸多并行应用的所做出的反应和调整，理应做为我们深度学习加速器研究的一个比较基准。

在面对日益常见的并行计算的应用需求之下，x86 架构引入了全新的名字、全新的计算能力、全新的指令集，SIMD 硬件和指令集就是在原来的 SISD（Single Instruction Single Data，单指令流单数据流）基础上引入的扩展的功能和内容。传统的 SISD 架构将一条操作执行到一条数据上面，而 SISD 架构致力于发掘数据流的并行性（Data parallelism）。数据流并行性是指具有统一类型的大规模数据，它们都需要完全相同的指令来进行处理。典型的数据并行性的例子就是在数字图像处理当中，将 RGB 图像反色：图像当中的每一个像素值都需要被遍历到，并且执行相同的减法操作——多数据单操作。当然对于并行问题的处理，在 SISD 架构之上，人们还会挖掘并行问题的另一种性质，即指令流并行性（Instruction Level Parallelism），主要目的致力于在相同的数据流上面执行多条指令，在深度学习的计算任务当中这样的需求并不多，因此这里也就不做深入讨论。

SIMD 架构当中最基本的单元是向量（vector），这也就是 SIMD 计算也被称作向量处理的原因，向量仅仅是一组独立的数字或者标量的集合。传统的 CPU 是作用于标量的（超标量 CPU 是指可以同时在多个标量上执行不同操作的架构），而 SIMD 当中的以向

量为基本操作单元就是指对向量当中含有的多个同一类型的标量，同时执行完全相同的数据操作。

这些向量在操作时被称为打包向量格式，数据都是以字节（byte，8-bit）或者字（word，16-bit）的形式存储，而后打包成向量继续操作。在设计 SIMD 架构时候一个很重要的问题在于同时执行多少个数据元素最为合适。如果想要执行 32-bit 的单精度浮点（single precision floating point）操作的话，那么在此 SIMD 架构之下可以同时对 4 个数据进行 4 路操作，也可以用 2 个 64-bit 的向量进行 2 路的双精度浮点操作。因此每个向量的大小就决定了使用者可以同时处理特定类型数据的数量。

对于我们本次要使用到的 SIMD 指令集 SSE（Streaming SIMD Extensions），是 Intel 在 x86 架构下设计的 SIMD 扩展指令集，它是 Intel 在 1999 年为了回应 AMD 发布的 3DNow!在 MMX（Multi Media eXtensions）指令集的基础上做的进一步扩展。

Intel 最开始的 IA-32 基础上的 SIMD 指令集是 MMX 指令集，但是其存在两个问题：它复用了现有的浮点数寄存器而不是单独使用自己的 SIMD 寄存器，这就导致了它不能同时进行浮点指令操作与 SIMD 指令操作；MMX 指令集也只能作用到整型数据上。SSE 指令集引入了全新的寄存器组（XMM 寄存器组），并且可以使用更多的整型指令与浮点指令，更易于应用到数字信号处理与图像处理领域。并且 SSE 指令集还有后续的扩展指令集，每一次扩展都会针对不同的问题和需求提供一系列的指令，例如 SSE4 中的指令多是针对文本的并行处理的。我们本次使用到的指令基本到 SSE2 就足够使用了。

3.2 SSE 编程环境

如果要使用 SSE 指令集，首先需要接触到的就是 SSE 指令集当中的寄存器组。图 3.1 展示了 SSE2 扩展指令集的编程环境，SSE2 扩展指令集并没有引入新的寄存器，而是使用了原来 MMX 引入的 MMX 寄存器组，SSE 引入的 XMM 寄存器组，以及传统的 IA-32 通用寄存器组：

XMM 寄存器组：总共有 8 个，均为 128 位，可以处理组合的或者标量的双精度浮点操作，其中标量双精度浮点操作是指可以处理存放在 XMM 寄存器当中低 64 位的双精度浮点数。XMM 寄存器还可以处理 128 位的组合整型数据，我们在本次实验当中主要使用的就是这一类型的操作。这 8 个寄存器的命名方式是 XMM0 到 XMM7

MXCSR 寄存器：仅有 1 个，32 位寄存器，可以显示在浮点操作当中的状态位和控制位。例如其中的 denormals-to-zeros 位与 flush-to-zero 位可以为处理类似溢出的非常规操作提供更高效的处理方法。

MMX 寄存器组：共有 8 个，均为 64 位，可以用来处理组合整型数据操作。也可以为了数据在 MMX 寄存器和 XMM 寄存器之间进行数据转移提供过渡，其命名方式为 MM0 到 MM7。

通用寄存器组：这就是最常见的通用 IA-32 寄存器，适用于 IA-32 架构的地址模型及寻址操作。可以被用来存储一些 SSE 指令的操作数，根据功能不同分别命名为 EAX，EBX，ECX，EDX，EBP，ESI，EDI，ESP。

EFLAGS 寄存器：32 位寄存器，用来存储一些比较操作的结果。

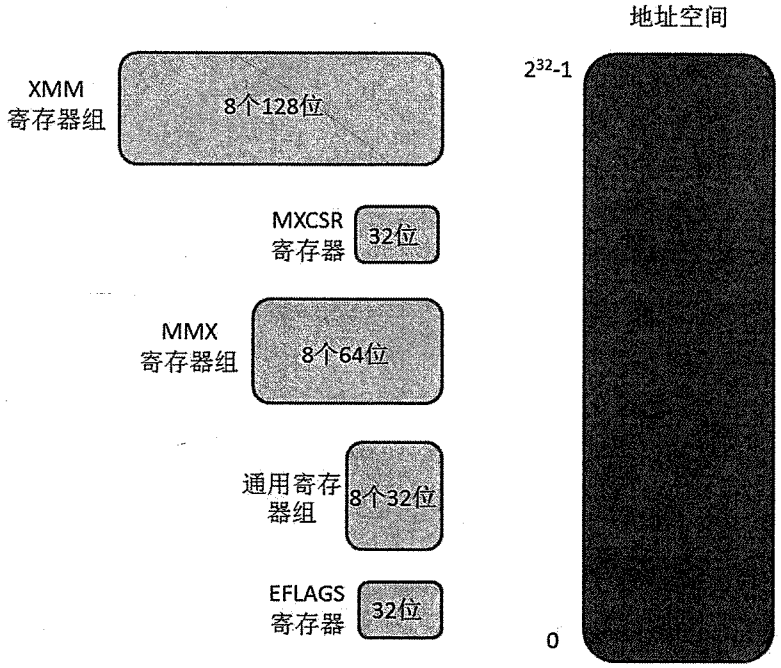

图 3.1　SSE 编程环境

SSE 当中的数据类型如图 3.2所示，主要是考虑到待处理数据类型的位数以及寄存器的位数综合决定最后采用的数据类型。在本次实验当中我们所要处理的是 16 位定点型数据，所要采用的是 128 位组合字整数的数据结构。这种数据结构在一个寄存器当中可以容纳 8 个 16 位定点数据。

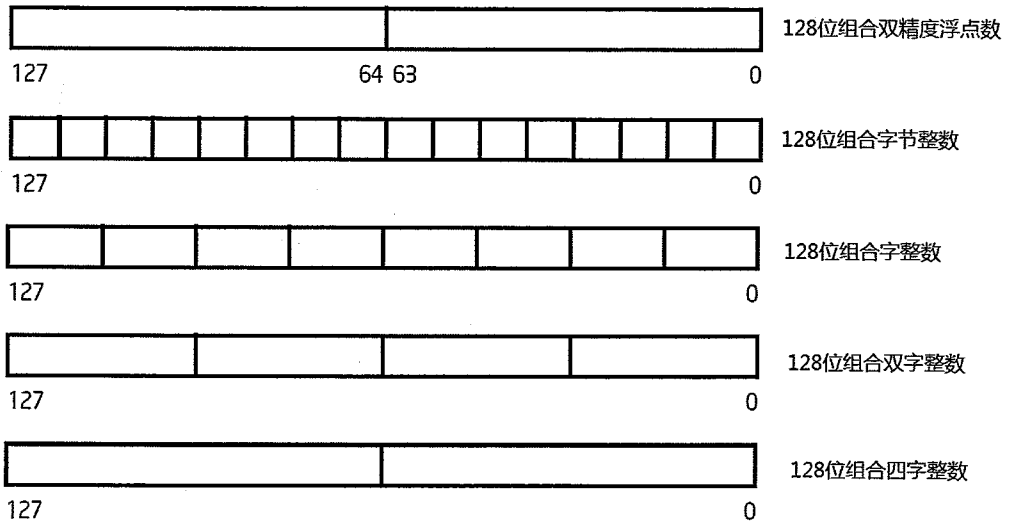

图 3.2　SSE 指令集数据结构

一般 SSE 指令集当中的指令都可以按照功能分为以下几组：

- 数据搬移指令
- 算术指令
- 比较指令
- 转换指令
- 逻辑指令
- 随机指令

典型的 16 位定点数的 SIMD 指令形式如图 3.3所示。

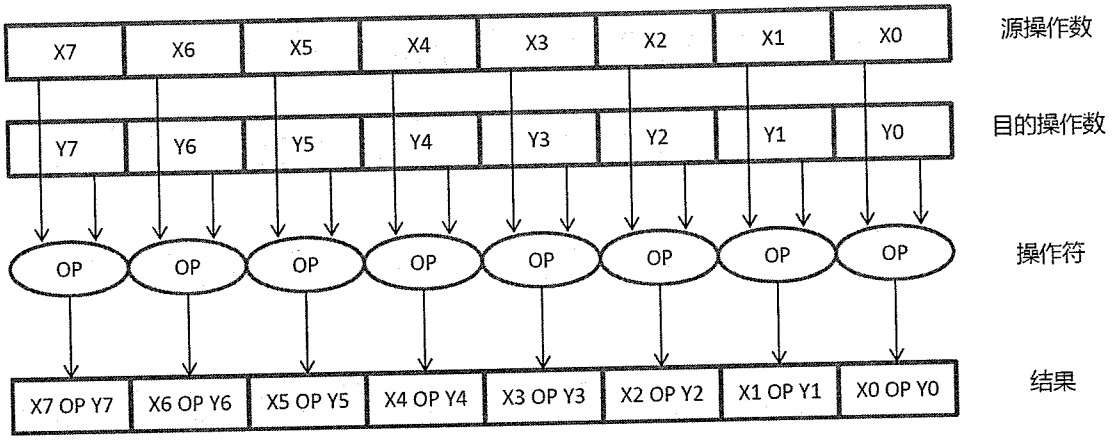

图 3.3 典型 16 位定点数 SIMD 指令形式

对于组合操作来说，源操作数（即源操作寄存器）都包含着 8 个 16 位定点数据，而目的操作数（即目的操作寄存器）也包含着 8 个 16 位定点数据，在每次操作的过程当中 16 个并行的操作可以同时进行，并且写入到目的操作寄存器的相应位置。而标量操作则是和传统的 x86 浮点操作相似，目的寄存器的高位不发生变化，而低位数据则和源寄存器进行算术或者逻辑等操作，写入到目的寄存器的低位。

我们这里主要使用的是组合操作指令。

3.3 SIMD 实现方法

由于要想对数据进行操作，首先要将数据从内存当中搬运到寄存器当中，因此在程序当中首先要使用的就是数据搬移指令。我们这里需要的是将 8 个非对齐 16 位定点数从内存当中搬移到寄存器当中，因此需要对需要处理的输入数据数组以及突触上的权值数组进行对齐及组合操作，保证数据搬移指令的正确处理。

然后需要对寄存器当中的数据进行算术指令操作。值得注意的是下面算法当中的第 2 行到第 5 行每一行都能在一次操作当中完成，这也是 SIMD 相比于传统 x86 架构的最大优势，并行性得到了一定程度上的利用。SSE 指令集当中不仅有上面提到的标准 SIMD 指令（如 PADDSW，PMULLW 等），也提供了类似 PHADDSW 垂直加法指令这样的特殊 SIMD 指令来供调用，充分实现了并行化（如第 5 行当中展示）。

算法 3.1 SIMD 实现 MLP 的核心算法

初始化：xmm0 寄存器作为临时加和的存储置为 0，即 xmm0[0-7]=0

1. **循环开始**
2. 将输入神经元权值载入到 xmm1 寄存器 xmm1[0-7]
3. 将突触神经元权值载入到 xmm2 寄存器 xmm2[0-7]
4. 做 SIMD16 位组合定点数乘法 xmm2[i]=xmm2[i]*xmm1[i]（0≤i≤7）
5. 做垂直加法：xmm0[0]=xmm0[0]+xmm0[1]，xmm0[1]=xmm0[2]+xmm0[3]，
   xmm0[2]=xmm0[4]+xmm0[5]，xmm0[3]=xmm0[6]+xmm0[7]，
   xmm0[4]=xmm1[0]+xmm1[1]，xmm0[5]=xmm1[2]+xmm1[3]，
   xmm0[6]=xmm1[4]+xmm1[5]，xmm0[7]=xmm1[6]+xmm1[7]
6. **循环结束**
7. 对 xmm0 寄存器自身做 3 次垂直加法，xmm0[0]即为所求加权和
8. 将 xmm0 寄存器数据写入到内存
9. 对加权和进行 stepwise sigmoid 操作

在实现以及调优的时候还需要考虑到 SSE 指令集当中每一条指令的延迟（latency）和吞吐率（throughput），延迟是指该条指令的结果在多少个周期（cycle）之后才能被使用，吞吐率是指在几个 cycle 之后才能再次发射同一条该种指令。过长的延迟和过高的吞吐率都会给系统带来性能瓶颈，在调优时利用循环展开、调整指令顺序等方式都可以提升程序的性能。

3.3.1 Sigmoid 函数的处理

另外在系统当中由于需要使用 sigmoid 函数，而 sigmoid 函数当中大多涉及到了指数操作，这在 SSE 指令集当中没有相应的支持，并且在传统 x86 的指数指令的执行速度要远低于乘加操作的处理速度，因此我们决定对 sigmoid 函数进行分段线性化。

算法 3.2 步进 sigmoid 算法

输入：x　　输出：y

1. 将输入数据定点化，x=int(t)
2. 查表得到 x 对应的斜率 k 与截距 b
3. 计算 y=k*x+b
4. 输出 y

对于

$$K(t) = \frac{1}{1+e^{-t}}$$

其中的 t 首先要进行近似为整数的操作 x=int(t)，对于 x 要进行查表计算 K(t)，表格如下：

表 3.1 步进 sigmoid 线性拟合系数表

x	k	b	x	k	b
小于等于-8	0	0	1	0.1498	0.5813
-7	0.0006	0.0050	2	0.0718	0.7373
-6	0.0016	0.0118	3	0.0299	0.8643
-5	0.0042	0.0278	4	0.0113	0.9369
-4	0.0113	0.0631	5	0.0042	0.9722
-3	0.0299	0.1357	6	0.0016	0.9882
-2	0.0718	0.2627	7	0.0006	0.9950
-1	0.1498	0.4187	大于等于 8	0	1
0	0.2311	0.5			

表中系数是按照最小二乘法结合原始 sigmoid 函数的各点取值拟合出来的结果。按照上面所示的线性步进 sigmoid 函数做出图像并且与原始的 sigmoid 函数的图像做出比较，得到下面的图像。

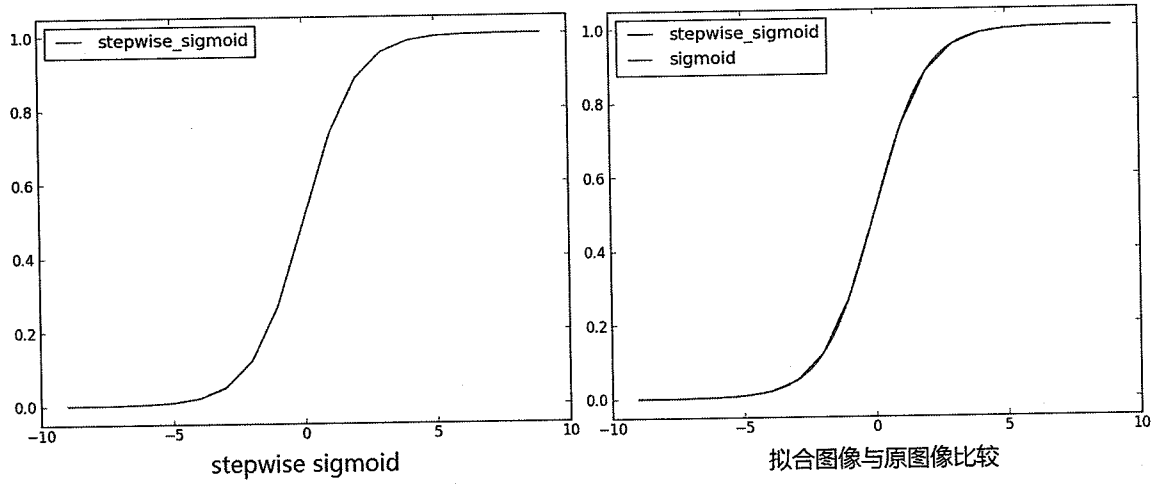

图 3.4 步进 sigmoid 线性拟合函数图像

从图 3.4中可以看到，线性拟合的步进 sigmoid 函数与原始 sigmoid 函数的行为非常相似，完全可以取代原来的 sigmoid 函数在神经网络当中被使用。并且步进 sigmoid 函数是分段线性函数，避免了非线性计算的过大开销。

因此在上面 mlp 层算法的最后，在每一层的加权和从 SSE 寄存器当中迁移到内存之后，就可以按照上面的步进 sigmoid 的方式线性计算出该层输出。

其它神经网络层的实现方式都与 MLP 层类似，关键的问题主要在于发掘各自网络层的计算的并行性，并结合 SSE 指令集选择合适的指令，充分利用数据并行性。上面提到的步进 sigmoid 函数也能够为整体实现提升速度。

第四章 神经网络的 GPU 实现

近年来，很多卷积神经网络和深度神经网络算法都是在 GPU 上实现的[3, 9, 37]，虽然这些实现要么面向的对象任务太过单一[9, 37]，亦或者是面向的数据规模太小[10]，但是它们表现出来的效率要比传统 SIMD 的比较基准要好，并且 GPU 也是最为常见和使用最为广泛的加速器，因此我们在选择神经网络加速器实现方式的时候，GPU 成为了不能错过的一个环节。

本章首先介绍了 GPU 的架构和 CUDA 编程模型，根据 GPU 本身的结构特点和 CUDA 编程的一般准则，结合之前的神经网络的 GPU 实现方式，考虑到对各层神经网络层次的 GPU 片上划分，分别介绍了分类层、卷积层和归并层的 CUDA 实现的执行流程。

4.1 GPU 架构

一般我们使用到的 GPU 是 GPGPU，其全称是通用图形处理单元（General Purpose Graphic Processing Unit），主要适用于计算机图形的处理工作，分担了原来传统 CPU 的部分工作。所有的 GPU 都提供了一套完整的操作集合来处理 GPU 可能遇到的数值及逻辑操作。最为重要的是，GPU 在现代计算机当中对于图形操作等并行操作能够比传统 CPU 更好地发掘其并行性，提高程序执行效率。

本次采用的主要是 GPU 的架构为 fermi 架构，而 GPU 在最近的发展历程上也经历了 G80 架构到 GT200 架构到 fermi 架构到 kepler 架构的转变。但是现代 GPU 的整体架构的框架却一直没有发生大的变化，下面就可以以 fermi 架构为例简单介绍一下本次使用到的实验环境当中的 GPU 架构。

现代的通用 GPU 架构是由很多固定的图形函数流水线构成的，这些流水线要么包含着处理几何图形的硬件要么包含着处理像素的渲染器：几何处理器可以处理一些涉及到原点、直线、三角形等的原语操作，而像素渲染器则能够处理一些用于填充图形内部光影效果的光栅插值。但是在 Tesla 架构推出之时推动的统一处理（Unified Processing）模型下面，现在的 GPU 架构更着重关注抽象的设计元素与更高效的资源利用率。

在现代 GPU 架构的框架中，GPU 是由 SM（Streaming Multiprocessor），DRAM 控制器以及片上的二级 cache 构成的。根据 GPU 型号的不同，每个 GPU 当中的 SM 的数量，以及每个 SM 中流处理器（Streaming Processor，SP）的数量是各不相同的。一般每个 SM 当中会有 32 个单指令流多线程流（Single Instruction Multiple Thread，SIMT）处理通道（SIMT lane），每个 SIMT 通道都可以在一个 cycle 下面在一个线程当中发射一条

指令，这样每个 SM 在每个 cycle 当中就可以同时发射 32 条指令。每 32 个线程被划分为 1 个 warp，在 GPU 程序执行的时候，warp 是 GPU 程序执行调度的最小单元，每一个 warp 当中所有的 32 个线程是共用一个程序计数器的（Program Counter，PC）。每一个 SIMT 通道都各自有自己的寄存器文件，其读写速度是非常快的，并且不同 SIMT 通道能够同时访问一个低延迟的片上共享存储（shared cache）。对于寄存器文件和共享存储的合理使用能够让 SM 充分利用带宽，来保证每一个操作符在每一个 cycle 当中都有两个输入和一个输出而发射执行，不需要因为带宽不够获取不到输入数据而空转。

GPU 的存储模型也是需要关注的一个重点。整体来看在 GPU 当中每个 SM 都会有相应的一级缓存，而 GPU 上的所有操作则是共用统一的二级缓存。分开来说：

- 寄存器：每一个 SM 都有 32K 个寄存器，在程序执行时候，每一个线程就会被分配到一定数量的寄存器，这些寄存器是不会被其它线程所使用的。一般在 CUDA 的核函数当中所允许使用的寄存器数量是 63 个，如果程序需要划分为相当多数量的线程来执行的话，那么每一个线程的核函数当中允许使用的寄存器数量最多则将为了 21 个。寄存器文件都有非常高的读写带宽，一般约为 8000GB/s。
- 一级缓存（L1 cache）与共享缓存（shared cache）：这两种缓存要放在一起是因为这两种存储本身都是片上存储，但是其用途可以作为一级缓存来存放每个线程各自独立的数据也可以作为共享缓存来存放多个线程需要共享的数据。一般片上存储有 64KB 大小，可以被配置为 48KB 的一级缓存加上 16KB 的共享缓存或者配置为 16KB 的一级缓存加上 48KB 的共享缓存。共享缓存是我们本次实现神经网络当中需要重点关注的对象，因为它可以在被同一个程序块（block）当中的所有线程来共享数据，对于卷积层当中共享核的存在，对于分类层当中每个中间节点都需要共享所有的输入数据的存在，以及其它很多神经网络当中数据复用情况的存在，这种共享数据的方式对于减少访存开销是非常有意义的。共享缓存可以让在一个线程块当中的线程协调合作，复用片上数据，能够大量地减少计算单元与存储之间的通信开销。一级缓存与共享存储有着较短的延迟，一般为 10 到 20 个 cycle，较高的带宽，一般为 1600GB/s。对于神经网络这样需要可复用存储来存放中间数据的程序来说，有效地利用共享存储可以大大减少中间数据在存储与计算单元之间传输的开销。
- 设备存储（Device Memory）：设备存储是用来存放没有被存放在寄存器以及共享缓存当中的数据的。有时候在线程当中可能需要存放大量的变量，而上面提到的寄存器文件是不能满足需求的，这就是本地存储发挥功用的时候。一般设备存储还支持存放以下类型的数据：常量大小的数组类型数据，超过寄存器文件支持上限的数组类型数据，从本次存储当中读取的需要应用到各个线程块上面的规模较大的数据等。有较大的读写延迟，一般为 400 到 800 个 cycle。
- 二级缓存（L2 Cache）：二级缓存是多个 SM 共用的，可以认为是处理单元从设备存

储当中读写数据的过渡缓存，二级存储也支持原子操作，但是这些原子操作所涉及的数据必须能够在不同线程块和核函数之间进行共享。

- 宿主存储（Host Memory）：CPU 存放数据的地方，一般在 GPU 程序开始时候会在 CPU 上开辟空间存放数据，然后搬移到 GPU 上的设备存储作为输入数据。在程序的结尾，也需要在 CPU 的宿主存储当中开辟空间来接收从 GPU 迁出的输出数据。

上面提到了 CUDA 的存储结构，而 CUDA 的计算单元可以称之为 CUDA 核（CUDA cores），包含了 GPU 上最核心的数据处理的模块。而这些模块一般分为：

- 整数算术逻辑单元（Integer Arithmetic Logic Unit）：支持所有符合传统编程需求的 32 位定点操作，并且也对 64 位以及更多扩展位的算术逻辑操作的精度进行了优化处理。这也就是说，传统编程环境当中所有可用的指令与操作在 GPU 上都是可以实现的。
- 浮点处理单元（Floating Point Unit）：实现了最新的 IEEE 754-2008 标准规定的所有浮点标准操作，并且提供单精度版本和双精度版本的聚合乘加（Fused Multiply-add）等指令。在每一个 cycle 当中，每个 SM 当中最多可以处理 16 个聚合乘加操作。

从 GPU 本身设计的出发点来说，它起初是面向图形处理运算的，因此其在可用操作和可编程性等方面都有着诸多的限制要求，根据 GPU 的架构设计，对于那些可以并行化处理的问题，GPU 虽然能够提供较为高效的解决方案，但是使用起来也必须要遵循一定的规则。

一个很重要的要求就是 GPU 所能处理的问题，与 SIMD 概念当中描述的那样，需要对一组数据流中的每一个数据都执行相同的操作。这些操作可以组合放置到一个函数当中进行实现，也就是核函数（kernel function）。GPU 可以并行化地将同一个核函数作用到数据流当中的每一个数据当中，当然其前提是数据流本身已经提供了数据并行性，而这条限制在神经网络算法当中是不成问题的。无论是在分类层、卷积层还是归并层当中，输入数据之间是相互独立的，一旦输入数据进入了核函数的入口，就可以相互独立的进行操作和变化，处理完成之后写入到各自的存储当中，而不需要考虑到是否有数据需要同时被读写发生混淆。另外对于 GPU 来说，如果执行的计算过程有着较高的计算密度，那么利用 GPU 对应用进行加速将会非常有利，而较低的计算密度则会成为加速的性能瓶颈，不同神经网络层次在计算密度上略有差别，这也能在最后的实验结果当中有所体现。

4.1.1 GPU 并行化特点

从上面的 GPU 的架构以及处理模型当中可以看到，虽然 GPU 也是多线程的并行，但是在多个线程上执行的相同的操作，而传统的 CPU 的线程并行，则是开辟更多的线程执行不同的操作，两者在本质上有着较大的区别：

CPU 并行化特点：

- 任务并行化
- 多个任务被映射到多个线程当中
- 不同的任务执行不同的指令
- 最多支持在几十个核心上支持几十个重量级线程并行执行
- 每一个线程都需要人为地精确管理与调度
- 每一个线程都需要分别编程控制行为

GPU 并行化特点：

- 线程并行化
- 同一任务被映射到多个线程当中
- 不同数据上执行相同指令
- 数百个核心上并行执行上万个轻量级线程
- 线程通过硬件来管理调度
- 编程框架已经实现了对线程的批量操作

4.2 GPU 编程模型

CUDA（Computing Unified Device Architecture）是 NVIDIA 制定的并行编程平台和编程模型，作用在 NVIDIA 生产的 GPU 上。CUDA 使得程序员能够方便使用 GPU 当中的指令集，并且对片上以及片下的存储空间进行直接的操作。在 CUDA 的帮助下，传统的 GPU 除了能够在图形处理方面发挥作用之外，在通用的并行问题处理当中也能有相当不错的表现。

程序员可以使用 CUDA 提供的加速库、第三方提供的预编译指令以及传统的编程语言来在 CUDA 平台上发挥能力解决问题。根据 GPU 架构的不断改进，CUDA 的版本也随之不断的更新，从最初 2007 年的 CUDA SDK 到现在的 CUDA 6.0，虽然 CUDA 的 API 函数以及实现方式可能发生了较多的修改，但是 CUDA 的实现都是向前兼容的，之前版本的程序可以不经修改直接运行于更高版本的 CUDA 上。

一般 CUDA 程序的执行步骤分为一下四步：

1. 拷入数据：将 GPU 需要处理的输入数据从 CPU 的内存当中（Host Memory）拷贝进入 GPU 的存储当中（Device Memory）。但是在这一过程当中，可能会因为系统总线的带宽上限和延迟带来一定的性能瓶颈。GPU 本身也考虑了这个问题，一般采用 GPU 本身的 DMA 控制器来保证数据的异步传输，充分利用带宽，尽量掩盖传输延迟造成计算核心空等的影响。
2. CPU 指导 GPU 处理数据：CPU 决定对 GPU 的硬件采用怎样的并行化分组进行处理，如对大规模问题划分为多个线程块进行执行，而每个线程块当中又有若干个线程分

别执行较小规模的同质化的问题。在 CUDA API 将数据从 CPU 传输到 GPU 之后，CPU 就可以为 GPU 预先划分问题规模，在 GPU 执行阶段就可以按照这样的规模划分执行细粒度的问题。

3. GPU 各个核心之间并行执行：在 GPU 进行执行的时候，各个核心就按照预先写好的 CUDA 核函数进行处理和执行，各个线程按照 warp 的发射顺序被分配到流处理器核心上，执行指令和操作。一个优秀的 kernel 函数可以指导硬件行为，大大提升资源使用率。但这同时也是通用 GPU 的一个制约，对于没有经验且不了解 GPU 本身设计架构的 CUDA 程序员来说，面对复杂问题写出高效的解决方案基本是不可能的。
4. 拷出结果：在 kernel 函数执行完毕之后，在 GPU 存储上的数据就是我们需要的结果，我们可以将 GPU 存放于设备存储当中的数据拷贝到 CPU 的宿主存储当中。当然这一过程也会受到 PCIe 带宽的限制和延迟的影响，这也成为了传统 GPU 实现的一个非常重要的瓶颈。一般解决方法就是对程序所使用到的数据进行考量，并且对 GPU 的片上存储中的数据进行尽可能地复用，减少数据在 GPU 和 CPU 之间的传递。

对于本次将要处理的神经网络层的 GPU 实现，主要困难在于各层次数据的划分与核函数的书写。另外需要说明的是，在传统的神经网络 GPU 实现当中，多数处理的是 32 位浮点程序，这在 GPU 当中有着大量的库函数可以使用[8, 52]，但本次我们由于面临的是 16 位定点数据的处理，就需要从最底层的数据划分与性能的优化来重新考虑每一个神经网络层的实现。这并不是毫无意义的重复性劳动，也不是闭门造车的行为。我们这样做一方面是一开始就选择了以 16 位定点数作为统一的处理标准，另一方面来说我们在第二章当中按照分块化思路对神经网络各层模型进行重新划分的算法在传统的 GPU 实现当中也是不曾出现过的[53]。对于全新的模型和全新的数据标准，从头进行设计和实现也非常自然。

4.3 卷积神经网络 CUDA 编程模型

4.3.1 分类层 CUDA 实现

从之前的分析可以看到，分类层算法的前一部分部分和的计算与向量矩阵乘法非常相似，后面的 sigmoid 函数并不是并行化的重点，之前 SIMD 当中实现 sigmoid 的方式基本可以照搬到 GPU 的实现当中，这里我们主要考虑的是 MLP 的矩阵向量乘的实现。原本的实现如图 4.1所示，输入数据是一个 1*TI 的向量，突触权值数据则相当于一个 TI*TN 的矩阵，最后的输出则向量与矩阵相乘得到的一个 1*TN 的向量。

单纯的以每一个 thread 来计算每一个输出的结果是最自然的想法，但是这样的话无法充分利用 GPU 上大量的 SM，导致 SM 的闲置，也导致了并行化程度不够高，最后的程序的效率自然也还是有充分的提升的空间。

图 4.2展示的就是改进之后的分类层的 GPU 实现方式。如图中展示，执行过程被分为了两步，第一步是求部分和操作，第二部是对部分和进行累加的操作。

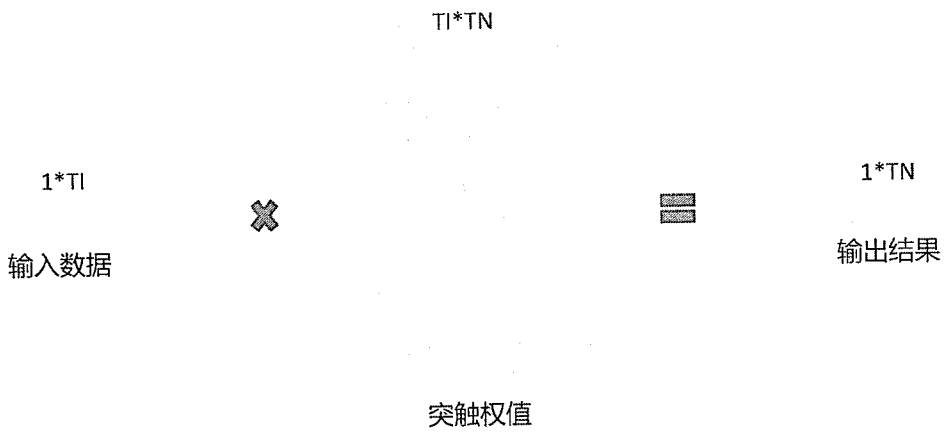

图 4.1 分类层基本算法框架

在第一部分当中，CPU 在为 GPU 设置 blocks 维度时候，采用了二维的维度即（TN/BLOCKSIZE，TI/ROWNUM），其中 BLOCKSIZE 和 ROWNUM 是预先设定的变量值。而 threads 的维度则放弃了最自然的用每个线程计算每个输出节点加权和的划分方式，而是直接采用了对输出节点数量 TN 进行划分为 TN/BLOCKSIZE，这样做的目的是可以调整 BLOCKSIZE 以适应 GPU 的计算能力，获得较好的程序性能。同样的，在每一个核函数当中，每一个线程需要对 ROWNUM 个输入数据进行乘法和加法操作，对于每一个输出节点来说，就相当于原来有 TI 次乘法操作之后得到 TI 个乘积，然后对这 TI 个乘积做加法操作，现在则只对每 ROWNUM 个数据做乘法操作，然后对每 ROWNUM 个乘积做加法操作，得到 TI/ROWNUM 个临时和，并且将这些临时和存放到 GPU 存储上面。这样做的目的在于，原本每一个输出节点都要进行 TI 次乘法及 TI 次加法操作，这样的并行化程度只能达到原来的 TN 倍，而在现在的情况下由于临时和的存在，并行化程度能够达到原来的 TN*TI/ROWNUM 倍，并且 ROWNUM 也可以根据具体的 TN 和 TI 做定制，一来是能够充分使用到硬件资源，二来则是根据程序的表现选择最优的参数。得到的中间和需要存放在一个 TI/ROWNUM*TN 的数组当中，这一点在图中也得到了体现。

第二步则是对这些中间和做累加，划分的维度则是和第一步当中划分的维度以及得到的结果相呼应的，其 blocks 维度为 TN/BLOCKSIZE，其 threads 维度为 BLOCKSIZE。在核函数执行之前不需要对数据进行迁移，因为在第一步当中已经有了同步操作，所有的 threads 的计算任务都已经完成，部分和都已经存放在了 GPU 的设备存储当中。核函数所要做的就是每个线程都要对 TI/ROWNUM 个临时和进行加和操作，并不会涉及到任何存储上的传输操作。

一个比较遗憾的地方就是部分和如果可以存放到共享存储当中的话会对程序的性能

还有进一步的提升，但是由于临时和的数量虽然相比元原始突触权值的数量有大规模的减少，但是将大规模的神经网络将临时和存放在共享存储当中还是不现实的事情，这里所做的妥协也是不可避免的，但是实际操作当中将输入数据放到共享存储当中对于程序性能还是有较大提升的，在这分类层的代码当中也是能够体现出来的。

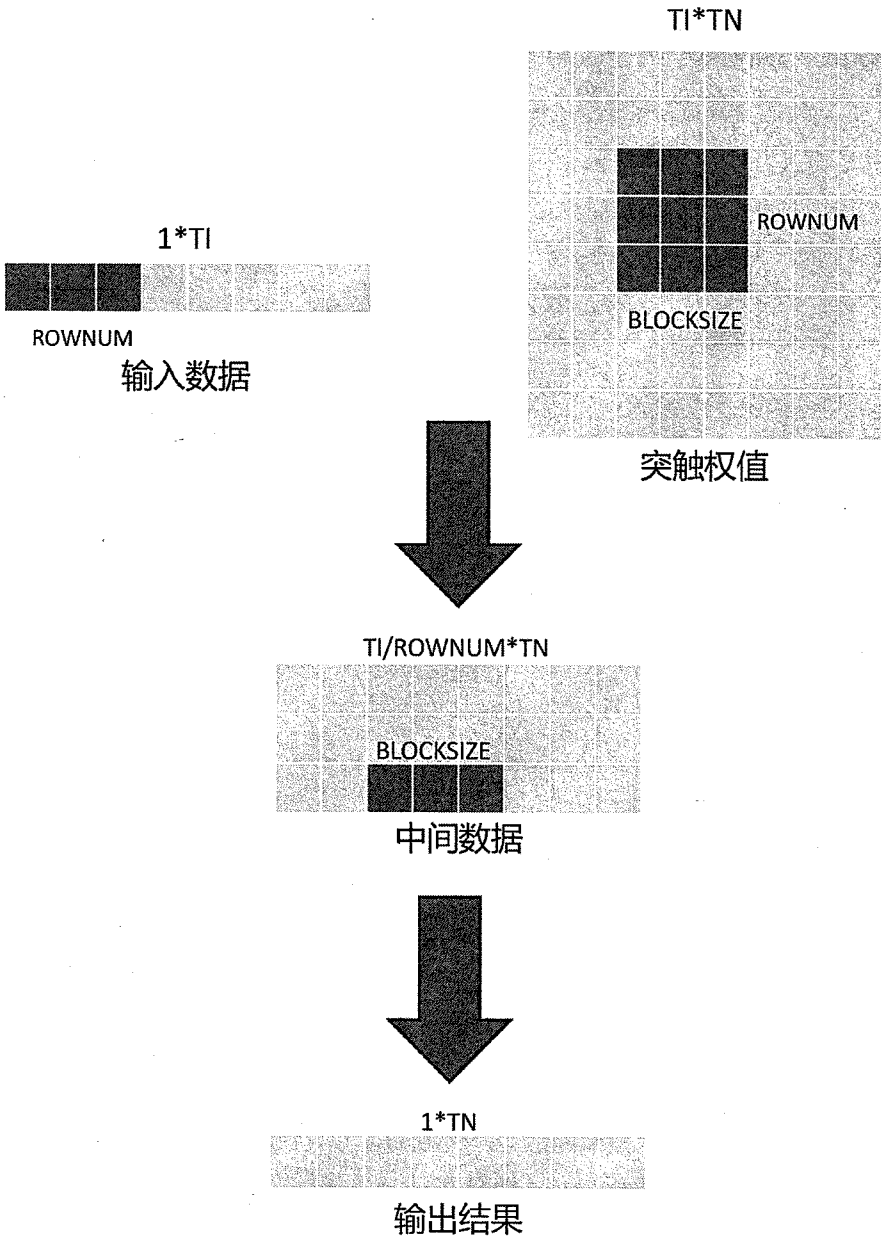

图 4.2 分类层的 GPU 实现流程图

4.3.2 卷积层 CUDA 实现

卷积层的原始程序正如图所示，可以看到其涉及到的循环层数非常的繁复，涉及到的输入数据和突触权值的数据的数据结构也非常的复杂，但是经过之前的网络描述的层次化，以共享核版本的卷积层为例可以描述为：对于 TI 个输入特征阵列，每一个阵列都

是 TY*TX 的二维数组，而为每一个输入特征阵列都有 TN 个突触权值数组，也就是对于 TI 个输入二维数组当中的每一个位置都有一个 KY*KX 的二维矩阵与之对应，这个阵列都会与以输入特征阵列某节点为中心的 KY*KX 的窗口进行卷积，得到的卷积和作为输出数据的一个点值，或者可以看作在 TI 输入特征阵列上同时有一个 KY*KX 的窗口在滑动，每经过一个位置都会与 TI 个同样大小的二维数组进行卷积和操作，而由于是私有核版本的卷积算法，这样的中间特征阵列有 TN 组。最后可以得到 TN 个 OY*OX（OY 与 OX 的大小与 TY 与 TX 的大小基本相当，差量在窗口大小上，为便于叙述，故而分为 OY*OX 与 TY*TX）个二维矩阵。

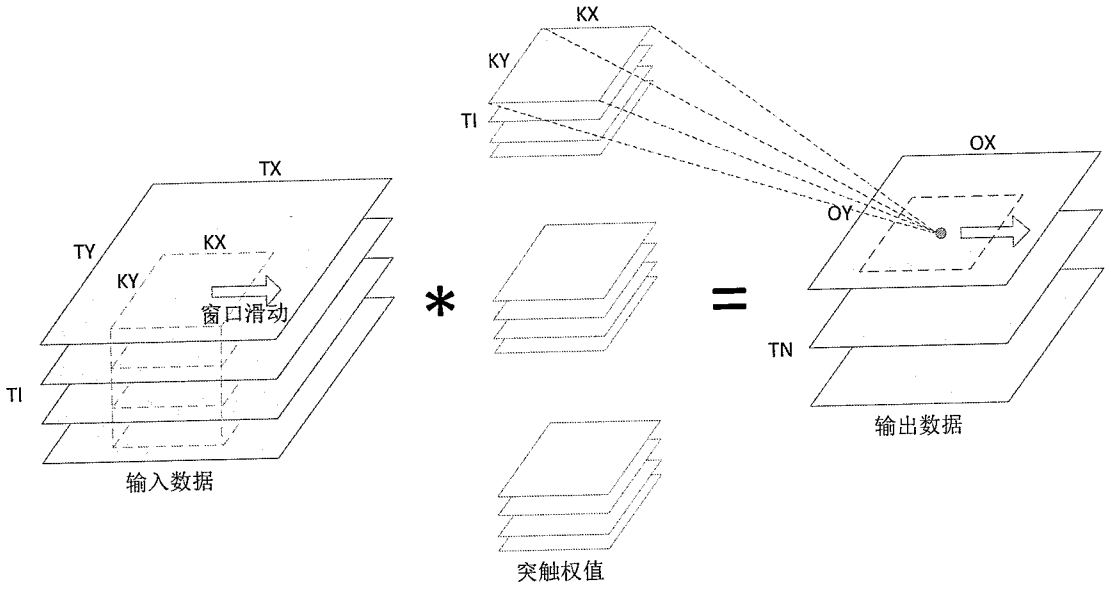

图 4.3 卷积层基本算法框架

更几何化的语言描述为：输入为一个 TI*TY*TX 的三维数据，权值为 TN*TI*KY*KX 的四维突触权值数据，最后得到的每一个输出点就相当于一个三维的 TI*KY*KX 的窗口在输入的三维数据中不断滑动求值的过程，每一次的求值是两个三维数组进行卷积求和操作得到的结果。

在进行 GPU 实现的时候，我们考虑了原始的求值顺序并且对它进行了修改就得到了我们现在的计算顺序，非常适合 GPU 的处理方式。如图 4.4图 4.5所示

其具体的执行方式也是分两步来做，第一步是进行卷积操作，第一步的 kernel 函数是按照<<<(TN, TI), (TY, TX)>>>的维度进行划分。这样对 blocks 和 threads 进行划分的原则是：

1. 进行 TN 划分：求输入的三维数组与 1 个三维突触权值的卷积和
2. 进行 TI 划分：求三维输入数组的每一层，也就是 TI 个二维数组与突触权值三维数组的每一层，也是一个二维数组的卷积和
3. 进行（TY，TX）划分：精确计算每一个输入节点的卷积值

上述划分过程没有对窗口进行更细粒度地划分主要是因为一般 KY 与 KX 的数值比

较小，没有进一步划分的必要。这里存在着两个复用的情况，一是每一个输入节点在窗口滑动经过该点的时候都会被复用到，在最内层循环总共被复用到 KY*KX 次，二是每一个突触权值也会被不断的复用，复用次数可以达到 TX*TY 次。这两部分数据都可以直接存放到共享存储当中，提升程序的整体效率。

在每一个线程当中所要执行的 kernel 函数就是要完成当前输入节点为开始点的一个 KY*KX 大小的输入数据矩阵与对应突触权值的加权和操作。得到的数据可以存放在大小为 TN*TI*OY*OX 的 GPU 片上存储当中，方便下一个阶段的使用。

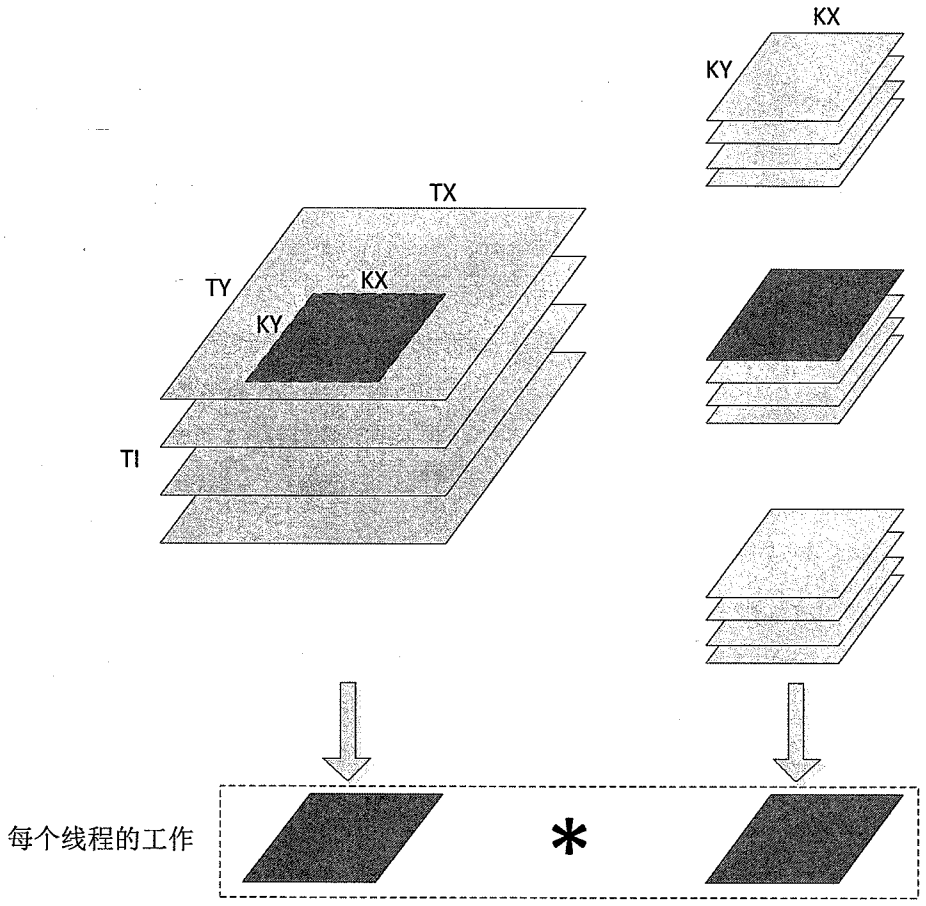

图 4.4 卷积层 GPU 实现流程图（1）

在计算的第二阶段就是要对这些加权和进行相加，这一部分的实现也非常的简单，如图 4.5所示。对 kernel 函数的划分为<<<(OY, TN), OX>>>，这三层划分很明确的就是将每一个输出节点的求值放到一个线程当中。而每一个线程的 kernel 函数就是对 TI 个 GPU 片上存储的临时数据进行加和操作。

在卷积层的实现过程当中最要的步骤是将原来的多层循环进行分解重组，得到适合 GPU 实现的循环逻辑，然后通过对线程块的划分以及线程块中线程的划分，使得每一个线程都能充分享受到并行执行带来的性能优势与共享内存带来的访存优势。

需要说明的是，这里的卷积层的划分是以共享核卷积为例来进行说明的。对于需要

处理的程序是私有核的情况，也可以按照这里的划分来进行处理，但是突触权值的数据就不能享受到共享存储带来的访存便利。并且私有核的突触权值的数据量要远比共享核的数据量来得多，这在 GPU 最后实现结果当中也可以得到充分的说明。

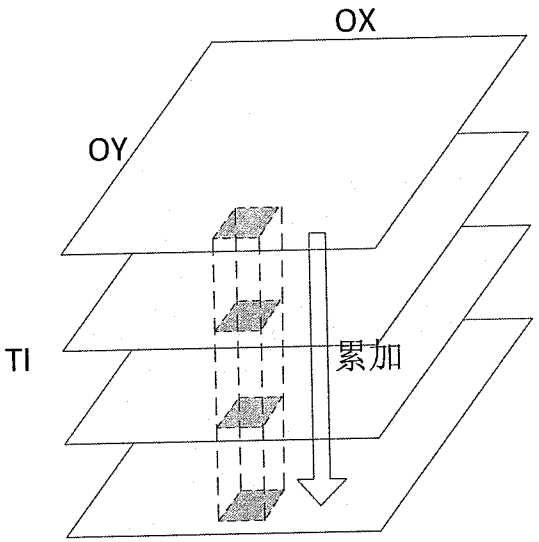

图 4.5 卷积层 GPU 实现流程图（2）

4.3.3 归并层 CUDA 实现

而对于归并层来说，归并层需要进行数据窗口的滑动，然后在每个窗口当中通过预先设定的归并规则来获得一个值，这样就能够使得整个网络层大大化简。归并层的问题描述是，对于一个输入数据是 TI*TY*TX 的三维数据，需要一个 KY*KX 的窗口进行不交叉平移滑动，然后在每个窗口当中得到一个数据，最常见的方法是选择最大的权重，或者计算当前窗口所有数据的平均值，最后得到一个 TI*OY*OX 大小的三维数据（OY 为 TY/KY，OX 为 TX/KX）。

从上面的计算分析可以看到归并层的数据基本没有数据复用，每一个数据都需要进行一次操作，因此对归并层的 GPU 实现当中的数据存储就没有办法利用到共享数据来实现性能的提升。

归并层具体的实现如图 4.6所示，对 kernel 函数的划分是<<<TI/BLOCKSIZE，BLOCKSIZE>>>，其中 BLOCKSIZE 是预先设定的 BLOCKSIZE 以增加对问题的划分粒度，在单一 warp 当中保证更多的运行时线程，在 warp 发射时候提升 warp 的使用率。在每一个 thread 当中所要做的仍然是对窗口当中 KY*KX 个数据进行归并操作。这里并没有对 KY*KX 次的比较操作进行进一步的划分主要原因是 benchmark 当中的 KY 和 KX 都比较小，当然也是因为在实际应用当中归并层窗口的选取粒度都不是很大，这就没有再对归并窗口进行进一步划分的必要。最后在每一个线程都完成对各自窗口的归并操作之后，整个程序就能够得到 TI*OY*OX 大小的三维数组。

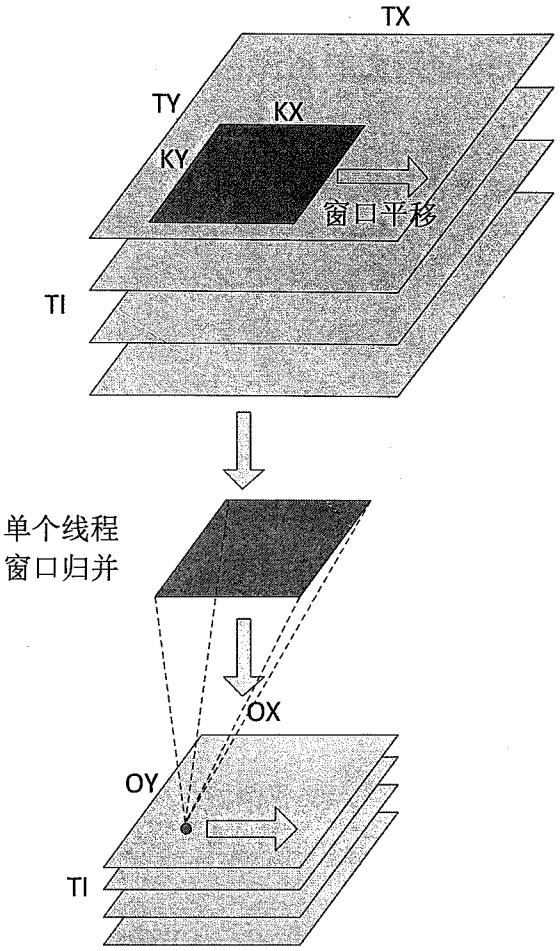

图 4.6 归并层 GPU 实现流程图

第五章 专用神经网络加速器实现

随着异构多核结构的火热发展，各种类型的加速器不断涌现。在神经网络方面，除了在绪论当中提到的[32-35]等之外，也出现了依托 FPGA+DSP 平台的[54]非线性神经网络控制器是，Poormann[55]虽然号称是神经网络的硬件实现，但其实也是在 FPGA 实现基础上实现的，其它诸如 Boser[56]只面向单一种类的分类层神经网络，Eldredge[57]则仍限定于特定的数字门电路模块。

而本章将要介绍的我们自己提出的 DianNao[6]则是最新出现的突破传统 FPGA、DSP 等架构限制的专用神经网络加速器。本章首先介绍 DianNao 的体系架构，然后介绍其硬件设计与实现，最后以分类层网络为例，介绍在 DianNao 上实现神经网络层次的控制代码以及流程。

5.1 专用神经网络加速器架构

DianNao 是一种针对神经网络的专用硬件加速器，能够实现对大规模卷积神经网络和深度神经网络正向传播的快速执行，执行时功耗相对传统架构较低，占用的片上面积也相对较低（0.98GHz 频率，65nm 工艺，3mm^2 片上面积）。其设计架构如图 5.1所示

该加速器由以下部件组成：输入缓冲区（NBin），输出缓冲区（NBout），突触权值缓冲区（SB），与 SB 相连的功能部件（NFU），控制模块（CP）。下面就按模块和功能来分别来介绍加速器的主要构成：

神经网络功能部件的作用将神经网络划分为一个个独立的计算模块，每个模块都能接收 TI 个输入或者权重并且有 TN 个输出。NFU 是为了完全浮现算法当中的核心计算流程而设计的，诸如分类层和卷积层源代码当中的 i 和 n 的循环，或者是归并层当中的 i 循环。

算术操作符是计算功能部件的核心，虽然深度学习算法没有那么多指令的需求，但也需要精心的分析与选择：不同神经网络层的类型虽有不同，但是计算步骤都可以被划分为 2 到 3 个阶段：对于分类层来说，计算阶段可以划分为突触权值与输入数据的乘法、所有乘积的加法、sigmoid 操作；对于卷积层来说，过程是基本一致的，只有最后一步会有所不同（有可能放弃 sigmoid 而选用其它类型的非线性函数）；归并层的计算指令流当中没有乘法操作，但增加了最大归并或者平均归并的指令需求。同时注意到上述加法操作基本都是多操作数的加法，因此加法器在设计的时候需要做成加法树的形式。另外，为了满足归并层的需求也需要增加专用的移位器和极值器。

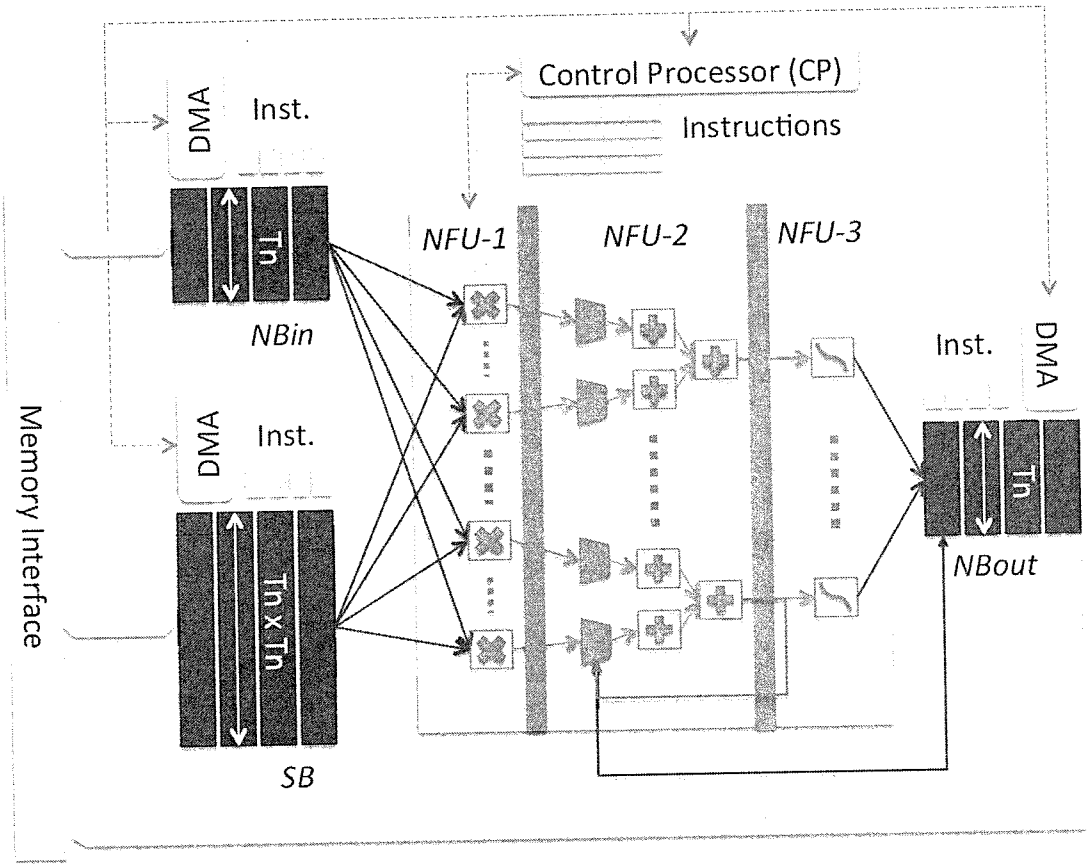

图 5.1 专用神经网络加速器架构图

不完全流水线是计算功能部件的精髓所在。无论是归并层的两阶段计算，还是分类层和卷积层的三阶段计算过程可以被流水线化，但各层之间存在的数据依赖关系又要求这样的流水线不能是完全流水线：分类层与卷积层的前两个阶段，以及归并层的第一阶段和正常的流水线的行为应该保持一致，但是第三阶段的 sigmoid 流水线应该在所有的加法操作或者极值操作执行完之后才能执行（对于卷积层来说可能没有第三阶段的 sigmoid 操作），在下面的叙述过程当中我们用 NFU-n 来指代神经网络功能部件流水线的第 n 个阶段。

NFU-3 功能实现是功能计算部件的必要补充，当然这在之前文献当中也有所涉及[29, 58]，对于卷积层和分类层的第三阶段的 sigmoid 操作可以使用分段线插值函数来逼近，

$$f(x)=a_i\times x+b_i, x\in[x_i,x_{i+1}]$$

这样的逼近带来的精度的损失是非常小的，在之前已经用图形形象地论证过了。其硬件实现如图 5.2 所示，由两个 16 路的多路选择器（根据输入选择分段），一个 16 位的乘法器和一个 16 位的加法器来实现插值操作。16 个分段的系数（a_i，b_i）存放在片上一个极小的 RAM 上，这样的设计保即便是选择其它非线性甚至线性函数的基础上，只要能够将系数提前放入 RAM，由于该阶段的待计算数据的输入和输出的连线都已经被硬件化写好到电路上了，整个加速器仍然可以正常运转。

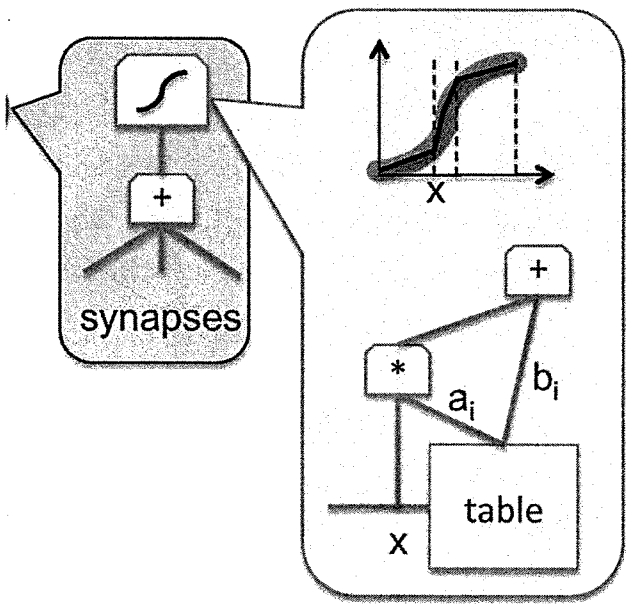

图 5.2 神经网络加速器实现 sigmoid 的硬件电路

需要一提的是，这里使用了 16 位的定点算术单元而非 32 位的浮点算术单元。正如我们在之前所说的，有很充分的证据表明即便是位数更少的算术操作（例如 8 位）都不会给神经网络的计算精度带来非常明显的误差。

加速器的存储部分包含 NBin，NBout，SB，NFU-2 阶段寄存器，这些存储就和传统 CPU 上的缓存作用类似，对于传统 CPU 来说缓存是非常优秀的存储结构和介质，但是考虑到 cache 的复用性并不是那么的出色（每次都要进行 tag 验证、关联性限制以及块大小限制等）并且还有缓存访存冲突这一问题的存在，在可定制的硬件电路上使用缓存就不是最优的选择了。在 VLIW 处理器当中最常见的解决方案就是使用的超高速缓存，但是这在实现上会有比较困难，但是在专用加速器当中使用超高速缓存的好处也是显而易见的：有效的存储访存方式，对于数据的摆放以及存储方式没有特殊要求非常便利（只有极少数算法需要对数据摆放进行手工调整）。在这种情况下，我们上面对各个神经网络层次的数据存储方式的分析及修改都可以直接翻译为对缓冲区的数据映射指令。

上面已经介绍过了，我们将存储部分分为三块，输入缓存区 NBin，输出缓存区 NBout，突触缓存区 SB，这样的存储方式被称为分块存储（split buffer）。

对存储进行分块带来的第一个优点就是可以对 SRAM 的读写位宽进行定制，NBin 和 NBout 的位宽都是 Tn*2 字节，而 SB 的位宽则为 Tn*Tn*2 个字节。而如果采用统一的缓存设计，面对每次读取或者写入的位宽的不同，缓存位宽大小的选择将会非常棘手。如果缓存行大小调整到 Tn*Tn*2，那么在读取 Tn*2 大小的输入数据和输出数据时候就会造成大量的能量浪费。图 5.3展示了在 TSMC65nm 工艺下的每次读取的功耗与缓存位宽选择的函数关系。而如果缓存行大小调整为神经元的个数，也就是 Tn*2，那么在读取突触数据 Tn*Tn*2 情况下，就不能在 1 个 cycle 当中完成数据读取，消耗的时间就会大

大增加。综上所述，将数据存储区进行分块设计能够使得读写数据的性能和功耗同时达到最佳。

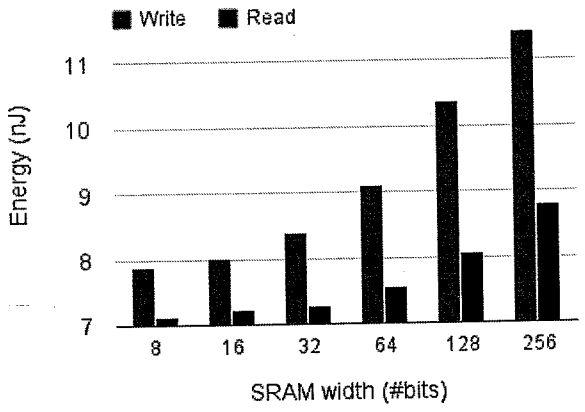

图 5.3 SRAM 读写功耗与 port 宽度关系

另外，将存储区各自独立的第二个好处就是避免了缓存读写可能出现的冲突。由于我们为了更高的效率和更低的功耗会将存储模块设计的非常小，那么访存数据时产生冲突就显得不可避免了。如果不对数据存储区进行独立和分离，那么高度关联性缓存是一个可用的替代方案，并且为了满足读取突触数据的需求，缓存行位宽（或者访存端口数量）就需要比较的大（Tn*Tn*2）。在一个 n 路缓存当中，每次读取都要并行地访问到所有的 n 路缓存行，这种情况下的高关联缓存的功耗就相当的可观。在 65nm 工艺下，8 路的 32KB 大小的缓存当中读取 64 字节的数据耗费的功耗就是从直联缓存当中读取 64 字节数据功耗的 4.15 倍[59]。在 Intel 的 core-i7 当中，一级缓存的大小是 32KB，8 路缓存，缓存行位宽是 64 字节，而我们要解决的问题规模要大于通用 CPU 的一级缓存的大小，因此如果采用单一的数据存储区，我们的缓存应当是缓存行位宽更大、关联度更高（对于 Tn=16 的情况下，我们需要的缓存行大小就要有 512 字节）。因此，高关联缓存是一个高功耗低效率的选择，而缓存独立则是较好的取代方案。

5.2 神经网络的硬件实现

DMA：从上面存储分离的设计来看，需要实现 3 个 DMA 控制器，两个 load DMA（用于读输入数据和突触权值），一个 store DMA（用于写输出数据）。以 NBin 为例，DMA 控制器通过指令的形式来控制 NBin，控制指令被存放在各自的队列当中，指令按照顺序进行发射执行。DMA 指令与相对应的 NFU 的算术指令是解耦的，即只要当前存储区有足够多的空间，DMA 指令可以提前被放到这些指令队列当中发射执行，实现数据预取，减少了载入载出输出的时间上延迟的影响。除了没有猜测相关的功能，这和传

统 CPU 流水线当中的预取（prefetching）功能非常的相似。

在卷积层和归并层的数据摆放在实际计算当中当中存在着一些不便，而 NBin 对归并层的数据输入则实现调整。在上面提到的代码当中，一般 Kx 和 Ky 的取值都比较小，基本在 10 以下，但是 Ni 的取值会很大一个数量级，因此对存储的读取方式肯定是一次读取一长列的方式更有效率。但这对于归并层的计算并不方便，传统的方式是对每一个输入特征阵列进行计算，每次计算一个输出点只需要读入 Kx*Ky 个数据（在卷积层当中，每个输出点需要读入 Kx*Ky*Ni 个数据，并没有类似的担忧）个数据，这在硬件实现上对性能有着极大的浪费。在加速器的设计结构当中，引入了转置模块可以对 Ky、Kx 以及 Ni 循环层进行本地转置，使得数据可以按照最内层循环为 Ni 来进行读取，计算的时候是按照 Kx、Ky 的循环被送入神经网络功能部件进行计算。其实现方式就是在数据读入 NBin 的时候就对数据进行交错存储，如图 5.4所示。

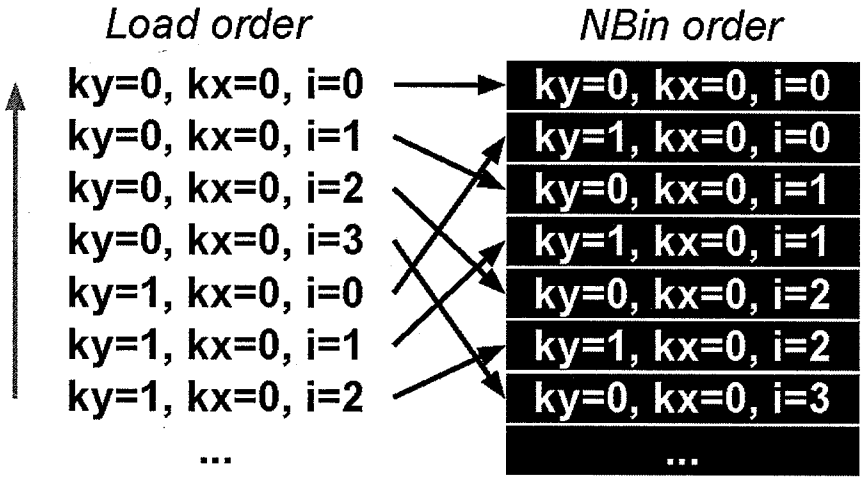

图 5.4　NBin 读入数据转置示意图

这样的实现方式可以保证每次 DMA 进行读取的时候都会读取较大的数据块，保证能量的高效使用。当然在计算时候需要对计算顺序进行调整，并且需要 NBout 当中的数据作为临时变量进行存储。

对于 SB 中的突触数据，实际上是没有复用的，当然只有卷积层的共享核是有复用可以发掘的。对于输出数据我们需要复用部分和，在下面会介绍如何利用这一部分的复用性。

在分类层和卷积层当中，做加权之后的临时乘积需要有临时存储区域进行存放，考虑到临时乘积需要在计算的时候被调入到流水线当中，然后在计算之后在存入到其它位置，整个过程实际上是一个循环的过程。如果考虑让临时乘积存放到 NBin 当中，那么它就需要在 NBin 和流水线之间搬进搬出，这对性能和功耗是巨大的浪费。因此我们在 NFU-2 阶段引入了专用的寄存器，专门用来存放这些临时乘积。

另外如果现在已经计算出了 Tn 个部分和数据，但是由于新的输入数据以及新的权

值需要做乘法，计算出来的 Tn 个新的临时乘积（可以看作做 0 次加法的部分和）也需要存放到专用寄存器当中，那之前那 Tn 个相关的部分和数据应该存放在哪里？我们的解决方法是存放在 NBout 当中。虽然 NBout 传统意义上认为是用来存储需要被写出到外存的部件，但是我们规定在 NBin 和 SB 当中的数据没有被完全使用之前，NBout 都是空闲可以被占用的，因此我们可以将 Tn 个部分和数据暂时存放到 NBout 当中。这样在设计上就要求 NBout 不仅要和 NFU-3 以及存储部件联系在一起，还要和 NFU-2 连在一起：NBout 的入口需要提供给 NFU-2，保证 NFU-2 当中的寄存器数据可以存放到 NBout 当中。

5.3 控制及实现

5.3.1 控制芯片

神经网络加速器的控制也是独立进行的，即在每次的控制代码当中都会对主要的 5 个模块进行分别控制，分别是控制芯片、SB、NBin、NBout 和 NFU。这样设计的目的是更深层的控制与调度，根据神经网络实现算法的修改（如分块）对各个层面的操作进行修改，为机器学习算法的硬件加速提供了更大的灵活性。虽然看起来对硬件的控制需要更详细的硬件知识，对使用者来说是不够友好的，但是加速器将算法的核心计算部分包装了起来，对使用者是透明的，在一个控制指令当中，加速器就能够完成分类层与卷积层的 ii，i，n 循环或者是归并层的 ii 与 i 循环。

在片上执行阶段，这些指令流被存放在于控制芯片（Control Processor）相连的 SRAM 当中，控制芯片根据控制代码的开关直接驱动其它三个缓存区以及 NFU 功能部件的执行。（这里虽然将控制芯片称之为芯片，但是其功能并不像传统意义上的 PC 的 CPU 的功能那样复杂，可以近似看作一个可配置的有限状态自动机）

CP	SB				NBin							NBout				NFU						
END	READ OP	REUSE	ADDRESS	SIZE	READ OP	REUSE	STRIDE	STRIDE BEGIN	STRIDE END	ADDRESS	SIZE	READ OP	WRITE OP	ADDRESS	SIZE	NFU-1 OP	NFU-2 OP	NFU-2 IN	NFU-2 OUT	NFU-3 OP	OUTPUT BEGIN	OUTPUT END

图 5.5 CP 控制代码各位置含义

5.3.2 网络层实现代码

每一条指令都有五个延迟槽，分别对应控制芯片本身，三个缓冲区以及 NFU 功能部件。如图 5.5所示。

由于控制芯片的指令都是非常底层的，需要代码生成机制，但是总共只需要三种类

型的代码（分别对应三种神经网络层），如果为该结构设计一个编译器显得大材小用了，直接为每一种类型的神经网络层撰写控制代码反而是比较高效的选择（如果以后该专用神经网络加速器能支持更广泛的应用就有必要为其设计专用的编译器了）。

下面是分类层的生成代码，由于 Tn=16（每个缓存区行都有 16 个 16 位的数据），NBin 有 64 个缓存区行，容量是 2KB，不能包含所有的输入数据（输入数据大小是 16KB）。因此数据需要被划分为 8 个 2KB 大小的数据块来处理，NBin 的第一条指令是 LOAD（从内存当中取数据），在 LOAD 指令之后被标识为复用（reused），接下来的指令是 READ 指令，正如前面提到的，输入缓冲区需要被用作循环队列来存放下一次要进入的数据（总共是 16KB/2KB=8 组），因此该 READ 指令的 reused 位也需要被标志为 1。在计算出乘积之后部分和数据需要被存储到 NBout 当中，也就是 NFU-2 的输出需要写入到 NBout 当中。在上面已经解释过了 NBout 在 NBin 和 SB 被清空之前可以被当作临时存储区域使用，因此 NBout 的第一个指令应当 WRITE，同时由于下一组的输入神经元要进入 NFU-2，因此 NFU-2 的 input 位应该标志为 reset，将其中的寄存器清空。当最后一个输入数据块被发射出去之后，NBout 的 DMA 应该将 512 字节的输出数据，也就是 256 个输出数据写到存储当中，因此 NBout 的指令应当是 STORE。利用这样的一步一步的详细分析可以分别得到另外两种神经网络层次的控制代码。而在实验时候，只需要按照已有的控制代码流模板进行修改即可。

CP	SB				NBin						NBout				NFU						
NOP	LOAD	0	0	32768	LOAD	1	0	0	0	4194304	2048	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	1	0
NOP	LOAD	0	32768	32768	READ	1	0	0	0	0	0	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	0	0
.....																						
NOP	LOAD	0	7864320	32768	LOAD	1	0	0	0	4225024	2048	READ	STORE	8388608	512	MULT	ADD	NBOUT	NFU3	SIGMOID	1	0
.....																						

图 5.6　神经网络加速器分类层部分控制代码

第六章 实验与结果分析

前面三章从理论上分别从 SIMD、GPU 和专用神经网络加速器的角度对神经网络的具体实现方式以及所做的修改做出了论述，在本章当中我们就这常见的三种神经网络层次在这 SIMD 比较基准和另外两种加速器上进行了实现并且对结果进行分析。

6.1 Benchmark

我们从最近一些大规模实际问题当中抽取出了深度神经网络以及卷积神经网络模型，包含了相当规模的分类层、卷积层以及归并层。这些神经网络层的详细信息及特点如表 6.1所示：

表 6.1 测试用 Benchmark

网络层	Tx	Ty	Kx	Ky	Ni	No	描述
CONV1	500	375	9	9	32	48	街景分析（CNN）[32]（例如楼房及车辆识别）
POOL1	492	367	2	2	12	-	
CLASS1	-	-	-	-	960	20	
CONV2*	200	200	18	18	8	8	YouTube 视频人脸识别 [13]（DNN）
CONV3	32	32	4	4	108	200	行车导航路标识别（CNN）[5]
POOL3	32	32	4	4	100	-	
CLASS3	-	-	-	-	200	100	
CONV4	32	32	7	7	16	512	谷歌街景（CNN）[47]
CONV5	256	256	11	11	256	384	自然图像多目标识别（DNN）[4]
POOL5	256	256	2	2	256	-	

6.2 实验平台

由于本次实验所设计的平台较多，需要分别论述。

SIMD 实验平台：我们采用了 GEM5+McPAT[60]的组合。我们选用的模拟器配置是四发射超标量 x86 核心，拥有 128 位的 SIMD 功能部件，使用 SSE/SSE2 指令集，频率是 2GHz。该核心有 192 路 ROB，64 路的 load/store 队列。其 L1 数据缓存是 32KB，L2 缓存是 2MB，这两个缓存都是 8 路缓存缓存行大小都是 64 字节，这些功能参数和 Intel Core i7 的参数是基本一致的。一级缓存的未命中延迟是 10 个 cycle，二级缓存的未命中延迟是 250 个 cycle，内存总线位宽是 256 位。对于程序的功耗分析我们采用的都是 McPAT 的功耗模型。

SIMD 的实现方式是核心部分用汇编直接书写来保证指令性能的充分发挥，而不是通过函数调用方式实现。在编译阶段我们使用了-O 的编译选项，来确保编译器不会对我们的核心代码进行调整修改。为了进一步探究 SIMD 核心对于性能的影响，我们还实现了不同 benchmark 程序的普通 C++版本。在一般情况下 SIMD 核心能够实现 3.92 倍的执行时间的提升以及 3.74 倍的能量使用效率提升。这也证明了我们 SIMD 代码加速的有效性。

GPU加速器的实验平台我们选取了NVIDIA的fermi架构的C2070GPU平台。C2070的参数如表 6.2所示：

表 6.2 C2070 参数列表

参数	C2070
频率	1.15GHz
GDDR5 带宽	144GB/s
SM 核心数量	448
每个核心 L1 缓存数量（包含共享缓存）	64KB
工艺	40nm
峰值性能	1.288TFLOPS
GDDR5 大小	6GB
Warp 大小	32
每个核心寄存器数量	32768
二级缓存大小	768KB

比较遗憾的是虽然现有的诸如 gpgpusim+GPUWattch[61]的实验平台号称能够给出 GPU 执行程序时候的功耗，但是其执行速度过慢，功耗模型不够先进（目前只有 GTX480 的功耗模型，该显卡是 NVIDIA 公司 2010 年之前推出的老旧类型）。NVIDIA 的 cuda sdk 自带的 profiler 工具也不支持本次实验采用的 c2070 型号的显卡，因此只好放弃对 GPU

程序功耗的测量，而采用 NVIDIA 官方提供的平均功耗进行定性分析。

神经网络的专用神经网络加速器用[6]在其综合之后进行验证实现的，其综合使用的 Tn 为 16，即有 16 个神经元，每个神经元都有 16 个突触连接，这要求对于 NFU-1 来说要有 256 个 16 位截断乘法器（在卷积层和分类层使用的），在 NFU-2 当中要有能做 15 个数加法的加法器，这样的加法器需要 16 个（如果归并层使用了平均归并，那么三种神经网络层都需要），以及 16 个移位器和最大值选择器（归并层需求），16 个 16 位截断乘法器和 16 位加法器（分类层和卷积层使用）。对分类层和卷积层来说，NFU-1 和 NFU-2 每个 cylce 都工作，因此每个 cycle 能够完成 496 个定点操作，在 0.98GHz 频率下的计算能力是 452GOP/s，在有数据达到 NFU-3 时候，NFU-1 和 NFU-2 当中依然有数据在计算，这个时候能够达到峰值性能，每个 cycle 能执行 528 个定点操作，即 482GOP/s。

6.3 结果分析

6.3.1 神经网络加速器与 SIMD 基准的结果对照

神经网络加速器与 SIMD 指令集的对照组的实验结果的加速比如图 6.1所示，由于我们使用了 128 位的 SIMD 指令，也就是每个 cycle 可以执行 8 个 16 位的定点操作，而上面提到的在神经网络加速器当中可以在卷积层和分类层 1 个 cycle 当中执行 496 个 16 位操作，也就是 SIMD 核心的 62 倍。观察我们的实验结果，在卷积层和分类层两层的 8 个结果当中，平均加速比是 117.87 倍，大概是上面提到的 62 倍理想加速比的 2 倍。并且根据 SIMD 核心的执行时间我们发现平均来算，SIMD 核心在 1 个 cycle 当中能够执行约 2.01 条 16 位指令。其原因可能存在于以下：

在 SIMD 核心没有将预取功能加入，而在神经网络加速器当中在 NBin 和 NBout 的缓冲区当中则很好地实现了预取功能并且对数据复用进行了利用，这样能够大大减少计算的延迟。这也就解释了 CLASS1，CLASS3 和 CONV5 的数据为何高于平均值（这 3 个网络层的平均值为 629.92 倍），因为这几个网络层都有着非常大的数据阵列，数据的复用性也能利用地最好，它们也就能从预取和复用当中获得最大的收益。如果在 SIMD 核心当中加入数据预取器就能够很好得弥补上面的性能耗费，提升 SIMD 的表现，从而使得 SIMD 表现和神经网络加速器的表现的比值趋于正常。NBin 的存储复用是由 DMA 控制器处理的特征阵列决定的，并且较小的 Ni 值使得 DMA 需要多次搬运小规模数据，这使得整体的性能有所下降。除了 CONV5 剩下的卷积层的平均加速比是 195.15 倍，CONV2 的加速比较小是因为 CONV2 使用了私有核，存储的数据复用性较低。而归并层基准测试程序的加速比很小是因为在 NFU-2 当中，仅有加法器被使用了，乘法器没有被使用，导致整体的加速比的理想值就非常低（240/8=30 倍），POOL3 的加速比是 25.73 倍，POOL5 的加速比是 25.52 倍和这个数据非常接近。关于 POOL1 当中的加速比比较

小的原因在[6]当中也有提到，这是因为 POOL1 当中的输入的特征阵列过小（Ni=12），在执行时候应用到特征阵列上的操作单元数量要少于 POOL3 和 POOL5，在最核心的循环当中，硬件资源的利用率没有发挥出来。在神经网络加速器的未来设计当中，如果能采用二维或者三维的 DMA 控制器，能够对于各种规模的神经网络进行自适应处理，就可以很好的解决这些问题。

表 6.3 神经网络加速器相比于 SIMD 实现的性能加速比结果

网络层	加速比	网络层	加速比
CLASS1	697.96	CONV4	224.97
CLASS3	685.52	CONV5	522.18
CONV1	196.45	POOL1	2.17
CONV2	130.64	POOL3	25.73
CONV3	251.21	POOL5	25.52

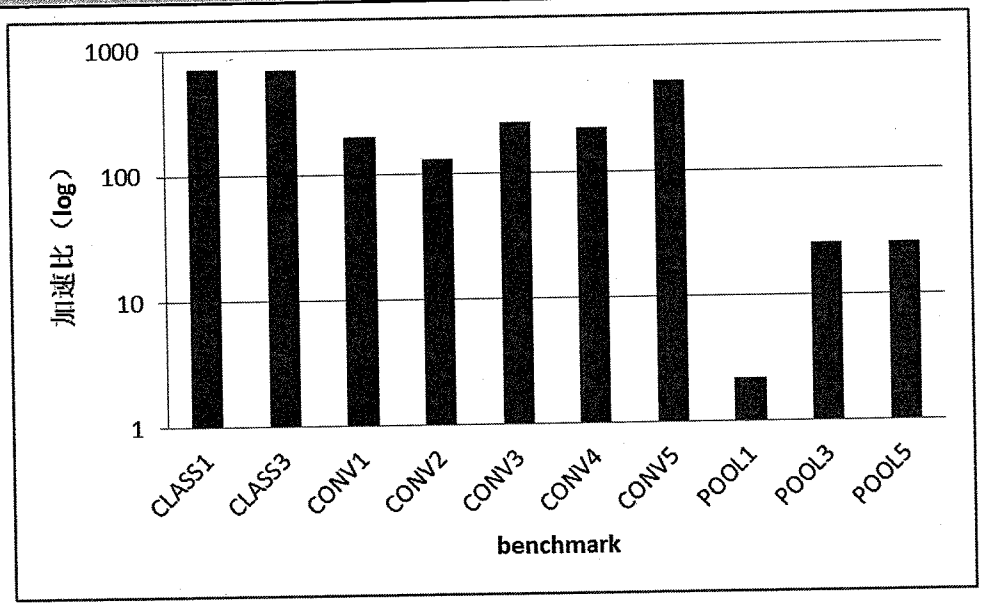

图 6.1 神经网络加速器相比于 SIMD 实现的性能加速比示意图

在功耗方面，神经网络加速器的功耗相比于 SIMD 的功耗平均上是有所下降的，其具体数值如表 6.4所示。

表 6.4 神经网络加速器相比于 SIMD 实现的功耗加速比结果

网络层	加速比	网络层	加速比
CLASS1	18.01	CONV4	20.73
CLASS3	18.24	CONV5	18.11
CONV1	28.62	POOL1	21.02
CONV2	17.67	POOL3	24.67
CONV3	25.45	POOL5	21.02

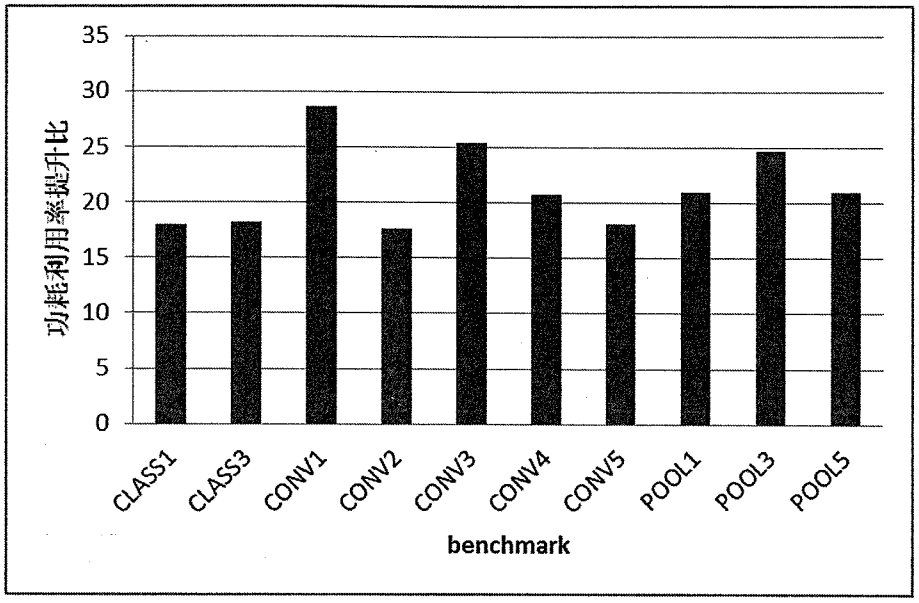

图 6.2 神经网络加速器相比于 SIMD 实现的功耗加速比柱状图

平均功耗利用率提升了 21.08 倍，这个数据看起来非常得不错，但是其它已经实现的加速器如 Hameed[16]当中的平均功耗提升是 500 倍，另有一个多层神经元的硬件实现[29]甚至能够达到 974 倍的功耗利用率提升相比，相比而言差距非常明显。最主要的原因在于其它神经网络的硬件实现都没有考虑到内存访问对于功耗的巨大影响，上面两篇文章当中的统计过程当中也没有涉及到存储访问功耗的统计比较。在很多类似的硬件设计当中，作者往往对计算的复用和重定义进行考量（类似我们使用到的神经网络加速器当中相当大量的 16 位乘法器），对功耗的比较分析也仅限于这一阶段，没有关注到内存访问的功耗降低，以及计算相关的存储部件的设计（如该神经网络加速器当中的 NBin，NBout，SB 和 NFU-2 当中的寄存器），这才是功耗产生的主要方面。在功耗模型上面也要有孰轻孰重的判断，以后对于加速器功耗的提升主要就应着眼于数据迁移的功耗的研究与优化。

6.3.2 神经网络加速器与 GPU 加速器的结果对照

表 6.5 神经网络加速器相比于 GPU 实现的性能加速比结果

网络层	加速比	网络层	加速比
CLASS1	0.905	CONV4	3.345
CLASS3	0.884	CONV5	0.648
CONV1	1.254	POOL1	0.008
CONV2	0.041	POOL3	0.022
CONV3	5.150	POOL5	0.032

而加速器对于对于 GPU 的 benchmark 性能上的加速比如表 6.5所示，从表中的数据可以看到，神经网络加速器在这些程序上的表现和 GPU 的表现是不相上下的，平均加速比为 1.22 倍，但是神经网络加速器的实际片上面积只有 GPU 片上面积的 0.56%

（GTX480 面积是 529mm^2，40nm 工艺），这样看来神经网络加速器的效率上要远胜 GPU。

虽然 GPU 的功耗不方便测量，但是根据 NVIDIA 官方的数据，即便是最新的 K20X 系列 GPU 包含 1.5MB 的片上 RAM，其平均功耗达到 235W，而加速器在这 10 个基准测试程序上的平均功率约为 64W，约为 GPU 平均功率的 27%左右。因此虽然神经网络加速器在执行时间上并无太大优势，但综合考虑片上面积以及功率上，已经完胜 GPU 的 SIMT 加速框架。

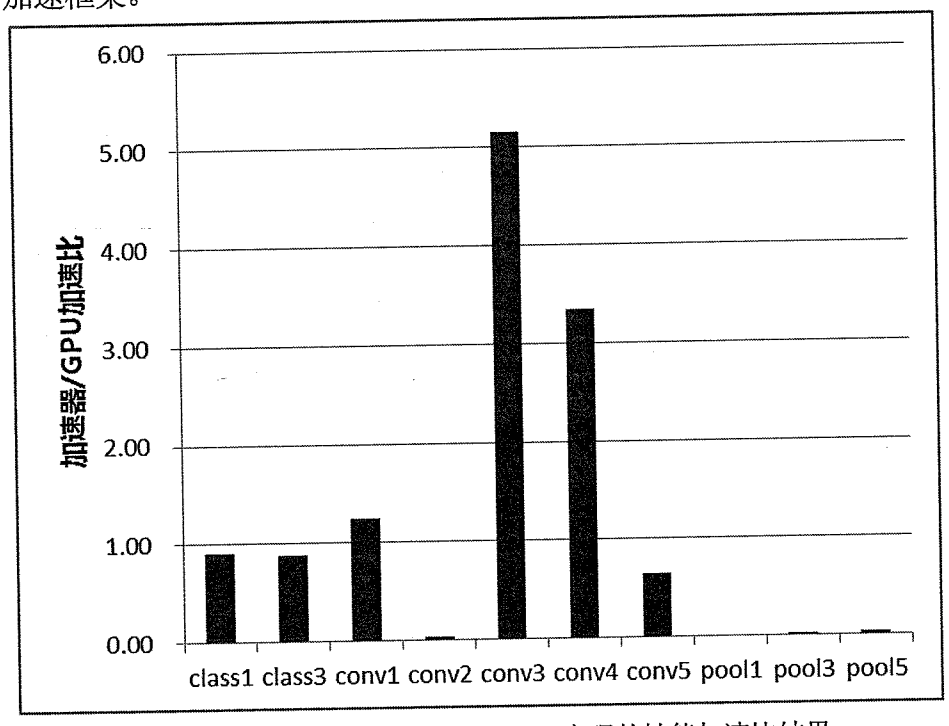

图 6.3 神经网络加速器相比于 GPU 实现的性能加速比结果

还有一点在实验当中体现的非常明显。在 GPU 平台上，数据在 CPU 的 host memory 与 GPU 的 device memory 之间的数据迁移是非常耗时的，原因一来是由于本身神经网络规模上去之后突触数量的大规模上涨，而 GPU 要求数据需要一次性的从 host memory 搬移到 device memory，在神经网络加速器上则是在计算的同时进行数据的迁移，用计算的时间掩盖了数据搬移的时间，二来是现有体系架构下连接 CPU 和 GPU 之间的 PCIe 带宽没有发展到很高的地步（在我们的实验平台上 PCIe 带宽为 8GB/s）。以上两点综合造成了数据搬移的耗时，这一部分的时间在上面的统计当中没有体现，但在实际执行的时候有非常大的影响，这也是 GPU 架构在神经网络算法实现上表现出来的一个劣势。另外，GPU 程序的撰写需要相当多的经验和技巧，不同的划分方式就有可能导致性能的巨大差异，学习成本较高；而神经网络加速器的指令集有限，并且只要按照已有的指令流模板就能快速学习上手，不需要过多复杂的技巧与经验。

第七章 结论及未来工作

一方面，随着传统通用 CPU 与 SIMD 在效能上的限制，越来越多的人投入到了各种加速器的研究工作当中。另一方面，随着深度学习的兴起，神经网络也受到了越来越多的关注。因此本文针对神经网络在各种加速器平台与基准平台上的实现展开研究。本文从对最为重要的卷积神经网络的分析入手，介绍了当前主流的各种平台的实现方式，最后对各种平台上的实现进行了分析和对照。

7.1 本文工作及结论

本文所做的工作如下：

1. 对以 SIMD 体系结构为代表的基准比较平台，以及各种加速器平台，诸如 GPU 加速器体系结构，以及我们自己提出的神经网络加速器的体系架构为代表，进行了分析。
2. 论述了常见神经网络算法如卷积神经网络的概念以及实现方式，并且对其算法进行了抽离和分层，从数据复用性的角度出发对卷积神经网络各个网络层的具体实现方式进行了重新改良，通过对神经网络层次的分块化强调了神经网络数据计算的复用性并且完成了对大规模问题的切分。
3. 在 SIMD 基准测试平台和 GPU 加速器平台和神经网络加速器这三种平台上实现了这三种神经网络层次，并对基准测试程序运行之后的数据结果进行了深入的分析。

从本次论文实验结果当中可以清晰的看到：

1. 相比于基准的 SIMD 实验平台，专用神经网络加速器的性能与功耗的优势非常的明显，专用的硬件加速器是未来解决计算效能问题的关键技术。
2. 相比于 GPU 加速器平台，虽然神经网络加速器在性能上并无太大优势，但是综合考虑芯片面积以及功耗，专用加速器的优势仍非常明显，并且 GPU 的实现需要大量经验与技巧，较费时的片上片下数据传输，使得 GPU 的劣势更为明显。
3. 从实现结果都可以看出，对内存访问的带宽限制今后是对神经网络算法实现的最大阻碍：SIMD 基准平台缺少必要的预取功能；GPU 上有着较高的峰值性能，但是其 PCIe 带宽以及编程成本成了性能执行的瓶颈；虽然在神经网络加速器计算部件的堆叠与计算阶段流水线的优化非常到位，但其主要性能和功耗都耗费在了访存行为上，都说明了下一步的优化重点应该放到对访存的优化上面。

7.2 未来研究方向

无论是传统的 SIMD 指令集还是 GPU 的 CUDA 编程框架，都没有专注于神经网络

算法的实现，专用的神经网络加速器则仍然有巨大的发展空间

1. 研究反向传播算法在并行架构下的应用，无论是在 SIMD 核心还是 GPU 加速器，亦或者是专用加速神经网络，都需要考虑其实现。虽然现阶段神经网络的线下训练并不会对神经网络的应用造成障碍，但是如果能够实现在并行框架下的正反向神经网络传播，那么搭建一套完整的神经网络训练应用流程仍然会有足够广阔的应用空间。
2. 对通用 GPU 或者神经网络加速器编程框架进行构建，编写一套在常用并行平台上执行的神经网络算法架构对于未来各种类型神经网络的实现有着非常重要的现实意义。
3. 通过对神经网络加速器核心的扩展，实现多核的神经网络加速器阵列，或者在已有的分布式系统，甚至是多核 GPU 上进行扩展，能够并行执行更大规模的神经网络算法，处理更大规模的数据，对于现实工程当中海量数据的超大规模神经网络具有参考价值。