密级:____

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

博士学位论文

机器学习处理器研究

作者姓名: 刘道福

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

培养单位: 中国科学院计算技术研究所

2015 年 5 月

Research on Machine Learning Processor

By
Daofu Liu

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

Institute of Computing Technology, Chinese Academy of Sciences

May, 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：　　日期：2015.5.28

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：　　导师签名：　　日期：2015.5.28

摘　要

随着机器学习重要性的不断增长，业界对于机器学习处理的速度和性能功耗比也提出了越来越高的要求。

一方面，大数据时代的到来使得互联网公司拥有海量的数据来进行机器学习，不断提高学习的精度。这使得机器学习的数据处理速度至关重要。工业界的一个常见问题是空有大量数据却来不及处理。例如，科大讯飞语料库的增长速度已经达到了其语音识别模型训练速度的 5 倍。这里一个重要原因是通用 CPU/GPU 的机器学习处理速度太慢。谷歌进行猫脸识别的训练甚至动用了数万个处理器核。更加严峻的是，正如 EMC 公司在 2011 年指出，互联网数据指数增长的速度，已经超过了摩尔定律的增长速度。这也就意味着，通用 CPU/GPU 的处理能力和对机器学习训练速度的需求之间的剪刀差，将会指数扩大。

另一方面，随着数据规模的进一步增大，以及受限于主频，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用，因此，急需提出一种高效的多核并行结构来进行大规模高性能的机器学习，而在所有多核并行体系结构中，核心问题就是存储一致性和缓存一致性协议。

本文的目标就是设计支持大规模并行数据处理和分析的高性能低功耗的多功能机器学习处理器。为了实现这个目标，本文从算法分析和访存优化，处理器核设计，多核并行扩展等三方面对通用机器学习处理器进行了研究，并分别提出了三种创新性技术。

- **基于分块调节的机器学习访存优化技术。**基于算法决定结构的设计思想，本文首先对应用最广泛，最典型的几种机器学习算法进行了分析，具体分析了包括运算特征和访存特征，并发现现有的算法存在对片外访存带宽过高的现象。为了解决这个问题，本文提出了一种分块调节的技术，可以大大减少片外访存带宽的需求，避免访存成为机器学习处理器的性能瓶颈。实验表明，分块调节访存优化技术可以将几种典型的机器学方法的片外访存带宽需求减少 46% 到 93% 不等。

- **多功能机器学习处理器。**基于前述算法分析，本文设计了一个支持多种机器学习算法（如 k-NN，k-Means，深度学习，支持向量机，朴素贝叶斯，分类树等等）加速的多功能机器学习处理器 -PuDianNao。PuDianNao 的设计过程中充分考虑了机器学习的运算特征和访存特征，可以很好的用于机器学习应用加速，并且避免存储墙的问题。同时，为了减少处理器的面积和功耗，我们基于对机器学习算法的精度需求分析，提出了不同流水级采用不同精度数据的机器学习运算单元。同时，为了提高通用性，本文为该机器学习处理器设计了一套指令集，该指令集可以自由组合，完成一些新的机器学习方法的加速和处理。实验表明，在 MNIST 和 UCI 的基准测试集中，相比顶级的 Nvidia K20 图形处理器（GPU），本文的机器学习处理器（PuDianNao）取得了 1.20 倍的加速比，却只消耗了 GPU 1/128.41 的能耗。

- **一种无共享信息的缓存一致性协议。**随着数据规模的进一步增大，以及受限于主频，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用。因此，迫切需要一种可扩展的多核（众核）结构的专用处理器。而在多核和众核结构中，缓存一致性对处理器的性能和功耗有显著影响。传统的缓存一致性协议都存在对性能影响过大以及为了维护缓存一致性需要额外的缓存目录面积的问题。本文为了解决该问题，提出了一种新的基于自无效和猜测执行的，无需存储共享信息目录的缓存一致性协议 NSI，该协议相比传统的 MESI 协议，可以显著地提高性能，减少片上网络通讯和片上网络通讯功耗。实验表明，在一个 16 核的多核处理器中，使用 SPLASH 和 PARSEC 基准测试集，相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储备份信息的目录以及其电路面积，而且平均可以提高 7.80% 的程序性能，减少 31.10% 的片上网络通讯，以及减少 12.39% 的功耗。

关键词： 机器学习，加速器，体系结构，访存优化，缓存一致性，缓存一致性协议

Research on Machine Learning Processor

Daofu Liu

Directed by Professor Yunji Chen

Abstract

In the era of data explosion, Machine Learning (ML) techniques have become pervasive tools in emerging large-scale commercial applications such as social network, recommendation system, computational advertising, and image recognition. Facebook generates over 10 Petabyte (PB) log data per month. Taobao.com, the largest online retailer in China, generates tens of Terabyte (TB) data every day. The increasing amount of data poses great challenges to ML techniques, as well as computer systems accommodating those techniques.

The most straightforward way to accelerate large-scale ML is to design more powerful general-purpose CPUs and GPUs. However, such processors must consume a large fraction of transistors to flexibly support diverse application domains, thus can often be inefficient for specific workloads. In this context, there is a clear trend towards hardware accelerators that can execute specific workloads with very high energy-efficiency or/and performance.

On the other hand, with the ever-increasing density of on-chip transistors and limited frequency increasing, Chip Multi-Processor (CMP) has become the mainstream in industry. In the CMP system, the most key problem is maintain cache coherence, as the core number increasing, a high-efficient cache coherency is needed.

In this paper, we propose three novel techniques related to design a high efficiency and high performance multicore machine learning processor.

- **Memory access optimization for machine learning based on tiling.**

First, we conduct a thorough analysis of the ML techniques to extract critical computational primitives and locality optimizations that need to be supported by the accelerator. Based on this analysis, we proposed a memory access optimization techniques based on tiling. The experiment results show that such technique can reduce the off-memory bandwidth from 46% to 93% for different ML techniques.

- **A machine learning processor architecture supporting multi machine learning methods.** Base on the thorough analysis on computational primitives and locality properties of different ML techniques, we proposed an ML processor called PuDianNao, which accommodates seven representative ML techniques, including k-means, k-nearest neighbors, naive bayes, support vector machine, linear regression, classification tree, and deep neural network. Benefited from our thorough analysis on computational primitives and locality properties of different ML techniques, PuDianNao can perform up to 1056 GOP/s (e.g., additions and multiplications) in an area of 3.51 mm^2, and consumes 596 mW only. Compared with the NVIDIA K20M GPU (28nm process), PuDianNao (65nm process) is 1.20x faster, and can reduce the energy by 128.41x.

- **A high performance and high effiecency cache coherence.** With the ever-increasing density of on-chip transistors and limited frequency increasing, Chip Multi-Processor (CMP) has become the mainstream in industry. In the CMP system, the most key problem is maintain cache coherence. In this paper, we reveal the interesting fact that sharer information is actually an unnecessary luxury in practice. Based on this key observation, we propose a lightweight novel scheme called *NSI (Non Sharer Information)*, which removes the sharer information and Invalidation/Ack messages, and efficiently maintains cache coherence using a novel *self suspicion + speculative execution* mechanism. Experimental results over various SPLASH2 and PARSEC2.0 benchmarks show that on a 16-core processor, NSI not only removes the chip area cost for recording the sharer information, but also improves processor performance by 7.80%, reduces overall network traf-

fic by 31.10%, and reduces energy consumption of the network by 12.39% (in comparison with traditional MESI protocol with full directory). Moreover, NSI does not involve any modification to programming languages and compilers, and hence is seamlessly compatible with legacy codes.

Keywords: Machine Learning, Accelerator, Micro Architecture, Cache Consistency, Cache Coherency

目　录

表　格

插　图

第一章 绪论

随着机器学习重要性的不断增长，业界对于机器学习处理的速度和性能功耗比也提出了越来越高的要求。一方面，大数据时代的到来使得互联网公司拥有海量的数据来进行机器学习，不断提高学习的精度。这使得机器学习的数据处理速度至关重要。工业界的一个常见问题是空有大量数据却来不及处理。例如，科大讯飞语料库的增长速度已经达到了其语音识别模型训练速度的 5 倍。这里一个重要原因是通用 CPU/GPU 的机器学习处理速度太慢。谷歌进行猫脸识别的训练甚至动用了 1.6 万个处理器核。更加严峻的是，正如 EMC 公司在 2011 年指出，互联网数据指数增长的速度，已经超过了摩尔定律的增长速度。这也就意味着，通用 CPU/GPU 的处理能力和对机器学习训练速度的需求之间的剪刀差，将会指数扩大。

另一方面，大数据可潜在提升机器学习的精度，但也不可避免地增大了机器学习模型的复杂度，这对硬件平台（如云服务器和移动终端）的性能和功耗带来了很大的挑战。近年来，随着深度学习等技术的提出，机器学习所建立的模型越来越复杂。谷歌 2012 年提出了一个 10 亿参数神经网络模型，2013 年就将其扩展至 110 亿参数。2014 年百度大脑甚至采用了一个拥有超过 200 亿参数的模型。在同样的硬件平台下，超大模型可能会提供更加准确的预测，但也会显著影响实时性。而实际应用往往对机器学习的预测有严苛的实时性要求(例如广告推荐必须在给定的极短时间内完成，否则无法出现在相关页面上)。当前的许多主流硬件平台（如移动终端上的 CPU，甚至云服务商的 GPU）往往都难以在短时间内完成超大模型的预测。更重要的是，移动终端（如手机）往往要频繁运行机器学习预测任务（如语音识别、图像识别等），如果不能以很高的性能功耗比进行机器学习预测，手机电池的续航能力将会大打折扣。

要解决大数据时代机器学习处理的速度和功耗问题，仅仅依靠传统的通用处理器是远远不够的。众所周知，通用处理器的发展受到功耗制约，几乎已经无法通过提高主频来进一步提升性能。虽然可以通过多个核/多个芯片的并行来提高性能，但是相应的，大数据处理系统的功耗也会急剧地增加，从而导致运营成本的上升。

为了实现大数据时代对机器学习应用的高性能低功耗处理，设计专用的机器学习处理器很有必要。在设计该专用机器学习处理器时，需要考虑以下问题。

首先，由于大数据时代的数据不仅数据量非常大，其处理和分析的种类需求也非常多，不同种类的大数据分析需要不同的机器学习方法进行处理，因此，所设计的机器学习处理器需要能够通用，具体来说，要能够支持多种机器学习算法，支持不同的机器学习场景（如分类，聚类，回归等）。其次，由于不同机器学习方法的特征差异很大，因此，需要仔细分析不同机器学习算法的计算特征和访存特征，抽取共性，从而指导机器学习处理器的设计。最后，机器学习处理器核心运算速度提升上去后，访存就会称为瓶颈，因此，对访存的优化就显得尤其重要。

另外一方面，由于大数据的量极大，受限于工艺和主频的限制，单核结构的处理器或加速器的性能已经无法满足未来的大数据处理的要求，因此，急需提出一种高效的并行多核结构来对大数据进行处理。

本文针对前述几个问题分别进行了研究，首先，本文分析了不同机器学习方法的运算特征和访存特征，并创新性的提出了分块调节的方式来优化机器学习应用的访存，避免访存成为性能瓶颈；其次，本文基于前述运算分析和访存分析，设计了一种支持多种机器学习方法，名为 PuDianNao 的机器学习处理器；最后，为了支持机器学习的多（众）核处理器处理，本文提出了一种高效的、无需存储共享信息的目录的、基于自无效的缓存一致性协议。

本章节将论述本文的研究背景，具体来说，主要从机器学习及其应用，并行编程模型，多核存储一致性和缓存一致性协议等几个角度来阐述。

1.1 机器学习及其应用

随着计算机技术（尤其是互联网和移动互联网技术）的进一步发展，每天有越来越多的数据生成，可以说大数据时代已经来临。据 IDC 公司统计，2011 年全球被创建和被复制的数据总量为 1.8ZB, 其中 75% 来自于个人 (主要是图片、视频和音乐), 远远超过人类有史以来所有印刷材料的数据总量 (200PB)。谷歌公司通过大规模集群和 MapReduce[1] 软件, 每个月处理的数据量超过 400PB[2]；百度每天大约要处理几十 PB 数据[2]; Facebook 注册用户超过 10 亿, 每月上传的照片超过 10 亿张, 每天生成 300TB 以上的日志数据[2]；淘宝网会

员超过 3.7 亿, 在线商品超过 8.8 亿, 每天交易数千万笔, 产生约 20TB 数据; 雅虎的总存储容量超过 100PB[2]。

在大数据处理和分析中，最常用也是最核心的数据处理和分析方法是机器学习。通过机器学习，可以挖掘大数据中有用的信息，根据大数据对未来数据进行预测等等。

机器学习是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。通常，机器学习基于已观测数据建立模型，并依靠模型来求解问题。近年来，随着大数据时代的来临，机器学习广泛应用于解决语音识别、图像识别、信息检索、广告推荐等重要问题。包括苹果、谷歌、IBM、Intel、微软、百度、华为、科大讯飞在内的多个国内外重要 IT 公司都已经投入了大量的资金和人力用于研发机器学习技术，并提供了多种相关的商业应用，例如苹果的语音控制工具 Siri、谷歌翻译、百度识图等等。可以说，机器学习已经通过云计算和移动终端深入到了普通人的日常生活中。

具体来说，如图1.1所示，机器学习的应用面涵盖了各种平台尤其是大数据平台的认知任务，如超级计算机上的商业分析和药物研制，数据中心上的广告推荐和自动翻译，智能手机上的语音识别和图像分析以及各种机器人和消费类电子的自动化系统。随着大数据时代的来临，越来越多的机器学习相关的应用必将继续涌现。机器学习应用会更加深入到科研，商业，以及普通人的生活中。

1.1.1 机器学习应用对性能要求越来越高

随着机器学习重要性的不断增长，业界对于机器学习处理的速度和性能功耗比也提出了越来越高的要求。一方面，大数据时代的到来使得互联网公司拥有海量的数据来进行机器学习，不断提高学习的精度。这使得机器学习的数据处理速度至关重要。工业界的一个常见问题是空有大量数据却来不及处理。例如，科大讯飞语料库的增长速度已经达到了其语音识别模型训练速度的 5 倍。这里一个重要原因是通用 CPU/GPU 的机器学习处理速度太慢。谷歌进行猫脸识别的训练甚至动用了数万个处理器核。更加严峻的是，正如 EMC 公司在 2011 年指出，互联网数据指数增长的速度，已经超过了摩尔定律的增长速度。这也就意味着，通用 CPU/GPU 的处理能力和对机器学习训练速度的需求之

图 1.1: 机器学习的应用领域

间的剪刀差，将会指数级扩大。

另一方面，大数据可潜在提升机器学习的精度，但也不可避免地增大了机器学习模型的复杂度，这对硬件平台（如云服务器和移动终端）的性能和功耗带来了很大的挑战。近年来，随着深度学习等技术的提出，机器学习所建立的模型越来越复杂。谷歌 2012 年提出了一个 10 亿参数神经网络模型[3]，2013 年就将其扩展至 110 亿参数[4]。2014 年百度大脑甚至采用了一个拥有超过 200 亿参数的模型。在同样的硬件平台下，超大模型可能会提供更加准确的预测，但也会显著影响实时性。而实际应用往往对机器学习的预测有严苛实时性要求(例如广告推荐必须在极短的给定时间内完成，否则无法出现在相关页面上)。当前的许多主流硬件平台（如移动终端上的 CPU，甚至云服务商的 GPU）都往往难以在短时间内完成超大模型的预测。更重要的是，移动终端（如手机）往往要频繁运行机器学习预测任务（如语音识别、图像识别等），如果不能以很高的性能功耗比进行机器学习预测，手机电池的续航能力将会大打折扣。

1.1.2 机器学习应用在传统平台不够高效

实际上，通用 CPU/GPU 在机器学习处理上的低效并不出人意料。无论是从计算单元还是存储层次上看，通用 CPU/GPU 考虑的应用面非常宽泛，并没

有对机器学习进行针对性的优化。在计算单元方面，通用 CPU/GPU 主要是提供算术运算（加减乘除）以及逻辑运算（与或非）的能力。诚然，以算术/逻辑运算为基础，可以构建出各种机器学习算法。但是这些基本算术/逻辑运算距离机器学习常用的非线性激活函数、计数、排序、熵计算等操作还有很遥远的距离。这些机器学习常用操作往往需要用成百上千个加减乘除与或非才能搭建出来。更重要的是，通用 CPU/GPU 每做完一次算术/逻辑运算，运算器都必须把结果花很大的延迟和能量代价写回寄存器堆。下一次进行相关的运算时又要不厌其烦地再把之前运算的结果从寄存器堆中读回来供给运算器。这里面数据搬运的延迟和能量损耗，甚至超过了算术/逻辑运算本身。

从存储层次上看，通用 CPU/GPU 主要是采用片上多级缓存加上片外内存的结构。出于通用性的考虑，CPU/GPU 的片上缓存规模并不大。例如，Intel 于 2014 年推出的高端服务器 CPU E7-8880L v2 片上三级缓存加起来也仅有不到 50MB。而 NVidia 的主流高性能计算 GPU 产品 K20X 所有两级缓存加起来也不到 3MB。而现有的机器学习模型越来越大，甚至可以达到数十 GB，难以在通用 CPU/GPU 芯片片上完全放下。这就迫使通用 CPU/GPU 频繁去访问片外内存。而根据 Nvidia 的实验，从片外搬运 64 位的数据到芯片里面所花的时间和能量，是进行 64 位乘法所花的时间和能量 10 倍以上。也就是说，绝大部分时间和能耗都将被消耗在访问外存上。此外，通用 CPU/GPU 对片上缓存的管理也并不令人满意。它们不能充分利用机器学习算法的特点，往往会用一些流式数据（如训练样本）把真正频繁访问的数据（如模型）从片上缓存挤到片外，从而进一步增大对片外内存的访问量。

1.1.3 机器学习处理器的价值

本文的机器学习处理器定位为大数据时代机器学习领域中的通用处理器，和信号处理领域的 DSP，图像处理领域的 GPU 是一样的地位。由于机器学习崛起迅猛，目前机器学习领域尚没有一款领域通用处理器，本项目研究的机器学习处理器，力求能够支持主流机器学习算法的共性运算和访存，从而能够适用于多种机器学习应用的加速，乃至未来的机器学习新应用新算法。本文在设计机器学习处理器的过程中，具体解决了前述两个问题，设计高效的运算单元，以及针对机器学习应用的访存优化和存储层次。同时，为了未来能够支持大规模的机器学习，扩展机器学习处理器到多核乃至众核，本文也对多核众核系统

中最重要的存储（缓存）一致性协议进行了研究。

1.2 多核处理器研究中的关键问题

随着机器学习的规模进一步扩大，受限于工艺和主频的限制，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用，因此，急需提出一种高效的多核并行结构来进行大规模高性能的机器学习，而在所有多核并行体系结构中，非常关键的问题就是存储一致性模型和缓存一致性协议。

1.2.1 存储一致性模型

为了解决程序在多核系统加速受限的问题，出现了串行程序不同的编程模式—并行编程。和串行程序不同，并行程序设计要求程序员显式的声明程序的那些部分是可以并行的。因此，对于一个习惯于串行程序编写的程序员来说，写并行程序难度要大。为了降低并行编程难度，需要为程序员提供统一的并行编程接口，多种并行编程模型被提出来了。

从存储管理及编程界面的角度看，并行编程模型可以分为共享存储和消息传递两类。在共享存储多处理器并行模型中，所有处理器共享主存储器（主存储器可以是主存，也可以是共享缓存），每个处理器（或核心）都可以访问（包括读取和写入）主存储器，处理器之前的通信通过访问共享存储来实现。在消息传递多处理器并行模型中，每个处理器都只能访问自己的局部存储器，处理器之间的通讯必须通过程序显式的消息传递来实现。

对于程序员来说，和消息传递相比，共享存储体系结构编程模型简单，程序访问在共享地址空间中的数据就如同访问在传统的虚存中的数据一样，不用考虑数据所在的具体位置。由于事实上共享存储系统里面存储的各部分是分布到各处的，访存还是需要通过低层的通讯（消息传递）完成的。最严重的是，由于数据在多处具有备份，导致数据在不同的备份会出现不一致的情况，存储一致性就是为了解决这个问题。

为了改善性能，共享存储系统依赖复制共享数据项（多个备份）和允许在许多结点上并发访问。但是，如果并发访问不仔细地加以控制，内存访问可能以不同于程序员所期望的次序被执行。非正式讲，一个内存是一致的，如果由读操作返回的值总归是程序员所期望的值。

存储一致性（Memory Consistency）模型实质上是软件和存储器间的契约[5]，它是说，如果软件同意服从某模型给出的规则，则存储器允诺软件在这种模型下正确地工作；否则，存储器操作的正确性将得不到保证。简单来说，存储一致性模型提供了一套并行程序在共享存储的多核处理器上执行的正确标准。

存储一致性模型依据对访问顺序的限制由紧到松依次分为严格一致性(strict consistency)，顺序一致性 (Sequential consistency)，因果一致性 (Causal Consistency)，一般一致性 (General Consistency)，处理机一致性 (Processor consistency)，以及弱一致性 (Weak consistency) 等等。

这些一致性模型对性能的影响越来越大，出于性能考虑，现在的处理器普遍采用弱一致性模型或者更弱的一致性模型，弱一致性模型的具体定义如下：

在一个多处理器系统中，访存操作符合弱一致性当且仅当：

1. 访问全局的同步变量是顺序一致的。
2. 在所有非同步变量写操作被全局观察到之前，不允许访问任何同步变量。
3. 在之前一个同步写操作被全局观察到之前，不允许访问任何非同步变量。

1.2.2 缓存一致性协议

存储一致性协议（Memory Coherence）是为了让处理器等硬件能够支持存储一致性模型所设计的一套协议。在片上多核乃至众核时代，处理器设计者主要关注片内分布式缓存的存储一致性协议。当前，在共享缓存领域，主要有两类存储一致性协议。基于广播的存储一致性协议和基于目录的缓存一致性协议。

在基于广播的缓存一致性协议中，每个核对缓存块的独占写请求（RdEx）都会通过广播立即被所有核观察到。受限于同时只允许一条广播存在，广播协议很难被扩展到众核。而基于目录的缓存一致性协议，则有更好的扩展性，已经被学术界[6]和工业界[7]广泛采用。基于目录的缓存一致性协议的基本思想是通过目录来跟踪共享缓存块的私有备份情况。当某个处理器核发出独占请求时，片上网络会根据目录通知之前拥有该缓存块备份的处理器 (或处理器核)，使其私有备份无效，这样就可以保证所有的独占操作（一般为写操作）立即被所有处理器观察到。

由于基于广播的缓存一致性协议扩展性差，本文主要关注基于目录的缓存一致性协议。当前，基于目录的缓存一致性协议主要有 MESI。然而，随着系统中处理器（核）数增加，目录所需要的存储空间也越来越大，同时，大量的无效信息也会导致大量的片上网络通讯乃至堵塞片上网络，造成系统性能的极大损失。本文的最后一个创新点就致力于解决该问题。

1.3 本文主要贡献

本文的目标就是设计支持大规模并行数据处理和分析的高性能低功耗的多功能机器学习处理器。为了实现这个目标，本文从算法分析和访存优化，处理器核设计，多核并行扩展等三方面对通用机器学习处理器进行了研究，并分别提出了三种创新性技术。

- **基于分块调节的机器学习访存优化技术。**基于算法决定结构的设计思想，本文首先对应用最广泛，最典型的几种机器学习算法进行了分析，具体分析了包括运算特征和访存特征，并发现现有的算法存在对片外访存带宽过高的现象。为了解决这个问题，本文提出了一种分块调节的技术，可以大大减少片外访存带宽的需求，避免访存成为机器学习处理器的性能瓶颈。实验表明，分块调节访存优化技术可以将几种典型的机器学方法的片外访存带宽需求减少 46% 到 93% 不等。

- **多功能机器学习处理器。**基于前述算法分析，本文设计了一个支持多种机器学习算法（如 k-NN，k-Means，深度学习，支持向量机，朴素贝叶斯，分类树等等）加速的多功能机器学习处理器 -PuDianNao。PuDianNao 的设计过程中充分考虑了机器学习的运算特征和访存特征，可以很好的用于机器学习应用加速，并且避免存储墙的问题。同时，为了减少处理器的面积和功耗，我们基于对机器学习算法的精度需求分析，提出了不同流水级采用不同精度数据的机器学习运算单元。同时，为了提高通用性，本文为该机器学习处理器设计了一套指令集，该指令集可以自由组合，完成一些新的机器学习方法的加速和处理。实验表明，在 MNIST 和 UCI 的基准测试集中，相比顶级的 Nvidia K20 图形处理器（GPU），本文的机器学习处理器（PuDianNao）取得了 1.20 倍的加速比，却只消耗了 GPU 1/128.41 的能耗。

- **一种无共享信息的缓存一致性协议。**随着数据规模的进一步增大，以及受限于主频，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用。因此，迫切需要一种可扩展的多核（众核）结构的专用处理器。而在多核和众核结构中，缓存一致性对处理器的性能和功耗有显著影响。传统的缓存一致性协议都存在对性能影响过大以及为了维护缓存一致性需要额外的缓存目录面积的问题。本文为了解决该问题，提出了一种新的基于自无效和猜测执行的，无需存储共享信息目录的缓存一致性协议 NSI，该协议相比传统的 MESI 协议，可以显著地提高性能，减少片上网络通讯和片上网络通讯功耗。实验表明，在一个 16 核的多核处理器中，使用 SPLASH 和 PARSEC 基准测试集，相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储备份信息的目录以及其电路面积，而且平均可以提高 7.80% 的程序性能，减少 31.10% 的片上网络通讯，以及减少 12.39% 的功耗。

1.4 论文的组织

本文共包括六章，具体组织如下:

在第一章中，也即本章中，我们介绍了机器学习加速器（处理器）研究的相关背景。

在第二章中，我们介绍了相关研究的现状。由于本文所研究的面向大数据时代的机器学习处理器是异构并行加速器的典型类别之一，第二章首先介绍了异构并行加速器的总体研究现状。接着介绍异构并行核 —机器学习加速器核的研究现状。最后关注了并行研究的现状，并且着重介绍了存储一致性模型和缓存一致性协议的研究现状。

在第三章中，我们详细论述了第一个创新点，机器学习访存优化技术。具体来说，在基于对机器学习的核心运算以及访存分析的基础上，提出了一套分块调节的访存优化技术，避免访存成为性能瓶颈。同时总结了机器学习的核心运算以及共性访存模式，为下一步设计通用机器学习处理器奠定了基础。

在第四章中，我们详细论述了支持多种常见机器学习方法（如 k-NN，k-Means，深度学习，支持向量机，朴素贝叶斯，分类树等等）的机器学习处理器的结构。并且，通过 EDA 工具对处理器的性能和功耗进行了评估，评估结

果表明，相比顶级的 Nvidia K20 GPU，本文的机器学习处理器取得了 1.20 倍的加速比，却只消耗了 GPU 1/128.41 的能耗。相比传统针对单一算法的机器学习处理器，本文的处理器更加通用，可以适应更多的数据和算法的变化，因为本文的机器学习处理器的设计思想是通过分析多种机器学习算法，抽取共性核心运算和共性访存特征，为所有机器学习方法的共性操作提供计算加速和访存加速。

在第五章中，我们设计了一个适用于多核众核架构时代，基于弱一致性的、无广播、无需存储共享信息的、无目录的缓存一致性协议 NSI。该协议能够通过新型的自我怀疑 + 猜测执行机制，消除用于存储共享信息的目录，以及所有写无效以及对应的应答消息。因此，NSI 明显降低芯片面积成本和能量消耗。同时，由于减少了片上网络的拥塞，处理器的性能也有所提高。

最后一章对全文工作进行了总结，并探讨了未来研究的方向。

第二章　研究现状

在本章中，我们将介绍相关研究的现状。由于本文所研究的面向大数据时代的机器学习处理器是异构并行加速器的典型类别之一，本章首先介绍了异构并行加速器的总体研究现状。接着介绍异构机器学习加速的研究现状。最后从体系结构层面关注了并行研究的现状，并且着重介绍了存储一致性模型和缓存一致性协议的研究现状。

2.1　异构并行加速器研究现状

高效能的异构并行设计中的一个重要挑战是如何设计合理的基础架构。当前，按其加速核之间的耦合程度划分，有两类异构并行基础架构，即松耦合结构和紧耦合结构。

在松耦合的异构处理器中，加速核是独立于主通用核的 IP。在这种架构中，当需要使用加速核时，通用处理器核通过片上网络向该加速核发起命令，之后通用核能够并行地处理其它任务。这类方法国外 Hauser 等人 1997 年就曾提出[8]。工业界最常见的异构处理器，其通用处理器核 + 图形处理器 GPU 的结构就是这类（如 AMD 的 APU[9]）。Kumar 等人还提出在同一个处理器上集成多个采用同指令集但是大小和性能不同的通用核[10]。这个技术目前已经应用于一些嵌入式 ARM 多核芯片中。近年来，AMD、ARM、Imagination、LG、高通、三星和德克萨斯仪器等有影响力的企业共同发起了异构系统架构基金会（Heterogeneous System Architecture Foundation，简称 HSA）[11]。由于 HSA 主要面向通用核 +GPU 的平台，它很难解决为其它类型的异构核编程的困难。

国内在松耦合的异构处理器方面也已开展深入的研究。岳虹[12] 于 2006 年设计了一个嵌入式异构多核处理器，该嵌入式多核处理器由一个主通用核和一个异构的多媒体应用核组成。陈芳园等人[13] 设计了嵌入式异构双核微处理器，该处理器和传统的异构处理器不同之处在于，两个核分别为同步电路核和异步电路核，异步电路核可以大大提高性能功耗比。先后曾在 TOP500 排名第一的天河 1A[14] 和天河 2 号[15]，也是基于通用核 +GPU 的松耦合结构搭建的。

松耦合的异构并行架构中通用核和加速核能并行地工作。但它的劣势在于通用处理器核和加速核之间的通讯，会导致额外的编程负担以及额外的延迟。这个开销可达数百甚至数千个周期。对于一些相对短的任务，执行这些通讯指令本身的时间甚至可能会抵消加速核带来的加速。因此，一般来说，该种架构仅仅适合于粗粒度的并行加速，如视频编解码和图形渲染等。

在紧耦合异构处理器中，加速核以类似功能部件的形式（类似定点或浮点功能部件）紧密镶入通用核中。在这种架构中，需要加速的任务以专门的硬件加速扩展指令的形式进行执行，比如，加解密运算可以通过实现专门的加解密功能部件完成。Lodi[16] 等人提出了一个基于 VLIW 指令集的面向嵌入式应用的紧耦合型异构处理器。N. Vassiliadis 等人则基于该结构提出了多种异构并行加速系统[17]，如 MPEG4 编解码[18]。

紧耦合异构处理器不需要额外的通讯或者控制指令，因为其通讯一般直接通过寄存器文件完成。因此它对用户友好，易于编程的。然而，该结构会占用通用核的指令窗口，很难做到真正的并发。另外，加速指令会占用宝贵的指令编码空间，因此紧耦合异构处理器的可扩展性会受到指令编码长度的限制。

2.2 机器学习加速器研究现状

为了加速机器学习，学术界和工业界从各个方面都进行了研究，主要包括两方面：一方面，通过寻找更高效的机器学习算法实现，从而减少机器学习算法的执行时间，让机器学习算法在实用性上得到提高；另一方面，通过提出性能更高的硬件架构，如集群，GPU，专用加速器等，来提高运行机器学习算法平台的计算能力从而加速机器学习的应用。在本项目中，我们主要关注第二方面的国内外研究现状。

国际上，为了加速机器学习应用，有大量的关于通过 CPU 集群或 GPU 来加速机器学习应用的研究，谷歌在 2012 年利用 16000 个 CPU 的集群加速训练了一个 10 亿参数神经网络模型[3]，并于 2013 年用 GPU 集群将其扩展至 110 亿参数[4]。

然而，无论使用使用 CPU 集群还是使用 GPU 来加速机器学习的算法，耗费的硬件资源和功耗都非常大。更加严峻的是，正如 EMC 公司在 2011 年指出，互联网数据指数增长的速度，已经超过了摩尔定律的增长速度 [21]。这也

就意味着，通用 CPU/GPU 的处理能力和对机器学习训练速度的需求之间的剪刀差，将会指数扩大。

为了解决通用处理器 CPU 以及图形加速器 GPU 在机器学习加速上的不足，学术界开始提出使用专用加速器来加速机器学习的应用，从而提高机器学习算法执行速度，同时降低所需要的功耗，达到很好的性能功耗比。早在 2007 年，Yeh 等人就提出了使用专用加速器来加速 k-近邻算法 (kNN)[19]，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[20]。Hussain 等人为 k-Means 算法设计了一个专用的 FPGA 加速器[21]，并将专用加速器和 CPU 以及 GPU 进行了比较[22]；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[23]。Meng Hongying 等人则为朴素贝叶斯算法 (Naïve Bayes, NB) 在图像识别上的应用提出了一个 FPGA 专用加速器[24]。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[25] 和 Markos[26] 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[27]。在神经网络方面，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[28] 以及大规模的卷积神经网络的在 FPGA 上的实现[29]，Farabet 还研究了卷积神经网络在机器视觉的具体应用[30]。

国内方面，浙江大学的蒋德等人研究了支持向量机加速器的设计，该设计的重要特点是能够支持在线学习[31]。2014 年，计算所本项目团队设计了一个小尺寸高吞吐量的神经网络加速器，该加速器可以在 $3mm^2$ 面积达到了 CPU 神经网络处理速度的 117.87 倍[32]。

然而，前述的研究工作的目标都是致力于加速某一种机器学习方法。而在实际应用中，由于应用特征的不同，适用的机器学习方法也不同。即使对同一应用，也可能需要用到不同的机器学习方法，比如在数据预处理阶段需要使用主成分分析 (PCA) 的方法，而在训练阶段需要使用支持向量机的方法。因此，迫切的需要有一种能适应更多的机器学习算法的加速器结构。针对此问题，近年来国内外也有一些相关的研究。

国际方面，2009 年，NEC 的 Hans 等人提出了一种基于 FPGA 的，能够同时支持两种最流行机器学习方法 (支持向量机和卷积神经网络) 的加速器结构[33]。而 Cadambi 和 Majumdar 则致力于设计一种可编程的，能够支持更多

机器学习算法（包括支持向量机，卷积神经网络，k-近邻，k-平均等）的通用机器学习加速器结构[34–36]。而国内方面，国防科大也研究了多种机器学习或模式识别方法的加速器的结构研究，具体来说，他们研究了适合 k-means，支持向量机，遗传算法，以及人工神经网络等机器学习算法硬件体系结构[37]。

然而，无论还是国内和国外，这些研究都存在着一些缺点。一方面，受限于研究平台，这些研究都是基于 FPGA 的，而在大规模机器学习应用中，FPGA 的成本和功耗都比较高。因此，迫切需要研究在专用集成电路上的机器学习加速器的实现。另一方面，这些研究缺乏对所有机器学习算法的共性研究，包括访存特点，公共计算因子等。因此，为了设计能够高效支持多种机器学习方法的通用机器学习加速器，需要对各类机器学习方法进行分析研究，并在此基础上提取共性，设计机器学习加速器。

2.3 并行编程模型和缓存一致性协议研究现状

为了解决程序在多核系统加速受限的问题，出现了串行程序不同的编程模式，并行编程。和串行程序不同，并行程序设计要求程序员显式的声明程序的那些部分是可以并行的。因此，对于一个习惯于串行程序编写的程序员来说，写并行程序难度要大。为了降低并行编程难度，为程序员提供统一的并行编程接口，多种并行编程模型被提出来了。

从存储管理及编程界面的角度看，并行编程模型可以分为共享存储和消息传递两类。在共享存储多处理器并行模型中，所有处理器共享主存储器（主存储器可以是主存，也可以是共享缓存），每个处理器都可以访问（包括读取和写入）主存储器，处理器之前的通信通过访问共享存储来实现。在消息传递多处理器并行模型中，每个处理器都只能访问自己的局部存储器，处理器之间的通讯必须通过程序显式的消息传递来实现。

和消息传递相比，共享存储体系结构编程模型简单，程序访问在共享地址空间中的数据正如同访问在传统的虚存中的数据一样，不用考虑数据具体所在的位置。由于事实上共享存储系统里面存储的各部分是分布到各处的，访存还是需要通过低层的通讯（消息传递）完成的。最严重的是，由于数据在多处具有备份，导致数据在不同的备份会出现不一致的情况，存储一致性就是为了解决这个问题。

为了改善性能，共享存储系统依赖复制共享数据项（多个备份）和允许在许多结点上并发访问。但是，如果并发访问不仔细地加以控制，内存访问可能以不同于程序员所期望的次序被执行。非正式讲，一个内存是一致的，如果由读操作返回的值总归是程序员所期望的值。存储一致性模型和缓存一致性协议就是为了解决该问题。

2.3.1 存储一致性模型

存储一致性模型实质上是软件和存储器间的契约[5]，它是说，如果软件同意服从某模型给出的规则，则存储器允诺软件在这种模型下正确地工作；否则，存储器操作的正确性将得不到保证。简单来说，存储一致性模型提供了一套并行程序在共享存储的多核处理器上执行的正确标准。

存储一致性模型依据对访问顺序的限制由紧到松依次分为严格一致性 (Strict Consistency)，顺序一致性 (Sequential Consistency)，因果一致性 (Causal Consistency)，一般一致性 (General Consistency)，处理器一致性 (Processor Consistency)，PRAM 一致性 (PRAM Consistency)，弱一致性 (Weak consistency)，释放一致性 (Release Consistency)，入口一致性 (Entry Consistency) 等等。下面我们将依次介绍这些存储一致性模型。

严格一致性定义为从存储器地址 X 处读出的值为最近写入 X 的值。严格一致性要求具有决定最近写的能力，因此它蕴涵请求的全序，所有共享访问事件有绝对的时间顺序。这在实际分布式系统中是无法实现的，所以没有任何分布式系统或多核处理器实现了该一致性。

顺序一致性由图灵奖获得者 Lamport 与 1979 年提出[38]，定义为“任意一次执行的结果都像所有处理器的操作以某种顺序的次序执行所得到的一样，而且各处理器的操作都按照各自程序所指定的次序出现在这个顺序中。”这个定义意味着，当程序在各个机器上并行运行时，任何一种有效的交错存储器访问顺序都是可认可的行为，但**所有处理器必须看见的是同样的访问顺序**。如果一个进程（处理器）看见的是一种交错，另一进程看见的是另一种交错，则这样的存储器不是一个顺序一致性的存储器。然而，同一程序再次并行运行，其存储器访问的交错次序会不同于上次的交错次序，这是允许的。在一块存储器中，若一个进程（或处理器）看到一种交错，另一进程看到另一个交错，这就不是顺序一致存储器。注意，这与时间无关，没有最近存入的概念。

因果一致性是由 Hutto 和 Ahamad 提出的[39]，因果一致性是顺序一致性的淡化，它按有无可能的潜在因果联系来区分各事件。假设进程 P1 写变量 x，然后 P2 读出 x，写入 y。这里读出 x 和写入 y 之间可能有潜在的因果联系，因为 y 的计算很可能决定于 P2 读到的 x 值（即 P1 写入的值)。另一方面，若两进程处理器同时地写两个变量，就没有因果联系。先有读操作之后执行写操作，两个事件就可能有因果联系。相似的，读和提供所读数据的写有因果关系。没有因果关系的操作称为并发的（concurrent)。定义为“因果一致的存储器应遵守以下条件：可能因果相关的写操作应对所有进程可见，且顺序一致。并发写操作在不同机器看来顺序可能是不同的”。

处理器一致性由 Goodman 提出[40]，定义为由一个处理器发出的写以它们所发出的同样次序被观察到。但是，从两个不同进程写的次序，它们自身或第三个处理器所观察到的次序不一定相同。即从不同的处理器的同时对同样的位置的两个读可以产生不同的结果。和顺序一致性相比，只需要同一个处理器的写操作或者不同处理器对同一地址的写操作在所有的处理器看到的顺序是一致的即可以（顺序一致性要求读写都一样)。

弱一致性模型的提出[5]，是因为人们观察到大多数并行程序排定几个进程对数据存取次序时都需要使用同步操作；而在同步操作之间，一个进程所进行的数据存取操作次序对其他进程会是不可见的。例如，某进程对同步变量实施加锁后进入临界段，已在临界段内所做的存储器数据读写操作，由于互斥性其他进程根本看不到，更不用说读写操作次序了。在一个多处理器系统中，访存操作符合弱一致性当且仅当：

1. 访问全局的同步变量是顺序一致的。
2. 在所有非同步变量写操作被全局观察到之前，不允许访问任何同步变量。
3. 在之前一个同步写操作被全局观察到之前，不允许访问任何非同步变量。

释放一致性[41] 本质上和弱一致性相同，主要区别是如果存储器能够区分进入还是离开临界区的话，进入临界区会执行获取操作，退出临界区会执行释放操作。共享存储器在遵守以下规定时就是释放一致的：

1. 在访问共享变量前，进程所有先前的获取操作都已经成功地完成；
2. 在允许释放操作前，进程先前的所有读写操作都已经完成；

表 2.1: 各种存储一致性模型比较

一致性模型	说明
严格一致性	所有的共享访问事件都有绝对时间顺序
顺序一致性	所有进程都以相同的顺序检测到所有因果联系的事件
处理器一致性	所有的进程按照预定的顺序检测到来自一个处理器的写操作，来自其他处理器的写操作不必以相同的顺序出现
弱一致性	同步完成后，共享数据才可能保持一致
释放一致性	当离开临界区时，共享数据就保持一致

3. 获取操作和释放操作必须是顺序一致的。

以上几种一致性的区别从紧到弱可以总结为表2.1。

2.3.2 缓存一致性协议

一方面，随着大规模集成电路工艺的发展，集成电路的密度越来越大；另一方面，现有技术已经很难再进一步提升处理器的主频。这两方面的原因导致多核乃至众核处理器越来越成为工业界的主流。至今为止，美国英特尔公司 (Intel) 已经发布了 8 核的至强 (Xeon) 处理器[42] 以及 32 核的 Larrabee 处理器[43]；AMD 已经发布了 16 核的 Opteron 6200 处理器[44]；ARM 发布了 64 核的 Centip3De 处理器[45];IBM 发布了 8 核的 Power 7 处理器[46]。随着工艺进一步发展，未来的处理器可能集成上百甚至上千处理器核。

片上多核处理器通常用复杂的缓存层次，具体来说，所有核一般会共享最后一级高速缓存 (Last-Level Cache, 简称 LLC)，同时，每个核心也有其自己的小的一级缓存 (L1 缓存，简称 L1C，包括 L1 指令缓存，L1 数据缓存，以及”受害者缓存 (victim cache)”)。缓存一致性协议负责不同类型缓存中存储的数据的一致性，当前主流的缓存一致性可以分为两类，基于目录的缓存一致性协议 (比如[47]) 和基于广播的缓存一致性协议 (比如[48])。在基于广播的缓存一致性协议中，每个核对缓存块的独占写请求 (RdEx) 都会通过广播立即被所有核观察到。受限于同时只允许一条广播存在，广播协议很难被扩展到众核。而基于目录的缓存一致性协议，则有更好的扩展性，已经被学术界[6] 和工业界[7] 广泛采用。基于目录的缓存一致性协议的基本思想是通过目录来跟踪共享缓存块的

私有备份情况。当某个处理器核发出独占请求时，片上网络会根据目录通知之前拥有该缓存块备份的处理器，使其私有备份无效，这样就可以保证所有的独占操作（一般为写操作）立即被所有处理器观察到。

传统的基于目录的缓存一致性协议主要有 MESI[7] 和 MSI[49]，其中 MSI 是 MESI 的简化版本。MESI 主要包括四个状态，更改 (Modified), 独占 (Exclusive)，共享 (Shared) 和无效 (Invalid)。各状态的具体含义为: 无效状态，表示该块的数据确定不是最新的；共享状态，表示该块是通过读操作从 LLC 获取的备份，且暂时没被其他写操作无效；独占操作，表示该块是通过写操作获取的备份；更改状态，表示 L1C 通过写操作获得 EXC 块之后，若 L1C 继续写该块，则 L1C 的缓存块状态为 MOD 状态。其中 MSI 相比 MESI，少了独占状态。

然而，使用目录存储共享信息有两个缺陷。第一个缺陷是根据记录的共享信息，目录发出许多无效消息给各个处理器核心，无效 (Invalidation) 消息是指 LLC 发出的一个请求，请求从一个共享者核心消除对应的私有 L1C 块。目录也会从核心收回等量的 Ack 消息。这些消息不仅增加了互联网络的能量消耗，还因为网络拥塞降低了处理器的性能。第二个缺陷是，共享信息随着核心数目的增加而急剧增加，导致芯片面积的明显的消耗，同时带来额外的芯片功耗消耗。

针对基于目录的共享存储的缺陷，国际上已经有若干研究致力于解决这些缺陷。

针对目录太大的问题，某些早期的工作[48,50,51] 的目标是减少用于每个单独的最后一级共享缓存 (Last Level Cache，简称 LLC) 块的共享信息的位数。例如，稀疏目录[50] 用目录项中的每一位来表示 N ($N > 1$) 个处理器核心和 1 级私有缓存 (L1 Cahce，简称 L1C)；如果这 N ($N > 1$) 个 L1C 有任意一个具有 LLC 块的备份，那么将该块的目录项的对应位设为 1。虽然共享信息的目录面积消耗减少了 N 倍，但是稀疏目录常常导致额外的无效/回应消息，因为即使此时只有一个核心有 L1C 备份，无效消息也会发送给所有的 N 核。所以，稀疏目录会增加 N 倍的无效/回应消息的数量。因此，稀疏目录会降低高达 10.4% 的性能[50]。

Zebchuk 等人提出了一种名为 TL 的无标签目录[52]。TL 利用了 Bloom 过滤器来保守记录共享核信息，并且通过牺牲网络通信和整体性能减少了 48%

的目录的面积成本。Cuckoo 目录[53] 使用了高效的哈希函数来减少目录面积。但是，Cuckoo 目录引入了额外的无效消息，因为 Cuckoo 的哈希函数插入了失败率。所以，Cuckoo 目录的性能并不比传统的具有完美目录的 MESI 好。

Cuesta 等人[54] 建议对私有的内存区域禁用高速缓存一致性，因为它只会被一个线程访问。所以，无需提供为私有的内存区域提供目录项。Cuesta 等人还提出了一种机制来决定内存地址是否对于一个线程是私有的还是线程间共享的。因此，他们的技术能够在不改变性能的情况下降低 8 倍的高速缓存目录大小。当一个内存地址既不是私有的，也不是只读的时，SWEL[55] 通过强制排除该地址来删掉了全部目录。所以，对于 HPC 应用或多个使用一小部分共享和频繁写内存地址的虚拟机来说，SWEL 非常有效且高效（SWEL 报告说整体性能增加了 2.5%[55]）。最近，Ros 和 Kaxiras 提出了无目录多核一致性协议[56]。然而，他们的协议需要对共享数据的采用写穿透策略（write-thought strategy），这不仅会导致极大的性能开销，还会导致额外的片上网络通信。

Denovo[57] 是最新的避免使用目录的技术。它需要程序员将全部内存空间分为多个区域，而后给每个区域插入自无效信息。通过这样的程序员和编译器的支持，每个核心能够适当的使自己的私有缓存 L1C 无效。所以不再需要 LLC 目录和无效/回应消息。然而，Denovo 需要一个新的编程模型和相应的编译器支持，所以不能够直接兼容现有的代码（即使是源代码）。

为了减少网络通信量，Keleher 等人提出了一种名为懒惰释放一致性 (LRC)[58] 的存储一致性模型。在 LRC 中，有两种同步：获取和释放。无效信息能够在获取时间传递。所以，网络传输量能够有效降低。然而，如果没有程序员的标注，硬件很难区分获取和释放（获取和释放多被设置为相同的硬件元素）。所以，用弱一致性模型写的传统的代码不能从 LRC 中获益。

Lebeck 等人提出的动态自无效 (DSI) 协议[59] 允许每个核心能够在没有收到任何来自共享缓存（LLC）的无效消息的情况下自行将自己的共享状态的私有备份改为无效。因为减少了无效消息，DSI 能减少整体网络通信量高达 26%[59]。Lai 等人[60] 设计了类似于 DSI 的自无效机制，但是他们的工作侧重于如何准确的识别/预测哪个共享缓存块应当被自无效。

也有一些技术通过减少访问共享缓存（LLC）来降低网络通讯量。传统的基于目录的高速缓存一致性协议中，当一个核心在一级缓存发生缺失时，它必须询问共享缓存来获得最新的数据备份，而共享缓存本身也可能需要访问其他

私有缓存来获取最新的数据。为了降低这样的通讯，Kaxiras 和 Keramidas[61] 建议在私有缓存发生缺失的核心直接询问另一个最有可能有有效最新 L1C 备份的核心。综合一些其他的减少网络通信量技术，他们的工作平均降低了 55.91% 的网络通信量。另一工作，动态目录[62] 通过将目录放在对应最活跃的节点上来减少不必要的网络通信，因而平均降低了 16.4% 的片上网络通讯。虽然这些技术在降低网络通讯时非常有效，但是这些技术都仍然依赖于目录的存在。

第三章　机器学习算法分析和访存优化

在机器学习领域，机器学习技术传统上来看是通过其数学模型（线性或非线性）、其学习方式（有监督或无监督）、其训练算法（最大似然估计或梯度下降）等等方式来描述其特征的。然而，当从计算机结构的角度来分析机器学习技术，它主要是通过计算原语和局部性性质来描述特征的。因此，在这章中，我们首先从传统的角度概括地对机器学习技术的特征进行整体描述和分类，而后着重从体系结构设计者的角度详细分析了七种比较具有代表性的机器学习技术（包括 k 均值聚类算法（k-Means）、k 近邻算法（k-Nearest Neighbors，简称 k-NN）、朴素贝叶斯算法（Naive Bayes，简称 NB）、分类树算法（Classification Tree，简称 CT）、支持向量机算法（Support Vector Machine，简称 SVM）、线性回归算法（Linear Regression，简称 LR）和深度神经网络算法（Deep Neural Network，简称 DNN））。对于每一种技术，我们的分析包括三部分，首先，概述每种机器学习技术的方法及用途；然后，对其计算过程进行分析，并定义出每种技术中最耗时的步骤；最后，我们分析了每种技术中最耗时的那部分的局部性性质，并且在后面的部分中对其进行分析和优化。这种分而治之的思想很好的体现了我们加速器的思想：用专用的硬件计算结构来加速机器学习技术中最耗时的操作，并优化片上缓存，用轻量而又通用的运算逻辑单元来支持剩下的操作。

关于我们的局部性分析，为了估测不同机器学习技术的片外访存带宽需求情况，我们使用了一个内置缓存模拟器的处理器分析片外访存带宽需求。所模拟的处理器特征包括，片内缓存大小为 32KB，主频为 1GHz，具有一个 256 位 SIMD 引擎，在每一拍都能完成一个基本的运算操作（如对应向量位宽的点积或距离），即我们假设 SIMD 引擎能够在一个节拍中能够计算三输入（每个输入 256 位，如 F(a,b,c)）的基本函数。另外，除特殊说明外，我们假设在评估中，每种机器学习方法都提供了足够多的训练/测试/参考样例，且每个样例都是一个 32×32 位浮点向量。

3.1 机器学习简介

机器学习作为人工智能的核心之一，目前被广泛采用的定义是“利用经验来改善计算机系统自身的性能”[63]。机器学习技术已经被广泛的应用到了搜索引擎、医学、生物学、图像识别、声音识别等等领域，取得了较为显著的效果。同时，机器学习技术也是一个庞大的家族，不同的分类标准可以将其分为不同类别，如按照学习方式分类、按照算法用途分类等等。

3.1.1 按照学习方式划分

机器学习技术主要可以根据提供给学习系统的学习的“信号”或者“反馈”的性质来分成三大类：有监督学习、无监督学习和强化学习。

有监督学习是指，计算机能够接收到一个样例输入和它的目标输出（即标签），目标是通过学习得到一个通用的规则将新的输入映射到输出上去，也称为监督训练或有教师学习。其典型特征是训练样本的标签是已知的。各种分类算法是其典型例子。

无监督学习是给出训练样例的输入，但是没有对应的标签，通过学习样例输入来寻找对应的标签或类别，可以说，无监督学习本身也可能是一个目标(在数据的隐藏样例中发现)，或者是一个趋于结束的手段。其典型特征是训练样本的标签是未知的。各种聚类算法是其典型例子。

强化学习是指计算机程序与一个动态环境进行交互，需要达到某个目标(如使奖励信号 (强化信号) 函数值最大)，但是在学习过程中，没有人会明确告诉你是否离这个目标更接近了。

除了有监督学习和无监督学习，还有一种介于两者之间的叫做半监督学习——提供给计算机一个不完整的训练信号：一个训练集，但是部分标签缺失。此类问题来源于现实应用中收集特征数据的代价很低，而将特征向量关联到标签的代价却很高这一情况。

3.1.2 按照算法用途划分

机器学习技术有较为广泛的应用性，源于它的算法适用于多种用途。常见的算法用途包括分类、回归、聚类、降维、概率建模等等。

分类问题是指根据一个训练集，包括输入数据和标签，训练出一个“分类器”。而后，利用该分类器将新的测试集分为两类或者更多类。该“分类器”即为一个分类模型，常为一个数学函数，用于将输入样例映射到其对应的标签。这是典型的有监督学习方法。譬如垃圾邮件过滤问题就属于该类问题，输入是一封邮件或者类似的信息，分类结果是属于垃圾邮件或者不属于垃圾邮件。又如，病人病情诊断问题，输入是病人可以观察到的特征和症状（性别、年龄、血压、是否发热咳嗽等等），输出是病人可能得的病。常用的机器学习分类方法有 k-NN[64]，支持向量机等。

回归问题类似于分类问题，用于估测变量之间的关系，包括建立模型和分析多个变量来建立一个或多个独立的变量与其标签之间的关系，因而也属于有监督问题。回归问题与分类问题间最明显的差别就是回归问题的输出是连续的，而分类问题的输出是离散的。

聚类问题是指将给定的输入分类，使得每一类中的样例与其所属的类更为相似，而与其他类相似性较低。不同于分类问题，聚类问题中的“类”是事先未知的，分类原则依赖于样例间属性的相关性，而不是给定的类别，属于典型的无监督学习任务。

降维是指通过将输入从高维度的空间映射到一个低维度的空间中来简化输入。数据变换时，常用主成分分析来进行线性模型的变换，当然也存在一些方法针对非线性问题。主成分分析的基本思想是，构造原变量的一系列线性组合形成几个综合指标，以去除数据的相关性，并使低维数据最大程度保持原始高维数据的方差信息。

除了上述的分类方法以外，机器学习技术还可以从数据对象方面分类，分为两类、多类、多标签等问题；从学习模型方面分类，分为神经网络技术、分类树技术等等。下文便介绍了从学习模型的方面可以分为七种的机器学习技术，它们都是具有代表性的技术。介绍每种技术时都是从三个方面进行描述：算法介绍、核心算法分析和访存特征及优化。通过下文的优化方式，可以有效利用每种技术的局部性，减少算法对片外访存带宽的需求，避免访存成为性能瓶颈，为下一节设计机器学习处理器提供了很好的铺垫。

3.2 k 近邻算法（k-NN）

3.2.1 算法简介

k-近邻算法 (k Kearest Neighbor，简称 k-NN)[64]，是一种非参数学习算法。k-NN 算法的核心思想是如果一个样例在特征空间中的 k 个最接近的样例中的大多数属于某一个类别，则该样例也属于这个类别，并具有这个类别中的样例的特性。其中，k 是一个由用户预先设定的整数，通常较小。特别地，当 k=1 时，结果可简化为仅由最近的一个样例决定，此时可称为“最近邻算法”。

k-NN 被称为机器学习中最简单的分类学习算法，属于懒惰学习法的一种，其无需提前训练相关模型。k-NN 不仅可以用于分类问题，还可以用于回归问题。

3.2.2 核心计算分析

选择离测试样例距离最近的 k 个样例，那么测试样例的分类标签与该 k 个样例中多数样例的类别标签相同。

对于每一个测试样例，k-NN 都有两个主要步骤，一是计算测试样例和参考样例间的距离，另一个是寻找参考样例中距离测试样例最近的 k 个，并寻找出这 k 个测试样例中出现次数最多的标签，将该标签设置为测试样例的标签 (分类问题)，或者求得 k 个最近参考样例的平均标签（回归问题)。其中，最耗时的操作是计算样例间的距离。在 Intel Xeon E5-4620 处理器上，针对 UCI Gas 数据集的计算分析计算耗费的时间中，距离计算平均占到了 84.44%。

3.2.3 访存特征以及分块调节

正如前文分析所言，k-NN 中最耗时的操作是距离计算，图3.1 展示了距离计算最初的伪代码。

我们观察到一个参考样例在 $N_b - 1$ 次距离计算（N_b 是参考样例的数量）后可以被重用，而一个测试样例总能够在自己的循环中重用。当 N_b 非常大时，参考样例的重用距离也会非常大。由于芯片内部缓存不能够同时存储所有的参考样例，就会导致频繁的片外存储器访问，使得对于不同的测试样例，总是重新获得相同的参考样例。为了利用参考样例的局部性，我们将测试样例和参考

样例分块展开，并定义每个分块展开的块大小位 T_i 个测试样例和 T_j 个参考样例，具体见图3.2伪代码。

```
//Na is the number of testing instances
//Nb is the number of reference instances
for (i = 0; i < Na; i++){
  for(j = 0; j < Nb; j ++){
    Dis[i,j] = dis(t(i),r(j));
}  }
```

图 3.1: k-NN 距离计算原始代码

```
//Ti is the tiled testing block size
//Tj is the tiled reference block size
for (i = 0; i < Na/Ti; i++){
  for(j = 0; j < Nb/Tj; j ++){
    //tiled block
    for (ii = i*Ti; ii < (i+1)*Ti; ii++){
      for(jj = j*Tj; jj < (j+1)*Tj; jj++){
        Dis[ii,jj] = dis(t(ii),r(jj));
} } } }
```

图 3.2: k-NN 距离计算分块代码

分块展开显著减少了参考样例的重用距离，也显著减少了片外访存所需的带宽。通过实验也证实了上述优点。在我们的实验中，比较了分块展开前后片外存储带宽的需求。令 $T_i = T_j = 32$ ，即在一个分块展开块中有 32 个样例（每个样例都是 32 维向量，每个维度均为 32 位的浮点数），参考样例和测试样例的子块（每个块均为 $32 \times 32 \times 4B = 4KB$）可以同时存放在 32K 的片内缓存中。我们观察到平铺展开能够降低距离计算带来的片外存储带宽需求的 93.9%。

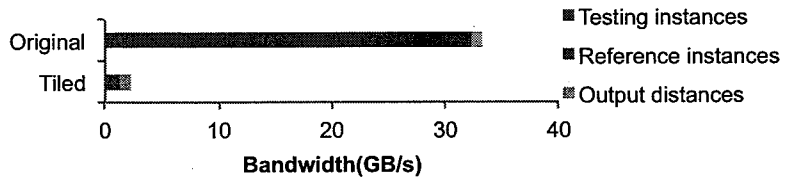

图 3.3: k-NN 原始版本和分块版本所需带宽对比

3.3 k 均值聚类算法 (k-Means)

3.3.1 算法简介

k-Means[65] 最初是信号处理中的矢量量化的一种方法，现在被数据挖掘中的聚类分析广泛使用。k-Means 的目标是将 N 个样例分成 k 组，使得每个样例属于离它均值最近的类，属于非监督机器学习技术的一种。

这个问题实际上是 NP 困难问题，但是在通常情况下，有许多有效的启发式算法，能够使得该问题快速收敛到局部最优值。

3.3.2 核心计算分析

k 均值聚类算法 (k-Means) 最初有 k 个随机的聚类重心，而后交替进行两个主要步骤：一是将每个样例分类，即分别计算出该样例与现有重心（均值）的距离，选择离它最近的重心即为该样例所属的那一类。二是为每一类更新重心，即计算该类现有所有样例的各个特征的均值，得到的即为该类的新的重心。在 k-Means 中最耗时的操作是距离计算。在 Intel Xeon E5-4620 处理器上，UCI Gas 数据集的计算耗时分析中，距离计算平均占到了 89.83%。

3.3.3 访存特征以及分块调节

k-Means 中最耗时的操作是距离计算，而距离计算可以用类似于 k-NN 中的优化方式（如，分块展开)。这里有一个小的区别就是用聚类重心（均值）和待聚类的样例来分别替代在进行 kNN 的局部性分析时的参考向量和测试向量。我们用与 k-NN 中分块展开实验的相同的设置（如样例长度和分块展开块大小等)，并观察到无论是聚类重心还是样例的分块展开都能减少距离计算的 92.5% 的片外存储带宽需求, 具体见图3.4。

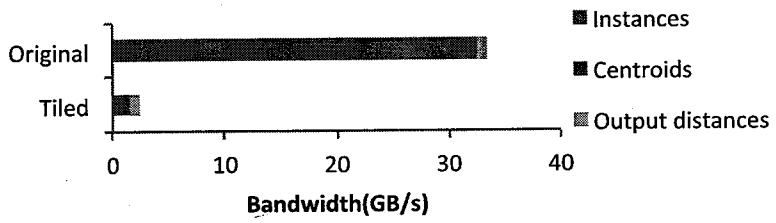

图 3.4: k-Means 原始版本和分块版本所需带宽对比

3.4 深度神经网络算法（Deep Neural Network）

3.4.1 算法简介

深度神经网络（Deep Neural Network, 简称 DNN）是在近十年中在机器学习界和工业界引起了广泛的兴趣的一种新型的多层感知器[66–68]。多层感知器（Multi-layer Perceptron, 简称 MLP）[69] 是典型的人工神经网络算法之一，能够在输入和输出之间建立非线性关系。DNN 由 MLP 发展而来，具有一种深度结构，即由许多隐藏层和相邻层间全相连的神经元组成。DNN 可以建立很多复杂的非线性关系，与类似的浅层模型相比，DNN 增加的隐藏层能够使算法从底层的数据特征中，利用复杂的数据来建立出更为合适的模型。

3.4.2 核心计算分析

DNN 包含三种计算模式，前馈计算，计算现有网络设置下通过每个给定输入计算网络输出（即预测计算）；预训练，局部调整相邻层的每对突触（连接权值）；全局训练，利用反向传播算法（Backpropagation algorithm, 简称 BP）全局调整突触[70]。

我们首先分析前馈计算。考虑用第 $g(g \geq 1)$ 层神经元的值来计算第 $g+1$ 层神经元的值。为了不失一般性，假设我们已经知道第 g 层神经元的值，该层具有 N_a 个神经元 $(x_1, \ldots, x_{N_a})$。我们要计算的第 $g+1$ 层神经元的值，该层具有 N_b 个神经元 $(y_1, \ldots, y_{N_b})$。对于第 $g+1$ 层的第 i 个神经元 y_i，我们利用公式 $y_j = f\left(\sum_{i=1}^{N_a} w_{ij}x_i + s_j\right)$ 来计算求值，其中，f 表示激活函数，如 sigmoid 和 tanh，w_{ij} 表示神经元 x_i 和 y_j 之间的突触，s_j 是偏移量。前馈计算中，计

算同一层所有 N_b 个神经元，能够写成形如 $Y = X \otimes W$ 的矩阵，其中:

$$W = \begin{bmatrix} s_1 & s_2 & \cdots & s_{N_b} \\ w_{11} & w_{12} & \cdots & w_{1N_b} \\ w_{21} & w_{22} & \cdots & w_{2N_b} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_a1} & w_{N_a2} & \cdots & w_{N_aN_b} \end{bmatrix} \tag{3.1}$$

$X = (1, x_1, \ldots, x_{N_a})$, $Y = (y_1, \ldots, y_{N_b})$, 运算操作 $\otimes$ 表示的运算类似于常见的矩阵/向量乘法，只不过是将乘法器的左边的每个行向量和乘法器的右边的每个列向量之间进行点积运算，也就是说，可以用 $p \cdot q$ 来替代 $f(p \cdot q)$，这里 f 表示神经元的激活函数。

预训练阶段，局部调整的相邻层间的每对突触（权重），可以作为全局训练的初始突触，能够用限制玻尔兹曼机（Restricted Boltzmann Machines，简称 RBMs) 来完成，其中最耗时的操作是迭代进行的由吉布斯采样算法（Gibbs sampling）支持的前馈计算和反馈计算（含有类似于前馈计算的运算)。全局训练使用反向传播算法来进行全局调整突触，最耗时的操作也是类似的前馈/反馈计算。

```
//x is the input neurons, y is output neurous
//w is the weights, w[j,0] = s[j]
y(all) = 0; //initialize all neurons
for(j = 0; j < Nb; j++){
  for(d = 0; d < (Na+1); d ++){
    y[j] += w[j,d]*x[d];
    if(d == Na)
       y[j] = f(y[j]);
  }
}
```

图 3.5: DNN 距离计算原始代码

3.4.3 访存特征以及分块调节

我们首先分析前馈计算。沿用上文中的参数，原始伪代码见图3.5

```
//Td is the tiled dimension block size
//Tb is the tiled synapse block size
//x is the input neurons, y is output neurous
//w is the weights, w[j,0] = s[j]
y(all) = 0; //initialize all neurons
for (d = 0; d < (Na+1)/Td; d ++){//tiling for
   dimension
  //only one neuron vector, no tiling for neurons
  for(j = 0; j < Nb/Tb; j ++){//tiling for
     synapses
    //instance and dimension tiled block
    for(jj = j*Tb; jj < (j+1)*Tb; jj++){
       for(dd = d*Td; dd < Td; dd ++){
         y[jj] += w[jj,dd]*x[dd];
         if(dd == Na)
            y[jj] = f(y[jj]);
       }
    }
  }
}
```

图 3.6: DNN 距离计算分块代码

一方面，从图中我们可以看出，第 g 层的神经元 ($x[i]$) 能够用 N_b 次来计算下一层的神经元 ($y[j]$)，每个突触 ($w[i,j]$) 只被用了一次。然而，在原始代码中，$x[i]$ 可以在总量为 $2N_a$ 次浮点运算 (乘法和加法) 重用。当向量 X 的突触数量 (如 N_a) 非常大时，重用距离也会非常大。另一方面，芯片的缓存也许不够大以做到同时存储下所有 N_a 个神经元的值，那么就不得不在重用时重新获取。为了减少内存传输，我们针对向量 X 中的神经元的循环进行了分块

展开。展开代码具体见图3.6。

我们用实验估计平铺展开前馈计算的影响，设定 $Na = 16384$(16384 个神经元总共需要 $16384 \times 4Byte = 64KB$)。我们观察到平铺展开能够减少前馈计算所需的存储带宽的 46.7%。具体结果见图3.7。

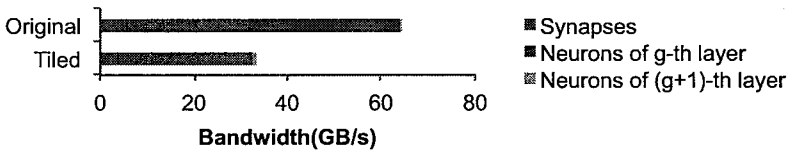

图 3.7: DNN 前向计算原始版本和分块版本所需带宽对比 ($N_a = 16384$)

类似的，由于预训练和全局训练阶段中最耗时的操作都类似于前馈计算，因而该局部性优化也能够适用于这两个阶段。

3.5 线性回归算法（Linear Regression）

3.5.1 算法简介

线性回归算法 (LR)[71] 是一种有监督学习技术，利用已知数据建立线性预测函数的模型，其中模型中未知的参数就从数据中估测得出，这些模型即为线性模型。常见的线性回归是用一个形如 $y = \sum_{i=0}^{d} \theta_i x_i$ （其中 x_0 恒等于 1）的线性函数模型表示标量响应变量 y 和一个 d 维矢量变量 $x = (x_1, x_2 ... x_d)^T$ 的关系。线性回归是第一个被细致研究的回归分析算法，并且在实践中应用广泛，这是因为根据数据建立线性模型，常常比非线性模型更容易更合适，因为这些估测结果的统计特性更容易确定。

3.5.2 核心计算分析

线性回归算法主要包含两部分，训练阶段和预测阶段：训练阶段是通过数据建立线性函数，预测阶段是利用现有的线性模型来预测每个测试样例的响应(标签)。由于预测阶段相对简单且基础，故我们从预测阶段开始分析。N 个不同的测试样例的预测响应可以由

$$Y = \theta X, \tag{3.2}$$

其中:

$$X = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \cdots & x_1^{(n)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & \cdots & x_2^{(n)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & x_d^{(3)} & \cdots & x_d^{(n)} \end{bmatrix}, \tag{3.3}$$

$$\theta = (\theta_0, \theta_1, \ldots, \theta_d) \tag{3.4}$$

$Y = (y^{(1)}, \ldots, y^{(n)})$ 是测试向量 $x^{(1)}, \ldots, x^{(n)}$ 的预测值。最耗时的操作可以看作是一个向量和一个矩阵的乘法。

训练阶段，目的是为了寻找合适的系数 $\theta = (\theta_0, \ldots, \theta_d)$ 来最小化 m 个训练样例 $\{x^{(1)}, y^{(1)}\}, \ldots, \{x^{(m)}, y^{(m)}\}$ 间的均方误差，其中 $y^{(i)}$ 是训练样例 $x^{(i)}$ 的响应（标签）。这个阶段最常用的解决方法是梯度下降法。简言之，梯度下降法是从 $\theta = (\theta_0, \ldots, \theta_d)$ 的初始值开始，沿着负梯度方向迭代更新。故该阶段耗时的操作是用最新的 θ 来计算 $\theta x^{(i)}$ $(i = 1, \ldots, m)$ 。

3.5.3 访存特征以及分块调节

预测阶段：由于该过程可以看作是一个向量和一个矩阵的乘法，所以局部性分析类似于 DNN 中的前馈计算。简单地说，系数 θ 需要被重用，在 X 中没有重用数据。当线性模型中的系数数量非常大的时候，我们需要平铺展开来减少这些系数的重用距离。我们的实验中，令 d=16384，结果证实了在线性回归的预测阶段，平铺展开是非常有效的，能够减少存储带宽需求的 46.7%。

训练阶段中的梯度下降的主要操作是用最新的 θ 来计算 $y^{(i)} = \theta x^{(i)}$ $(i = 1, \ldots, m)$。这个操作也可写成 $Y = \theta X$ 的形式，将其看作是一个向量和一个矩阵的乘法，那么我们就可以采用类似的优化方式，对向量 X 中的变量进行平铺展开，从而减少存储的需求。

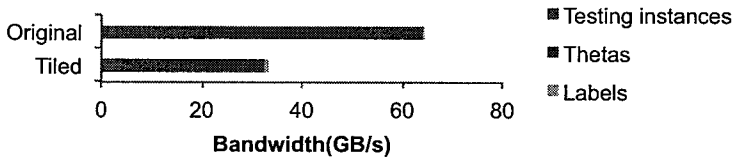

图 3.8: 线性回归预测阶段原始版本和分块版本所需带宽对比 ($N_a = 16384$)

3.6 支持向量机 (Support Vector Machine)

3.6.1 算法简介

支持向量机 (Support Vector Machine，简称 SVM)[72] 是一种有监督学习模型，常作为一种学习算法来分析数据、识别样例，处理分类问题和回归问题。其核心思想将低维空间中无法线性可分的样例映射到高维空间，而后在高维空间中寻找一个超平面，使得样例可以线性可分。当超平面和最近的训练样例间的距离（几何间隔）越大，分类的效果会越好。图3.9即为一个低维空间中线性不可分的样例，使用高斯核函数将样例映射到高维空间中变成线性可分的情况。

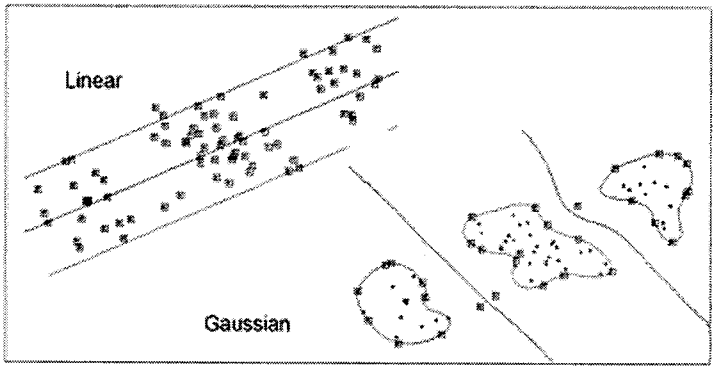

图 3.9: 支持向量机的核函数可将线性不可分的数据映射到高维成为线性可分的数据

具体来说，就是给定一组训练样例，每一个样例都被标记为两类中的一类，而后利用 SVM 训练算法建立 SVM 模型，把样例映射成为某一空间中的点，通过一个非常明显的间隔来将两类样例分开，因而这个间隔是越宽越好的，形

成一个非概率二元线性分类器。预测时，新的样例也被映射到同一个空间中，而后根据之前的样例建立好的间隔位置和宽度，预测新的样例属于哪一类。

3.6.2 核心运算分析

SVM 的想法是在一个高维空间中找到一个把样例分成两类的超平面。若用 x 表示输入样例，y 表示其标签，那么 SVM 的模型的形式可以表示为：

$$y = \sum_{i=1}^{N} \alpha_i y^{(i)} k(x, x^{(i)}) + b \tag{3.5}$$

其中 N 是训练样例的数量，$x^{(i)}$ 和 $y^{(i)}$ 分别是第 i 个训练样例及其标签，α_i 是常数系数，$k(\cdot,\cdot)$ 是核函数（如径向基函数，tanh 函数），有点类似高维空间中的两个样例间的点积运算，b 是一个常数。若 $x^{(i)}$ 不是支持向量，则它对应的系数 α_i 等于 0。那么显然，SVM 也会分为两个阶段，用于通过训练数据构造模型的训练阶段和用训练好的模型来预测每个测试样例的标签的预测阶段。

因为 SVM 的预测阶段相对简单，所以我们先考虑预测阶段，即使用已经构造好的 SVM 模型来预测 N_b 个测试样例的标签。主要过程为输入新的测试样例 x，而后计算与训练样例的每一项间的核函数值，即 $k(x, x^{(i)}$，而后再乘上训练样例对应的系数 α_i 和标签值 $y^{(i)}$，并求出乘积和，加上偏移量即为最终结果。显然，在该过程中，最为耗时的运算是计算测试样例和每个训练向量之间的核函数的值。

SVM 的训练阶段，目标是寻找合适的系数 α_i 来使得几何间距最大化。一种常用的训练算法，序列最小化优化算法（Sequential Minimal Optimization，简称 SMO）[73] 能有效地解决该问题。SMO 算法是一种很快的二次规划优化算法。其主要利用了坐标上升的思路，通过改变一个变量，固定其他变量，来只考虑一个变量的对优化结果的影响。

SVM 训练阶段最耗时部分是计算 $N \times N$ 的核矩阵 K（N 是训练样例的数量），该矩阵用于记录训练样例的所有可能对的核函数值，是一个对称矩阵。令 k_{ij} $(i, j = 1, \ldots, N)$ 表示矩阵 K 中第 i 行第 j 列的数据，即训练样例 $x^{(i)}$ 和

$x^{(j)}$ 的核函数值，也即 $k_{ij} = k(x_i, x_j)$。对于核矩阵的计算，具体包括了向量的点积运算和标量的非线性函数运算。

3.6.3 访存行为分析和分块调节

接上一节的内容，我们首先分析 SVM 的训练阶段，当用 SMO 算法寻找到了合适的系数 α_i 来使得几何间距最大化时，最耗时的步骤是计算用于记录训练样例的所有可能对的核函数值的对称矩阵——核矩阵 K，因为有 N 个训练样例，所以矩阵的大小为 $N \times N$。具体来说，利用公式 $k_{ij} = k(x_i, x_j)$ 计算出训练样例 $x^{(i)}$ 和 $x^{(j)}$ 的核函数的值，保存在核矩阵 K 的第 i 行第 j 列中。比较常用的核函数有径向基函数，tanh 函数等。无论采用哪种核函数，我们可以发现，核矩阵的计算和 k-NN 中距离的计算有相似的局部性性质，只是，对于每对样例来说，核矩阵计算是计算核函数的值，而不是距离。因此，我们重用计算距离时的平铺展开的方式。平铺展开后，核矩阵计算减少了所需的存储带宽的 93.9%。

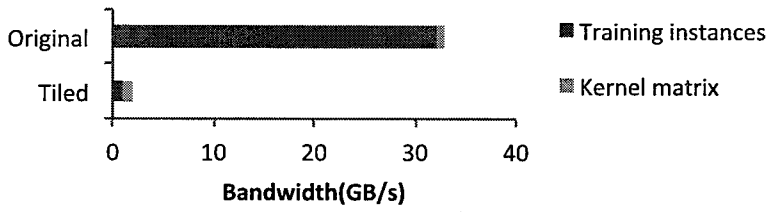

图 3.10: SVM 原始版本和分块版本所需带宽对比

在 SVM 预测阶段，我们已经构造好了一个 SVM 模型，包括 N_a 个支持向量 (支持向量是训练样例中最靠近决策超平面的样例)。我们使用该模型来预测 N_b 个测试样例的标签，并已分析得最为耗时的计算是计算测试样例和每个测试向量之间的核函数的值。类似的，采用和训练阶段的优化方法，对支持向量和测试样例进行平铺展开，减少存储带宽需求。

3.7 分类树算法（Classification Tree）

3.7.1 算法简介

分类树（Classification Tree，简称 CT）是一类有监督学习技术，是数据挖掘中常用的方法之一。其目的是根据训练样例来构造一个树状的分类器，可以预测新的样例的目标值。分类树的内部结点和每个输入变量相关，连接其孩子结点的便表示该输入变量可能的取值。叶子结点表示目标值，即标签，对应的就是从根结点到该叶子结点的路径上的输入变量。较为常用的算法有 CART[74]、C4.5[75]、ID3[76] 等。

3.7.2 核心运算分析

分类树算法由两个阶段组成，通过训练样例来构造树状分类器的训练阶段和预测测试样例标签的预测阶段。

在训练阶段，分类树算法从表示全部训练样例的集合的根结点开始，而后递归地将非叶结点分裂成两个更小的结点。分裂时需按照特定的学习标准进行，不同的分类树算法会有不同的学习标准，例如，CART 使用 gini 系数[74]，ID3 使用信息增益[75]，C4.5 使用信息增益率[76]。然而，对于所有的分类树算法最耗时的操作是计数操作，也是不同学习标准计算求值时都不可或缺的步骤。

在预测阶段，构造好的分类器能够预测每一个测试样例的标签。在这个过程中，每一个测试样例需要沿着根结点走到叶结点，而每个叶结点表示测试样例的标签。因此，计算方面，该阶段不会造成很大的计算负担，但是有一个问题，就是当分类器的尺寸非常大时，缓存可能无法存下整棵树，那么当我们对每个测试样例重新载入整棵树时就可能会有不能接受的开销。

3.7.3 访存特征以及分块调节

该算法中，无论选择哪种学习标准，最耗时且都不可或缺的操作是计数操作，于是我们对该操作的特征进行了分析。通常，计数任务分为三种类型：计算特定特征具有特定值的样例数量（对于离散特征空间），计算特定特征超过阈值的样例数量（对于连续特征空间）和计算具有特定标签值的样例数量。在每一轮计数中，相同训练样例（向量）的相同特征值（向量组成）将会频繁重

用。举例来说，当它包含其中时，按位与操作来比较它和一些候选值，来确定选取该特征（比较是为了选取正确的计数器的条件分配/分支语句）。因为特征值的每一次重用会紧接着上一次使用之后，所以无需平铺展开样例及其特征值。而在不同轮计数中，相同的训练样例也会重用，但是，重用距离会根据数据特征来决定（而不是算法特征），所以不具有确定性。因此，我们无法有一个预先定义好的平铺展开策略来减少相关存储访问。

在预测阶段，当分类器尺寸非常大时，缓存可能无法存下整棵树的问题，我们采用了分而治之的策略，把树分割成了很多子树，每一个都能存到缓存中。当一棵子树存到缓存中后，它能够处理所有未被标记的测试样例。这个策略也可以解释为树的平铺展开，避免了频繁重载整棵树。

3.8 朴素贝叶斯算法（Naive Bayes）

3.8.1 算法简介

朴素贝叶斯（Naive Bayes，简称 NB）算法是基于贝叶斯法则和最大后验概率（Maximum A Posteriori，简称 MAP）的概率型分类器[77]。朴素贝叶斯基于样例特征相互独立这个强假设，即在一个特定特征的值和任何其他参数（或特征）的存在或者不存在不相关。在有监督学习领域，相比于很多概率模型，朴素贝叶斯分类器是非常有效的一种模型。在很多实践应用中，朴素贝叶斯模型中的参数估计使用最大似然估计法。

3.8.2 核心运算分析

朴素贝叶斯算法是基于贝叶斯理论和最大后验概率思想的概率型分类器。由于它基于一个很强的假设，即样例间相互独立，所以在其训练阶段的主要任务是估测训练样例的独立条件概率，而预测阶段则是直接计算样例属于各类的概率，并进行分类。

训练阶段的目标是建立一个概率表，用于估测后验概率 $p(C|F_1,\ldots,F_d)$，其中 C 表示标签（类别索引），F_i $(i=1,\ldots,d)$ 表示第 i 个特征。这里 C 和 F_i 都是随机变量，不是特定的值。通过应用贝叶斯法则和独立性假设 $p(C|F_1,\ldots,F_d)$，可以写成 $(1/Z)p(C)\prod_{i=1}^{d}p(F_i|C)$，其中 Z 是比例因子，不会影响分类结果。训练阶段的核心任务是估测条件概率 $p(F_i|C)$ $(i=1,\ldots,d)$。这

里我们考虑该算法的离散形式，所以所有的条件概率都能通过计算训练数据的频率得到。假设 F_i $(i = 1, \ldots, d)$ 只能从 $\{f_{ij}\}_{j=1}^{a}$ 取值，C 只能从 $\{c_k\}_{k=1}^{b}$ 取值，那么每个条件概率可以写成 $p(F_i = f_{ij}|C = c_k)$，那么总共有 $d \times a \times b$ 个类似的项。为了估算频率，每一项都有一个临时计数器。统计完训练样例的特征和标签，基本也就完成了频率的估测，归一化后便可以得到所有的条件概率。因此，可以说，在这个过程中，最耗时的操作就是计数操作。

预测阶段相对简单，对于每个测试样例 $x = (\hat{f}_1, \ldots, \hat{f}_d)$ ，计算其后验概率 $p(C = c_k|F_1 = \hat{f}_1, \ldots, F_d = \hat{f}_d)$ $(k = 1, \ldots, b)$，然后比较这些后验概率值，值最大的类别即为样例 x 的标签。该过程中，最耗时的操作就是 d 个数字的乘法。

3.8.3 访存特征以及分块调节

训练阶段：在训练一个朴素贝叶斯的分类器时，每个样例的每一维度（特征值）都会取到缓存中一次，但当它们在缓存中时会被重用很多次，例如，采用某一特征前，将它和其他多个候选值进行比较时的多次按位与操作（比较是为了选择正确的临时计数器相加)。因为一个特征值的每一次重用几乎紧挨着下一次的使用，所以没有必要平铺展开样例和它们的特征值。然而，计数过程中，不同候选值对应的临时计数器会被更新。当有太多的条件概率需要估测时，缓存不能够存下所有的临时计数器，为了减少临时计数器的片外访存，可以进行预处理训练样例，即根据他们的标签或候选值进行分组。而且，从片外存储器中获取不同样例的相同维特征，而不是获取一个样例的整体（这样一次能获取更多的样例，相对减少了临时计数器的片外替换次数)。

预测阶段最耗时的操作就是 d 个数字的乘法，没有明显的局部性，因为无法预测一个没见过的样例的特征值，也就不能确定后续乘法所需的数据。

3.9 小结

本章中，我们主要是从体系结构设计者的角度对这七种具有代表性的算法进行了分析。从每种算法的核心运算入手进行分析提取，而后对其核心算法中的主要运算步骤通过平铺展开进行优化。

首先，对于每种算法，我们进行了核心运算分析，如表3.1。我们采用 MNIST 数据集和 UCI 数据集中的“Gas Sensor Array Drift Dataset at Different

表 3.1: 不同算法核心运算分析

Algorithm	Test bench	Core Computation	Proportion
k-NN	Gas	Computing Distance	84.44%
	MNIST	Computing Distance	93.95%
k-Means	Gas	Computing Distance	89.83%
	MNIST	Computing Distance	63.58%
ID3	Gas	Gain	77.91%
NB Train	MNIST	Counting	99.62%
	Gas	Counting	99.62%
NB Test	MNIST	Multiply	100%
	Gas	Multiply	100%
LR Train	Gas	Computing Gradient	100%
LR Test	Gas	Multiply+Add	100%
SVM Train	Gas	POL(+dot)	94.96%
SVM Test	Gas	selecting working set	91.80%
MLP	Gas	Multiply+Add	100%
BP for MLP	Gas	Computing Gradient	97.28%

Concentrations”数据集进行核心算法分析，结果如表3.1，其中“Gas Sensor Array Drift Dataset at Different Concentrations”数据集均被简略地称为 Gas。我们从图中可以看到，对于 k-NN 和 k-Means 的最核心的运算为距离计算，而对于支持向量机、深度神经网络以及线性回归，其最耗时的操作分别为点积运算、计算梯度以及乘法和加法的结合，而这些计算无论是矩阵和向量之间的运算还是矩阵与矩阵之间的运算，均可以转化为类似于点积的运算。而朴素贝叶斯和 ID3 算法的主要任务即为计数。

接着，我们分析了每个算法的访存特征。具体来说，如图3.11, 大部分机器学习算法（包括 k-NN，k-Means，深度学习，线性回归，以及支持向量机等）的访存都体现了“三个柱子”的特征。不同的柱子对应不同的重用距离，不同的重用距离对应不同的访存带宽，对于重用距离短的柱子对应的数据，其所需要的片外访存带宽小。对于重用距离长的柱子对应的数据，其所需要的片外访存带宽大。为了减少机器学习算法对片外访存带宽的需求，可以通过分块调节

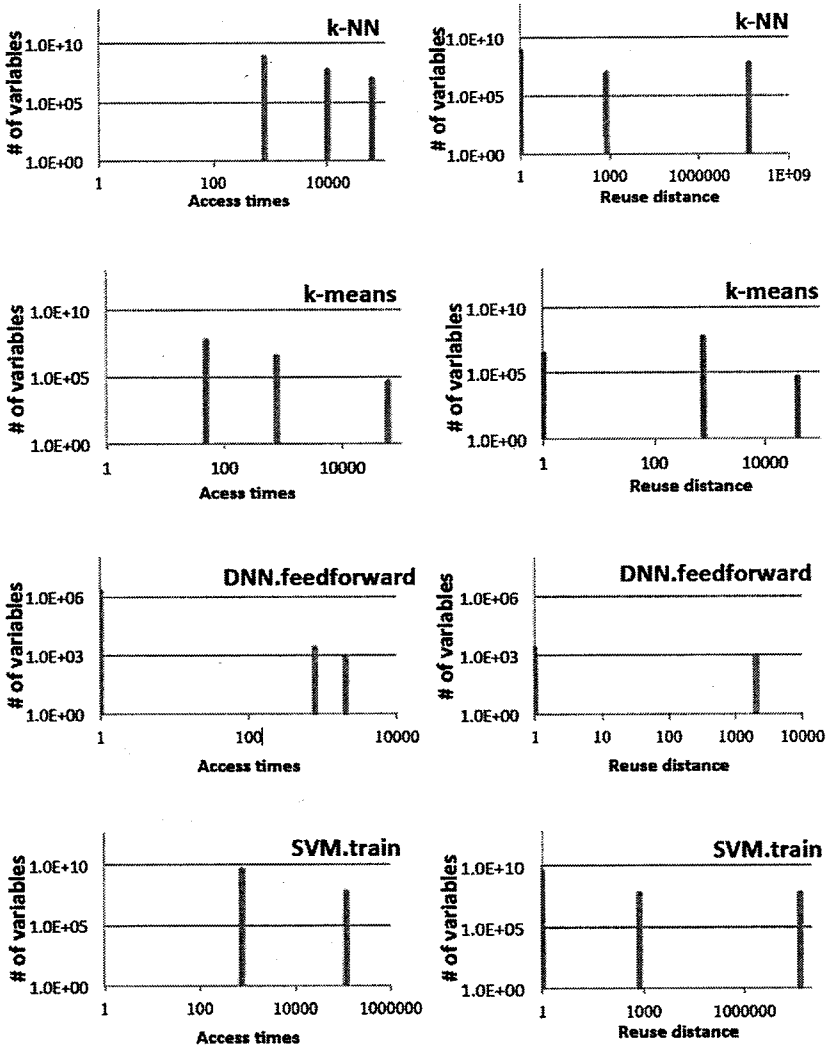

图 3.11: 向量类机器学习方法访存特征

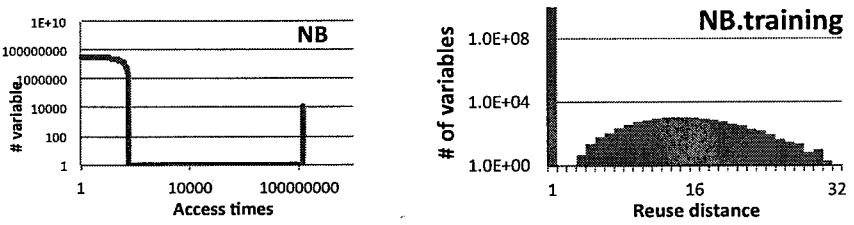

图 3.12: 概率计数类机器学习方法访存特征

的方法，将重用距离长的数据的重用距离变小，从而减少片外访存带宽需求。

而另外一些概率和计数类的机器学习方法，其“三个柱子”特征则没有那么明显。其访存行为与训练和测试数据有关，其现象具有一定随机性，比如，对于朴素贝叶斯，其访存现象如图3.12所示。

如前所述，分块展开主要分为两类运算的分块展开，一类是距离计算的分块调节，如 k-NN，k-Means 的分块展开；另一类是进行矩阵/向量运算 (包括点积和类点积的运算，类点积运算主要是点积运算的结果加一个非线性函数的运算) 的分块展开，其典型代表是深度学习，SVM，线性回归中的运算。

其实，前述几种分块调节的手段可以看成是同一种，并且可以写成统一的形式。我们可以将以上几种核心运算都看成两层循环运算，分块调节的本质是将循环重新分块展开，改成更多层循环，从而保证每层循环的数据重用距离都较小，可以放在片内缓存中。在这里假设我们处理器的每个运算单元可以同时处理 T_m 维的数据，共有 T_f 个运算单元，且片内缓存可以同时存放 T_a 外层循环数据，以及 T_b 个内层循环数据，以及其对应的结果数据。统一的分块调节代码可以改写成图3.13所示。其中，T_a 是外层循环分块调节块大小，T_b 是内层循环分块调节块大小，T_d 是针对高维数据维度分块调节块大小 (对于维度很高的数据，片内缓存连一个向量也无法存下，这时候就需要维度分块调节)。

在图3.13中，T_d 是维度分块大小，这是为了防止有时数据维度太大，单个向量就无法放在片内缓存中。N_a 和 N_b 分别是外层和内层循环总数据大小。

分块调节可以大大的减少片外访存带宽需求，从而避免了访存成为瓶颈。实验表明，分块调节访存优化技术可以将典型的机器学方法的片外访存带宽需求减少了 46% 到 93% 不等。可以解决专用处理器设计中最常见的问题，访存

```
Res(all) = 0; //initialize all result
for (d = 0; d < D/Td; d++){//tiling for dimension
  for (i = 0; i < Na/Ta; i++){//tiling for outer
     loop instance
    for(j = 0; j < Na/Tb; j ++){//tiling for inner
       loop instance
      //instance and dimension tiled block
      for (ii = i*Ta; ii < (i+1)*Ta; ii++){
        //FU inner loop instance tiled block
        for(jj = 0; jj < Tb/Tf; jj++){//tiling for
           FU inner loop instance
          //function unit tiled inner instance
             block
          for(jjj = j*Tb+jj*Tf; jjj < j*Tb+(jj+1)*
             Tf; jjj++){
             for(dd = 0; dd < Td/Tm; dd ++){//
                tiling for MLU dimension
               //MLU dimension tiled block, this
                  for performs in MLU in each
                  cycle
               for(ddd = 0; dd < d*Td+dd*Tm; ddd
                  ++){
                 Res[ii,jjj] += op(x[ii,ddd],y[jjj
                    ,ddd]);
} } } } } } } }
```

图 3.13: 分块调节通用代码

带宽无法满足计算需求（因为专用处理器往往具有强大的峰值计算能力，但受限于访存带宽限制，往往无法完全发挥出来）。

第四章　通用机器学习处理器设计

在这一章中，我们主要将介绍本文所设计的机器学习处理器 -PuDianNao 的结构。首先，我们介绍了现有的一些相关研究工作，并指出本文的工作与他们的区别。接着，我们重点介绍了本文设计的机器学习处理器核的结构。主要是从整体结构、存储层次以及控制与指令的设计三部分进行描述。基于第二章中的分析可知，依据算法的核心运算可以将算法分为三类，第一类是 k-NN 和 k-Means，常用的是距离的计算；第二类是深度神经网络、线性回归算法和支持向量机，常见的是矩阵/向量运算，具体包括了向量点积运算，以及非线性函数运算；第三类是朴素贝叶斯和分类树算法，以计数操作为主。因而，在本章后半部分对于该机器学习处理器对实际算法的应用时，也是主要分为上述三种情况进行描述的。

4.1　现有研究

由于机器学习越来越重要，学术界越来越多关于使用专用硬件来加速机器学习应用的研究。早在 2007 年，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）[19,78]，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[20]。Hussain 等人为 k-Means 算法设计了一个专用的 FPGA 加速器[21]，并将专用加速器和 CPU 以及 GPU 进行了比较[22]；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[23,23,79]。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[24]。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[25] 和 Markos[26] 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[27]。在神经网络方面，则有更多的工作被提出[32,80,81]，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[28] 以及大规模的卷积神经网络的在 FPGA 上的实现[29]，Farabet 还研究了卷积神经网络在机器视觉的具体应用[30]。然而，这些加速器都有一个共同的特点，就是

虽然相比于通用处理器（CPU 和 GPU），它们在能源效率/性能上有明显的提升，但是他们都只针对单一的技术或技术类。一旦学习任务转变了，例如，分类问题转化为聚类问题，或者用户想使用另一种机器学习技术，这些加速器就可能没有了用武之地。

最近，Majumdar 等人设计了一款叫 MAPLE 的机器学习加速器，它能加速 5 种机器学习算法（包括神经网络、SVM 和 k-Means）中的矩阵/向量操作和排序操作[35,36]。然而，仍有许多具有广泛应用的机器学习技术（如决策树和朴素贝叶斯），由于其主要计算元素既不是矩阵也不是向量操作（如计数、线性插值），这些算法很难被被 MAPLE 支持。并且，相比本文的 PuDianNao，MAPLE 并没有致力挖掘机器学习算法访存特征，受限于访存墙，其性能很难进一步提升。

本章设计的机器学习处理器（PuDianNao）与上述加速器相比，更具有普遍性，在设计之初就考虑了多种具有代表性的机器学习技术（包括 k 均值聚类算法（k-Means）、k 近邻算法（k-Nearest Neighbors，简称 k-NN）、朴素贝叶斯算法（Naive Bayes，简称 NB）、分类树算法（Classification Tree，简称 CT）、支持向量机（Support Vector Machine，简称 SVM）、线性回归算法（Linear Regression，简称 LR）和深度神经网络算法（Deep Neural Network，简称 DNN））。另外，本文在设计该机器学习处理器之前，详细分析了每种算法的核心运算和特征和访存特真，从而在提高核心运算单元的运算效率同时，也降低了片外访存的存储带宽需求。虽然本机器学习处理器还没能支持所有具有代表性的机器学习技术，但是设计思路和方法对未来设计更为通用的机器学习加速器提供了一定的帮助。我们日后的工作也是改进本机器学习处理器的结构，以扩大其适用范围，从而支持更多的机器学习技术。

4.2 PuDianNao 整体结构设计

基于前一章对各个算法的运算特征和访存特征的分析，我们设计了一个名为 PuDianNao 的机器学习处理器。在这里，普的意思是能够支持多种机器学习算法，电脑的意思是能够思考的计算机，即机器学习处理器。

本文中的机器学习处理器的结构设计主要包括两部分，运算单元设计和存储层次设计，分别对应于机器学习的运算特征和结构特征。其中运算单元的设

计的出发点是高效实现机器学习最频繁的运算操作，而存储层次的设计则主要根据访存特征，提高各机器学习算法中数据的片内重用，减少片外访存带宽的需求，从而充分发挥运算单元的计算能力，避免片外访存成为性能瓶颈。

基于前一章对各种机器学习的运算特征和访存特征的分析，本文设计的机器学习处理器 –PuDianNao 的整体结构见图4.1，其主要包括运算部件、存储部件和控制部件三大部分。运算部分是由十六个运算单元 (Function Unit，简称 FU) 组成，每个运算单元包含一个机器学习单元 (Machine Learning Unit，简称 MLU) 和一个算术逻辑单元 (Arithmetic and Logic Unit，简称 ALU)。存储部分专指数据存储，包括三个数据缓存 (HotBuf，ColdBuf 和 OutputBuf)，分别存储不同使用频率的数据，存储的数据内容会根据算法的不同进行相应的调整。它们连接同一个 DMA(全称为 Direct Memory Access，译为直接内存存取部件)，用于传输数据。控制部分包括一个指令缓存 (InstBuf) 和一个控制模块 (Control Module，简称 CM)。

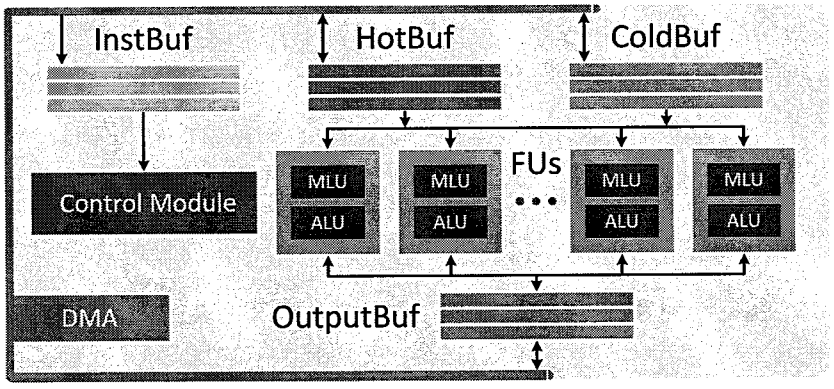

图 4.1: PuDianNao 整体结构

4.3 PuDianNao 运算单元设计

运算单元是机器学习处理器的基础执行单元，主要的计算操作都通过运算单元来实现，其他一些比较耗时的操作，如 log，sigmoid 等函数的运算也是通过其执行。为了设计高效的运算单元，正如前一章所言，机器学习的核心运算主要包括：

1. **向量点积**，支持向量机，深度学习，线性回归等算法都有大量的向量点积

运算，向量点积运算可以拆解成两部分，第一部分是乘法，第二部分是加法树。

2. **向量距离**，k-NN，k-Means 等算法都有大量的向量距离运算，向量距离运算可以拆解成三部分，第一部分是减法，第二部分是乘法，第三部分是加法树。其中后两部分可以和向量点积重用。
3. **计数**，决策树，朴素贝叶斯等算法都有大量的计数操作，计数操作主要是加 1 操作。
4. **排序**，k-NN，k-Means 等算法都有排序操作，且比较耗时，由于这两种算法的排序都不是全排序，因此，可以通过硬件进行加速。
5. **非线性函数**，支持向量机，深度学习等机器学习算法向量点积后都有一个非线性函数运算，使用普通的运算单元完成非线性函数需要通过迭代实现，非常耗时，本文的机器学习处理器使用专门的硬件对此类非线性函数予以支持。

为了提高性能，PuDianNao 中共集成了 16 个运算单元，其中每个运算单元都是由两部分组成：机器学习单元（Machine Learning Unit，简称 MLU）和算术逻辑单元（Arithmetic and Logic Unit，简称 ALU）。其中 MLU 主要是用于加速前述在各种机器学习方法大量出现的核心运算，而 ALU 是用于计算一些零碎的运算。

4.3.1 机器学习单元（MLU）

机器学习单元（MLU）是用来支持各种机器学习方法中共有的核心运算，具体包括：点积（LR,SVM,DNN）、距离计算（k-NN，k-Means），计数（决策树和朴素贝叶斯），排序（k-NN，k-Means），非线性函数计算（如 sigmoid 和 tanh）等等。根据前面的分析，所设计的 MLU 结构如图4.2所示，MLU 被分成了 6 个流水线阶段（计数器阶段，加法器阶段，乘法器阶段，加法树阶段，累加器阶段和 Misc 阶段）。

计数器阶段，每一对输入数据被送至按位与单元或者比较单元中进行比较（需要比较是因为大部分计数都是先需要判断和某值是不是相等），结果值送至累加器中相加（实现 +1 操作）。在此之后，累加器的结果被直接送到结果缓存中而不是下一个阶段。这个阶段主要用于加速朴素贝叶斯和分类树的计数操作。对于没有计数操作的机器学习算法，计数阶段会被直接略过。

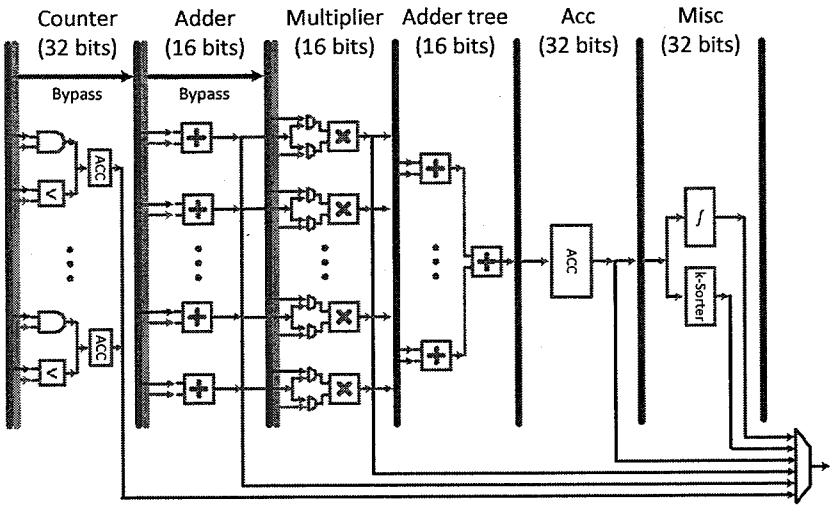

图 4.2: 机器学习单元 MLU 实现

加法器阶段是用来进行向量加法或减法的，这是在机器学习技术中非常普遍的一个运算（比如距离中的差或绝对值)。不过，加法器阶段也可以略过，譬如在深度学习中，计算结果被直接送至下一个阶段或者输出口。

乘法器阶段是用来计算向量乘法的，输入可以使前面阶段的结果或者直接从输入缓存中读取的数据。乘法器的结果可以被送至下一个阶段或者输出口。

加法树阶段和累加器阶段是用来把之前所有乘法器的结果加起来。加法树和乘法器一起完成点积运算，在机器学习技术中广泛应用的运算（LR，SVM，DNN)。当数据维数大于加法树的大小时，这个阶段的结果就是部分和，在累加器阶段再进行累加。在获得了点积或距离的最终结果之后，累加器阶段可以被送至下一个阶段或者输出口。

Misc 阶段用于完成距离或点积运算结果之后的一些其他运算，具体为非线性函数或者排序，在硬件上实现了两个模块，线性插值模块和最小 k 排序 (k-sorter) 模块。线性插值模块是用来近似计算机器学习技术中的非线性函数的值，例如神经网络中的 sigmoid 和 tanh。不同的非线性函数对应着不同的插值表，插值表可以根据不同算法所需要的非线性函数不同由程序员进行初始化。最小 k 排序模块是用来寻找累加器阶段输出值中最小的 k 个值。这是 k-Means 和 k-NN 中常见的操作。线性插值模块和最小 k 排序模块的结果分别

表 4.1: 不同位宽数据的训练精度比较（结果归一化到全部为 32 位的版本）

机器学习算法	精度	
	16 位版本	16 位 32 位混合版本
SVM	37.7%	98.2%
k-NN	99.9%	100%
k-Means	93.9%	100.1%
LR	78.2%	99.0%
DNN	99.4%	100.1%

被送到输出口。最后，输出口将会在这六级流水的结果中选择一个作为 MLU 模块的最终结果输出来。

需要说明的是，我们为了减少加速器面积/功耗的消耗，在加法器、乘法器、加法树阶段都设置了 16 位浮点运算单元，而在剩下的三个阶段（计数器阶段，累加器阶段和 Misc 阶段）依然采用 32 位运算单元，以此来避免潜在的溢出情况。我们通过实验验证了上述硬件优化对 k-NN，k-Means，SVM，LR 和 MLP 等多种机器学习方法的精度影响有限，其误差损失可忽略不计，详见表4.1。在该表中，我们分三种情况讨论精度，第一种情况是所有阶段都是用 32 位浮点运算单元，第二种情况是所有阶段都使用 16 位浮点运算单元，第三种情况是对不同阶段采用 32 位和 16 位相结合的浮点运算单元，也即我们最终采取的方法。在该表中，我们将 32 位浮点运算单元的精度设为 1，并按比例得到 16 位浮点运算单元和 16 位与 32 位结合的浮点运算单元的相对精度（即相对于 32 位浮点运算单元的精度之比）。从图中，我们可以看到，降低浮点运算单元的位数对于错误率的影响并不大，相比之下，16 位的差距较大的一个原因是在这些运算中产生了溢出的现象，因而，为了获得更精准的加速结果，我们的机器学习处理器选择了折中的方法，即 32 位和 16 位结合的浮点运算单元，一方面可以有效降低处理器的面积和功耗（根据我们的实验，16 位浮点运算的乘法器的面积仅为 32 位的 20.07%）；另一方面能够有效避免潜在的溢出的情况影响处理器运行的结果。这里我们没有考虑朴素贝叶斯和分类树技术，因为他们不包含加法器、乘法器和加法树阶段，也不会影响到 16 位运算单元。

4.3.2 算术逻辑单元（ALU）

算术逻辑单元（Arithmetic and Logic Unit，简称 ALU）用于处理机器学习技术中偶尔出现但 MLU 不支持的其他运算，如除法运算、条件转移等。这些运算虽然在机器学习技术中并不多，但是如果用主机上的通用核来执行它们依然会造成长距离的数据移动和同步开销。因此，我们在每个功能单元中添加了一个 ALU，包括一个加法器、乘法器、除法器和一个能够支持 16 位到 32 位浮点转换和 32 位到 16 位的浮点转换的转换器。该单元还可以进行对数运算（log）的泰勒展开来近似计算 log 的值，以满足某些技术（比如 ID3 决策树中熵的计算）中的 log 运算。

4.4 PuDianNao 存储层次设计

本文的机器学习处理器 PuDianNao 设计了 3 个片上数据缓存：HotBuf (8KB)，ColdBuf (16KB) 和 OutputBuf (8KB)。HotBuf，存储输入数据，即具有最短重用距离的数据，ColdBuf 存放相对较长重用距离的输入数据，OutputBuf 存储输出数据或者临时结果。

这样设计的原因有二：一方面，由前一章小结中图3.11我们可以看出，在利用了机器学习技术的局部性进行分块展开之后，变量的重用距离可以分为两类或三类，因此，机器学习处理器中设计了 3 个片上数据缓存。

另一方面，考虑到数据读取宽度，我们选择使用多个缓存。这里我们用 k-Means 为例，计算展开后的距离计算时，假设一个 MLU 一次能够处理一个测试样例的 f 个特征（维度）和一个重心的 f 个特征。加速器共有 u 个 MLU，那么在一个节拍中能够计算一个重心和 u 个测试样例的平方距离。而重心是 16 位浮点数组成的向量，测试样例是 16 位无符号浮点数组成的向量。由于一个重心对应多个样例，重心和样例的读取宽度不同，因此我们设置了两个分开的缓存来减少不同的宽度带来开销。HotBuf 存储重心，读取宽度就是 $f \times 16$ 位浮点数；ColdBuf 存储测试样例，读取宽度是 $u \times f \times 16$ 位浮点数。

前述三个缓存连接同一个 DMA。另外，我们使用单端口的 SRAM 来构造 HotBuf 和 ColdBuf，因为这两个缓存只需要读出数据；而运算单元既可以读又可以写 OutputBuf，所以我们使用双端口 SRAM 来构造 OutputBuf。这样做，明显降低了面积和功耗的开销，具体见图4.3。

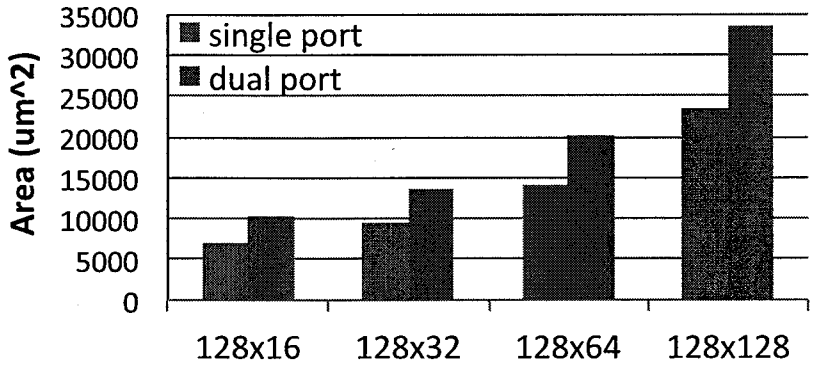

图 4.3: 单端口和双端口 RAM 面积比较

4.5 控制模块与指令设计

控制模块实现的最简单方法是对这七种机器学习技术进行硬编码（类似专用集成电路（ASIC）的方式)。但是，如果用户想要使用另一种机器学习技术，只要这个技术和硬编码的机器学习技术相比有一点不同，我们就不得不提供用户一个新的加速器。为了提高 PuDianNao 的灵活性，有效避免算法的局部改动导致本处理器无法使用，我们设计了控制模块来控制运算单元。由于所有的运算单元同时执行同样的操作，所以只需要一个控制模块即可。该模块从指令缓存（InstBuf）中获取指令并解码，而后将信号传送到所有的运算单元中，并控制运算单元进行计算（包括输入输出数据来源，需要经过那些流水级)。PuDianNao 的控制模块采用的指令格式如表4.2所示，其类似于通用处理

表 4.2: 通用机器学习处理器指令格式

CM	HotBuf				ColdBuf				OutputBuf								FU						
Inst Name	READ OP	READ ADDR	READ STRIDE	READ ITER	READ OP	READ ADDR	READ STRIDE	READ ITER	READ OP	WRITE OP	READ ADDR	WRITE ADDR	READ STRIDE	WRITE STRIDE	READ ITER	WRITE ITER	MLU-1 OP	MLU-2 OP	MLU-3 OP	MLU-4 OP	MLU-5 OP	MLU-6 OP	ALU OP

器领域的超长指令字 (Very Long Instruction Word，简称 VLIW)，指令主要包括 CM 域 (表示指令类型和指令名字)，HotBuf，ColdBuf，OutputBuf 域 (这三个域分别表示指令输入输出所需要的数据地址和地址范围)，以及 FU 域 (表示指令需要经过哪些流水级运算)。

4.6 编程方法

由于 PuDianNao 采用了类似于通用处理器中的 VLIW 指令集，因此，只要为其开发编译器，理论其可以支持使用高级语言 (如 C，C++ 等) 直接编程。然而，受限于时间，本文暂时没有为 PuDianNao 设计专用的编译器，因此，现阶段主要采用专家直接使用汇编编程的模式。或者，普通用户调用专家写好的汇编库的编程模式。下面，我们讲通过一个简单的例子说明如何使用 PuDianNao 的指令进行机器学习方法的编程。

在表4.3我们提供了一个 k-Means 代码的示例。在这个例子中，每一个样例的特征数量 $f = 16$，重心数量 $k = 1024$，测试样例数 $N = 65536$。重心存储在 HotBuf(8KB) 中，测试样例存储在 ColdBuf (16KB) 中。为了能够隐藏在计算后的 DMA 内存访问时间，我么采用乒乓的方式来使用 HotBuf 和 ColdBuf。具体来说，第一条指令是加速器通过 DMA 从内存中 LOAD 128 个重心 (4KB) 和 256 个测试样例 (8KB)，分别占用了一半的 HotBuf 和 ColdBuf。DMA 完成后，处理器将计算载入进来的重心和测试样例间的距离。同时，另外的 256 个测试样例被载入到了另一半的 ColdBuf。当第一条指令的计算和 DMA 都完成后，开始执行第二条指令。在第二条指令中，第一条指令载入的 128 个重心会被重用，即从 HotBuf 中 READ (而不用从内存中 LOAD)。当这 128 个重心和所有 65536 个测试样例间的距离计算处理完 (在第 256 条指令之后)，新的 128 个重心组成的块被载入 (在第 257 条指令)。这个过程会重复到所有重心和测试向量间的距离计算完毕。

表 4.3: k-Means 代码示例 ($f = 16$, $k = 1024$, $N = 65536$)

CM	HotBuf				ColdBuf				OutputBuf								FU						
k-means	LOAD	0	16	128	LOAD	16384	256	16	NULL	STORE	NULL	1064960	NULL	16	NULL	16	NULL	SUB	MULT	ADD	NULL	SORT	NULL
k-means	READ	0	16	128	LOAD	20480	256	16	NULL	STORE	NULL	1065216	NULL	16	NULL	16	NULL	SUB	MULT	ADD	NULL	SORT	NULL
k-means	READ	0	16	128	LOAD	24576	256	16	NULL	STORE	NULL	1065472	NULL	16	NULL	16	NULL	SUB	MULT	ADD	NULL	SORT	NULL
.....																							
k-means	LOAD	2048	16	128	LOAD	16384	256	16	LOAD	STORE	1064960	1064960	16	16	16	16	NULL	SUB	MULT	ADD	NULL	SORT	NULL
.....																							

4.7 算法映射

本节将介绍不同机器学习方法如何映射到 PuDianNao 中，具体包括不同算法如何使用 PuDianNao 的运算单元以及片内缓存。

4.7.1 k 均值聚类（k-Means）和 k 近邻算法（k-NN）

k-NN，k-Means 中常用的距离的计算，该运算可以分为三个阶段，两个输入向量的相关维度的减法，所得差值的平方，以及平方之和。对于 k 均值聚类算法，为了避免读取的数据的宽度不同，HotBuf 中存储 k 个重心的数据，ColdBuf 存储待聚类样例数据。对于 k 近邻算法，HotBuf 中存储待分类点的数据，ColdBuf 存储参考样例数据。

k-NN，k-Means 在 PuDianNao 中计算过程位，PuDianNao 将数据读入 MLU 中，利用加法器阶段进行向量加法，乘法器阶段进行向量乘法，得到的数据通过加法树累加乘法器输出的结果，得到部分维度的平方和。再输入到累加器阶段中得到两个样例的所有维度的平方和，即距离值。最后在 Misc 阶段选出距离值中最小的 k 个，输出即为所求结果，输入值 OutputBuf 中保存。

4.7.2 朴素贝叶斯算法（NB）和分类树算法（CT）

朴素贝叶斯和分类树算法中常见的计数运算，主要包含两个阶段：比较输入特征值和目标值，给计数寄存器加 1 或 0。所以，加法器首先在计数阶段进行按位与操作或者利用比较器进行比较，输出值利用累加器相加后直接送到输出缓存中。而后，利用 ALU 进行其他操作，如在 ID3 算法（分类树算法的一种）中需要计算的 log 运算（具体使用泰勒展开），在朴素贝叶斯算法中需要进行的除法运算等等。

4.7.3 支持向量机（SVM）、线性回归算法（LR）和深度神经网络算法（DNN）

深度神经网络、线性回归算法和支持向量机中常见的矩阵/向量运算，该运算可以分为两个阶段：两个输入向量的相关维度的乘法和乘积间的求和。这个过程类似于点积运算，就是利用 MLU 中的加法树和乘法器阶段完成的。当数据维数过大时，还需要利用累加阶段进行累加。结果若不需要其他操作，直接输出即可，否则，输入到 Misc 中进行线性插值，如 SVM 中使用 sigmoid 核函数的情况和神经网络中使用 tanh 函数时的情况等。

4.8 性能评估

4.8.1 评估方式

我们使用了 C 模拟器和 Verilog+EDA 工具来对 PuDianNao 的进行性能评估和功耗评估。

本文中 Verilog 实现使用了台积电（TSMC）65 纳米 GP 工艺，采用标准阈值电压（Standard VT），并通过 Synopsys 公司的 Design Compiler 工具进行了综合，使用 Synopsys ICC Compiler 进行布局和布线的。同时，我们使用了 Synopsys VCS 进行设计的模拟和确认。基于 VCS 仿真的 Value Change Dump (VCD) 文件，使用了 Synopsys Prime-Tame PX 来估测功耗。

由于 VCS 的模拟速度较慢，在 PuDianNao 的 Verilog 实现上运行一个大规模数据集是非常消耗时间的。为了解决这一问题，我们为 PuDianNao 一个节拍精确的 C 模拟器，并且在小规模数据集上将它和 Verilog 设计仔细进行校

表 4.4: 本文所用的测试基准程序和数据集大小

ML technique	Dataset	Problem Size
k-NN	MNIST	60000 reference instances, 10000 testing instance 784 features, k=20
k-Means	MNIST	60000 instances, 784 features, k=10
DNN	MNIST	data size same with k-NN, L1=784,L2=L3=L4=L5=4096,L6=10
LR	MNIST	data size same with k-NN
SVM	MNIST	data size same with k-NN
NB	UCI Nursery	12960 instances, 8 features, 5 classes
Classification Tree(ID3)	UCI Covertype	522000 training instances, 59012 testing instances

对。这个模拟器允许的存储带宽为 250GB/s，可用来估测 PuDianNao 在大规模数据集上的性能。

我们用 7 种机器学习技术对 PuDianNao 和基准处理器进行了评估，包括 k-NN，k-Means，深度神经网络，线性回归，支持向量机，朴素贝叶斯，和分类树（具体使用了 ID3 算法)。我们使用 MNIST[82] 和 UCI[83] 作为基准数据集，具体数据集的选用和规模详见表4.4。

由于 GPU 相比于 SIMD CPU 在速度上的优势，所以在工业上 GPU 广泛应用于支持机器学习技术。正因如此，我们选取了一款高端的 GPU 计算卡(具体型号为 NVIDIA K20M, 峰值性能 3.52 TFlops, 5GB GDDR5, 208GB/s 访存带宽, 28nm 工艺, 支持 CUDA SDK5.5) 作为基准。为了便于验证这款 GPU 基准在机器学习应用方面的性能，我们将它和一款 256 位 SIMD 的 CPU(Intel Xeon E5-4620 Sandy Bridge-EP, 2.2GHz, 1TB 内存，便用 gcc 4.6 编译，编译选项为“-O3 -ftree-vectorize -march=native”) 进行对比，其结果见图4.4。如图所示，我们用来作为基准的 GPU 比 SIMD CPU 设备平均提速 17.74x。这个结果符合了两个最近的研究结果，即对于机器学习的应用，GPU 的性能比

SIMD CPU 的性能加速 15x-49x[84] 和 10x-60x[85] 。

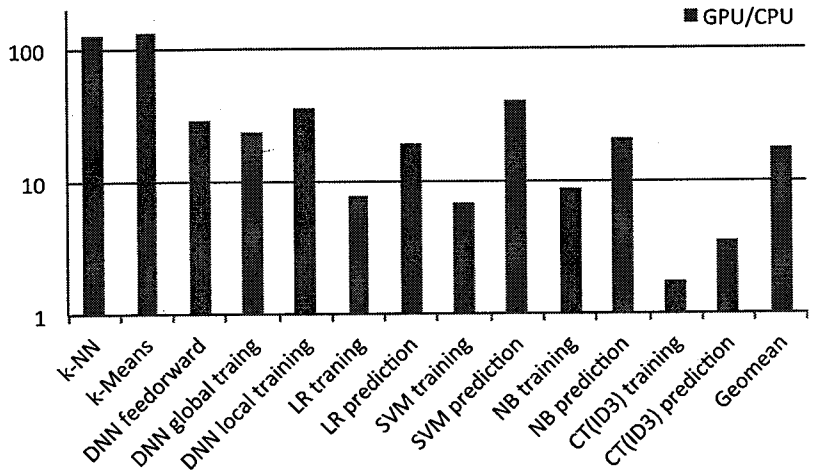

图 4.4: GPU 和支持 SIMD 的 CPU 的性能比较

4.8.2 评估结果

4.8.2.1 芯片参数

本文的 PuDianNao 包含 16 个 MLU，每个 MLU 在每个周期中能够处理 16 个样例特征（维度）。每个 MLU 包含 16+16+15+1+1=49 个加法器，分别来自于计数器阶段、加法器阶段、加法树阶段、累加器阶段和 Misc 阶段。每个 MLU 包含 16+1=17 个乘法器，分别来自乘法器阶段和 MISC 阶段。因此，PuDianNao 能够在 1GHZ 频率下达到 $16 \times (49 + 17) \times 1 = 1056$ Gops 的峰值性能，接近于一个高端 GPU 的性能。

然而，PuDianNao 的面积和功耗都显著比一个高端 GPU 小两个数量级。根据 EDA 工具 DC 和 ICC 报告，PuDianNao 的总面积是 $3.51mm^2$，总功耗是 596 mW。PuDianNao 关键路径延迟是 0.99ns，这说明 PuDianNao 能在 1GHz 频率下工作。具体的面积/功耗值见表4.5。在 PuDianNao 中，片上缓存消耗的面积和功耗分别占总面积和总功耗的 62.64% 和 31.37%。所有的模块中，面积占用最多的部分是 ColdBuf（32.22%）。所有 16 个功能单元（FU）占用了 19.38% 的面积和 35.57% 的能耗。具体的布局见图4.5。其中 CM, FU, HB, CB,

和 OB 分别表示控制模块（Control Module），运算功能单元（Functional Unit），热缓存（HotBuf），冷缓存（ColdBuf），以及输出缓存（OutputBuf）。

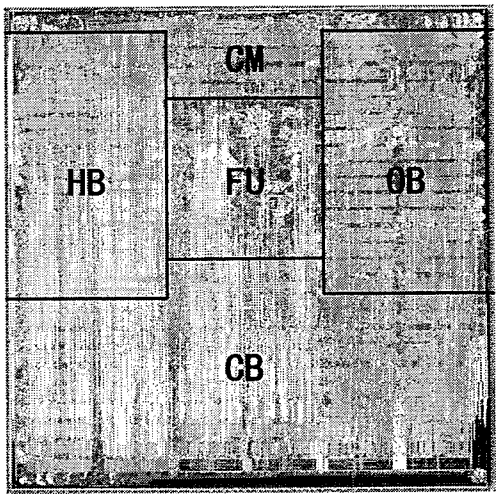

图 4.5: PuDianNao 的版图

表 4.5: PuDianNao 基本参数

Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,513,437		596		0.99
Combinational	771,943	(21.97%)	173	(29.02%)	
On-chip buffers	2,201,138	(62.64%)	187	(31.37%)	
Registers	200,196	(14.23%)	86	(16.10%)	
Clock network	40,154	(1.14%)	143	(23.99%)	
Function Units	681,012	(19.38%)	117	(35.57%)	
ColdBuf	1,167,232	(33.22%)	78	(16.44%)	
HotBuf	578,829	(16.47%)	47	(9.56%)	
OutputBuf	586,361	(16.68%)	51	(10.23%)	
Control Module	481,737	(13.71%)	127	(21.30%)	
Other	18,266	(0.52%)	41	(0.06%)	

4.8.2.2 性能和功耗

我们将 PuDianNao 的性能和作为基准的 K20 GPU 进行了比较，比较内容包括 7 种机器学习方法，除 k-NN 和 k-Means 外，每种方法分训练和测试两个阶段（其中深度学习 MLP 训练分为局部训练和全局训练），因此共 13 个阶段。如图4.6所示，13 个阶段中，有 6 个阶段，PuDianNao 表现都比 GPU 要好，而且 PuDianNao 的平均加速是 GPU 的 1.20 倍。最大加速比是在 SVM 预测阶段（具体为 2.92 倍），一个重要的原因是 PuDianNao 提供了专用的线性插值功能，能高效计算核函数。最差的加速是 NB 的预测阶段（具体为 0.37 倍）。这个阶段为了获得全部的后验概率，需要不同数字间频繁的乘法（条件概率）。PuDianNao 不像 GPU，没有一个大的寄存器文件，因此不得不在功能单元和片上缓存间频繁移动数据，导致明显的性能损失。相比之下，NB 的另一个阶段（NB 的训练阶段）没有那么多类似的操作，所以 PuDianNao 在这个阶段相比于 GPU 达到了 2.22 倍的加速。

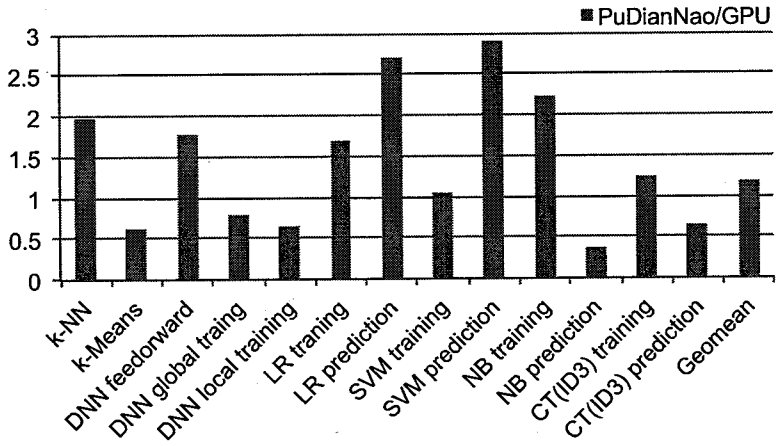

图 4.6: PuDianNao 和 K20 显卡性能比较

功耗方面，如图4.7所示，PuDianNao 平均比 K20 GPU 减少了 128.41 倍的功耗（也即 PuDianNao 功耗只有 K20 GPU 1/128 不到），这是因为 PuDianNao 根据不同机器学习算法的运算特征和访存特征设计了专用的运算单元和专用的片内缓存。PuDianNao 能耗减少最大的是 k-NN（262.20 倍），主要是因为 PuDianNao 能有效支持排序操作，而排序是 k-NN 中频繁且耗时的操作（我

们的实验是每次在 60000 个参考样例中寻找最合适的 20 个样例)。相比之下，GPU 用它通用的功能单元进行排序就会消耗比较显著的能量。能耗减少最小的是 CT (ID3) 预测阶段 (50.32 倍)，这是因为决策树的数据重用性差，需要 PuDianNao 频繁使用 DMA 来将数据在片内缓存和主存之间搬运。

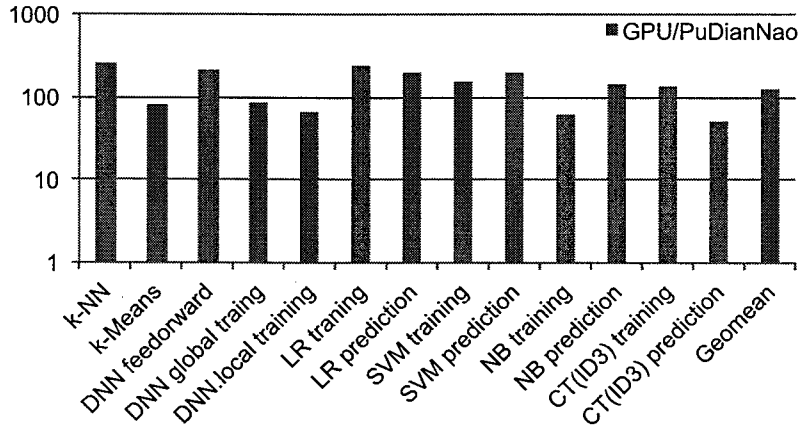

图 4.7: PuDianNao 和 K20 显卡能耗比较

4.9 小结

在本章中，我们介绍了支持多种典型机器学习场景 (如分类，聚类，回归)，同时也支持多种常见机器学习方法 (如 k-NN，k-Means，深度学习，支持向量机，朴素贝叶斯，分类树等等) 的机器学习处理器 -PuDianNao 的结构。并且，通过 EDA 工具对处理器的性能和功耗进行了评估，评估结果表明，相比高端图形处理器 Nvidia K20 GPU，本文的机器学习处理器取得了 1.20 倍的加速比，却只消耗了 GPU 1/128.41 的能耗。相比传统针对单一算法的机器学习处理器，本文的处理器更加通用，可以适应更多的数据和算法的变化。因为 PuDianNao 的设计的思想是分析多种机器学习算法，抽取共性核心运算和共性访存特征，以灵活的指令的形式为所有机器学习方法的共性操作提供计算加速和访存加速。

第五章　一种无共享信息的高速缓存

随着机器学习的规模进一步扩大，受限于工艺和主频，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用，因此，急需提出一种高效的多核并行结构来进行大规模高性能的机器学习。具体来说，有两方面的原因：一方面，随着大规模集成电路工艺的发展，集成电路的密度越来越大；另一方面，现有技术已经很难再进一步提升处理器的主频。这两方面的原因导致多核乃至众核处理器越来越成为工业界的主流。至今为止，英特尔（Intel）公司 2012 年推出了 50 核 Phi 众核处理器[86]，AMD 公司 2011 年推出了 16 核的 Opteron 6200 处理器[44]，ARM 公司于 2012 年推出了 64 核的 Centip3De 处理器[45]。随着工艺的进一步发展，未来的处理器甚至能在片内集成上百甚至上千个处理器核心。

片内多核处理器（Chip Multi-Processor，简称 CMP）往往具有复杂的高速缓存（Cache）层次，具体来说，其高速缓存通常由一个大容量的最后一级高速缓存（Last-level Cache，简称 LLC）以及第一级高速缓存（L1 Cache，简称 L1C）组成。LLC 一般是所有核心共享的，而 L1C 则是每个核心私有的，一般来说 L1C 上的数据是 LLC 上数据的子集的备份。各级 Cache 之间的数据一致性是通过高速缓存一致性协议来保证的。通常来说，有两类高速缓存一致性协议：基于广播的一致性协议[47] 和基于目录的一致性协议[48]。在基于广播的一致性协议中，每次独占写操作（RdEx）的请求会广播给所有处理器核心（无论其是否有该块的备份），从而通知所有核心核心无效其私有备份（假如有）。随着处理器核心数目的增加，广播需要的成本越来越高，因此，无论是学术界还是工业界，都倾向于使用基于目录的一致性协议[6][7]。基于目录的一致性协议基本思想是使用一个专门的目录用来记录每个 LLC 缓存块在 L1C 的备份情况。一般来说，每个块的目录都是一个位宽等于处理器核数的位向量，当某个处理器核（L1C）有该 LLC 缓存块备份时，则对应位的位向量为 1。在基于目录的一致性协议中，当某个核心发生独占写（RdEx）操作时，LLC 会根据目录信息，向对应目录位向量为 1 的（即含有该块备份的）核心的 L1C 发去无效的信息，对应 L1C 收到无效信息后会将其私有备份无效。通过这种方法，所有

独占写操作都能马上被所有核心全局观察到，从而保证了高速缓存的一致性。

然而，基于目录的缓存一致性协议有两个缺陷。第一个缺陷是目录位向量的位宽随着核心数目的增加而急剧增加，因而会消耗大量而宝贵的片上 RAM 面积和功耗。比如，对于一个 256 核的 CMP 处理器，假如其缓存块大小（Cache Block Size）是 256 位，则每个缓存块的目录也是 256 位，因此，其目录就占了所有高速缓存 RAM 的一半，浪费了大量的 RAM 面积和芯片功耗。第二个缺陷是，基于目录的缓存一致性协议会导致大量的写无效消息和写回应消息，这些消息不但会大大增加片上网络的功耗，而且可能造成片上网络拥堵，从而降低性能。

在本文中，我们发现，在实际大量应用的多核/众核系统中，可以不存储共享缓存块的共享信息。因为现在大部分（假如不是全部）的多核/众核系统，采用的一致性协议都是弱一致性或者更松的一致性协议，在弱一致性协议中，某个核心的写操作允许不立即被其他核心观察到，直到同步点才被其他核心观察到。基于这个发现，我们提出了一种不用记录共享信息的，也即无需目录的一致性协议（Directoryless Shared Cache, 简称 NSI)。在该协议中，通过在同步点，对于不确定是否被其他核心更改的缓存块主动无效的方法，从而在不需要存储共享信息的目录的情况下来保证多核系统符合弱一致性协议。为了避免无效掉未来可能还会使用的有效块，我们同时引入了猜测执行的技术，从而降低由此带来的性能损耗。

实验表明，在一个 16 核的多核处理器中，相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储备份信息的目录以及其电路面积，而且平均可以提高 7.80% 的程序性能，减少 31.10% 的片上网络通讯，以及减少 12.39% 的功耗。而这一切，只需要改变处理器缓存协议的设计，并不需要改变现有的编程语言和编译器，因此，该协议无需更改或重新编译即可以兼容现有的代码。

5.1 缓存一致性协议介绍

为了解决程序在多核系统加速受限的问题，出现了串行程序不同的编程模式 —并行编程。从存储管理及编程界面的角度看，并行编程模型可以分为共享存储和消息传递两类。和消息传递相比，共享存储体系结构编程模型较为简单，

程序访问在共享地址空间中的数据就如同访问在传统的虚存中的数据一样，不用考虑数据所在的具体位置。所以，现在的片上多核处理器一般都使用共享存储编程模型。

片上多核处理器通常采用复杂的缓存层次。具体来说，所有核一般会共享最后一级高速缓存 (Last-Level Cache, 简称 LLC)，同时，每个核心也有其自己的小的一级缓存 (L1 缓存，简称 L1C，包括 L1 指令缓存，L1 数据缓存，以及受害者缓存 (victim cache))。缓存一致性协议负责不同类型缓存中存储的数据的一致性，当前主流的缓存一致性可以分为两类，基于目录的缓存一致性协议（比如[47]）和基于广播的缓存一致性协议（比如[48]）。在基于广播的缓存一致性协议中，每个核对缓存块的独占请求 RdEx 都会通过广播立即被所有核观察到。受限于同时只允许一条广播存在，广播协议很难被扩展到众核。而基于目录的缓存一致性协议，因为具有更好的扩展性，已经被学术界[6]和工业界[7]广泛采用。基于目录的缓存一致性协议的基本思想是通过目录来跟踪共享缓存块的私有备份情况。当某个处理器核发出独占请求时，片上网络会根据目录通知之前拥有该缓存块备份的处理器，使其私有备份无效，这样就可以保证所有的独占操作（一般为写操作）立即被所有处理器观察到。

然而，使用目录存储共享信息有两个缺陷。第一个缺陷是根据记录的共享信息，目录会发出许多无效消息[1]给核心，并从核心收回等量的 Ack 消息。这些消息不仅增加了互联网络的能量消耗，还因为网络拥塞降低了处理器的性能。第二个缺陷是，共享信息随着核心数目的增加而急剧增加，导致芯片面积的明显的消耗。虽然，有很多技术通过模糊或者不正确地记录共享者的信息来减少共享信息的面积消耗 (如, 稀疏目录[50]，有限的指针目录[48], TL[52], 和 Cuckoo 目录[53])，但是，这些技术会给非共享核心带来导致额外的写无效/写应答消息，从而导致性能进一步降低。最近，Choi 等人设计了 DeNovo[57] 来消除目录。然而，它需要对编程语言和编译器进行较大的改动，因此和传统的代码并不兼容。

[1]在本文中，无效 (Invalidation) 消息是指 LLC 发出的一个请求，请求从一个共享者核心消除对应的 L1C 块，而 Intervention 消息指的是 LLC 请求获得一个专用/修改的 L1C 块的内容。

5.2 NSI 基本思想

在本文中，我们重新考虑了缓存一致性协议和在弱一致性模型下[87] 的共享信息的必要性。弱一致性模型是实际上的标准存储一致性模型[2]。弱一致性模型允许两个的同步操作之前的访存指令自由重排，因此能够容纳许多关于指令重排的硬件/软件技术，包括乱序执行、猜测执行、指令调度、软件并行、自动向量化、循环展开等等，而这些技术往往可以提高处理器的性能。即使某个特定的处理器符合一个更强的一致性模型 (如 TSO 模型[88]), 但是，因为所有处理器共用一个编译器，其指令会根据编译器进行重排，系统（硬件 + 操作系统 + 编译器）作为一个整体，仍然表现出弱一致性。因此，大多数（如果不是全部）共享存储并行系统表现为弱一致性。因而，大多传统的并行程序是依据弱一致性模型编写的。

本文中，我们的所有研究都是基于一个关键的发现，即将共享缓存的共享信息保存在目录里的主要目的是每个写操作能够被其他拥有该缓存块备份的核心立即观察到。而实际上，在弱一致性模型中，写操作立即被全局观察到的要求是非必要的，因为**弱一致性模型暗含在下一次同步前，允许写操作对其他核心不可见**。基于这个关键的发现，我们设计了无需存储共享缓存备份信息的无目录的缓存一致性协议（NSI），这是一个轻量级缓存一致性协议，能够在弱一致性模型条件下既不使用共享者信息，也无需额外的写无效 (Invalid)/无效应答（Ack）信息来有效保持缓存一致性。因此，芯片的面积及和网络通讯需求都能够明显降低；从而提升芯片的整体性能和降低片上网络功耗。此外，NSI 不涉及任何编程语言和编译器的修改，因此能与传统的代码无缝结合。

本文通过一个新型的**自我怀疑 + 猜测执行**机制来具体实现 NSI，可以利用图5.1中的简单例子说明（其中 Inv., ShdInt., ExcInt. 分别代表 Invalidation（无效）, ShdIntervention（共享无效）, ExcIntervention（独占无效），详见表5.1)。当一个核心的写操作（设为核心 0 的 $S(x)$）发生了一级缓存缺失（原因是其缓存被另一核心 1 共享)。如图5.1(b)，在传统的基于目录的缓存一致性协议中，共享缓存 LLC 将向核心 1 发送无效（Invalid）信息，核心 1 将无效其 L1C 备份并发送一个回应 (Ack)（即核心 0 的写操作对核心 1 立即可见了)，共享缓存 LLC 才会更新自己的对应缓存块。而在 NSI 中，共享缓存 LLC 不会向核心 1

[2]存储一致性模型是程序员和并行系统之间的一个“协议”，它定义了在一个并行系统上并行程序的合规行为，即何种行为是正确的。

发送无效 (Invalidation) 信息，核心 1 无需发送回应 (Ack)，核心 1 的 L1C 私有备份缓存块也不会立即被无效。

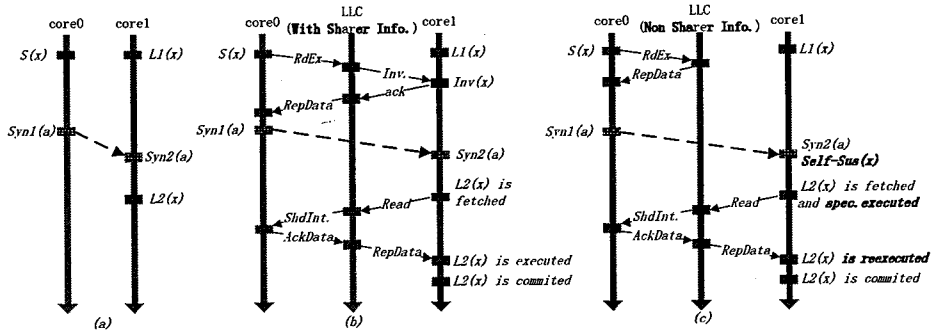

图 5.1: 在弱一致性模型 (a) 上传统基于目录的缓存一致性协议 (b) 和本文 NSI (c) 比较

相反，在 NSI 中（详见图5.1(b)），核心 1 下一次同步时，$S(x)$ 才变得对其可见，具体来说，核心 1 自行将它所有的共享 (SHD) 状态的 L1C 块标记为 *SUS (可疑)*，表示核心 1 怀疑该共享（SHD）L1C 块可能是过期的。当核心 1 的读操作命中一个可疑（SUS）的一级缓存块（L1C）时，核心 1 先用该可疑 (SUS) 块的数据进行猜测执行，同时向对应的最后一级共享缓存块（LLC）发出一个读请求，从而确认猜测执行的正确性。正如我们在5.5所指出的，无论 SUS 块是否是过期的，猜测执行带来的性能（如：L1C 缺失率、延迟）损失相对于传统的基于目录的协议是微不足道的。

我们在一个 C 语言的模拟器上实现了 NSI，并使用 SPLASH2[89] 和 PARSEC2.0[90] 作为测试基准集进行了实验。实验结果表明，在一个 16 核的处理器中，和传统的具有全目录编译器的 MESI 协议[7] 相比，NSI 协议平均提高了 7.80% 的程序性能, 平均减少了 31.10% 的片上网络（Network On Chip，简称 NOC）通讯，并且平均减少了 12.39% 的网络能量消耗。而这一切，NSI 只需要改变处理器缓存协议的设计，并不需要改变现有的编程语言和编译器，因此，该协议无需更改或重新编译即可以兼容现有的代码。这使得 NSI 成为一个共享存储领域有前景的、实用的解决方案。

本章的剩余部分的组织如下：5.3 节简要介绍了 NSI 协议的基本原理和思路。5.4 节说明了 NSI 协议的具体设计和实现。5.5 节详细分析了 NSI 的对性能的影响。5.6节介绍了实验设置和实验结果。5.7 节讨论了和 NSI 协议相关的

一起其他问题（如 NSI 协议对单线程程序的影响）。5.8节介绍了一起和本章相关的工作，5.9节对本章进行了小结。

5.3 原理和思路

本文的 NSI 协议是为支持弱一致性模型的共享存储系统设计的。弱一致性模型[87] 由 Dubois 等人于 1986 年提出，其具体定义如为：在一个多处理器系统中，访存操作符合弱一致性当且仅当：

1. 访问全局的同步变量是顺序一致的。
2. 在所有非同步变量写操作被全局观察到之前，不允许访问任何同步变量。
3. 在之前一个同步写操作被全局观察到之前，不允许访问任何非同步变量。

相比于顺序一致性需要保证所有访存操作被所有处理器核心以相同的顺序观察到，弱一致性只保证同步变量以相同的顺序被观察到即可（即只要求同步操作符合顺序一致性），对于非同步变量，弱一致性协议允许其延迟到同步点再被其他处理器核心观察到。

因此，在弱一致性存储模型中，非同步变量（即普通变量）的写操作并不需要立即被全局观察到，而是可以延迟到同步点才被其他处理器观察到。从而，任何一个写操作都广播或者根据目录发送写无效消息（Invalidation，简称 Inv.）是一个非必要的操作，写无效操作可以延迟到同步点进行。

直接遵循前述关于弱一致性模型的定义，弱一致性只要求保留弱一致性定义中的因果相关 (happen-before) 操作的顺序，也就是线程间同步操作 -同步操作的顺序和线程内同步操作 -普通操作/普通操作 -同步操作的顺序，以及这些顺序的传递闭包。因此，在弱一致性模型条件下，当一条写指令和一条读指令没有一个因果相关 (happen-before) 的顺序时，无需让这个写指令（例如，在图 5.1 中的 $S(x)$ ）对这个读指令（如图 5.1(a) 中的 $L1(x)$）是立即可见的。也就是说，在弱一致性模型中，在下一次同步点到来之前，一个远程写指令可以保持对一个核心不可见。简言之，在弱一致性模型中，写操作立即被全局观察到的要求是非必要的（而基于广播和传统基于目录的协议都会让写操作立即被观察到）。

图 5.1的简单例子可以说明上述事实。在这个例子中，核心 0 在同步点 $Syn1(a)$ 前对地址 x 执行了写操作 $S(x)$；核心 1 对地址 x 执行了两个读操作

$L1(x)$ 和 $L2(x)$，并且在两个读操作前有一个同步点 $Syn2(a)$。其中，程序员指定了 $Syn1(a)$ 必须在 $Syn2(a)$ 前发生。在弱一致性模型下，$L1(x)$ 不需要关注 $S(x)$ 写的值，因为 $L1(x)$ 能够在 $S(x)$ 前发生。然而，弱一致性模型要求，$L2(x)$ 必须能够观察到 $S(x)$ 写的值，原因是两个操作存在因果 (happen-before) 关系 $S(x) \rightarrow Syn1(a) \rightarrow Syn2(a) \rightarrow L2(x)$。所以，当核心 0 执行 $S(x)$，我们无需像传统基于广播或目录的缓存一致性协议中（参见图 5.1(b)）一样，立即将核心 1 中的 x 的共享（SHD）状态的一级缓存块（L1C）备份标记为无效来使 $S(x)$ 对核心 1 可见。相反，核心 1 能够在执行同步操作 $Syn1(a)$ 时自行将地址 x 对应的共享（SHD）状态的一级缓存块（L1C）备份无效，从而保证程序的执行符合弱一致性模型的要求，在同步点后让 $S(x)$ 对 $L2(x)$ 可见。

基于上述观察，本章提出的 NSI 缓存一致性协议无需存储 LLC 的共享信息，同时也无需目录和广播。而是通过一种新型的**自我怀疑 + 猜测执行**的机制来保证共享缓存私有备份的一致性。我们以图 5.1(c) 来说明该**自我怀疑 + 猜测执行**机制。当核心 0 的一个写操作 (图 5.1(c) 中的 $S(x)$) 在其私有缓存 L1C 中发生不命中，该核心将向共享缓存 LLC 发送一个独占请求 RdEx，然而，核心 1 对应的私有缓存 L1C 并不会因为这个 RdEx 请求而无效，因此核心 1 的读操作 $L1(x)$ 会读到该缓存块的旧值。在核心 1 的下一次同步 (图 5.1(c) 中的 $Syn2(a)$) 时，核心 0 的 $S(x)$ 才会对核心 1 可见，也就是核心 1 把它所有的 SHD L1C 块都标记为 SUS (可疑)，表示核心 1 怀疑该 L1C 缓存块可能是无效的，可能需要更新。当核心 1 的读指令 $L2(x)$ 访问包含 x 的 L1C 块（其状态为 SUS）时，核心 1 知道其私有备份可能需要更新（也可能不需要更新，取决于其他核心有没有修改该缓存块），但为了性能，核心 1 先用 SUS 块的数据进行执行，同时，给 LLC 对这个块发出一个读请求。最后，根据 LLC 的返回值，确定其本地的 SUS 状态的 L1C 块是否是最新的，假如是，猜测正确，猜测执行的结果直接提交；假如不是，则回滚并用最新的值重新执行（在图 5.1中，猜测是失败的，需要重新执行）。

使用可疑状态（SUS）块中的数据进行猜测执行对性能的影响不大。原因是猜测执行 SUS 块中的数据有两种情况，无论哪种情况，NSI 总能获得与传统的基于目录的协议的读指令几乎相同的 L1C 缺失率和延迟，具体如下：

1. 如果 SUS 块未被更新，猜测读和之后依赖于该猜测读的所有指令都能够成功提交，同时，该 SUS 块会返回 SHD 状态。这种情况不会对性能造

成影响。相反，由于是自己主动怀疑数据无效（SUS），减少了 LLC 发送写无效（Invalidation）和 L1C 回应（Ack）的时间，性能反而能够提高。

2. 如果 SUS 块已被更新，猜测读需要重新执行。但是，在这种情况下，即使是基于目录的协议，由于此时核心 1 的 x 对应的 L1C 是无效的，$L2(x)$ 也需要等待一段时间，其等待的时间是从 LLC 取回最新数据的时间。而在 NSI 中，这段等待的时间被用来猜测执行了，只不过猜测失败了。但当 LLC 最新数据回来时，NSI 和传统基于目录的协议能同时使用最新的数据执行。

总之，由于 NSI 缓存一致性协议不再需要通过目录存储共享缓存的共享者信息，同时在写操作时无需给其他共享者核心的发送写无效（Invalidation）和接收无效响应（Ack）网络消息。相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储共享者信息的目录以及其电路面积，同时，可以减少片上网络通讯，降低网络拥堵，从而提升整体性能，降低片上网络功耗。

5.4 设计和实现

本文提出的 NSI 缓存一致性协议是一个轻量级的协议。NSI 可以通过对多种传统的基于目录的缓存一致性协议（如 MESI,MSI 和 MOESI）进行简单的修改来实现。由于 MESI 缓存一致性已经广为知晓，本文重点介绍如何对 MESI 进行修改来实现 NSI。正如前文所言，相比传统的缓存一致性协议，NSI 既不需要共享者信息，也不需要广播，但需要在每个共享缓存 LLC 块的标签中有一个专用的属主域，用来保存拥有此块独占（EXC）或修改（MOD）状态的 L1C 的备份的核心编号（ID），若没有任何一个核心拥有该块的独占（EXC）或修改（MOD）状态的备份，则该域的值为 -1。

在本节的剩余部分，我们将介绍 NSI 的设计细节，包括高速缓存的状态，网络消息类型，高速缓存状态转换规则，关键操作（如同步识别）的硬件实现。同时，我们也对 NSI 进行了代价分析，并用一个样例程序的执行过程来进一步说明 NSI 和传统基于目录缓存一致性协议的区别。

5.4.1 高速缓存状态

具体来说，NSI 采用一个 3 位寄存器来表示一级私有缓存 (L1C) 块的 5 种可能的状态，包括 INV（无效状态，表示该块的数据确定不是最新的）；SHD (共享状态，表示该块是通过读操作从 LLC 获取的备份，且暂时没被其他写操作无效)；EXC（独占操作，表示该块是通过写操作获取的备份)；MOD（更改状态，L1C 通过写操作获得 EXC 块之后，若 L1C 继续写该块，则 L1C 的缓存块状态为 MOD 状态)；SUS（可疑状态，表示一个 SHD 的 L1C 块不确定是否被其他处理器的写操作更新过)。在前面的 5 个状态中，其中前四个都是 MESI 缓存一致性协议中有的，且意义也相近（读指令能够访问 SHD/EXC/MOD L1C 块，写指令能够访问 EXC/MOD L1C 块)，而 SUS 状态是 NSI 所特有的。NSI 和 MESI 间 L1C 状态的主要区别包括：同一个 LLC 块的 SHD L1C 备份能够和 EXC/MOD L1C 备份共存，无论 SHD L1C 中的数据是否已经更新；一个 SHD L1C 块必须能够同步变换为 SUS 状态（NSI 对每个 N1C 块使用专用的寄存器来表示 SUS 状态，所以所有的 SHD L1C 块能够在一个周期内转换为 SUS 状态)；一个访问 SHD L1C 块的读指令能够使得猜测执行和读请求从 LLC 中检索到数据（一个访问 SUS L1C 块的写指令能引起一个正常的高速缓存缺失)。

NSI 对共享缓存 LLC 块定义了三个可能的状态，包括 INV（无效状态)、EXC（独占操作）和 MOD（更改状态)。LLC 的 EXC 状态表示这个 LLC 块是有效的，而且它的数据也是最新的（没有核心修改过它的 L1C 块)。MOD 状态表示这个 LLC 块是有效的，但它的数据未被更新（最新的数据在某个 L1C 备份中)。和 MESI 不同的是，NSI 中共享缓存 LLC 没有共享（SHD）状态，因为 NSI 中的 EXC/MOD LLC 块也允许一个核心获得 SHD L1C 备份（更多细节参见5.4.3部分)。

5.4.2 网络消息

在 NSI 中，大部分用于保证一致性的网络消息都和其在 MESI 对应消息类似。比如，无论是在 NSI 还是 MESI，对于一个读操作，L1C 发生缓存缺失 (Cache Miss) 时，L1C 会发送一个读请求（Read）给 LLC；对于一个写操作，L1C 发生缓存缺失（Cache Miss）时，L1C 会发送一个读独占（RdEx）请求给 LLC；对于 Read 请求，LLC 将返回一个共享回应（RepShd）消息，同时

返回一个共享（SHD）块给 L1C；对于 RdEx 请求，LLC 将返回一个独占回应（RepExc）消息，并同时返回一个独占（EXC）块给 L1C。当 LLC 收到 L1C 的一个 Read 请求时，假如 LLC 上缓存块的状态是 MOD 状态，则 LLC 发送一个 ShdIntervention 请求给 L1C；当 LLC 收到 L1C 的一个 RdEx 请求时，假如 LLC 缓存块的状态是 EXC/MOD 状态，则 LLC 会给 L1C 发送一个独占无效（ExcIntervention）消息。

NSI 和 MESI 的最显著区别是 NSI 没有无效（Invalidation）消息以及无效应答（Ack）消息。这两个消息是某个 L1C 写操作时无效其他 L1C 上的 SHD 块的请求和应答消息。另外，在 NSI 中，共享无效（ShdIntervention）消息会将 L1C 上 MOD 块转成 EXC 状态（而 MESI 中转成了 SHD 状态）；独占无效（ExcIntervention）消息将 L1C 上的 EXC/MOD 块转成了 SHD 状态（而 MESI 中转成了 INV 状态），NSI 允许一个 LLC 块在有不多于一个 EXC/MOD L1C 块同时有多个 SHD L1C 块。

更多关于 NSI 和 MESI 中网络消息请参见表5.1 ，其中 NSI 和 MESI 的区别已用斜体标示出来了。

5.4.3 高速缓存状态转换

图5.2比较了 NSI 和 MESI 中私有缓存 L1C 缓存状态的转化。其中图5.2(a)是 MESI 的 L1C 状态转化图，图5.2(b) 是 NSI 的 L1C 状态转化图；Inv., ShdInt., ExcInt. 分别代表 Invalidation（无效），ShdIntervention（共享无效），ExcIntervention（独占无效），详见表5.1。

如图5.2(b) 所示，当 LLC 上对应块有属主（Tag 中属主域值不为 -1）时，INV 的 L1C 块会被 Read 操作转化成 SHD 块，当 LLC 上对应块没有属主时，INV 的 L1C 块会被 Read 操作转化成 EXC 块。没有属主时读操作会将 L1C 转化成独占 (EXC) 是为了保证任何一个块都至少有一个属主，且 LLC 缓存块的属主属于最近写过该块的 L1C 或者在没有 L1C 写过该块情况下第一次读该块的 L1C。这还能保证私有地址在 L1C 处于 EXC/MOD 状态，从而保证单线程程序不会被自我怀疑机制影响性能，详见5.7节讨论。L1C 上的 INV 块也可以直接被 RdEx 操作转成 EXC 状态。

如果遇到 RdEx 操作，L1C 中的 SHD 块会转成 EXC 状态；如果碰到同

表 5.1: NSI 中片上网络消息

斜体标示的是与传统的 MESI 协议不同的部分

消息	描述
核心 →LLC	
Read	当读指令发生 L1C 不命中时，该核心将发送一条普通读 (Read) 请求。当收到一个 Read 请求时，*如果 LLC Tag 中属主域为 −1，即该缓存块没有被任何 L1C 拥有，LLC 会回复一个 RepExc 消息给请求核，并且将 LLC 块的属主域的值设置为请求核的 ID*; 否则回复一个 RepShd 消息，表示 L1C 收到是一个 SHD 的块。
RdEx	当写指令发生 L1C 不命中时，该核心将发送一条独占读 (ReEx) 请求。当收到一个 RdEx 请求时，LLC 向属主核心发送一个独占无效 (ExcIntervention) 请求，然后回复给请求核一条 RepExc 消息，并且将 LLC 块的属主域的值设为请求核的 ID 。
Upgrade	当核心想修改其 EXC L1C 备份时，该核心会发送一条 Upgrade 请求给 LLC。当 LLC 收到一条 Upgrade 请求时，LLC 会将 LLC 块的状态改为 MOD。
Replace	当核心想替换出一个 EXC/MOD L1C 备份时，该核心会发送一条 Replace 给 LLC。当 LLC 收到一条 Replace 请求时，*LLC 会将块的属主域的值修改为 -1，并将 LLC 块的状态改为 EXC 状态。*
AckData	当核心收到 ShdIntervention/ExcIntervention 请求时，会给 LLC 发送一条包括最新数据的 AckData 消息作为回复。
~~*Ack*~~	*在 NSI 中，Ack (回应 Invalidation) 消息已被消除。*
LLC→ 核心	
ShdIntervention	当碰到一条 Read 请求访问 MOD LLC 块时，LLC 发出一条 ShdIntervention 消息给属主核心。当属主核心收到 ShdIntervention 请求时，该核心会给 LLC 回复一条 AckData 消息，*并将其状态改为 EXC*。
ExcIntervention	当碰到一条 RdEx 请求访问一个 EXC/MOD LLC 块时，LLC 发出一条 ExcIntervention 消息给属主核心。当属主核心收到该 ExcIntervention 请求时，该核心会给 LLC 回复一条 AckData 消息，*并将其状态改为 SHD*。
~~*Invalidation*~~	*在 NSI 中，Invalidation (LLC 排除 SHD L1C 备份) 消息已被消除。*
RepShd	当 LLC 收到一个 Read 请求，*且当 LLC 块已经有了 EXC/MOD L1C 备份 (例如，属主域的值不为 -1，* 会给该核心回复一个包含最新数据的 RepShd 消息，请求核心收到后会将其 L1C 状态改为 SHD。
RepExc	当 LLC 收到一个 RdEx 请求，会给该核心回复一个包含最新数据的 RepExc 消息，请求核心收到后会将其 L1C 状态改为 Exc；或者当 LLC 收到一个 Read 请求，*且当 LLC 块没有 EXC/MOD L1C 备份 (此时属主域的值为 -1)，会给该核心回复一个包含最新数据的 RepExc 消息，请求核心收到后会将其 L1C 状态改为 Exc*。
AckChange	当 LLC 收到一个 Upgrade/Replace 请求，会给该核心回复一个 AckChange 消息。

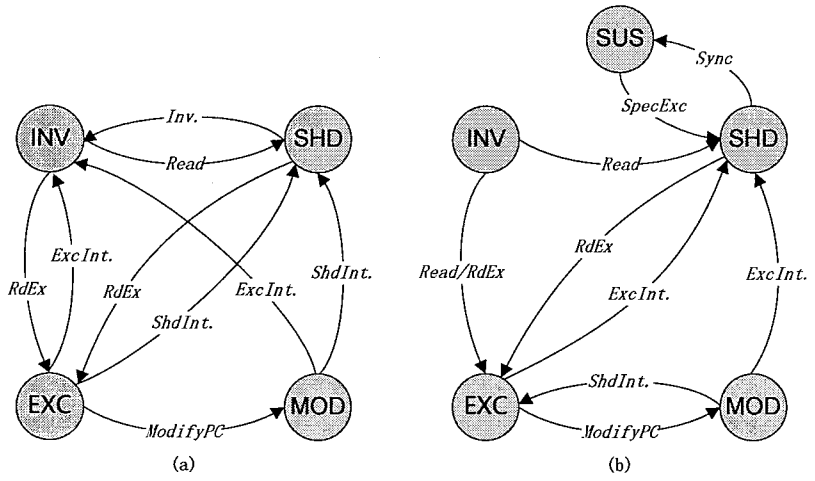

图 5.2: MESI (a) 和 NSI (b) 的 L1C 状态转换比较

步操作 (Sync)，则 SHD 块会转成 SUS 块，表示该块可能在同步点之前被其他块修改过。

如果遇到对 EXC 块的操作，L1C 的 EXC 块会转成 MOD 状态；如果收到一个独占无效（ExcIntervention）消息，则 EXC 块会转成 SHD 状态。

如果收到独占无效（ShdIntervention）消息，L1C 的 MOD 块会转成 EXC 状态，如果收到独占无效（ExcIntervention）消息，则转成 SHD 状态。

当一个 SUS 块遇到读操作时，会转化成 SHD 状态（假如 SUS 块的数据是最新的，则直接转化，假如 SUS 块的数据不是最新的，则从 LLC 取回最新块后再转成 SHD 状态)。

在 NSI 中，LLC 的状态转化相对简单。对于 INV 的 LLC 块，当遇到一个 Read/RdEx 操作，会转化成 EXC。当某个 L1C 发生对 EXC 块的写操作（LLC 会收到 Upgrade 请求），EXC 的 LLC 块会转成 MOD 状态。当收到 Read/RdEx 请求时，MOD 的 LLC 块会转成 EXC 状态。

5.4.3.1 硬件（模拟器）实现

从前文的 NSI 设计中可以看出，NSI 的实现是相当轻量级的，因为大多数由 NSI 定义的高速缓存的状态、网络消息和状态转换都和传统的基于目录的协议（如 MESI）一样。因此，本文采用了在原本由 MESI 实现的处理器的 C 模

拟器上通过轻量级的修改来实现 NSI。对一个基于 MESI 协议的 C 模拟器做出的具体修改总结如下：

1. 在 LLC 中，去除了 Tag 中目录，增加了只用记录一个处理器 ID 的属主域。

2. 一个 LLC 块能拥有多个 SHD L1C 备份，但无论 SHD L1C 备份中的数据是否已过期，都最多有一个 EXC/MOD L1C 备份。

3. NSI 用每个 L1C 块的寄存器的一个额外的位来表示 SUS 的状态。同步时，每个核心可以利用该位把每个 SHD 状态的 L1C 块设置为 SUS 状态。

4. 读指令访问一个 SUS 块时能够用 SUS 块中现有的数据进行猜测执行，并给 LLC 发送一个读请求。LLC 回应给 L1C 最新的数据。只要发现该模块已经过期，那么读指令和其相关的指令都需要重新执行。如果 SUS 块的数据依然是最新的，那么读指令和其相关的指令的猜测执行的结果（假设和其他猜测不相关）就可以成功提交，同时将 SUS 块返回到 SHD 状态。

5. NSI 会无需写无效（Invalidation）和写回应（Ack）消息，因为 NSI 不要求写指令迅速全局可见。

6. NSI 中，ShdIntervention 请求可以将 MOD L1C 块转变为 EXC 状态（而不是 MESI 中的 SHD 状态）。在其对应的应答（AckData 消息）中，核心必须给 LLC 发送这个块最新的数据。

7. NSI 中，ExcIntervention 请求将 EXC/MOD L1C 块改变为 SHD 状态（而不是 MESI 中的 INV 状态）。在其对应的应答（AckData 消息）中，核心必须给 LLC 发送这个块最新的数据。

8. NSI 中，更换 SHD/SUS L1C 块不需要通知 LLC。

9. 对于没有属主的 LLC，无论是普通读请求 Read，还是独占读请求 RdEx，LLC 都会回复一个 EXC 块给 L1C。而在 MESI 中，只有独占读请求会回复 EXC 块。

5.4.3.2 同步识别

在 NSI 中，将和锁/原子/同步（如 LL/SC、test&set）相关的任意指令都认为是同步指令。为了在所有的同步之间保持严格的顺序一致性（从而保证整个程序符合弱一致性），NSI 把上述所有的指令看做一类特定的写指令（请注意 NSI 保持了冲突写指令之间的因果关系）。这些特殊指令也可以看做是相应核心流水线的“障碍”：在这些指令完成之前，先前的指令都必须退出了指令窗口并成功提交；在这些指令从退出指令窗口并成功提交之前，之后的指令都不允许执行。为了解决这个繁忙 -等待锁和其他可能导致死锁的问题，NSI 定期用硬件进行同步。

5.4.4 成本分析

正如前文所述，从面积上来讲，NSI 主要的成本是每个 LLC 块用于表示属主域的 log_2^p 位（p 是核心数）和每个 L1C 块用于表示 SUS 状态的 1 位。显然，这些成本相比于传统的用目录保存共享者信息的成本相比是微不足道的（目录需要 p，当 p 很大时，$log_2^p + 1$ 远小于 p）。值得一提的是，很多先进的处理器已经能够支持 NSI 需要的对读指令的猜测执行，所以，因为猜测而带来的额外的实现成本也很小了。

5.4.5 程序执行样例

为了更好的说明和比较本文提出的无共享者信息无目录的 NSI 和基于目录的 MESI 缓存一致性协议的区别，如图5.3所示，我们将展示一个简单的程序（toy program）在 NSI 和 MESI 上执行过程。为了便于说明，我们假设变量 X, Y 和 Z 在不同的高速缓存块，X, Y 和 Z 同时也代表其所在的缓存块。如图5.3所示，我们可以看到，在本文的 NSI 和基于目录的 MESI 协议上，高速缓存的状态转换不完全相同。

在时间点①，执行完指令 `X=Y=1`，无论是 NSI 还是 MESI，核心 0 的 L1C 块 X 和 Y 都处于 MOD 状态。

在时间点②，执行完指令 `Z=X+Y`，核心 1 的 L1C 块 Z 处于 MOD 状态，而 L1C 块 X 和 Y 都处于 SHD 状态。在 NSI 中，核心 0 仍然有处于 EXC 状态的 L1C 块 X 和 Y，因为 NSI 允许 SHD L1C 备份和一个 EXC/MOD L1C

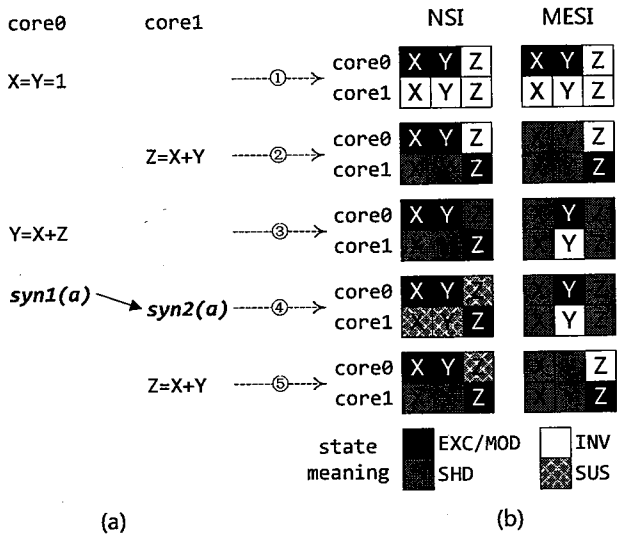

图 5.3: NSI 和基于目录的 MESI 协议执行对比

备份共存；然而，在 MESI 中，核心 0 就不得不把 L1C 备份 X 和 Y 从 MOD 状态转变为 SHD 状态。

类似的，在时间点③，在执行完指令 `Y=X+Z`，NSI 允许核心 0 有一个 Z 的 SHD L1C 备份，同时核心 1 有一个 Z 的 EXC L1C 备份，然而 MESI 必须将核心 1 的 Z 的 MOD L1C 备份降为 SHD 状态。在 NSI 中，L1C 块 Y 也不会改变它的状态，因为核心 0 已经有了 Y 的 EXC 备份，LLC 不需要令核心 1 的 Y 的 SHD 备份无效。但是在 MESI 中，核心 1 的 Y 的 SHD 备份就会由核心 0 设为无效。

同步时，NSI 要求核心 0 和核心 1 将他们的 SHD L1C 块变为 SUS 状态，而 MESI 不会改变 L1C 块的状态。

同步后，核心 1 执行指令 `Z=X+Y`。在 NSI 中，核心 1 会发现 X 和 Y 都在 SUS 状态，并猜测执行指令。需要注意的是，在这里，核心 1 的 SUS 块 X 含有最新的值，而 SUS 块 Y 的值已经过期 (在核心 0 处的远程写指令 `Y=X+Z` 改变了 Y 的值)。

所以，当核心 1 从 LLC 处收到最新的 Y 的值就会重新执行指令。而在 MESI 中，核心 1 也需要从 LLC 检索得到 Y，因为它的 Y 的 SHD L1C 备份

在时间点③被核心 0 设为了无效。所以，在执行指令时，NSI 和基于目录的 MESI 具有相同的 L1C 缺失率和近乎相同的延迟。

在时间点⑤执行完指令 `Z=X+Y` 后，NSI 中，核心 1 有两个 SHD L1C 块，X 和 Y 两个缓存块都从 SUS 状态中恢复。X 的状态转变是由于从一个不正确的怀疑中恢复回来（X 的 SUS 块确实是最新的），而 Y 的状态转变（成 SHD）是由于怀疑正确，需要从具有最新版的 L1C 块重新填充引起的简言之。在 MESI，核心 0 的 EXC L1C 块 Y 会变为 SHD 状态，而其 SHD L1C 块 Z 被核心 1 设为无效。核心 1 会将自己的 INV L1C 块 Y 升为 SHD 状态，而 SHD L1C 块 Z 变为 EXC 状态。

5.5 性能分析

为了分析 NSI 的性能影响，我们比较了 NSI 中和基于目录的 MESI 协议中的读指令的 L1C 的缺失率和延迟（我们本节只分析读指令的 L1C 的缺失率和延迟的原因是，NSI 的自我怀疑（在同步点将 SHD L1C 块转换为 SUS 状态）显然不会增加 L1C 的缺失率或者写指令的延迟，因为访问 SHD L1C 块的写指令也会引起一个 L1C 缺失。）如图 5.4 例子所示，其中，该核心会有一个 SUS L1C 块，该块中含有被读指令 $L(x)$ 访问的内存地址 x，其中，图中坐标轴的阴影条表示有效执行的与 $L(x)$ 相关的指令，“written back” 表示指令用流水线方式写到寄存器文件中。

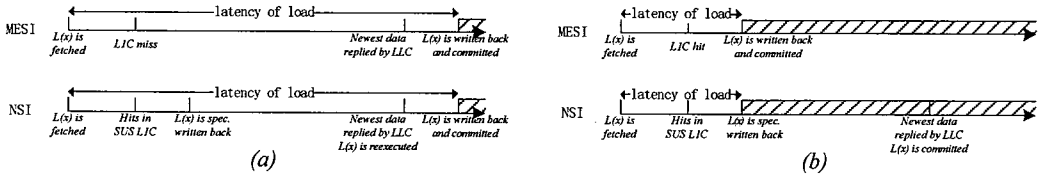

图 5.4: 在访问 SUS 块时，NSI 对性能的影响示意图：(a) SUS 块有 x 的过期版本, (b) SUS 块有 x 的最新版本

按 SUS 块中的数据是最新还是过期的，我们分两种情况分析 NSI 对性能的影响：

(a) 如果 SUS 块有 x 的过期的版本： 在 NSI 中，核心使用 SUS 块的值猜测执行 $L(x)$ ，同时，从 LLC 取回 x 的最新版本。当 LLC 返回了 x 的最

新版本，发现之前使用的 L1C SUS 块是过期了，那么 $L(x)$ 被重新执行（如图 5.4(a) 所示）。从图中可以发现，这里必须在 $L(x)$ 前有个远程（其它核心）写指令 $S(x)$ 来写最新版的 x。因此，在 MESI 中，核心的 x 的 SHD L1C 块必须在 $L(x)$ 执行前，被远程写指令 $S(x)$ 设为无效。因此，$L(x)$ 在 MESI 会产生一个 L1C 块缺失，也不得不从 LLC 取回最新版本的 x（如图 5.4(a) 所示）。和 MESI 相比，NSI 包含了一个无效猜测执行；但是，这个猜测执行可以和取回 x 的最新版本并行，从而被从 LLC 获取 x 最新版的延迟所隐藏，而延迟是两种协议都需要的。所以，在 NSI 中的 $L(x)$ 的延迟和 MESI 中的是近乎相同的。简言之，无论是在 NSI 还是在 MESI 中，$L(x)$ 都不能够从 L1C 直接获得最新版本的 x，也就是都会产生 L1C 缺失，都需要从 LLC 取回 x 的最新版本。

(b) 如果 SUS 块有 x 的最新的版本: 在 MESI 中，核心能够直接拥有 SHD L1C 块，因此能够快速写回寄存器，然后继续执行相关的指令。在 NSI 中，核心利用 SUS 块中的数据猜测执行 $L(x)$（和 $L(x)$ 相关的指令）。当 LLC 返回 x 的最新版本时，SUS 块发现和最新版的 x 一致；那么，猜测执行 $L(x)$ 及其相关指令（假设和其他猜测不相关）就能被成功提交（如图 5.4(b) 所示）。与 MESI 相比，NSI 包含了从 LLC 重新检索 SUS 块，但是其延迟是重叠的，所以可以被猜测执行所掩盖。所以，NSI 中的 $L(x)$ 的延迟和 MESI 中的也近乎相同。简言之，无论是在 NSI 还是在 MESI 中，$L(x)$ 都能够从 L1C 直接获得最新版本的 x，都不会因为 L1C 缺失导致执行堵塞，影响性能。

综上所述，本文的 NSI 和传统的 MESI 具有相同的 L1C 缺失率和近乎相同的读指令的延迟。所不同的是，NSI 消除了所有的写无效（invalidation）/无效响应 (Ack) 消息，所以大大减少了网络拥塞（由 SUS 块检索和获取修改块的标志所增加的通信量可以忽略不计）。因此，正如我们通过实验观察到的，NSI 相比于基于目录的高速缓存一致性协议，NSI 能明显降低芯片面积和能量消耗，同时能提高处理器性能。

5.6 实验

在本节中，我们将介绍对 NSI 的性能功耗评估实验，具体包括实验方法，实验结果和实验分析。

5.6.1 实验方法

本文的实验是基于一个事件驱动，能精确到每拍的处理器模拟器实现的，该模拟器是基于 SimpleScalar 工具集[91] 开发的，能够支持 MIPS 指令集体系结构 (ISA)。为了评估 NSI 片上网络的能量消耗，我们在该模拟器上集成了 Orion[92] 网络功耗模型。Orion 使用的参数是 65nm 工艺，1.0v 的核心电压，1Ghz 的主频，以及 2500μm 的核间连线长度。

表 5.2: 系统的参数

参数	值
处理器参数	
核心数	16
处理器核心	1G 主频，乱序 4 发射
分支预测	GShare，4096 项 PHT
ALU/FPU	2/2
ROQ 尺寸	64 项
L1/指令缓存 (L1C)	64KB，4 路，3 拍延迟
共享 L2 缓存 (LLC)	16MB，4 路，10 拍延迟
缓存块大小	256 位
网络参数	
片上网络拓扑	4 × 4,2D 网格
路由延迟	2 拍
线延迟	2 拍
链路宽度	**128 位**
能耗参数	
路由能耗	3.77e-10J
链路能耗	2.22e-10J
总线仲裁能耗	9.01e-13J

在该模拟器中，我们集成了一个 16 核处理器，每个核心集成了一个 4 发射，9 级流水的乱序执行单元，每个核心分别有 64KB 的 L1 数据缓存和 L1

指令缓存。所有核心共享 16MB 的 L2 缓存（LLC 缓存），这 16MB 的 L2 采用静态非一致缓存体系结构（Static Non-Uniform Cache Architecture，简称 S-NUCA)[93]。访问不同处理器核心的 LLC 缓存可能会有的不同延迟。该多核模拟器的片上网络采用了 4×4 的二维（2D）网格（Mesh）拓扑结构，具有 128 位的数据通路，其模拟的处理器的具体参数详见表 5.2和图 5.5 所示。

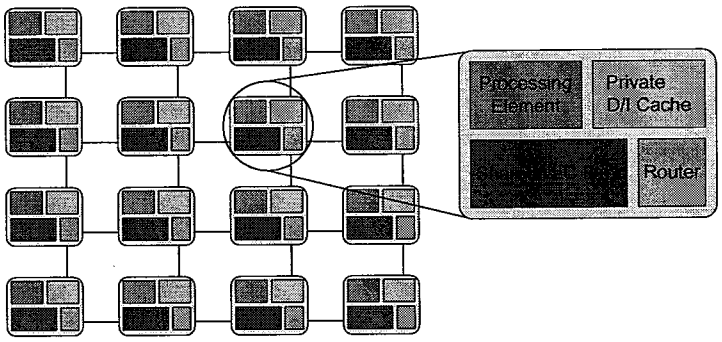

图 5.5: 本文模拟的具有二维网格结构的 16 核处理器架构示意图

为了充分验证 NSI 协议的性能和功耗，本文实验采用了 SPLASH2 基准程序[89]、PARSEC2.0[94] 基准测试程序和随机产生的程序。SPLASH2 和 PARSEC2.0 程序的输入数据集详见表5.3。

5.6.2 实验结果

为了验证本文提出的 NSI 的性能和功耗优势，我们首先比较了 NSI 和 MESI 协议的整体性能，而后，我们比较在两种协议下，私有缓存 (L1C) 的缺失率，片上网络通讯量和片上网络功耗方面的实验结果。最后，我们使用了随机生成的程序来观察同步频率和访存区间（Memory Footprint）对性能的影响。

整体性能比较:

在本文的试验中，并行程序的整体性能采用了总执行时间来测量，而不是采用每拍执行的指令数（instruction per cycle，简称 IPC)，原因是对于 IPC，不同的线程可能差别很大。图5.6给出了每个基准测试程序的在 NSI 上的执行时间，其执行时间归一化到了 MESI 的执行时间。总体而言，相比于 MESI，NSI 的性能优势很明显，在 10 个基准测试程序中，有 8 个程序（除了（fmm 和 raytrace)）NSI 的性能比 MESI 要好。其中，性能提升最大的是 *ocean-non* 程

表 5.3: 测试集和输入大小

测试集	规模
barnes	4096 particles
blackscholes	16K options
cholesky	d750.0
dedup	hamlet.dat
fft	65,536 data points
fmm	256 particles
lu	512×512 matrix
ocean-con	130×130 grid
ocean-non	130×130 grid
streamcluster	16,384 points
swaptions	10,000 simulations
radix	262,144 Keys
raytrace	teapot
water-nsquared	3 steps, 512 molecules

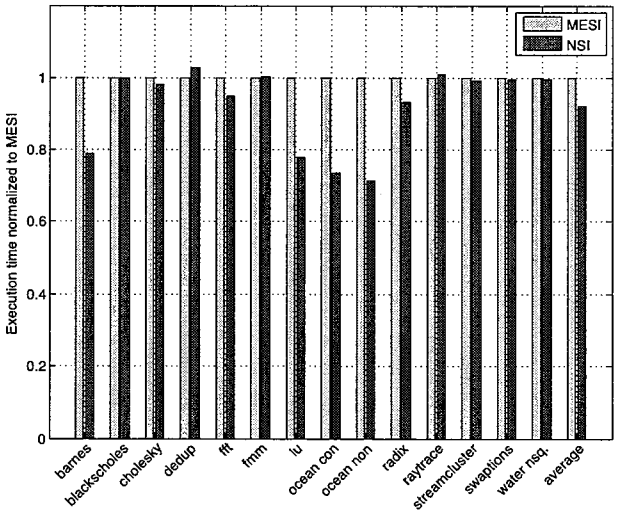

图 5.6: MESI 和 NSI 性能比较

序，NSI 将性能提升了 28.34%。平均而言，NSI 相比于 MESI 整体性能提高了 7.8%，NSI 明显的提升主要归功于 NSI 可以消除写操作中非必要的无效/Ack 消息。

我们也估计了不同程序的 SUS 块具体的分布。从图5.7 可以看出，大多 (平均 68.58%) 的 SUS 块在被替换出去之前不能够被相应的核心访问，也就不会引起猜测执行。同时，平均 28.07% 的 SUS 块仍然是最新的数据，因此，用这些数据进行猜测执行是能够被顺利提交的。只有剩下部分的 SUS 块（平均 3.35%）的数据是过期的，而且对应的猜测执行是无效的，应该被作废并重新执行。总而言之，NSI 的自我怀疑机制只会引起一小部分无效的猜测执行。

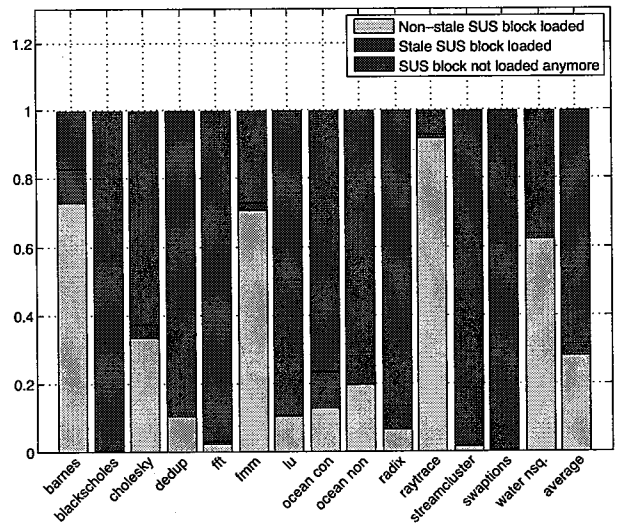

图 5.7: SUS 块的分布

网络通讯的比较:

因为 NSI 明显减少了不必要的核间通信，使得网络拥塞得到缓解，因而相比 MESI，还降低了功耗。在图5.8我们展示了 NSI 和 MESI 网络通讯和能耗的比较。

NSI 能消除所有的写无效（Invalidation）和无效响应（Ack）消息，相比传统的 MESI 协议，NSI 能显著地减少网络通讯。图.5.8 比较了 NSI 和 MESI 网络通讯情况。如图所示，本文的 NSI 可以大大地减少网络通讯。具体而言，对于最好的情况（在程序 lu），NSI 可以将网络通讯减少 81.92%，即使对于最

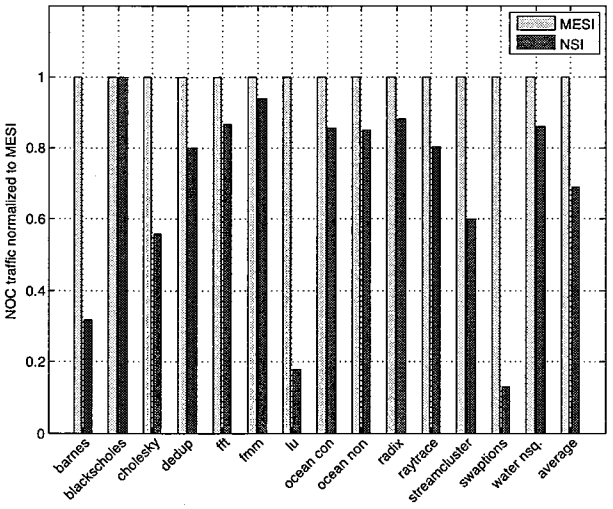

图 5.8: MESI 和 NSI 网络通讯比较

差的情况（fmm），NSI 仍然可以减少 6.23% 的网络通讯。相比于 MESI，网络通讯的显著减少解释了 NSI 相比于 MESI 的性能提升。平均上，NSI 能够减少 30.10% 的网络跳（flit）。实际上，NSI 的高性能应该归功于它能够减少网络通讯，因为网络阻塞现象已经显著减少了。然而，在通讯的减少和性能的提高间不存在强相关性（例如，对于 ocean-non 基准测试程序，NSI 减少的网络通讯并不是最多的，而其提升的性能却是最大的），可能的原因是对不同程序，网络拥堵不一样，对拥堵程序大的程序，减少一些网络通讯也可以极大的提升性能。

NSI 减少了网络通讯同时，能够进一步降低片上网络 (NoC) 的能量消耗，因为排除了写无效（Invalidation）和无效响应（Ack）消息不仅能够降低每个周期内路由器缓存和物理链路带来的能量消耗，而且能够降低整体执行时间。正如图5.9所示，对于所有的评估程序，NSI 平均降低了片上网络 12.39% 的能量消耗。尤其是对 barnes 程序，NSI 甚至降低了网络 46.52% 的能量消耗。

私有高速缓存（L1C）性能:

为了进一步分析性能提升的原因，我们使用前述基准程序，评估比较了 NSI 和 MESI 的 L1C 缺失率。有趣的是，如图5.10所示，在所有基准程序中，除了 fmm，NSI 降低了 L1C 缺失率（注：对于一个访问 SUS L1C 块的读指令，如果随后发现其猜测执行的数据是过期的，我们会认为这是一次 L1C 缺失，因为最新的数据应该从共享的 LLC 中获取）。一个明显的例子是程序 lu，其 NSI

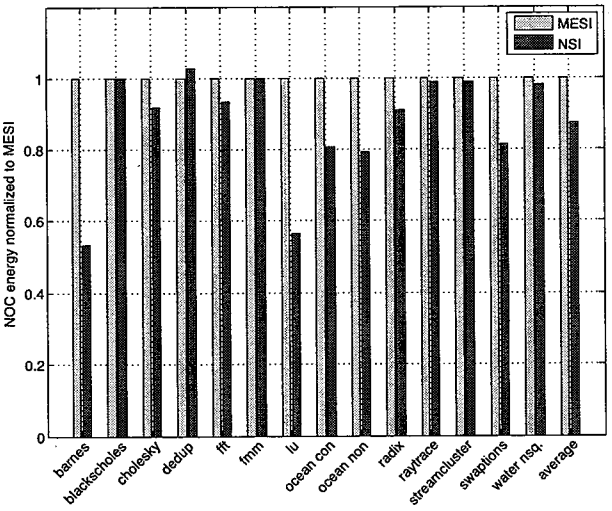

图 5.9: MESI 和 NSI 网络功耗比较

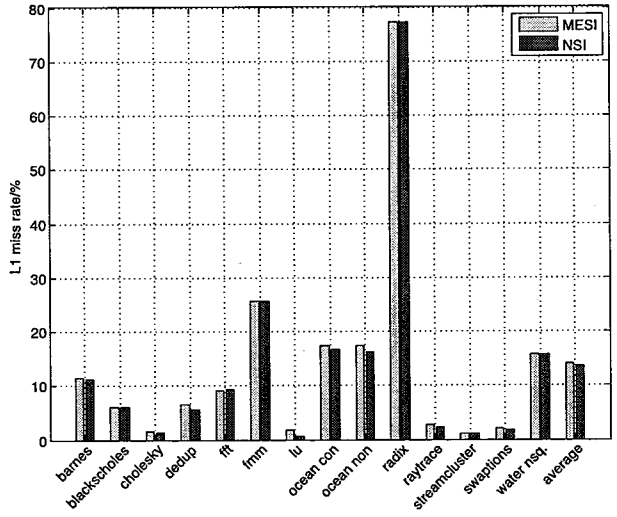

图 5.10: MESI 和 NSI 的 L1C 缺失率比较

的 L1C 缺失率 (0.61%) 只有 MESI(1.85%) 的 1/3 左右。一个可能的原因是对于在 NSI 的 L1C SHD 块，MESI 认为是 INV 状态的块 (在 NSI 中，RdEx 操作并不会将 L1C 中的 SHD 无效成 INV 块，只有在同步点才会猜测成 SUS 块，而在 MESI 中，RdEx 会将其他 L1C 上的对应 SHD 块无效成 INV 块，导致下一次访问时出现缓存缺失)，所以，在 NSI 中，有效的副本在 L1C 中会保留得更长，从而降低了 L1C 缓存缺失率。

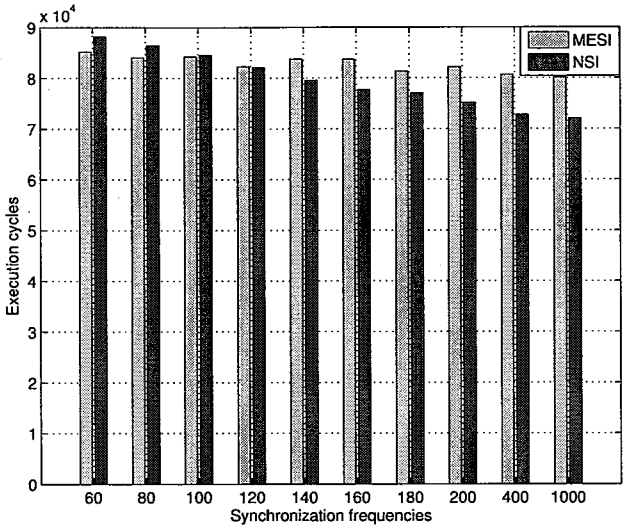

图 5.11: MESI 和 NSI 在不同同步频率下的性能比较

同步频率对性能的影响:

为了研究同步频率对 NSI 的性能影响，我们随机产生了十个只有内存指令和同步指令的并行程序，其同步频率分别是每 60, 80, 100, 120, 140, 160, 180, 200, 400, 1000 条指令同步一次。如图5.11所示，即使当同步频率在每 120 条指令同步一次时，NSI 相比于传统的 MESI 也能提升性能。每 120 条指令同步一次比大部分 (如果不是全部) 实际并行程序的同步频率要高很多。这也很好的证实了 NSI 能够提升实际程序的性能。

访存区间对性能的影响:

访存区间是指在程序执行中，程序所访问的所有内存段。为了研究不同的访存区间 (Memory Footprint) 对 NSI 的性能影响，我们随机产生了十个具有访存区间 (Memory Footprint) (分别为 0.25KB, 1KB, 4KB, 16KB, 64KB,

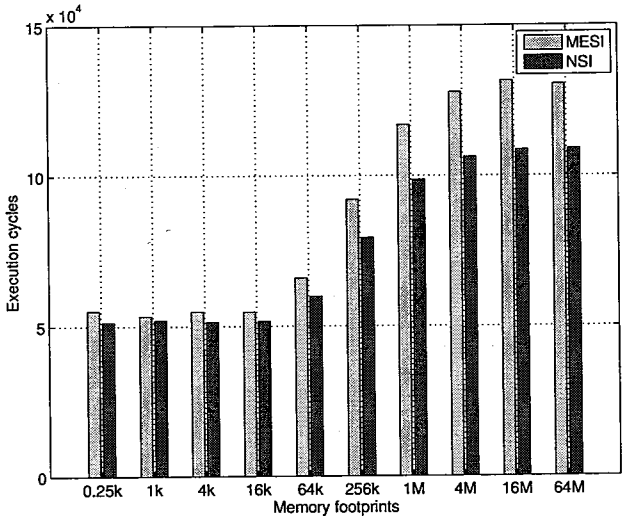

图 5.12: MESI 和 NSI 在不同内存占用时的性能比较

256KB, 1MB, 4MB, 16MB, 64MB）的程序。对于所有的访存区间情况，NSI 都比传统的 MESI 具有更好的性能（如图5.12所示）。实际上，访存区间越大，NSI 性能优势越明显，因为 NSI 能够比传统的 MESI 更有效的解决 LLC 替换问题（在 NSI 中，替换出去一个 LLC 块无需消除对应的 L1C 备份）。

5.7 讨论

在本节中，我们将讨论一些和 NSI 相关的其他问题。

单线程程序: 在提高了多线程并行程序的性能同时，NSI 并没有影响单线程程序的性能。这是因为单线程程序访问的内存区域是在这个程序中只属于这一个线程。当运行这个线程时，核心的 L1C 发生一个读缺失或者写缺失，目标 LLC 块不会有一个属主核心。因此，LLC 总是回复 EXC 块给 L1C。因此，该单线程程序的所有的 L1C 块都是在 EXC/MOD 状态，因此不会被 NSI 自我怀疑机制影响。

写操作的原子性和因果性: 显然，NSI 不能够保持写操作的原子性，一方面是因为对于一个相同的变量，两个核心可能会有不同的状态，虽然在一次同步之后就会消除这种不一致。另一方面，NSI 保持了写操作的因果性，也就是原本独有的属主总是提供最新的数据给请求这个块的核心。这是因为 LLC 通

过 AckData 消息（回复 ShdIntervention/ExcIntervention 给的消息）获得了最新的数据，并且通过 RepShd/RepExc 消息回复最新的数据。

伪共享：伪共享，是指两个变量属于同一个缓存块。伪共享并不会影响 NSI 的正确性。这是因为即使访问缓存块的一部分，只要整个缓存块是可疑 (SUS) 的，处理器核心也会重新从 LLC 取回最新的值。

处理非弱一致性的程序：虽然大多数传统的程序是依据弱一致性（WC）编写的，但是仍然有一些程序使用了较强的内存一致性。我们可以通过对这些程序增加插入一些栅栏（fence）来保证其结果在 NSI 情况下是正确的。另外，有一些有错误的程序，虽然它们是符合弱一致性的，但是有一些错误（例如包括数据竞争）。为了避免在执行这些有错误的程序时产生死锁，NSI 定期利用硬件进行同步（如，每 100,000 周期一次）。

释放/获取的区别：某些弱一致性的变量（如，释放一致性），同步能够进一步将其分为释放和获取。然而，如果没有程序员/软件的帮助，硬件很难自动识别释放和获取。所以，NSI 不区分出释放和获取，统一把他们当成同步操作，从而保证和传统代码间的兼容性和正确性。

独享的（Exclusive）LLC：在这篇文章中，我们主要关注的是在包含共享的 LLC（inclusive shared LLC）上实现 NSI。NSI 也能够应用于独享的或私有的 LLC。在这些情况下，NSI 可以不使用共享信息的标志位（属主位），同时使用相同的缓存状态和缓存状态转化规则。

5.8 相关工作

5.8.1 减少目录面积成本

由于随着核心数目的增加，存储共享信息的目录带来极大的面积消耗，之前的研究者已经提出了很多技术来减少 LLC 目录的面积。

针对目录太大的问题，某些早期的工作[48,50,51] 的目标是减少用于每个单独的最后一级共享缓存（Last Level Cache，简称 LLC）块的共享信息的位数。例如，稀疏目录[50] 用目录项中的每一位来表示 N ($N > 1$) 个处理器核心和一级私有缓存（L1 Cahce，简称 L1C）；如果这 N ($N > 1$) 个 L1C 有任意一个具有 LLC 块的备份，那么将该块的目录项的对应位设为 1。虽然共享信息的

目录面积消耗减少了 N 倍，但是稀疏目录常常导致额外的无效/回应消息，因为即使此时只有一个核心有 L1C 备份，无效消息也会发送给所有的 N 核。所以，稀疏目录会增加 N 倍的无效/回应消息的数量。因此，稀疏目录会降低高达 10.4% 的性能[50]。

Zebchuk 等人提出了一种名为 TL 的无标签目录[52]。TL 利用了 Bloom 过滤器来保守记录共享核信息，并且通过牺牲网络通信和整体性能减少了 48% 的目录的面积成本。Cuckoo 目录[53] 使用了高效的哈希函数来减少目录面积。但是，Cuckoo 目录引入了额外的无效消息，因为 Cuckoo 的哈希函数插入了失败率。所以，Cuckoo 目录的性能并不比传统的具有完美目录的 MESI 好，而 NSI 的性能表现优于传统的 MESI。

Cuesta 等人[54] 建议对私有的内存区域禁用高速缓存一致性，因为它只会被一个线程访问。所以，无需为私有的内存区域提供目录项。Cuesta 等人还提出了一种机制来决定内存地址是否对于一个线程是私有的还是线程间共享的。因此，他们的技术能够在不改变性能的情况下降低 8 倍的高速缓存目录大小。当一个内存地址既不是私有的，也不是只读的时，SWEL[55] 通过强制排除该地址来删掉了全部目录。所以，对于 HPC 应用或多个使用一小部分共享和频繁写内存地址的虚拟机来说，SWEL 非常有效且高效（SWEL 报告说整体性能增加了 2.5%[55]）。最近，Ros 和 Kaxiras 提出了无目录多核一致性协议[56]。然而，他们的协议需要对共享数据的采用写穿透策略（write-thought strategy），这不仅会导致极大的性能开销，还会导致额外的片上网络通信。

Denovo[57] 是最新的避免使用目录的技术。它需要程序员将全部内存空间分为多个区域，而后给每个区域插入自无效信息。通过这样的程序员和编译器的支持，每个核心能够是适当的使自己的私有缓存 L1C 无效。所以不再需要 LLC 目录和无效/回应消息。然而，Denovo 需要一个新的编程模型和相应的编译器支持，所以不能够直接兼容现有的代码（即使是源代码）。相比于 Denovo，我们的工作具有相当的性能，同时又能够保持和传统代码的兼容性。

总而言之，没有一个能够理想的目录方案能够和传统代码相兼容，并且消除目录的芯片面积、网络通信量和能量成本，同时有效提高处理器性能。相比之下，本文提出的 NSI 是第一个能够同时满足上述理想特征的方案。

5.8.2 减少网络通信

传统的基于目录的高速缓存一致性协议中，写无效（Invalidation）和无效响应（Ack）消息占用了很大一部分的网络通信量。

为了减少网络通信量，Keleher 等人提出了一种名为懒惰释放一致性(LRC)[58] 的存储一致性模型。在 LRC 中，有两种同步：获取和释放。无效信息能够在获取时间传递。所以，网络传输量能够有效降低。然而，如果没有程序员的标注，硬件很难区分获取和释放（获取和释放多被设置为相同的硬件元素）。所以，用弱一致性模型写的传统的代码不能从 LRC 中获益。与像 LRC 这样延迟无效消息的模型相反，NSI 消除了无效/Ack 消息。更重要的是，NSI 着眼于弱一致性模型，也就是实际并行系统中的使用的内存模型。所以，NSI 并不区分获取和释放，使得 NSI 对大多数传统代码非常有吸引力，使得它们能够从中获益。

Lebeck 等人提出的动态自无效 (DSI) 协议[59] 允许每个核心能够在没有收到任何来自共享缓存（LLC）的无效消息的情况下自行将自己的共享状态的私有备份改为无效。因为减少了无效消息，DSI 能减少整体网络通信量高达 26%[59]。Lai 等人[60] 设计了类似于 DSI 的自无效机制，但是他们的工作侧重于如何准确的识别/预测那个共享缓存块应当被自无效。

也有一些技术通过减少访问共享缓存（LLC）来降低网络通讯量。传统的基于目录的高速缓存一致性协议中，当一个核心在一级缓存发生缺失，它必须询问共享缓存来获得最新的数据备份，而共享缓存本身也可能需要访问其他私有缓存来获取最新的数据。为了降低这样的通讯，Kaxiras 和 Keramidas[61] 建议在私有缓存发生缺失的核心直接询问另一个最有可能有有效最新 L1C 备份的核心。综合一些其他的减少网络通信量技术，他们的工作平均降低了 55.91% 的网络通信量。另一工作，动态目录[62] 通过将目录放在对应最活跃的节点上来减少不必要的网络通信，因而平均降低了 16.4% 的片上网络通讯。虽然这些技术在降低网络通讯时非常有效，但是这些技术都仍然依赖于目录的存在。相反，我们的 NSI 方案完全移除了 LLC 目录，并且平均降低了 31.10% 的网络通讯。

5.8.3 高速缓存取值猜测

取值猜测是体系结构领域广泛采用的技术[95-98]。研究者已经通过猜测无效的 L1C 块的正确值来减少从 LLC 检索块的延迟[99-101]. 以 Huh 等人[101] 的工作为例，当一个读指令访问一个无效缓存块，Huh 等人提出使用猜测缓存查找 (SCL) 的方法来产生一个猜测值继续猜测执行。同时，核心也从 LLC 中获取正确的值。如果猜测的缓存值和正确值吻合，那么访问无效缓存块的读延迟能够被隐藏。为了提高值猜测的准确性，他们还提出了一种在一个读指令访问前更新无效块的值的方法。然而，这样对无效块的更新需要依赖于基于侦听（类似广播）的缓存一致性协议，该协议很难扩展到核心数目多的处理器。

我们的工作目标是一个不同的问题。不同于高速缓存的值猜测技术，它们致力于猜测一个无效块的具体值，而我们的 NSI 方案是在下一次同步之前判断一个 SUS 块是否已经被一个远程写指令设为无效了。所以，我们的工作是和 Huh 等人[101] 的工作及其他人关于缓存值猜测的技术相正交。事实上，NSI 在以后的工作中可以考虑和缓存值猜测进行无缝结合，进一步提高性能（事实上，NSI 只猜测了 SUS 块（这是和 MESI 共享的块），而不是[101] 中的 INV 块。即使当 NSI 的预测是正确的，也只能够抵消有自我怀疑机制造成的性能损失。因此，SUS 块的猜测执行并不是 NSI 性能提升的主要原因。NSI 相比于 MESI 性能提升主要归功于网络通信量的明显降低)。

5.9 小结

在共享存储并行系统中，存储一致性模型[87,102-106] 和高速缓存一致性协议是两个至关重要的方面。存储一致性模型定义了在共享存储系统中，何种执行行为（访存序列）是正确的，而高速缓存一致性协议则从底层定义了何种缓存层次和行为（包括状态转化，片上网络消息等）可以符合该一致性模型。存储一致性模型和高速缓存一致性协议是紧密相连的，因此，设计高速缓存一致性协议必须充分考虑到存储一致性模型的特性。

在本文中，我们研究了在当今使用最广泛的弱一致性模型[87] 下，共享存储并行系统中的缓存一致性问题。由于大多数（如果不是全部）的实际系统工作时遵循弱一致性模型行为，研究弱一致性下的缓存一致性对于并行系统的设计和并行编程时间具有直接而深远的影响。我们主要观察到，弱一致性模型远

程写指令并不需要立即被其他核心（或 L1C）观察到，而是可以延迟到下一个同步点观察到；因此，在传统的 MESI 缓存一致性协议中，将其他共享核心的 L1C 备份设为无效来使得每个写指令立即全局可见是不必要的。

基于上述观察，我们针对使用弱一致性的实际系统提出了 NSI 缓存一致性协议。该协议能够通过新型的**自我怀疑 + 猜测执行**机制，消除 LLC 的共享的 L1C/core 信息和所有写无效（Invalidation）和无效响应（Ack）消息。因此，NSI 明显降低了芯片面积成本和能量消耗，并且不会增加 L1C 缺失率。同时，由于减少了互联网络的拥塞，处理器的性能也有所提高。实验表明，在一个 16 核的多核处理器中，相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储备份信息的目录以及其电路面积，而且平均可以提高 7.80% 的程序性能，减少 31.10% 的片上网络通讯，以及减少 12.39% 的功耗。另外，本文的一致性协议能够和传统并行程序无缝结合，清楚证明了 NSI 在未来的商业处理器上的可应用性。

本章提出的缓存一致性协议 NSI，不仅可以适用于传统的同构多核/众核处理器，也可以适用于异构多核处理器，包括本文致力于的面向大数据时代的多核机器学习处理器。

第六章　总结和展望

随着机器学习重要性的不断增长，业界对于机器学习处理的速度和性能功耗比也提出了越来越高的要求。一方面，大数据时代的到来使得互联网公司拥有海量的数据来进行机器学习，不断提高学习的精度。这使得机器学习的数据处理速度至关重要。工业界的一个常见问题是空有大量数据却来不及处理。例如，科大讯飞语料库的增长速度已经达到了其语音识别模型训练速度的 5 倍。这里一个重要原因是通用 CPU/GPU 的机器学习处理速度太慢。谷歌进行猫脸识别的训练甚至动用了数万个处理器核。更加严峻的是，正如 EMC 公司在 2011 年指出，互联网数据指数增长的速度，已经超过了摩尔定律的增长速度。这也就意味着，通用 CPU/GPU 的处理能力和对机器学习训练速度的需求之间的剪刀差，将会指数扩大。

另一方面，大数据可潜在提升机器学习的精度，但也不可避免地增大了机器学习模型的复杂度，这对硬件平台（如云服务器和移动终端）的性能和功耗带来了很大的挑战。近年来，随着深度学习等技术的提出，机器学习所建立的模型越来越复杂。谷歌 2012 年提出了一个 10 亿参数神经网络模型，2013 年就将其扩展至 110 亿参数。2014 年百度大脑甚至采用了一个拥有超过 200 亿参数的模型。在同样的硬件平台下，超大模型可能会提供更加准确的预测，但也会显著影响实时性。而实际应用往往对机器学习的预测有严苛实时性要求（例如广告推荐必须在极短的给定时间内完成，否则无法出现在相关页面上）。当前的许多主流硬件平台（如移动终端上的 CPU，甚至云服务商的 GPU）都往往难以在短时间内完成超大模型的预测。更重要的是，移动终端（如手机）往往要频繁运行机器学习预测任务（如语音识别、图像识别等），如果不能以很高的性能功耗比进行机器学习预测，手机电池的续航能力将会大打折扣。

本文的目标就是设计支持大规模并行数据处理和分析的高性能低功耗的多功能机器学习处理器。为了实现这个目标，本文从算法分析和访存优化，处理器核设计，多核并行扩展等三方面对通用机器学习处理器进行了研究，并分别提出了三种创新性技术。

- **基于分块调节的机器学习访存优化技术。**基于算法决定结构的设计思想，

本文首先对应用最广泛，最典型的几种机器学习算法进行了分析，具体分析了包括运算特征和访存特征，并发现现有的算法存在对片外访存带宽过高的现象。为了解决这个问题，本文提出了一种分块调节的技术，可以大大减少片外访存带宽的需求，避免访存成为机器学习处理器的性能瓶颈。实验表明，分块调节访存优化技术可以将几种典型的机器学方法的片外访存带宽需求减少 46% 到 93% 不等。

- **多功能机器学习处理器。**基于前述算法分析，本文设计了一个支持多种机器学习算法（如 k-NN，k-Means，深度学习，支持向量机，朴素贝叶斯，分类树等等）加速的多功能机器学习处理器 -PuDianNao。PuDianNao 的设计过程中充分考虑了机器学习的运算特征和访存特征，可以很好的用于机器学习应用加速，并且避免存储墙的问题。同时，为了减少处理器的面积和功耗，我们基于对机器学习算法的精度需求分析，提出了不同流水级采用不同精度数据的机器学习运算单元。同时，为了提高通用性，本文为该机器学习处理器设计了一套指令集，该指令集可以自由组合，完成一些新的机器学习方法的加速和处理。实验表明，在 MNIST 和 UCI 的基准测试集中，相比顶级的 Nvidia K20 图形处理器（GPU），本文的机器学习处理器（PuDianNao）取得了 1.20 倍的加速比，却只消耗了 GPU 1/128.41 的能耗。

- **一种无共享信息的缓存一致性协议。**随着数据规模的进一步增大，以及受限于主频，单核结构的处理器或加速器的性能已经无法满足未来的机器学习应用。因此，迫切需要一种可扩展的多核（众核）结构的专用处理器。而在多核和众核结构中，缓存一致性对处理器的性能和功耗有显著影响。传统的缓存一致性协议都存在对性能影响过大以及为了维护缓存一致性需要额外的缓存目录面积的问题。本文为了解决该问题，提出了一种新的基于自无效和猜测执行的，无需存储共享信息目录的缓存一致性协议 NSI，该协议相比传统的 MESI 协议，可以显著地提高性能，减少片上网络通讯和片上网络通讯功耗。实验表明，在一个 16 核的多核处理器中，使用 SPLASH 和 PARSEC 基准测试集，相比于传统基于目录的 MESI 协议，NSI 不仅可以完全消除用于存储备份信息的目录以及其电路面积，而且平均可以提高 7.80% 的程序性能，减少 31.10% 的片上网络通讯，以及减少 12.39% 的功耗。

在未来，对 PuDianNao 的改进主要有三个方向的工作。首先，我们将继续分析更多的机器学习算法，改进机器学习处理器核心结构，让本机器学习处理器支持更多的机器学习方法；其次，我们将结合本文的 NSI 缓存一致性协议，设计一个多核的机器学习处理器，并将其应用于大规模机器学习应用中；最后，为了便于用户编程，我们将为 PuDianNao 开发专用的编译工具链，支持 PuDianNao 在 C，C++ 等高级语言的编程。