基于“神威·太湖之光”的并行深度学习训练系统

（申请清华大学工学博士学位论文）

培养单位：计算机科学与技术系
学　　科：计算机科学与技术
研 究 生：方佳瑞
指导教师：杨广文教授

二〇一九年六月

Parallel Deep Learning Training System on Sunway TaihuLight

Dissertation Submitted to
Tsinghua University
in partial fulfillment of the requirement
for the degree of
Doctor of Philosophy
in
Computer Science and Technology

by

Jiarui Fang

Dissertation Supervisor : Professor Guangwen Yang

June, 2019

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：______________ 导师签名：______________

日　　期：______________ 日　　期：______________

摘　要

深度学习是目前最成功的人工智能技术，有望带领人类真正进入智能时代。巨大的计算需求正驱动着深度学习系统软件和超级计算机的结合。因为有美国对我国高性能芯片禁售的前车之鉴，规划中的下一代国产超算系统将全部采用国产众核处理器制造。但是，国产超算上的深度学习系统软件的研究仍是一片空白，它的构建过程临着多方面挑战：一是缺少适合国产众核处理器创新硬件架构特点的优化指导方法；二是缺乏从复杂深度学习计算核心到全新体系结构的映射方法；三是国产编译工具和系统库使用时仍有待克服的技术障碍；四是需要创新的优化方法来解决网络、I/O 等硬件模块在超大规模扩展时遇到的问题。

为解决以上挑战，本文以我国最快的超级计算机—采用国产“申威 26010”异构众核处理器的“神威 · 太湖之光”为目标平台，针对深度学习训练任务提出了一套系统化的软件设计方法。为了更高效进行开发和调优，本文采用模块化的软件组织方法，将深度学习训练系统分解为矩阵乘法、深度学习算子、自动代码调优和并行通信等功能模块。具体来说，本文的主要贡献如下：

第一，本文设计了适合“申威 26010”创新体系结构特性的性能分析模型和张量化编程模型。在性能分析模型指导下，使用以张量为操作目标的访存和计算原语来表达算法，可以弥合硬件使用方式和算法设计之间的鸿沟。为了实现张量化编程模型所需要的关键计算原语，本文提出了面向众核核间通信机制的矩阵乘法。

第二，本文将性能分析模型和张量化编程模型应用于深度学习计算核心的优化中，并提出了自动化的代码调优方法来减少工程负担。首先，在“申威 26010”上设计了常见的复杂深度学习算子优化方法，包括卷积、全连接、LSTM 等。然后，为深度学习算子设计了端到端的自动调优和代码生成方法，使优化后的算子实现获得了超过 GPU 上 cuDNNv7.5 的运算效率。

第三，本文研究了超级计算机上深度学习并行训练的关键技术，在系统和算法层面突破了扩展瓶颈。系统层面上，本文在“神威 · 太湖之光”上实现了一个并行训练框架，通过对网络通信、I/O、内存管理等方面定制优化后，可以在 1024 节点上完成目前常用的深度学习模型的训练任务。算法层面上，本文使用残差梯度压缩方法减少需要通信的数据量，在不损失模型精度条件下，提升了系统的可扩展性。它不仅在最新的 GPU 超级计算机上显著加速了曾经难以扩展的深度学习训练过程，还能为下一代国产超级计算机上深度学习系统软件设计提供参考。

关键词：神威 · 太湖之光；深度学习框架；自动优化；并行计算；高性能计算

Abstract

Deep learning is currently the most successful artificial intelligence technology, and it is expected to lead human beings into an intelligent era. The huge computing demand is driving the combination of deep learning and supercomputers. Since the US banned the sale of high-performance-computing chip to China, the planned next-generation Chinese supercomputers will all be manufactured using domestic many-core processors. However, there is still no research on deep learning software systems designed for the domestic supercomputers, and the implementation of such a system faces many challenges: First, there is no systematic optimization guidance designed for the innovative hardware features of the domestic processor. Second, it is hard to map complex computation patterns of deep learning to the new architecture. Third, compilation tools and system libraries on the domestic supercomputers are hard to be leveraged. Fourth, innovative optimization methods are required to solve the problems of hardware modules such as network and I/O when scaling to large-scale.

In order to solve the above challenges, this thesis proposes a systematic methodology of building a deep learning system on Sunway TaihuLight, which is the most powerful Chinese supercomputer adopting a domestic heterogeneous many-core processor called SW26010. In order to develop a deep learning system more efficiently, a modular software design is adopted which decomposes the system into different functional modules, including GEMM, deep learning operator, automatic code-tuner and network communication. The main contributions of this thesis are as follows:

First, a performance analysis model and a tensorization programming model customized for the innovative features of SW26010 architecture are proposed. Under the guidance of the performance analysis model, the tensorization programming model expresses the optimal algorithm workflow as a combination of tensor-operated memory access and computation primitives. In this way, the gap between hardware usage and algorithm design is easily bridged. In order to implement the important GEMM primitives, a matrix multiplication algorithm based on register communication feature of SW26010 many-core processor is designed.

Secondly, applying the performance analysis model and the tensorization programming model, a set of deep learning operators are optimized on SW26010, including convolutional, fully-connected, LSTM operators. In addition, an end-to-end automatic code tuning method is proposed to reduce the engineering burden. As a result, the computational efficiency of tuned operators on SW26010 is better than cuDNNv7.5 of GPU.

Thirdly, this thesis studies the key techniques of scaling deep learning training on supercomputers, and breaks through the bottleneck of scalability at the system and algorithm level. At the system level, a parallel training framework is implemented on Sunway TaihuLight. After optimizing of the modules such as network communication, I/O, memory management, It is able to train the popular deep learning models on 1024 node scale. At the algorithm level, to reduce the amount of data to be communicated, a data parallel method based on residual gradient compression is designed, which improves the scalability of the system without losing accuracy. It not only significantly speeds up the deep learning training tasks that were difficult to scale on the latest GPU supercomputer, but also provides a reference for deep learning system software design on the next generation of domestic supercomputers.

Key Words: Sunway TaihuLight Supercomputer; Deep Learning System; Automatic Optimization; Parallel Computing; High-Performance-Computing

目 录

第 1 章　绪论

1.1　人工智能与深度学习概述

如同工业时代的蒸汽机技术、电气时代的发电机技术、信息时代的计算机和互联网技术，人工智能（Artificial Intelligence，简称 AI）技术正成为推动人类社会进入智能时代的主要动力。人工智能[1-2] 是研究如何模拟、延伸并扩展人类智能的一门科学技术，它的目的是促使计算机像人一样会听（语音识别）、会看（图像、文字识别等）、会说（语音合成、人机对话等）、会思考（人机对弈、定理证明等）、会学习（机器学习、知识表示等）、会行动（机器人、自动驾驶汽车等）。从应用角度，人工智能可以分为专用人工智能和通用人工智能，专用人工智能系统是面向需求明确的单一问题（比如识别人脸、下围棋等）的智能系统，通用人工智能系统则如人类大脑一样，能够处理视觉、听觉、推理、学习等各类问题，达到举一反三、融会贯通、“一脑万用”的效果。

我们尚处在从专用人工智能领域向通用人工智能过渡阶段。人类近年来在专用人工智能领域取得突破性成绩，首先要归功于深度学习（Deep Learning）技术。2018 年的“图灵奖”被授予推动深度学习发展的三位关键人物—Yoshua Bengio, Geoffrey Hinton 和 Yann LeCun。正如颁奖词①所言，“近年来，深度学习方法是计算机视觉，语音识别，自然语言处理和机器人等技术长久以来取得惊人突破的根本原因”。虽然深度学习并非人工智能唯一解决方案，但它是目前引领人工智能发展的关键因素却是不争的事实。

深度学习[3] 是一种新型的机器学习技术[4]，它可以克服传统机器学习技术在处理原始形式数据表示时的诸多不足。几十年来，构建传统机器学习系统需要所在领域的专业知识来设计相应的特征提取方法。与传统机器学习方法不同，深度学习被视为一种表示学习方法[5]，它使用模仿人类大脑的深度神经网络（Deep Neural Network，DNN）[6] 来“学习”如何从大量数据中自动地提取有用的特征和模式。因此，深度学习非常擅长发现高维数据中复杂的结构，使机器从原始数据中发现学习过程所需的中间表示。

深度学习已成功应用于众多人工智能领域。自 2009 年在 ICDAR 手写识别竞赛中②首次打破记录以来，深度学习已经在在图像识别[7-9] 和语音识别[10-11]，预测潜在药物分子的活动[12]，分析粒子加速器数据[13]，重建脑回路[14] 等众多专用智能

---

① https://amturing.acm.org/

② http://people.idsia.ch/ juergen/handwriting.html

任务上打破了传统机器学习方法的记录。甚至很多领域中，深度学习驱动的人工智能系统取得了超过人类的表现。例如，人脸识别系统超越人类的水平[15]，AlphaGo系统在围棋比赛中已经可以战胜人类冠军[16]，诊断皮肤癌人工智能系统超过了专业医生水平[17] 等等。

神经网络并非全新的概念。它的前身诞生于 1942 年，最早仅仅是一个从神经科学角度出发设计的简单线性模型[18]。线性模型有很多局限性，其中无法学习异或运算的缺陷直接导致了神经网络研究热潮的第一次大衰退。神经网络的第二次浪潮在 20 世纪 80 年代出现，1986 年，反向传播理论[19] 指明了有效训练神经网络模型的方法，将神经网络再一次拉回到了人们的视野。但是，当时人们仍无法解决将网络变深带来的诸多问题：计算机的计算能力还远远达不到深度神经网络的运算需要，深度学习赖以施威的海量数据集还没有准备好。相比传统机器学习方法，神经网络在精度和速度上都没有显著优势，此时人们对它只是浅尝辄止，并未成为人工智能领域研究的主流。

到 2010 年前后，深度神经网络所依赖的计算能力和高质量的数据得到了满足，神经网络才迎来真正的复苏。一方面，计算机此时已经能够提供相对强大的计算能力。1965 年提出的摩尔定律[20] 曾指出：集成电路芯片上所集成的电路的数目每隔 18 个月就翻一番。自 1970 年到 2005 年，通用处理器 CPU 的运算速度都如此定律预期那样增长。另外，新出现的以图形处理器 GPU 为代表的众核协处理器为深度学习运算提供更强大的算力支持。在 2010 年前后，计算机的计算能力已经可以支持一些初级的深度神经网络的训练任务。另一方面，科研人员也意识到数据规模和质量对深度学习效果的影响。如图1.1，和传统机器学习方法相比，深度学习只有在更大的数据规模上才能显现它们的威力。随着互联网时代到来，各式终端设备和传感器每天都在互联网上制造海量数据，获取高质量大规模的数据因此变得更加容易。在传统机器学习方法研究阶段，人们就认识到了数据对于模型表现的重要性，更多的训练数据意味着可以得到更好的模型，而对于深度学习来说，这种情况尤为明显。相比传统机器学习在大规模数据下相对有限的性能提升，深度学习性能提升更加显著。一个典型例子是 ImageNet 数据集[21]，它包括超过 320 万张 5247 类的高清图片和它们的分类标注。2012 年，AlexNet[7] 在 ImageNet 数据集上使用深度学习显著提升了传统机器学习的记录，从而导致深度学习从此而一炮走红。近年来，Google 研究人员[22] 发现深度学习在视觉任务的性能表现随着数据量的大小以对数方式增加。鉴于 ImageNet 的巨大成功，许多其它领域，如人脸识别、目标检测、手写体识别、文本处理、语音识别等，在近十年来也建立了有助于深度学习研究的大型公开数据集。

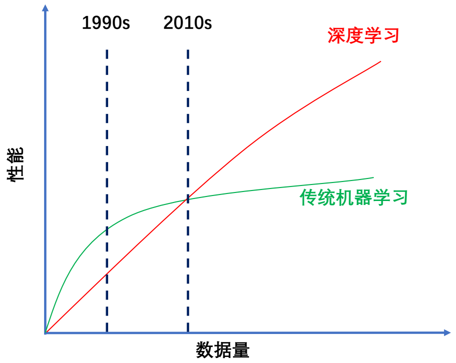


图 1.1　数据规模对模型性能的影响示意。

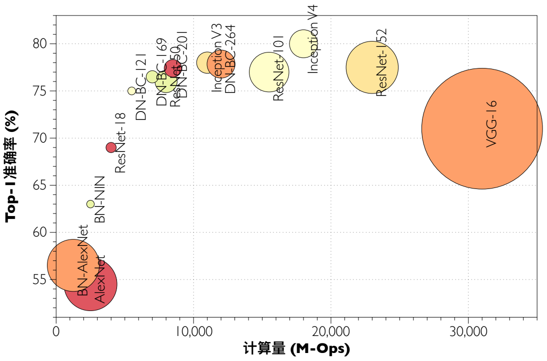


图 1.2　计算机视觉应用领域，深度学习模型的计算量和模型表现的关系，气泡大小反映模型尺寸。

有了提供强大算力的计算硬件和大量高质量的训练数据做基石，近十年来各式创新的深度学习算法如雨后春笋般涌现。通过使网络变得更深、更复杂，深度学习在各个领域攻城掠地，不断刷新着各种基准测试的记录。然而，相比应用上的空前成功，深度学习近十年来在理论方面的突破却乏善可陈。人们仍然没有全面了解神经网络的优化过程和内部组织结构的联系，深度神经网络的训练技术仍然被 80 年代的反向传播算法所支配。网络结构的设计原则、超参数选择方法等没有有效的理论支持，深度学习常常被当做神秘的“黑匣子”来使用[23-24]。目前最佳实践仍然是组合一些经验上的技巧[25]，寻找更好表现的网络结构的唯一办法是通过消耗计算资源来进行大量试验，因此，业界和学界将训练深度神经网络比作古代的“炼丹术”。正如量子力学的本质虽然尚无定论，但是并不妨碍它在半导体、量子计算、激光、电子显微镜等领域的成功应用。虽然深度学习理论突破有待时日，但其应用层面进步的脚步并不会停息。如同物理学家以“Shut up and calculate！①”态度来推动量子力学应用的发展，计算机科学如今正在以“Shut up and experiment！”的方式驱动着深度学习的进步。

既然深度学习是计算驱动的技术，随着数据量增多、模型更加复杂，计算资源会逐渐成为深度学习技术进步的瓶颈。一方面，训练单个网络任务的计算需求越来越大。如图1.2展示了在计算机视觉领域，主流深度学习模型的计算量，尺寸和学习效果之间的关系。可以观察到，为了获得更好学习效果，需要更大的计算资源和更多的存储资源对训练过程给予支持。即使现在的深度神经网络看似已经很庞大了，但从神经元的个数来讲，它们比无脊椎动物（如青蛙）还要少。而真正想达到人类大脑的水平还需万倍的神经元，还需要大幅投入相应规模的计算和存

---

① 这句物理学领域广为流传名言来自于 N. David Mermin 的文章 Physics Today 42, (1989); doi: 10.1063/1.2810963

储资源。另一方面，“以计算换智能”将逐渐成为未来深度学习发展的趋势，算法创新也需要大量计算资源支持。对于一个新的任务，深度学习的网络结构、参数调整都需要反复试验来调优。为了使实验人员即使不具备深度学习专业知识也能训练出高质量的深度学习模型，近年来基于 NAS（Neural Architecture Search）[26] 方法的自动网络结构搜索也引起人们关注。该方法的初步版本就调用了 450 块 GPU，且需要 3-4 天的训练才能找到表现优异的网络结构。综合以上两点可知，深度学习研究对计算的需求是没有止境的。

1.2　超级计算机系统概述

纵观深度学习理论的发展历史，人们已经逐渐意识到计算能力才是真正制约未来深度学习技术发展的关键因素。但是，随着集成电路制造工艺极限的限制，摩尔定律所带来的单个芯片性能提升红利即将终结。单个计算节点早已无法满足深度学习的训练需求，多节点并行训练单个网络已成深度学习的标准配置。为了满足未来计算需求，扩展并行规模势在必行。

而要说地球上能提供最大计算资源的硬件，非超级计算机（简称“超算”）莫属。自 1965 年第一台超算 CDC 6600[27] 被设计制造以来，其组织方式先后经历向量机 (Vector Machine)[28]、对称多处理机 (Symmetric MultiProcessing)[29]、大规模并行处理机 (Massively Parallel Processing) [30] 等若干次重大演进。如今，主流超算均采用由数以万计的处理器组合而成的大规模并行集群（Cluster，如美国的 Summit 和中国的“神威 · 太湖之光”超算）的方式。目前最快的超级计算机 Summit 可以提供 142 PFLOPS 的运算性能，对它来说 1 分钟进行的运算任务，个人笔记本电脑运算则需 30 年才能完成。

随着功耗墙的阻碍，超算的计算部件已从传统通用处理器逐渐替代为异构众核处理器。异构众核架构[31] 通过采用众多低频简化的核心作为运算单元辅佐通用处理器的方式，可以保证在相对较低功耗情况下提供较高的计算性能。如表格 1.1所示，最新（2018 年 11 月）世界超算 Top-500 榜单[32] 的前 10 名中的 8 台超算都采用异构众核处理器。排名第一的 Summit，第二的 Sierra，第五的 Piz Daint，第七的 ABCI 的超算都采用了 CPU+NVIDIA GPU 的异构组织方式。排名第四的天河-2A 采用 CPU+ 国产众核协处理器，排名第六的 Trinity 超算采用 CPU+Intel 众核协处理器的异构组织方式。神威太湖之光采用片上异构的组织方式，这是本文第 3章详细介绍的对象。只有排名第八和第十的超算采用同构的组织结构。

超级计算机在科学计算领域发挥着重要作用，常被用来解决包括量子力学[33]、天气预报[34]、气候研究[35-36]、地震模拟[37]、海洋科学[38]、生物制药[39]、宇宙科

表 1.1　2018 年 11 月 Top-500 超算榜单前十名

排名	名称	国家	架构	核心数	持续运算性能
1	Summit	美国	POWER9+Volta GV100	2,397,824	143.5 PFLOPS
2	Sierra	美国	POWER9+Volta GV100	1,572,480	94.6 PFLOPS
3	太湖之光	中国	申威 26010	10,649,600	93.0 PFLOPS
4	天河-2A	中国	Xeon+Matrix2000	4,981,760	61.4 PFLOPS
5	Piz Daint	瑞士	Xeon+Tesla P100	387,872	21.2 PFLOPS
6	Trinity	美国	Xeon+Xeon Phi 7250	979,072	20.1 PFLOPS
7	ABCI	日本	Xeon+Tesla V100	391,680	32.6 PFLOPS
8	SuperMUC-NG	德国	Xeon	305,856	19.7 PFLOPS
9	Titan	美国	Opteron 6274+K20x	560,640	17.6 PFLOPS
10	Sequoia	美国	Power BQC	1,572,864	17.2 PFLOPS

学[40-41] 和密码分析[42] 等领域科学问题。近年来，深度学习已经逐步成为各大超算的“杀手应用”。使用 Cori 超算的 9600 个节点训练的深度学习模型被用来对气候数据中极端天气进行定位和分类[43]。在 Cori 超算工作基础上，使用 Piz Daint 超算的 5300 个 GPU 和 Summit 的 27360 个 GPU，深度学习技术正被用来识别气候模拟中的极端天气现象，如热带气旋，大气河流等[44]。另外，使用 Cori 超算的 8000 个节点训练的深度学习模型被用来从暗物质模拟的结果中预测宇宙学参数，并达到了前所未有的精度[41]。在 Titan 的 18000 节点上，深度学习还被用来分析中微子与普通物质相互作用。在国产的天河-2A 上也运行着一系列深度学习应用程序[45]，包括肿瘤诊断，视频分析和智能交通等。

随着深度学习技术的普及，它和超算的结合将变得越来越紧密。2018 年 11 月刚刚发布的世界第一、第二、第七超算 Summit、Sierra、ABCI，在制造时就宣称是为 AI 定制的。如果使用 FP-16 浮点运算精度，Summit 已经可以为深度学习应用提供 3.3 Exaops 的运算能力。制造真正“E 级”（提供 EFLOPS 的双精度浮点计算能力）超算系统需要突破内存墙、功耗墙、通信墙等诸多困难，因此软硬件协同设计愈发重要。目前，各国的 E 级超算计划中都将人工智能列为未来机器核心应用来协同设计芯片架构和组织方式。在中国下一代超算系统制造计划中，在兼顾传统数值模拟应用的同时，明确指出将深度学习设为重点支持的目标[45]。

1.3　基于国产超算的深度学习训练系统：机遇与挑战

超级计算技术和深度学习技术作为打开智能时代大门的两把钥匙，也是国家之间进行科技角力的两把利器。自 1983 年银河 1 型超计算机制造以来，国产超算

技术已经经过了 36 年的发展。恰如美国的 Intel、IBM 和 Cray 三大超算公司形成了三足鼎立的局面，我国的天河、神威和曙光系列已经出产了一系列世界领先超算--“曙光 4000A”（2005，11.2 TFLOPS）；“天河-1A”（2011, 4.7 PFLOPS，Top-500 冠军一次）；“神威蓝光”（2011，1PFLOPS）；“天河-2A”（2013，33.83 PFLOPS，六次 Top-500 冠军）；“神威 · 太湖之光”（2016，93.0 PFLOPS，四次 Top-500 冠军）。在最新的 2018 年 11 月 Top-500 榜单中，中国超算的总数量已经超过美国，居全球首位。

尽管具备了领先的超算制造技术，但是我国超算制造所使用的芯片长期依赖美国进口。曾经的世界冠军超算“天河-1A”和“天河-2A”分别采用美国 NVIDIA 和 Intel 公司制造的众核协处理器。核心关键技术是要不来、买不来的、讨不来的，2015 年，美国政府以“存在或参与违反美国国家安全或外交政策利益的重大风险”为由，决定禁止向中国超算中心出售 Intel Xeon 系列高端芯片。2018 年，美国商务部曾禁止美国企业向中国电信制造商中兴通信销售芯片在内的零件。这让我们意识到中国超算不能完全依赖进口芯片，必须让“中国芯”走上自主研发的道路。2016 年，我国发布了世界首台设计性能超过 100 PFLOPS 的“神威 · 太湖之光”超算，它完全采用国产异构众核架构芯片。2017 年，“天河-2A”的升级过程中，使用国产众核协处理器飞腾 Matrix 2000 替换 Intel 制造的 Xeon Phi 协处理，性能从 33.83 PFLOPS 提升至 61.4 PFLOPS。在中国下一代超算计划中，中国三大超算系列都将采用国产众核处理器作为主要运算部件。神威系列将使用下一代申威异构众核处理器；天河系列将使用国产飞腾处理器；曙光系列将基于 AMD 授权的技术来设计自主高性能处理器。与此同时，在超算领域，中国正抱着更积极的态度寻求国际合作。中国两家超算中心的研究人员撰文[45] 指出“中国的超算社区愿意广泛参与国际合作来实现 E 级计算的目标，超算不应该是一种新的军备竞赛，而是让所有人受益的技术”。

在人工智能技术领域，近年来有关中美之间存在人工智能技术竞赛的说法甚嚣尘上。中国政府在 2017 年发布了《新一代人工智能发展规划》[46]，计划 2030 年在人工智能理论、技术与应用达到世界领先水平。对此，美国学者[47] 甚至出于人工智能的军事用途考虑，将由中国人主导的人工智能未来类比冷战时期的“苏联人造卫星时刻”。所谓匹夫无罪，怀璧其罪，只有提升自身科技实力，才能掌握创新发展的主动权，才有参与国际合作的话语权。由于海量人口和发达的互联网基础设施，中国在数据方面占有人工智能领域的创新优势。据估计，2020 年中国将拥有全球数据的 20%，到 2030 年这个比例将提升至 30%[48]。相比数据方面具有的先天优势，牛津大学学者[49] 认为计算能力反而会成为制约中国人工智能 2030 计

划的一大短板。而利用国产超级计算机来满足我国未来对深度学习的计算需求正是一个重要突破口。

表 1.2　神威太湖之光的主要性能指标

峰值计算速度	125.436 PFLOPS	**持续运算速度**	93.015 PFLOPS
内存总容量	1024TB	**访存总带宽**	4473.16 TB/s
系统功耗	15.37 MW	**性能功耗比**	6051.13 MFLOPS/W

利用我国现阶段世界领先的国产超算技术，来提升我国人工智能发展水平具有重要意义。本文选择以中国**第一台全部采用自主技术构建的超级计算机**—“神威·太湖之光”为目标，探索其进行人工智能计算的能力。“神威 · 太湖之光”蝉联了第 47 届 ~50 届（2016 年 6 月 ~2017 年 11 月）世界 500 强超算榜单冠军。“神威 · 太湖之光”的运算核心采用国产“申威 26010”①处理器，它已经达同时代发布的 NVIDIA 和 Intel 的众核架构芯片的制造水平和性能指标[50]。“神威 · 太湖之光”由超过四万块“申威 26010”全新异构众核处理器组成，峰值运算速度达到每秒 12.54 亿亿次，是世界上第一台峰值运算速度超过 100 PFLOPS 的超级计算机。表3.1具体展示了该系统的各项性能指标。

具备硬件制造水平之后，国产超算还需要高性能软件系统来配套，才能让超算真正具备解决应用问题的能力。“行百里者半九十”，对于超算发展而言高性能软件系统的开发与整个硬件系统制造同等重要。一个典型的表现是，在 Cray、Intel 这样的大型超算制造公司，软件与硬件工程师的比例高达 4:1 甚至 5:1。相比其它众核架构上高性能软件设计方法上的长期技术积累，面向“神威 · 太湖之光”的系统软件设计的理论方法仍相对落后，亟待我们探索。在“神威 · 太湖之光”上“白手起家”搭建高效深度学习并行训练系统主要面临如下挑战。

第一，任务部署层面挑战：如何设计灵活高效的软件系统组织方式来支持深度学习训练任务。深度学习训练任务和传统数值模拟任务有很大区别。传统数值模拟程序的代码结构往往是静态不变的，它们的优化方式可以归纳为不断迭代“热点分析-提取热点-手动优化热点”的过程。“神威 · 太湖之光”上许多数值模拟软件的开发，如大地震模拟[37]、地球系统模式[51-52]、大气动力学[35] 等，都依赖于这种优化范式。深度学习软件执行的代码结构是动态变化的，由于用户的需求不同，有时甚至执行完全不同的代码结构。因此，深度学习系统软件优化方法要考虑通用性和泛化性，不仅要对已经见过的网络模型能够高效运行，对于新定义的全新模

---

① 因其制造单位“上海高性能集成电路设计中心”位于上海，故取名为“申”，同时也有与“神威”双关之意。

型也要能够充分支持。另外，相比传统高性能计算应用，深度学习的算法变化速度快，因此系统软件要能够灵活增加新的功能，及时支持算法变化。

第二，硬件架构层面挑战：如何用好全新的国产众核处理器的硬件架构特点来充分发挥它的计算潜力。不同于主流的GPU或者Xeon Phi众核架构，“申威26010”处理器有许多创新的硬件特性，比如分布式的Cache组织方式、核间寄存器通信机制等，因此它需要全新的编程模型来指导并行算法设计。相比NVIDIA和Intel众核架构卷帙浩繁的编程指南和汗牛充栋的研究成果，对申威架构的使用方式的研究还十分初级。在优化深度学习应用之前，需要对硬件的高效使用方式进行深入的研究。

第三，算法设计层面挑战：如何设计从深度学习复杂计算核心到申威异构众核体系结构的映射方法。首先，深度学习的计算热点并不集中，同时存在访存密集型和计算密集性的运算核心，这就需要一对一地针对不同核心定制化设计并行算法才能充分发挥体系结构特性。如果映射方法不能顾及各种算子的特殊访存计算模式，则性能不佳的算子实现会严重拖慢整个训练过程。其次，每个深度学习计算核心设计都需要兼顾效率和通用性。深度学习的目标数据结构为多维张量，理论上每种参数条件下最优的并行算法实现都是不同的，因此同时变动的多个维度参数将给并行算法设计的健壮程度带来巨大的考验。

第四，工具生态层面挑战：如何弥补国产异构处理器在编译工具和系统库方面存在的劣势。不同于商业化运作的硬件公司，如Intel、NVIDIA、IBM，在技术层面有数十年积累，并维持大量工程师团队来迭代优化编译工具和系统库构成的软件生态。国产众核处理器由于问世时间较短、维护人员有限等因素，编译工具常常无法和其他商用众核处理器相比。对极致的性能追求，往往意味着大量的人力成本，如何利用更加自动化的方式完成代码调优将是一个开放性的难题。

第五，并行扩展层面挑战：如何高效扩展深度学习训练任务到多个计算节点并行执行。随着扩展规模增大，深度学习的训练在系统和算法层面都面临巨大压力。在系统层面，扩展规模增大导致通信总量、I/O总量也会随之增大，训练任务的主要瓶颈会逐渐从计算硬件转移到网络和I/O上。比如，为了连接数以万计节点，“神威·太湖之光”的互联网络采用了独特设计，如果直接应用传统的通信优化方法会面临“水土不服”的困境。在算法层面，考虑到超算处理器计算能力的提升速度远快于网络通信带宽的提升，未来将会面临即使非常充分利用网络传输带宽也无法消除通信瓶颈的情况。亟待解决的难题是如何通过并行算法和并行系统协同设计，在不损失训练精度前提下，从根本上减少对网络硬件带宽的需求。

1.4　本文主要贡献和行文结构

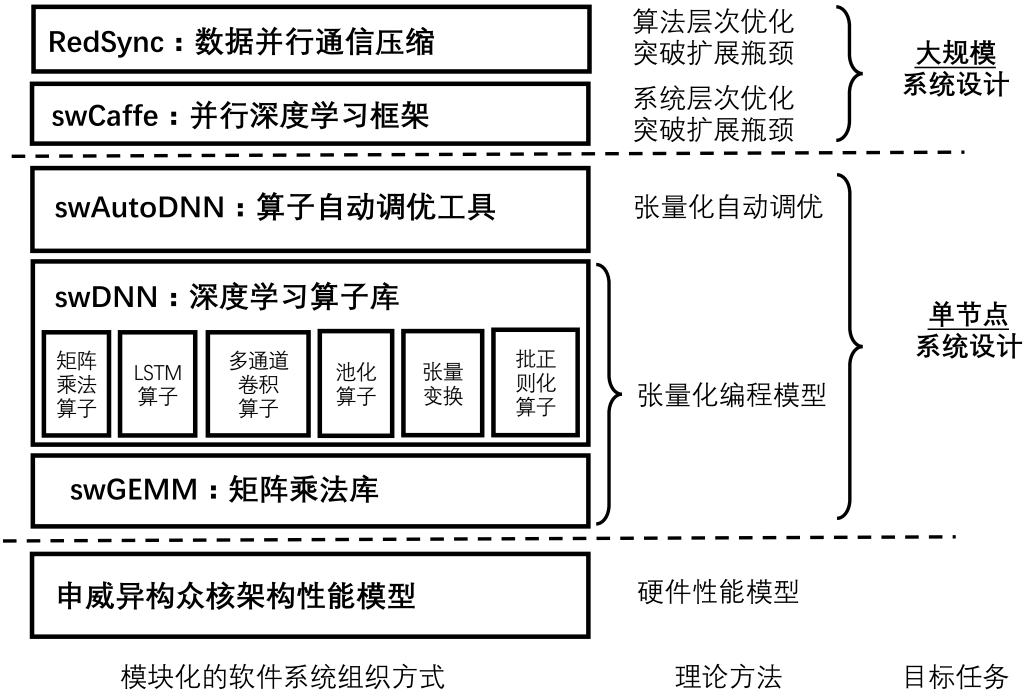


图 1.3　本文各章节在软件系统组织、理论方法和目标任务层面的关系

本文将首先介绍深度学习训练方法和相关工作（详见第 2章）。然后展示一套以“神威 · 太湖之光”超级计算机为目标平台的系统性深度学习并行训练方法。图1.3展示了本文的组织结构和主要贡献之间的对应关系。

贡献一：本文提出了针对深度学习训练系统的**软件组织模块化**方法。通过将系统分解为耦合度比较低的不同模块，可以更好管理软件质量，控制开发进度。这些模块包括矩阵乘法模块、深度学习算子模块，自动代码调优模块和网络通信模块。

贡献二：本文提出了**硬件性能模型**和**张量化编程模型**来指导“申威 26010 上”并行算法和硬件的映射方法。通过对申威异构众核处理器硬件特性进行深入分析，提出了一套性能分析模型来指导算法的设计和调优。如同向量化编程模型之于 SIMD 众核架构，本文针对核间互联特性的 MIMD 众核架构提出张量化编程模型（详见第 3章）。鉴于最新深度学习专用芯片的设计也采用了相似的核间通信网络，如 Google 的 Tensor Processing Unit（TPU）和 NVIDIA 的最新款 Turing 架构 GPU 中的 Tensor Core，本文提出的思路对它们也有借鉴意义。

贡献三：本文将性能分析模型和张量化编程模型应用于深度学习算子优化中，并提出了**张量化自动调优**方法来减少工程负担。本文提出了基于众核核间通信的

矩阵乘法方法（详见第 4章），并使用它实现了 swGEMM 矩阵乘法库，相对 BLAS 获得显著提升。本文提出一套名为 swDNN 的面向申威异构众核处理器深度学习算子的优化方法（详见第 5章）。本文提出了一个端到端自动化工具 swAutoDNN 来为深度学习算子张量化优化进行自动调优和代码生成（详见第 6章）。不仅显著提升手动优化的性能和通用性，采用本文的方法对复杂的计算密集型算子实现效率甚至远超过同代 GPU Kepler 架构上 cuDNN 算子库。

贡献四：本文解决了“神威 · 太湖之光”上增加深度学习并行训练系统扩展性的关键难题。通过针对“神威 · 太湖之光”硬件系统特点为通信、I/O、内存等模块进行相应的定制设计，本文实现了名为 swCaffe 的深度学习并行训练框架（详见第 7章），对使用 ImageNet 为数据集的深度学习训练任务进行了目前极限规模的扩展实验。

贡献五：本文研究了有助于增加深度学习并行训练系统扩展性的通信压缩算法。本文提出了一种名为 RedSync 的通信压缩方法（详见第 8章），在保证模型精度和收敛速度的同时，可以大幅减少通信带宽需求。此方法不仅可以应用于最新的 GPU 超算，还可以成为下一代“神威”系列超算[53] 上深度学习系统软件设计的参考。

本文最后对研究工作进行了总结，并对未来研究方向进行了展望（详见第 9章）。

第 2 章　研究背景及现状分析

2.1　深度学习训练方法

神经元是构成神经网络的基本单元，它对输入数据进行加权求和，然后经过一个非线性激活函数 σ 得到输出结果。如图 2.1右所示，深度学习中若干神经元排列成网络层（Layer）的结构，它们之间互相没有连接，但与前一层神经元的输出和后一层神经元的输入建立连接。网络层的输入和输出都表示成多维张量（Tensor）的形式，输入张量经过网络层变换得到输出张量的运算方式被称为**深度学习算子**（Deep Learning Operator）。网络层神经元的连接方式决定了深度学习算子计算方式。比如，全连接层（Fully-Connected Layer, FC Layer）的前后层神经元完全互连接，采用矩阵乘法作为算子计算方式。卷积层（Convolutional Layer, CONV Layer）的前后层神经元以稀疏共享权重方式连接，采用多通道卷积作为算子计算方式。深度学习算子的高效实现方式是本文第 5章研究重点。届时，笔者会具体介绍各种网络层对应算子的计算方式。

深度神经网络的计算过程可以表示为计算图的形式：图中的边表示算子计算；点表示网络层输入输出张量。根据网络对应的计算图是否有环，神经网络模型目前主要分为前馈神经网络（Feedforward Neural Network）和循环神经网络（Recurrent Neural Network，RNN）两类。

前馈神经网络的计算图把算子和张量按照有向无环图形式组织起来。因为输入数据按照顺序流经各个网络层就可以得到输出结果，所以被称为前馈的。比如，图2.2展示了目前最成功的前馈神经网络变种—卷积神经网络（Convolutional Neural Network，CNN），它是为了处理以图像信息为代表的网格结构数据而设计的，深度学习在视觉领域的巨大成功主要得益于卷积神经网络出色表现。图中展示了经典 CNN 组织方式，其中各个网络层首尾相连，呈线性的计算图结构。近年来出现了许多更加复杂的前馈神经网络计算图组织方式，比如间隔较远节点间有互联通路（ResNet[54]、DenseNet[55] 等），或某个节点后产生多个分支然后再汇聚在一起（Inception[56]、GoogleNet[57]）。

如图2.3所示，循环神经网络（RNN）的计算图把算子和神经元按照有环的方式组织。将计算图递归地展开，最终也可以得到一个所有层共享网络权重的前馈网络。在 RNN 中，将展开后的一个网络层称为一个时间片。正是由于 RNN 中不同时间片共享相同的中间状态，从而可以持久地记忆学习到的参数信息。RNN 非常适合处理序列结构的数据，比如文本处理和语音识别。早期的 RNN 主要采用首尾

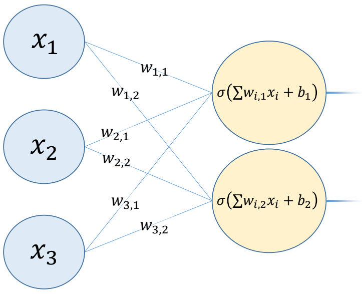


图 2.1　深度学习网络层示意

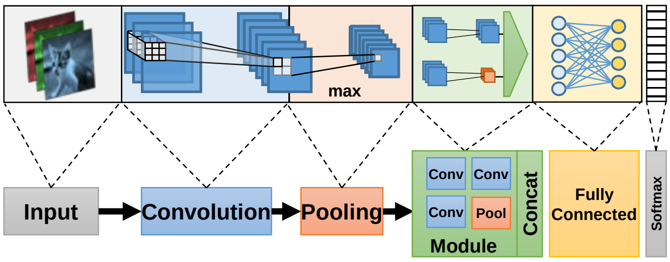


图 2.2　简单的前馈神经网络示意

循环连接若干全连接层的方式搭建，对于非常长的序列，这种 RNN 容易产生梯度散失或梯度爆炸的问题[58]，导致无法训练出正确的网络模型参数。目前，基于门控 RNN 可以消除上述缺陷，它们采用长短期记忆单元（Long Short-term Memory，LSTM）[59] 和基于门控循环单元（Gated Recurrent Unit，GRU）[60] 作为算子，这种算子依靠若干门电路来控制信息流动，每个时间步可以遗忘或积累信息，从而使网络变得易于学习。

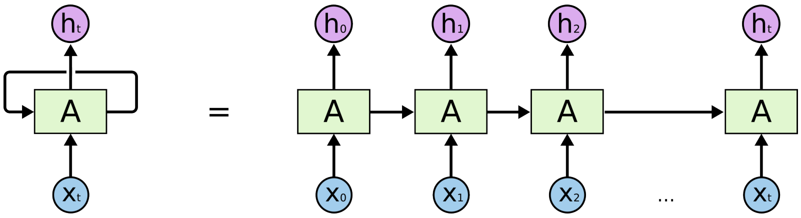


图 2.3　循环神经网络（RNN）的计算图展开方式示意

深度学习目前应用非常广泛，覆盖了有监督学习（Supervised Learning）、无监督学习（Unsupervised Learning）和强化学习（Reinforcement Learning）三大类主要机器学习问题。有监督学习用神经网络来寻找数据和标注之间的映射关系，它的应用最为广泛，因此也是本章接下来介绍的重点。无监督学习用来学习输入数据的隐藏特征，它的两个主要变种分别是自动编码器（Autoencoder）和生成对抗网络（Generative Adversarial Networks，GAN）。自动编码器[61] 通过训练神经网络使模型预测的结果和输入数据相同。GAN[62] 是近年来才提出的深度学习问题，它通过一个生成器神经网络和一个鉴别器神经网络互相对抗地来训练彼此，最终达到输入随机噪声也能生成与输入数据分布相同结果的效果。GAN、自动编码器的训练方式和有监督学习类似，都采用反向传播算法，因此本章介绍的训练优化方法对它们同样适用。与有监督和无监督学习相比，强化学习中使用神经网络的方式则比较特殊，它并没有明确的输入输出数据对应关系，而是通过和环境的交互，来学习行动方式以最大化目标收益。训练强化学习的方法不同于有监督和无监督

学习，通常采用 Deep Q Learning[63] 和 A3C[64] 算法而非反向传播。强化学习的训练超出了本文讨论范围。

有监督学习的目的是找到分布 $\mathcal{D}$ 上输入数据 z 和输出结果 $h(z)$ 的映射函数 f^*。深度神经网络学习出一个映射函数 $f_w(w)$，使之尽可能地近似 f^*。网络训练过程是为了寻找到参数 w，使之能最小化公式2-1中目标函数 $L_{\mathcal{D}}(f_w)$ 的泛化误差，其中 z 是概率分布 $\mathcal{D}$ 中的一个样本; $\mathcal{H}$ 为参数 w 的所有可行解；$\mathbb{E}$ 表示概率的期望；ℓ 是损失函数。

$$w^* = \mathrm{argmin}_{w\in\mathcal{H}} L_{\mathcal{D}}(f_w) = \mathrm{argmin}_{w\in\mathcal{H}} \mathbb{E}_{z\sim\mathcal{D}}\left[\ell(w,z)\right], \tag{2-1}$$

损失函数 ℓ 用来描述模型预测结果和真实标注之间的偏差，为使优化问题可解，它必须是连续可导的。深度学习中最常用的损失函数是交叉熵损失函数。首先，网络的输出分布 z 用 *softmax* 函数 $\sigma(z)_i = \frac{\exp(z_i)}{\sum_k \exp(z_k)}$ 进行归一化。然后，使用交叉熵损失函数计算预测与真实标签“分布”的差异：$\ell(w,z) = -\sum_i h(z)_i \log\sigma(f_w(z))_i$。优化问题可以通过梯度下降法（Gradient Descent）来求解。由于我们无法完全观测到概率分布 $\mathcal{D}$，因此获得其无偏梯度估计是不可行的，可以采用随机梯度下降法（Stochastic Gradient Descent，SGD）从数据集中随机抽取的样本 z 来估计损失函数的梯度期望 $\mathbb{E}_{z\sim\mathcal{D}}[\nabla\ell(w,z)]$。对于数据集 S 样本数很多的情况，它需要很多次迭代才能收敛。因此，深度学习最通用的优化方法是批量随机梯度下降方法 (minibatch Stochastic Gradient Descent，minibatch SGD)。算法 1描述了它的流程，这种方法每次从数据集 S 中随机抽取一个批量（minibatch）大小为 $N << |S|$ 的实例，通过计算它们的梯度来更新当前参数。

Algorithm 1 Minibatch 随机梯度下降法

```
for $t = 0$ to $\frac{|S|}{B} \cdot epochs$ do
    $\vec{z} \leftarrow$ 从数据集 $S$ 中随机抽样 $B$ 个数据实例 //获取数据
    $w_{mb} \leftarrow w^{(t)}$ //加载参数
    $f \leftarrow \ell(w_{mb}, \vec{z}, h(\vec{z}))$ //网络正向传播
    $g_{mb} \leftarrow \nabla\ell(w_{mb}, f)$ //使用反向传播计算梯度
    $\Delta w \leftarrow u(g_{mb}, w^{(0,\dots,t)}, t)$ //参数更新规则
    $w^{(t+1)} \leftarrow w_{mb} + \Delta w$ //存储参数
end for
```

算法1中梯度更新规则 u 对于收敛速度和训练精度有很大的影响。表格2.1展示了目前主流的权重更新规则。梯度更新可以看成以本次迭代梯度和历史参数

表 2.1　深度学习 SGD 算法的权重更新规则

方法	$w^{(t+1)}$ **更新方法**	**定义**
Learning Rate	$w^{(t)} - \eta \cdot \nabla w^{(t)}$	$\nabla w^{(t)} \equiv \nabla \ell(w^{(t)}, z)$
Adaptive Learning Rate	$w^{(t)} - \eta_t \cdot \nabla w^{(t)}$	
Momentum[65]	$w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)})$ $-\eta \cdot \nabla w^{(t)}$	
Nesterov Momentum[66]	$w^{(t)} + v_t$	$v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad[67]	$w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t}} + \varepsilon}$	$A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)}\right)^2$
RMSProp[68]	$w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t}} + \varepsilon}$	$A'_{i,t} = \beta \cdot A'_{t-1} + (1 - \beta) \left(\nabla w_i^{(t)}\right)^2$
Adam[69]	$w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}} + \varepsilon}$	$M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m)\left(\nabla w_i^{(t)}\right)^m}{1-\beta_m^t}$

$w^{(0)}....w^{(t)}$ 为输入的函数。最简单的方式 Learning Rate 使用梯度和学习率 η 乘积的相反方向改变可学习参数。其他流行的权重更新规则包括动量（Momentum）的方式，它在更新中加入了当前和过去参数之间的差 $w^{(t)} - w^{(t-1)}$ 来避免参数更新在局部最小值周围的反复冗余震动。AdaGrad，RMSProp，Adam 可以自适应地调整参数中每个元素学习率，从而对某些部分进行稀疏更新。

求解算法1中第 5行的梯度大小需要通过反向传播方式进行计算。首先，通过一次正向传播（Forward Propagation）获得网络的损失值（Loss），正向传播过程的中间结果被称为**特征图**。然后，进行反向传播（Backward Propagation）得到每个网络层参数的导数，即模型的**梯度**，和特征图的导数，被称为**敏感度**。对于如池化层这种没有参数的网络层则不需要计算梯度。

对于前馈网络神经，假设整个网络的参数由 n 个张量组合而成 $w = \{w^1, ..., w^n\}$，输入节点对应张量集合 $\{u^1, ..., u^p\}$，中间结果的特征图对应 $u^{p+1}, ..., u^{q-1}$，代价函数是计算图的输出节点 $L = u^q$。正向传播从输入节点开始计算其子节点，$u^i = f(Dep(u^i))$，其中 $Dep(u^i))$ 表示所有 u^i 的父节点，最后得到代价函数 L 的值。反向传播计算某项参数梯度和某个特征图对应的敏感度，方法如公式 2-2所示。对于循环神经网络，由于一个节点的父节点可能是自己本身，因此上述方法不再适用。BPTT（全称 Backpropagation Through Time）方法[70] 将循环的网络层按照一定序列长度展开成一个很长的前馈网络，并使用相同权重对每层进行反向传播计算。

$$\frac{\partial L}{\partial w^j} = \sum_{i:j\in Dep(u^i)} \frac{\partial L}{\partial u^i}\frac{\partial u^i}{\partial w^j} \qquad \frac{\partial L}{\partial u^j} = \sum_{i:j\in Dep(u^i)} \frac{\partial L}{\partial u^i}\frac{\partial u^i}{\partial u^j} \tag{2-2}$$

深度学习的训练成果是一套参数 w，推理时网络以 w 为参数对输入数据进行一次正向传播获得预测结果。尽管训练和推理任务的计算方式非常类似，但是它们的优化重点截然不同：训练端任务关注增大多节点计算的吞吐率，推理任务关注于降低单节点计算的延迟、功耗和内存使用；训练任务需要多次正反向项传播来迭代更新参数，推理任务仅需一次正向传播；训练任务计算量巨大，常常需要数天甚至数周时间，推理任务计算量非常小，常常只需要几毫秒即可完成；推理任务被部署在单节点的嵌入式设备，如 FPGA、低功耗 CPU 上，训练任务部署在多节点的众核 GPU 或其他加速设备上。虽然二者优化方法存在相当紧密的联系，由于本文的关注重点是众核处理器的训练任务优化方法，因此笔者在这里并未对推理端的优化针对性地进行阐述。

2.2　单节点深度学习训练性能优化研究

单处理器内深度学习系统优化集中在两方面，一方面关心各种深度学习算子的运行效率，另一方面关注于网络训练过程对应计算图的运算效率。本小节分别对它们的研究现状进行简要介绍。

2.2.1　深度学习算子库

卷积算子是公认最难优化的算子，并且占据 CNN 操作 90% 以上的计算时间。相比传统 CPU 架构的简单串行执行逻辑，卷积算子在众核架构上的并行优化具有更大的挑战。众核架构上卷积算子优化方法概括起来可以分为三类：

基于空间域的方法：cuda-convnet[7] 是深度卷积神经网络的开山鼻祖 AlexNet 采用的 GPU 卷积实现方法。它按照 2D 卷积定义在 GPU 上实现多通道卷积算子，但是由于它无法利用数据缓存的局部性，已经被其他优化方案取代。Chellapilla 等人[71] 提出将输入张量转化为 Toeplitz 矩阵形式，然后使用矩阵乘法来实现卷积，这种方法更广泛地被称为 *im2col* 的方式。因其可以直接利用充分优化的 GEMM（GEneral Matrix-Matrix multiplication）例程，而被早期流行的深度学习框架 Caffe[72] 采纳为 GPU 上卷积实现方案。为了减少 *im2col* 的内存移动的开销，cuDNN 早期工作[73] 将卷积隐式地转化为分块矩阵乘法操作来优化。maxDNN[74] 扩展了类似的思路，并对 Maxwell 架构的 GPU 做了定制化处理。

基于 Winograd 方法：Lavin 等人[75] 提出使用 Winograd 方法来针对小卷积核进行算法层次优化，Winograd 方法通过对输入和卷积核进行形式变换从而减少乘法次数，从而在 GPU 上相对于 cuDNNv3 取得了显著加速。

基于频率域方法：通过对输入、卷积核张量进行快速傅里叶变换（Fast Fourier Transform，FFT），空间域上卷积操作可以转换为傅里叶域上的矩阵点乘操作。fbfft[76] 介绍了在 GPU 上用 FFT 方法实现卷积操作，并给出了对 FFT 和 IFFT 操作在卷积算子中的应用给出了定制化的优化方法。ZNNi[77] 进一步优化了 FFT 方法，对输入补零的卷积进行了优化，使用剪枝 FFT 方法使特定的卷积操作在 GPU 上获得显著加速。

除了卷积操作，通用矩阵乘法—GEMM 运算也是深度学习算子中的重要操作。如上一段介绍，很多卷积操作优化需要转换为 GEMM 运算，并且矩阵乘法也是全连接层和 RNN 层的核心计算。众核架构上的基本线性代数库（Basic Linear Algebra Subprograms, BLAS）提供了 GEMM 例程的高效实现，因此它常常被视为算子库的重要补充。cuBLAS[78] 和 MKL[79] 提供了 GPU 和 Xeon Phi 架构的高效线性代库。

目前有一些工作在使用 BLAS 中的 GEMM 基础上，针对 RNN 算子进行优化。Appleyard 等人[80] 介绍了 GPU 上的一系列技巧来优化 LSTM 算子内 GEMM 和向量点乘等 BLAS 例程的调用方式。但深度学习中使用矩阵的形状常常是狭长细小的，即其中一个维度相对较小。同时，由于 BLAS 库中 GEMM 的初始目标是科学计算应用，因此有时无法适应深度学习中需要的 GEMM 计算方式。针对这种特点的矩阵的 GEMM 操作，Haidar 等人[81] 介绍了特殊形状矩阵乘法的优化方法。

目前，工业应用级的深度学习算子库的开发和维护常常被硬件制造商所承担。一方面，高效的算子库设计往往需要对底层硬件架构的精细掌控，而有些硬件使用细节，比如 NVIDIA GPU 的 Warp 调度策略，是秘而不宣的。另一方面，算子库需要不断融合各种最新的研究成果，随着硬件推陈出新而不断自我迭代，这需要大量的人力物力投入。比如，cuDNN[82] 是 NVIDIA 公司维护的闭源的算子库，Neon 是 Intel Nervana 公司维护的算子库，它们分别在 NVIDIA GPU 和 Intel Xeon Phi 上为深度学习算子进行了全面优化。

随着深度学习应用更加广泛，将深度学习计算高效部署在不同硬件上成为研究重点。尽管这类工作的研究重点是如何在嵌入式设备或者 CPU 上适应各种网络推理任务，但是众核架构上的训练任务也因此受益，产生了一系列“深度学习编译器”的研究。受到图像处理领域算子自动优化工具 Halide[83] 启发，近年来产生了一系列深度学习算子的自动优工具，比如，TensorFlow XLA[84]、Tensor Comprehensions[85]、Latte[86] 和 TVM[87] 等。

2.2.2 深度学习训练框架

在充分优化深度学习算子运算的基础上，深度学习训练框架为研究人员者提供简单便捷的计算流图构建方法。早期深度学习训练任务都是手动编写程序实现的，比如 2012 年训练 AlexNet 的 cuda-convnet 就以一段手动编写的程序方式运行。寻找更好网络结构需要按照不同计算图描述组合运行各种算子，每次都手动编写会产生大量重复的工作。深度学习框架可以通过简单的脚本语言或配置文件快速构建训练程序，让开发人员关注深度学习算法本身而不是底层优化细节。在 AlexNet 诞生后很长一段时间，研究人员借助 Theano[88] 来完成网络训练任务。Theano 原本是为科研人员在 CPU 和 GPU 上运算复杂数学表达式而设计的，“他山之石可以攻玉”，它的计算流图构建和执行的方式可以被用来解决神经网络的训练任务。2014 年问世的 Caffe[72] 是深度学习训练框架的鼻祖，用户可以使用配置文件形式自动生成 C++ 编写的网络训练程序。随后，各种深度学习训练框架如雨后春笋涌现，大型 IT 公司投入大量人力物力开发自己的深度学习框架，代表性工作有 Google 的 Tensorflow[89]、Facebook 的 Pytorch[90]、Amazon 的 MxNet[91]、Microsoft 的 CNTK[92] 等等。

对计算图优化方法而言，目前框架可以归类为“静态图”或者“动态图”两种方式。静态图框架，如 Caffe[72]、Tensorflow[89] 等采用“定义并运行”的方式，其计算图在模型运行之前定义，训练的每次迭代过程都将使用不可再修改的计算图，因此被称之为“静态”。这样就可以在运行之前使用各种图优化提高运行时性能。“动态图”框架，如 Pytorch[90]、Chainer[93] 和 Dynet[94] 则采用“运行时定义”的方式，其计算图在运行模型之前未定义，而在正向传播时递增方式地构造出来。用户根据框架提供的函数接口定义计算流图，随着这些函数被执行，计算图将逐步被构建，直到调用完正向传递的最后一个函数后，整个网络被构建完毕。与“静态图”框架相比，“动态图”框架具有一系列优点：首先，它更容易被调试，用户可以在调试器中逐步运行正向传递过程。其次，逐步运行还提供了相当大的灵活性，每次迭代都可以构建完全不同的计算图。所以动态图结构对于处理变长输入序列和树状、图状结构递归神经网络[95-96] 尤为有效。但是，这种灵活性往往使“动态图”框架相对“静态图”框架更加低效。另外，“动态图”无法执行图层次的优化，例如节点的融合、增加和删减等。总体来说，为了保证性能，目前流行深度学习的训练任务还是以静态图方式为主。唯一的例外是动态框架 Pytorch，因为其调试方式的灵活性，它常被视为一个研究性框架而非工业级的软件来使用。对于一些非常复杂的计算图方式，“静态图”框架提供“动态图”接口作为补充，比如 Tensorflow Fold[97] 就是封装在 Tensorflow 上的动态接口。

除了对框架计算方式进行优化，内存使用优化也引起了研究人员的关注。在使用 GPU 训练深度学习时，由于其有限的片上存储空间导致很多深层模型无法有效地训练。Chen 等人[98] 在 CNN 反向传播时重新计算网络正向传播的中间结果，以此来减少中间结果存储空间。受此启发，Gruslys 等人[99] 使用动态规划来平衡中间结果缓存和重计算开销，从而对 RNN 训练时的内存使用进行优化。vDNN[100] 使用 CPU DRAM 当做外层缓存不断换入换出 GPU 显存的数据，并设计了数据移动和计算的重叠机制来减少开销。SuperNeurons[101] 以张量为单位来进行生存时间分析，重复计算，维护统一的内存池等方式动态管理训练过程内存的使用。

2.3　多节点深度学习训练并行优化研究

算法1中主要计算部分 (行4-5) 可以分布到多个计算节点上并行运行。深度学习并行训练可以分为数据并行、模型并行和流水线并行三种方式。

数据并行: 使用这种方法在 N 个计算节点并行训练时, 通过对数据的 minibatch 维度进行切分，每个节点读取 minibatch 大小/N 个输入实例，反向传播后用所有节点加权平均后的梯度来更新本地参数。这个方法简洁有效，已经被 1990 年代早期的浅层神经网络[102-103] 所使用。首个在 GPU 上采用数据并行训练深度网络要追溯到 2009 年 Raina 等人训练 Belief 网络的工作[104]。数据并行时，不仅节点间的计算存在并行性，单节点内网络层的通信和计算也存在并行性。某一层反向传播的计算任务和后一层的梯度信息的通信任务同时进行，因此可以利用通信和计算重叠来隐藏一部分通信开销。这种方式被广泛应用于采用 CPU-GPU 异构计算节点的并行深度学习框架中[89,105-106]，但是数据并行的扩展性会受到泛化能力和通信开销两方面的限制，目前有许多研究致力于解决这两个问题。

首先，minibatch 的大小不能无限扩大，数据并行的扩展规模受到一定限制。实验结果[107] 和理论分析[108] 都发现，增大数据并行的 minibatch 大小会降低神经网络的泛化性。使用小 minibatch 训练则会找到相对尖锐位置的局部最优解。大 minibatch 训练得到的模型是解空间平面相对平坦位置的局部最优解，导致在测试集上无法得到和小 minbatch 一样的测试精度。但是，为了保证计算节点都有足够多的计算任务来发挥众核架构的计算能力，每个节点的 minibatch 不能过小。针对于 ResNet 和 AlexNet 两种广泛使用的 CNN，已经有工作提出了随 minibatch 大小变化的学习率调整策略 LARS 方法[109] 来提高大 minibatch 网络的泛化性。它们分别将 minibatch 大小增大到 8K[110]、32K[109]、64K[106,111]，并保证训练得到的模型和小 minibatch 的模型有相似的泛化能力。

其次，每个迭代步都需要同步所有节点的参数，这会带来非常大的通信量，从

而导致大规模数据并行时通信成为性能瓶颈。目前，很多工作致力于减少通信开销，其关键在于发掘时间和空间上的异步性。挖掘时间上的异步性方案被统称为异步 SGD，它放松了每个迭代步后同步参数的限制，打破多节点模型的一致性。每个节点计算完毕后可以立即更新全局的参数，这样就可以重叠不同节点间的计算和通信。Hogwild[112] 算法首先提出了共享内存情况下并行系统设计的异步 SGD 方法，随之被工作[113-114] 扩展到分布式内存的并行系统中。完全异步的方式收敛速度随着扩展规模而变慢[115-117]，陈旧同步更新（Stale-Synchronous Parallelism，SSP) 方法[118] 在同步和完全异步更新上做了一个折中，这种方法中，如果某个节点参数延迟过大，会被要求强制进行全局参数同步。比异步 SGD 更为激进地放松模型一致的方法是 Model Averaging[119]，它独立运行多套 SGD，并在若干迭代步才执行一次参数平均，代表性工作有 EASGD[120] 和 NG-SGD[121]。时间异步类方案由于其不确定性，给调试带来了很大困难，另外和同步 SGD 相比，其收敛速度和模型精度也会收到影响。因此，近年来随着越来越多的高质量网络硬件被深度学习训练任务所采用，挖掘时间异步性的研究明显降温[122]。

相比异步 SGD 的不稳定性，通过压缩传输数据的方法来挖掘空间的异步性成为近年来研究热点。它仍然采用同步 SGD，但每次通信一小部分重要梯度元素来降低通信量。残差梯度压缩（RGC）方法[123-127] 是目前最有效的梯度稀疏化方法，该方法在获得良好的压缩比的同时，还能确保最终训练出的模型不会有精度损失，它仅传输一小部分梯度，并将剩余的梯度部分保存为残差，用以添加到下一次迭代计算出的梯度上。残差梯度压缩由[123] 提出，最初实现版本使用基于阈值的方法对全连接层进行压缩，它仅仅发送数值大于预定义某阈值的梯度。考虑到预定义的阈值很难被恰当地选择，Aji 等人的工作[124] 建议选择绝对值最大的 1% 梯度元素，以此来提高残差梯度压缩方法的健壮性。由于这两种实现只针对某些特定的网络结构进行了调优，Chen 等[125] 指出应用该方法到其他网络结构会造成严重的精度损失，为了增加残差梯度压缩方法的泛化能力，最新的残差梯度压缩变种，如 AdaComp[125]、DGC[126] 等，引入一系列算法层次优化技巧，达到以 0.1% 压缩比压缩梯度，并在主流深度神经网络训练任务上几乎没有任何精度损失。目前，RGC 方法还处于理论阶段，系统方面的优化还有众多问题亟待解决，本文第8章将解决 RGC 系统实现上的问题。

通过互联网络来同步不同节点参数的操作可以分为参数服务器（Parameter Server, PS）和 Allreduce 两种方式。PS 方式[128] 将全局参数统一维护在参数服务器上，这里的服务器是一个虚拟概念：它既可以是一个物理节点，也可以是多个节点，每个节点存储全局参数的一个切片，这种情况被称为 Sharded PS。在分布式

深度学习框架还没有成熟之前，早期的数据中心大规模深度学习实践，如 DistBelief[113], Adam[129] 都采用了 Sharded PS 方案。PS 方案对低质量、非全连接的互联网络硬件，比如数据中心的以太网，是非常有效的解决方案。另外，异步 SGD 可以非常简单地映射到 PS 方式的并行系统上。但是对于配置了高质量全连接结构网络的高性能集群，由于多个消息传递到同一个物理节点会造成消息阻塞，PS 方案并不是最优方案，反而 Allreduce 方案更为高效。Allreduce 可以直接采用高性能消息传接口（Massage Passing Interface，MPI）[130] 的 `MPI_Allreduce` 例程来实现。Thakur 等[131]、Chan 等[132] 和 Hoefler 等[133] 的工作系统性地研究了 Allreduce 的多种实现方案。这些实现方案在解决并行深度学习的参数同步问题上又被重新发扬光大，比如，NCCL[134]，BaiduAllreduce[135] 采用环状通信连接方式来支持 GPU 间高效 Allreduce 实现。Facebook 推出了 Gloo[136]，Uber 推出了 Horvord[137]，Intel 推出了 MLSL[138] 来支持大规模集群上多种高效 Allreduce 实现。

模型并行：这种方案将网络层的参数进行切分，每个节点维护 $1/N$ 的参数信息，并使用相同的数据作为输入。每次迭代，模型并行更新参数的一个子集的全部梯度，而数据并行得到的是全部参数的部分梯度，它的显著优势在于可以减少模型 GPU 上显存的消耗。由于 GPU 显存空间限制，2012 年 AlexNet[7] 的训练就是在两块 GPU 上进行模型并行完成的。模型并行通信并不占优，只对模型参数相对输入数据规模小的情况有效。Lee 等人[139] 使用模型并行解决传统机器学习问题中“大模型”类问题，如主题模型、矩阵分解和 Lasso 问题。针对深度学习任务，krizhevsky 的早期工作[140] 采用混合并行方式，对 AlexNet 的卷积层进行数据并行而全连接层使用模型并行。Gholami 等人[141] 对这种混合并行方式进行了深入的理论分析，并将其类比为 1.5D 矩阵乘法。

流水线并行：这种方法是对数据并行和模型并行的另一种混合方案。不同于 Gholami 等人[141] 混合并行中模型并行部分是对每个网络层参数都进行切分，它的模型并行部分将网络沿着深度方向切分为多个阶段，每个阶段包含若干个网络层。每个阶段都映射到一个单独的计算节点来执行该阶段中所有层的正向和后向传播，不同节点可以顺序地接力完成训练。在此基础上引入数据并行，对 minibatch 维度切分成 microbatch，不同 microbatch 的不同阶段可以在多个节点中以流水的方式并行执行。1993 年，Petrowsk 等人[142] 提出了可以采用这种方式训练前馈神经网络。但是，由于节点间任务执行顺序的依赖关系容易导致计算资源闲置，这种方法在此后很长一段时间并没有得到和数据并行相似的关注度。PipeDream[143] 通过自动划分每个节点的任务规模来进行流水线并行，但是，它在反向传播过程引入了更新参数的异步性，容易引起模型精度的损失。与之类似，DualPipe[144] 通过设

计一种预测反向传播未来模型参数的方式来尝试缓解异步更新的问题。目前最新相关工作为 GPipe[145]，它通过在所有 microbatch 反向传播后显式同步梯度，消除了参数不一致或异步更新问题，该方法成功使用 8 块 GPU 训练了巨型卷积神经网络 AmoebaNet[146]。

目前，有许多基于高性能集群的深度学习实践。早先深度学习并行系统实现多基于低质量网络和 CPU 数据中心集群[113,129]。自 2013 年，在相同的深度学习训练任务上，Coates 等人[147] 使用配备了 16 块 GPU 卡的集群打败了 Google 公司上万节点的 CPU 数据中心[113]，人们开始意识到使用高性能集群在深度学习训练上的必要性。但是，再高性能超级计算机上进行并行深度学习训练的先锋工作要追溯到 2016 年的 FireCaffe[148]，它在 Titan 超算上用 128 个 GPU 实现了 CNN 的数据并行训练，并指出在超算上使用高带宽网络连接硬件进行 Allreduce 方式通信比 PS 方式更加有效。S-Caffe[149] 设计 CUDA-Aware MPI 进行 GPU 和 GPU 显存之间直接数据传递，改进了 128 个 GPU 节点规模 Allreduce 操作的效率。Yang 等人[150] 在 GPU 超算和 Xeon Phi KNL 上对 CNN 进行并行训练，结果也表明使用 Allreduce 同步的 SGD 相对于 PS 方式实现的异步 SGD 性能更佳。最新的大 minibatch 数据并行的工作[106,109-110] 分别使用 Gloo、MLSL、NCCL 提供的 Allreduce 接口进行同步 SGD 训练。在应用层面上，使用超算系统的深度学习训练技术在 2018 年左右大范围开花结果。2017 年，Cori 超算在使用 CNN 对科学计算图像数据进行分类任务上达到 15PFlop 性能[43]。2018 年，Summit 超算上的 Goden Bell 奖工作[44] 利用 Tensorflow 和 Horovod 搭建的系统软件，使用 CNN 分析遥感图像中极端气候模式。“风物长宜放眼量”，随着新型超算得到更广泛的使用，在超算上利用深度学习技术的科研成果会越来越多。

2.4　本章小结

本章介绍了深度学习训练的理论基础，以及它在众核处理器上性能优化和多节点上并行优化的研究现状。单节点内优化方法研究主要针对于成熟的商用众核处理器，如 NVIDIA GPU 和 Intel Xeon Phi，而国产众核处理器上的研究仍是一片空白。本文后续章节致力于填补这项研究空白，第5章将介绍申威处理器上深度学习算子的优化方法，第6章将介绍算子的自动优化和调优方法，第7章将介绍并行深度学习框架的设计与优化。

现今最大规模的深度学习应用都使用数据并行的方式训练。对于数据并行的通信方式，目前主要有参数服务器和 Allreduce 两种方案，后者更适用于超级计算机上的高质量的网络连接硬件。本文第7章将介绍如何利用“神威”网络硬件特点减

少 Allreduce 开销。

为了突破扩展性瓶颈，可以通过挖掘时间和空间异步性来减少通信开销，由于时间异步性对模型收敛性和精度的影响，空间异步性已经成为最近的研究热点，本文第8章将介绍如何利用空间异步性来设计扩展性更好的数据并行系统。

第 3 章　申威架构的性能模型和张量化编程模型

随着摩尔定律即将终结，众核架构处理器已经成为超计算机系统所采用的主流芯片。由于其架构的复杂性，在众核处理器上进行并行程序设计比通用处理器更加困难。在搭建深度学习并行训练系统之前，本章将带领读者揭开申威架构国产异构处理器的神秘面纱。通过对其硬件特点进行深入研究，本章建立了定性和定量的性能模型来指导程序设计。为了弥合众核架构和应用程序设计之间鸿沟，众核处理器需要针对架构特点设计相应的编程模型。在性能模型基础上，本章提出了“张量化”编程模型，并对并行程序的优化技巧进行总结。

3.1　申威异构众核处理器架构

“申威 26010”处理器（SW26010 Many-core Processor）是完全使用我国自主技术研制的第四代高性能处理器。它采用片上计算阵列集群和分布式共享存储相结合的异构众核体系结构。方便起见，本文简称此体系结构为**申威架构**。本小节先简单介绍这种架构的基本情况，随后会详细介绍其访存、计算和核间通信等方面的特点。

3.1.1　概况

如图3.1展示了申威架构硬件细节。一块“申威 26010”芯片上包括四个**运算核组**（Core Group，CG）共 260 个运算核心。每个核组采用主从异构的结构，包括一个**主核**（Management Processing Element，MPE）和一个**从核阵列**（Computing Processing Element Mesh，CPE Mesh）。每个核组还集成了 8 GB 的 DDR3 内存，并由内存总线通过存储控制器（Memory Controller，MC）与主核和从核阵列相连。四个核组通过高速片上网络（Network on Chip，NoC）互连。共计 32 GB 的内存物理空间被统一编址，任意主核和从核均可以访问芯片上的所有主存空间。片上配备有可提供 16GB/s 双向峰值带宽的 8 通道 PCI-E 3.0 标准接口，用于和其他芯片互联。

主核：主核的工作频率为 1.45 GHz，采用 64 位的基于 RISC 的**第四代申威指令集**。每个主核都包含两级 Cache，一级 Cache 包含指令 Cache（ICache）和数据 Cache（DCache）两种，大小均为 32 KB。二级 Cache 被指令和数据共用，称之为 SCache，大小为 512 KB。为了处理共享存储器空间的 Cache 一致性访问，在每个核组还设置了一个二级 Cache 的标记副本，被称之为 CTAG，与主核的 SCache 一

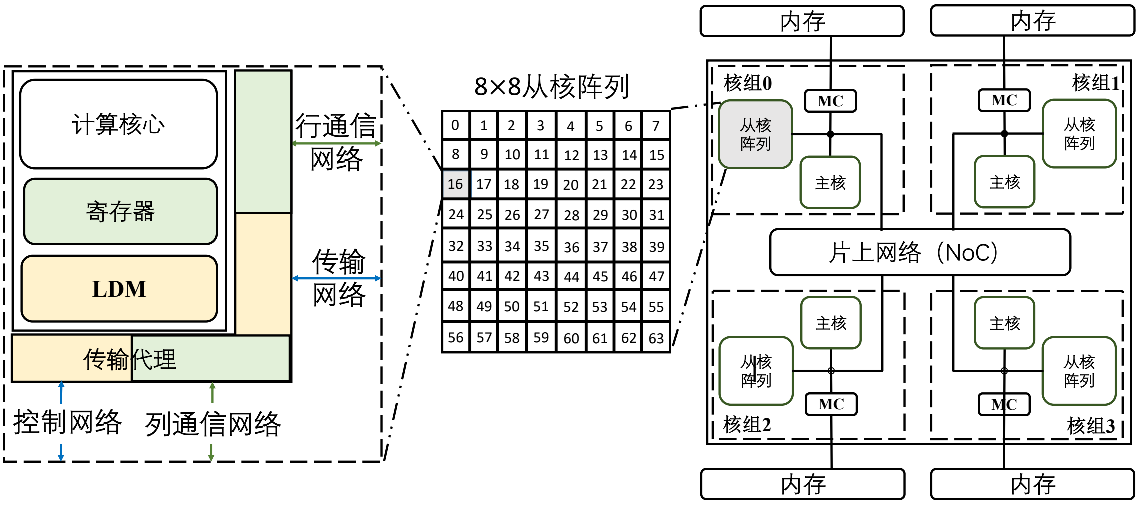


图 3.1　申威 26010 众核处理器架构图

一对应。主核支持中断，同时可以运行操作系统和用户程序，进行计算、存储资源的管理，并提供消息、文件、调试、低功耗管理等服务。

从核阵列：从核阵列由 64 个相同的从核排布成 8 行、8 列拓扑结构。从核的工作频率同样是 1.45 GHz。其指令集兼容大部分主核指令，并添加了一些从核特有的寄存器通信相关指令。每个从核拥有独立的大小为 16 KB 的一级指令 Cache。大小为 64 KB 的二级指令 Cache 被 64 个从核共享。每个从核拥有独立的 64KB 可重构局部数据存储作为数据 Cache，本文称之为 **LDM（Local Direct Memory）**。它采用便笺内存（Scratchpad Memory）的组织形式，并不能像传统 Cache 一样可以自动决定内存数据的缓存，程序员需要通过软件方式显式决定 LDM 数据的换入和换出规则。每个从核拥有 32 个 256-bit 的通用寄存器，可供计算单元进行运算使用。从核访问主存可以有两种方式，一种通过全局加载和存储指令（Global Load/Store，gld/gstd）直接从内存中读取向量或标量数据到寄存器。另一种方式先使用**直接内存访问（Direct Memory Access，DMA）**方式将数据从内存移动到从核 LDM，然后再读取 LDM 的数据到寄存器。每个从核运行一个轻量级操作系统，以进行打印、文件、断点、线程切换、容错等处理。

从核阵列配有阵列数据传输网络，包括 4 个行向总线作为数据通路和指令装填通路，相邻两列 16 个从核共享一套这样的总线。从核阵列虽然在 LDM 级别没有提供数据共享机制，但是为了提供方便的数据交互，申威架构提供了独特的**寄存器通信**机制。从核阵列配有寄存器通信网络，互联同行/同列的从核，它由一级交换开关进行连接，可以支持从核间低延迟、高带宽的同行同列寄存器通信。

存储控制器：每个核组的主从核都需要经由同一个存储控制器访问内存，因此它们共享内存带宽资源。整个芯片集成 4 块 8GB DDR3 SDRAM 形式内存，每

块内存的数据接口位宽 144 位，理论最大主存数据带宽为 134.4 GB/s[151]。

表 3.1　申威 26010 众核架构的主核从核特点

	主频	向量长度	指令 Cache	数据 Cache	内存带宽
主核	1.45 GHz	256 bit	32 KB L1	32KB L1	33.6 GB/s
			256KB L2（数据指令共享）		
从核			16 KB L1 64KB L2 (64 从核共享）	64KB (SPM)	

介绍完申威架构的概况后，本节接下来分别对从核阵列访存特性、核间通信特性、指令执行方式三个方面进行介绍，最后将它们和其它主流众核架构进行比较。

3.1.2　从核访存特性

申威架构的从核阵列提供了主要计算能力，能否向从核的计算单元高效地传输数据对并行程序整体性能至关重要。申威架构提供了两种内存到从核内寄存器的数据移动方式，一种是从核通过全局访存指令 gld/gstd 直接细粒度访问主存，另一种是通过 DMA 方式批量访问主存。本节将研究两种访存模式的带宽使用特性。

在传统的基于 Cache 的芯片架构中，内存是以 Cache 行（Cache Line）为粒度进行访问的。比如采用传统硬件 Cache 策略的申威主核，它的 L1 数据 Cache 就采用 4 路组相联结构读写都可装填的 Cache 策略，Cache 行大小为 128 字节。从核并没有基于 Cache 的缓存机制，但和以 Cache 行为单位访问类似，从核的内存访问也需要以 128 字节**内存事务**（Memory Transaction）为单位进行组织。即使对某内存事务块实际请求量不足 128 字节，也需要将 128 字节的内存事务全部访问，其中浪费的部分本文称之为**填充空间**。

全局内存访问十分低效，无法承担大量连续内存访问任务。全局内存访问 gld/gstd 最多请求 32 字节的数据，有效请求仅为一个内存事务大小的 1/4。根据 Lin 等人[152] 的测量结果，gld 的启动延迟为 177 个时钟周期，gld 和 gstd 一起使用延迟将达到 278 个时钟周期。据测量，单核组从核的 gld/gstd 操作的峰值带宽在 3 GB/s 左右，仅为利用了峰值带宽的 10%。

DMA 以异步形式完成访存操作，相比 gld/gstd 形式更为高效。从核指令流水线发射 DMA 请求指令后，请求进入从核中的通道缓冲，同时指令流水线继续运行。通道缓冲最多可以容纳三个未发出从核的通道操作，只有当通道缓冲被占满时，指令流水线才暂停运行。从核通过阵列控制网络的控制连线向全局控制器

(Global Controller，GC) 提交 DMA 请求，由于内存数据传输总线是两列 16 个从核共享的，所以需要等待全局控制器仲裁后，从核才能将发起的 DMA 请求送入全局控制器。全局控制器经过多列间的仲裁后，将 DMA 命令提交传输控制器（Transfer Controller，TA）中的 DMA 引擎进行处理。随后，DMA 引擎将 DMA 操作拆成多个基本操作，通过阵列数据网络，完成 LDM 与主存的数据交换。完成 DMA 传输后，传输控制器生成回答字，并通过阵列数据传输网络送入发起 DMA 的从核，从核通过读取 LDM 中的回答字的方式来判断 DMA 操作何时完成。

DMA 可以完成内存数据的连续访问（Continuous Memory Access）和跨步访问（Strided Memory Access）。两种访存方式的细节可以通过改变 DMA 指令的描述字来设置。如图3.2，跨步访问的方式需要提供数据分块大小、跨步大小、数据块个数信息。

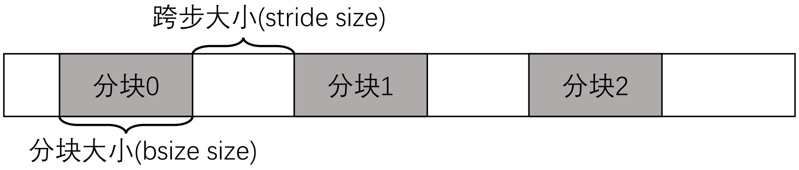


图 3.2　跨步 DMA 示意图

3.1.3　核间通信特性

由于从核的局部存储 LDM 空间有限，为了能够更好的利用片上存储资源，实现从核间的高效信息交互，申威处理器提供了寄存器级的通信机制。寄存器通信通过生产者—消费者的方式完成与同行/同列其它从核的点对点传输或广播传输。发送端执行 *Put* 类指令将某个通用寄存器内 256-bit 数据经过寄存器通信网络送入一个或多个目标从核的接收缓冲，接收端执行 *Get* 类指令从接收缓冲中读取队首数据，并送入本地通用寄存器文件。发送从核将通用寄存器文件中的数据送入发送部件，*Put* 类指令立即执行完成，指令流水线可继续执行。对于发送从核，寄存器通信指令是异步的，只有当发送部件的发送缓冲被占满时，发送从核的指令流水线暂停，直到发送部件有空间接收发送数据为止。但是对于接收从核，其工作方式是同步阻塞的，目标从核使用 *Get* 类指令，从接收缓冲中获取首个有效数据，当接收缓冲空时接收暂停，目标从核的流水线停止，直到其他从核生产出数据为止。寄存器通信的延迟为 11 个时钟周期，如果寄存器传输完全流水线化，则整体 P2P 和广播带宽可分别达到 2549 GB/s 和 4461 GB/s[152]。因此，相比 200 时钟周期的延迟和 134.4 GB/s 理论带宽的内存访问，寄存器通信是一种更高带宽、更低延迟的数据传输方式。

3.1.4　指令执行特性

申威架构的每个从核内部包含两个指令执行部件，分是执行部件 0（$P0$）和执行部件 1（$P1$）。申威架构的指令执行部件包括 8 个功能子部件，执行的功能分别是：地址仲裁，Tag 查询，指令读出，指令译码及缓存，指令译码及发射，读操作数，操作执行，结果写回。两个指令执行部件在指令译码及缓存，指令译码及发射，读操作数，操作执行上使用不同的功能子部件，其他部分共用相同的功能子部件。指令进行译码及发射阶段后，根据指令的类型被发射到某一条流水线，并从这条流水线流入相应的指令执行部件中执行。指令的类型和它们可分派的执行部件见表3.2。

从核有独特的指令发射规则，其原则是将尽可能多的指令前推，其基本算法分为如下两个步骤：

第一步：先确定指令是否有机会被发射。指令 I 可以进入流水线 P（P,n 为 0 或 1）的条件是：a) 指令 I 有效；b) 指令 I 可以在流水线 P 执行；c) 流水线 P 本时钟周期接收指令。

第二步：根据指令可以进入的流水线和两条指令的先后顺序，选择流水线的规则为：

a) 如果前导指令（即指令对中执行顺序在前的指令，通常为 PC 值小的指令）不能进入任意一条流水线，后续指令（即指令对中执行顺序在后的指令，通常为 PC 值大的指令）也不能进入。换而言之，流水线内指令是按序发射 (Issued In Order) 的。

b) 如果前导指令只能进入一条流水线，先根据前导指令选择。

c) 如果前导指令可以进入两条流水线，先根据后续指令选择。如果后续指令也可以进入两条流水线，则本条指令优先选择流水线 1。因此，指令流水线间是可以乱序发射（Issued Out of Order）的。

在指令发射规则基础上，指令执行的延迟同样重要。在前一条指令没有执行完毕的情况下，它的目的寄存器无法被下一条指令使用。因此，指令的延迟性质对高效的指令排布方式设计至关重要。表3.2列出了第四代申威指令系统中主要指令的可分派流水线及指令执行延迟信息，以供后面章节索引。

3.1.5　和其他众核架构比较

图 3.3抽象了申威架构存储层次（Memory Hierarchy）模型。和同世代超算所采用的众核架构芯片（如美国 Titan 超算采用的 NVIDIA Kepler K40 和 Cori 超算采用的 Intel Xeon Phi Knights Landing，KNL）相比，“申威 26010”架构有很多不同

表 3.2 第四代申威指令系统主要指令的可分派流水线及延迟时钟周期信息

指令类型	指令助记符	流水线	延迟
整数运算类指令			
标量加减	ADDW、SUBW、ADDL、SUBL、S4ADDL、S4SUBL、S8ADDL、S8SUBL	$P0/P1$	1
比较	CMPEQ、CMPLT、CMPLE、CMPULT、CMPULE		
向量加减	VADDW、VSUBW、VADDL、VSUBL	$P0$	1
整数乘法	MULW、UMULW、MULL、UMULH	$P0$	5
标量移位	SLL、SRL、SRA	$P0/P1$	1
浮点运算类指令			
浮点加减	FADDS、FADDD、FSUBS、FSUBD、VADDS、VADDD、VSUBS、VSUBD	$P0$	7
浮点乘法	FMULS、FMULD、VMULS、VMULD		
浮点乘加	FMAS、FMAD、VMAS、VMAD		
访存类指令			
标量访存	LDW、LDL、STW、STL	$P1$	访问 LDM 3~4 拍 访问主存节拍不定
向量访存	VLDS、VLDD、LDDE、VSTS、VSTD		
其它指令			
向量整理	VINSW、VEXTW、VINSF、VEXTF VSHFF、VSHFW	$P0$	1
通信指令	GETR、GETC、PUTR、PUTC	$P1$	1
访存并发送	LDR、VLDC、LDDER、LDDEC	$P1$	1
同步指令	SYNR、SYNC	$P0/P1$	至少 14 拍
控制指令	HALT、MEMB	$P1$	1

之处。

并行架构: Intel KNL 和 NVIDIA Kepler 是采用 SIMT（Single Instruction Multiple Thread）方式进行并行计算的架构。严格按照 Flynn 对体系结构分类标准[153] 来看，它们属于 SIMD（Single Instruction Multiple Data）架构。SIMD 架构中单个线程对多个操作数执行向量化的指令，而 SIMT 中多个线程向多个操作数发出向量化的指令。KNL 的每个计算核心提供操作目标为 512-bit 长的向量化指令，Kepler 向量化方式是通过在一个 Warp 内 32 个线程使用相同指令对 32 个数据同时运算。但是，申威处理器常常需要不同从核执行不同的指令，所以它的核心计算部件—从核阵列属于 MIMD（Multiple Instruction Multiple Thread）并行架构。相比而言，这种架构的并行方式更复杂。

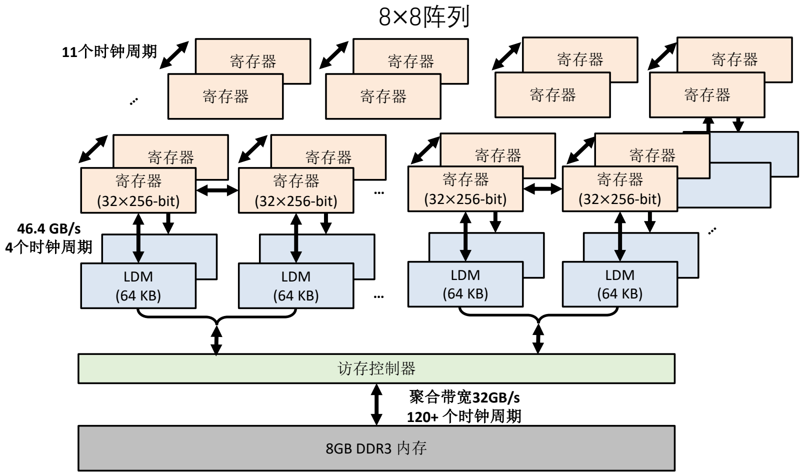


图 3.3　申威架构单核组的存储层次模型示意图。大括号内表示两列 16 个从核共享一个总线的特点。

内存体系：申威处理器 256 个从核都有独立的 64KB LDM 作为数据缓存。每个核组内 64 个从核构成的从核阵列可以通过寄存器通信总线交换数据，并通过数据传输网络连接到 8 GB 的 DDR3 内存。不同核组的内存之间通过 NoC 方式互相连接。

Kepler K40 架构由 15 个流式多处理器（Streaming Multiprocessors，SMX）组成，每个 SMX 包括 192 个流处理器（Streaming Processor，SP），共计 2880 个 SP 计算核心。同一个 SMX 上的 SP 共享 64KB 的高速 L1 缓存，15 个 SMX 共用一个 1.5 MB 的 L2 Cache。L2 Cache 通过存储控制器和 12 GB 的 DDR5 片上显存连接，显存通过 PCI-E 和 CPU 内存连接。

Intel KNL 架构上有 68 个计算核心。每个核心拥有独立的 32 KB L1 Cache，每 2 个核心组成一个分块，共享一个 L2 Cache。不同分块的 L2 Cache 通过总线连接，并维护 Cache 的一致性。内存控制器连接 16 GB 高速 MCDRAM 和 384 GB 相对慢速的 DDR4 内存。相比较而言，KNL 和 Kerper 都提供了不同计算核心间 Cache 层次的高速缓存共享机制，而申威架构从核间只能进行寄存器级别的数据交换。

地址空间：申威架构的主存地址空间被主核和从核共同编址并可以共同访问。在 Kepler 架构中，GPU 的片上显存和片外 CPU 内存地址分离，CPU 内存和 Kepler 显存的 L2 Cache 通信需要经过 PCI-E 总线，并显式调用相应编程接口。KNL 有片上高速缓存 MCDRAM 和 DDR4 内存，它们之间可以配置成 Cache 模式、Flat 模式和混合模式。后两种模式中，全部或者部分 MCDRAM 可以和 DDR4 内存共同编址。KNL 的片上内存空间最为宽裕，其次是申威架构，Kerpler 架构片上内存最小，常常需要通过低速 PCI-E 总线和 CPU 主存显式地交换数据。

访存计算比：相比 Kepler 和 KNL 架构，申威提供了类似的浮点运算性能，但

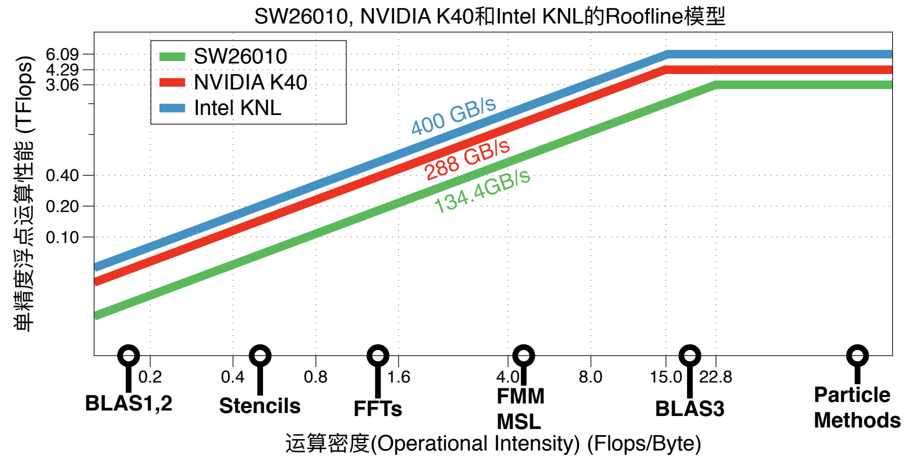


图 3.4　三种架构芯片的 Roofline 模型示意图

是在内存带宽方面则相形见绌。图3.4展示了三种架构的 roofline 模型[154]。申威架构提供 3.06 Tflops 的单精度浮点运算性能，而 DDR3 内存接口为每个核组提供 33.6 GB/s 的理论峰值带宽，处理器四核组的总带宽为 134.4 GB/s。NVIDIA K40 GPU 具有 4.29 Tflops 单精度浮点运算性能，可提供 288 GB/s 内存带宽。Intel KNL 具有 6.09 Tflops 单精度浮点运算性能，可提供 400 GB/s 对 MCDRAM 的访问带宽。相比较而言，申威架构计算访存比率远高于 K40 和 KNL，这意味其上运行的程序很容易陷入访存受限的境地。

3.2　性能分析方法

根据申威架构的硬件特性建立性能分析模型可以给并行算法设计提供指导。Xu 等[155] 工作尝试对申威架构建立精确性能模型，然而他们工作局限于不存在寄存器通信情况下，因而无法分析申威架构核间共享数据的潜力。本小节首次分析了寄存器通信的影响，并建立了定量和定性的性能模型指导程序设计。

3.2.1　核间通信的性能影响

申威提供了两种方式完成数据从内存到计算单元的流动，一种方式是从核寄存器直接对内存进行访问，另一种方式是从核寄存器通过三级（REG-LDM-MEM）存储器层次结构访存。如章节3.1.2所描述，全局访存效率极低，因此三级存储器访问模式是申威众核架构的主要编程方式，也是本文研究的重点。

从核阵列的每个从核可以独立发起 DMA 操作请求，并由同一个全局控制器进行仲裁。因为从核的阵列通信网络只有 4 条，因此从核阵列以每次 4 个从核为一组同时进行 DMA 访存，不同组的从核间会形成竞争 DMA 访存带宽的现象。另

一方面，从核的计算部件却是私有，只要数据在 LDM 内就绪后，从核随时开始计算。因此，竞争性访存会导致每个从核开启计算时间点不同。

如图3.5上半部分所示，如果从核间没有时间同步的需求，DMA 的访存时间和计算操作时间存在一定程度的重叠。如图3.5下半部分所示，同步操作和寄存器通信会导致从核间存在数据依赖关系，进而破坏了这种并行流水，降低从核的执行效率。

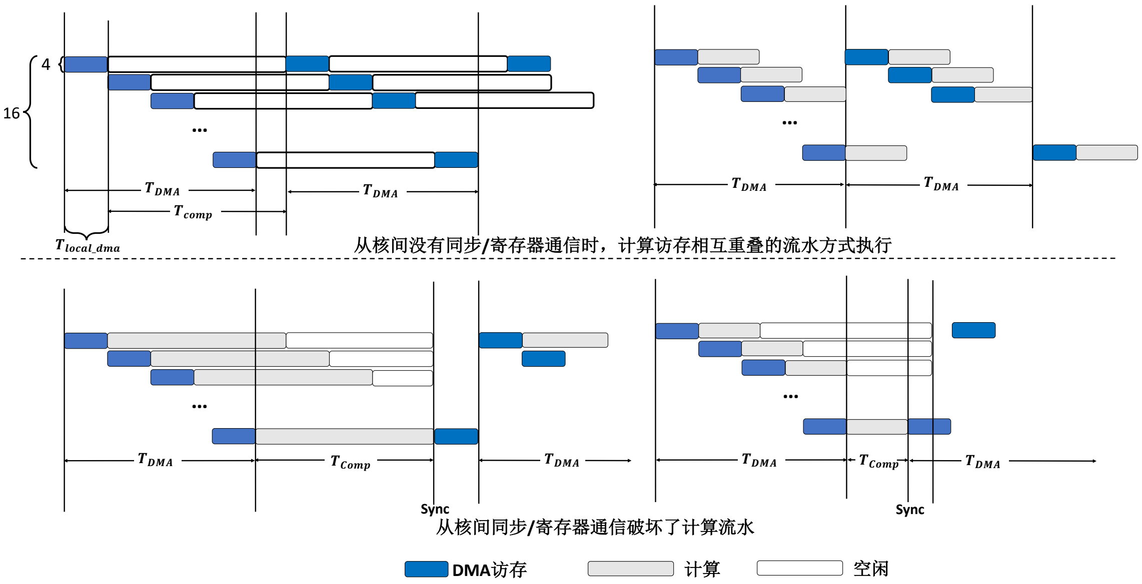


图 3.5　DMA 访存与计算时间硬件重叠

本研究使用一个简单的测试用例来验证寄存器通信对 DMA 访存计算硬件流水并行的影响。测试用例实现简单的 $C = A * B$ 功能，其中 A、B 和 C 都是内存中的数组，每个从核使用 DMA 读取该数组的一部分进行计算，图3.6展示了不同计算方式的耗时对比。图中 Get+Compute+Put 表示每个从核发起一次数据 DMA 读入，计算和数据写出操作。图中 Compute 表示只进行计算而不进行 DMA 读取的时间。图中 DMAGet+DMAPut 表示只进行 DMA 读写而不进行计算的时间。图中 Sync 表示使用 SYNR 和 SYNC 指令使 64 个从核全部完成计算操作后才发起 DMA 操作的耗时。图中 RegComm 表示每次让第 i 行和第 i 列从核发起行列广播寄存器通信的耗时，i 为当前迭代数和 8 取模的结果。通过实验结果可以得出以下结论。

DMA 访存计算硬件流水并行真实存在。从核间计算没有相互的数据依赖时，64 个从核完成访存和计算的总时间小于单独进行 DMA 访存和单独进行计算二者时间的加和。

寄存器通信可能会破坏 DMA 访存计算硬件流水并行方式。图中 DMAGet+Compute+Sync+DMAPut 和 DMAGet+Compute+RegComm+DMAPut 的大

小基本相同。由此可见，寄存器通信会使不同从核运行时序产生依赖关系，类似于同步操作，从而破坏 DMA 访存计算硬件流水并行。

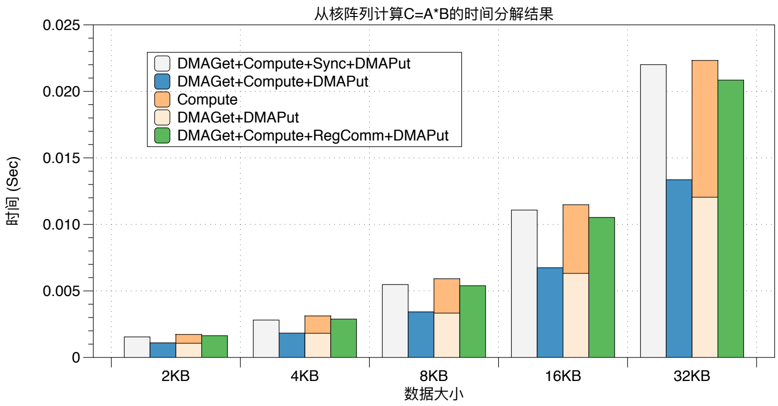


图 3.6　使用不同方式在从核阵列上进行数组点乘运算（$C = A * B$）的时间分解，横轴表示参与计算的数组长度。

3.2.2　定量的性能模型分析

基于内存-DMA-寄存器三层次访存方式，可以建立定量的性能模型以估计程序的整体运行时间。程序的执行时间 T_{exec} 可以用公式3-3表示。执行时间由从核阵列进行 DMA 访存总时间 T_{DMA}，单个从核使用 LDM 内数据计算时间 $T_{compute}$ 和访存计算硬件流水线的重叠部分时间 $T_{overlap}$ 共同决定。

根据本章3.1.2节介绍的事务型内存组织原理，可以为 T_{DMA} 设计精确的时间估计模型。连续访存的 T_{cont_DMA} 由公式 3-1计算得到，跨步访存 T_{stride_DMA} 通过公式3-2计算得到。$T_{latency}$ 是一次 DMA 启动延迟，为 200 个时钟周期。$PEAK\_BW$ 是单核组真实峰值带宽，通过实际测量[151] 为 30.94 GB/s。$block\_size$ 是连续访存数据总量，或跨步访存中数据块大小（Block Size，bsize）。$waste\_size$ 是内存填充数据大小，对于连续和跨步访存需要分别处理。在连续 DMA 访存中，起始位置已知，终止位置可以根据访存长度得到，由此可以推理出内存事务边界的填充尺寸；在跨步 DMA 访存中，访存由若干数据块的访问组成，需要对每个数据块按照连续访存估计，前一个数据块终止位置加上跨步大小（Stride Size）可知每个数据块访存的起始位置，通过数据块大小计算终止位置，有了起始和终止位置就可以推理出该数据块的填充大小。

图 3.7展示了跨步 DMA 性能模型公式估计值和实测的结果的对比。可以观察到，对于跨步大小不同的情况，性能模型和实测数据符合的很好。

$$T_{cont\_DMA} = T_{latency} + \frac{size + waste\_size}{PEAK\_BW/\#CPE} \tag{3-1}$$

$$T_{stride\_DMA} = T_{latency} + \frac{\sum_{i=1}^{block\_num} block\_size + waste\_size_i}{PEAK\_BW/\#CPE} \tag{3-2}$$

$T_{compute}$ 等于计算单元执行所有指令的总时间，换句话说是 $P0$ 和 $P1$ 运算的最长时间，它可以由公式3-3得到。$\#P0\_cycle$ 和 $\#P1\_cycle$ 分别表示 $P0$ 和 $P1$ 执行的时钟周期。公式中 1.45 GHz 是从核的时钟周期。$\#P0\_valid$，$\#P1\_valid$ 表示有效发射的指令数目。$\#P0\_idle$，$\#P1\_idle$ 表示执行单元闲置的时钟周期数目，它是由于依赖关系而造成指令延迟发射的等待开销。

$T_{overlap}$ 表示计算和访存重叠的时间，它需要根据程序类别分类讨论分析。如图3.5下半部分所示，对于使用从核同步操作或寄存器通信的从核程序，$T_{overlap}$ 近似为 0。如图3.5上半部分所示，对于从核间无时序依赖关系的程序，重叠现象仍可分为两类。第一类情况，当 $T_{DMA} - T_{local\_DMA} > T_{compute}$ 时，其中 $T_{local\_DMA} = 1/16T_{DMA}$ 是 4 个从核组的 DMA 时间，如图3.5右上所示，此时计算时间被全部隐藏，$T_{overlap} = T_{compute}$。第二类情况，当 $T_{DMA} - T_{local\_DMA} <= T_{compute}$ 时，$T_{overlap} = T_{DMA} - T_{local\_DMA}$。

$$\begin{aligned}
&T_{exec} = T_{compute} + T_{DMA} - T_{overlap} \\
&T_{DMA} = total\_data/MBW_{MEM->LDM} \\
&T_{compute} = max(\#P0\_cycle, \#P1\_cycle)/1.45GHz \\
&\#P0\_cycle = \#P0\_valid + \#P0\_idle \\
&\#P1\_cycle = \#P1\_valid + \#P1\_idle
\end{aligned} \tag{3-3}$$

综上所述，定量性能模型可以准确估计程序的 DMA 时间，而计算和二者重叠时间需要根据程序具体情况来讨论。本文后面部分（章节4.2.3和章节6.2.5等）会有应用于具体问题的例子。

3.2.3　定性的性能分析模型

定量性能模型中指令级别的执行时间往往难以精确估计，在算法设计阶段需要一个定性的性能模型，它由执行效率、需求带宽和实测带宽三个性能指标共同

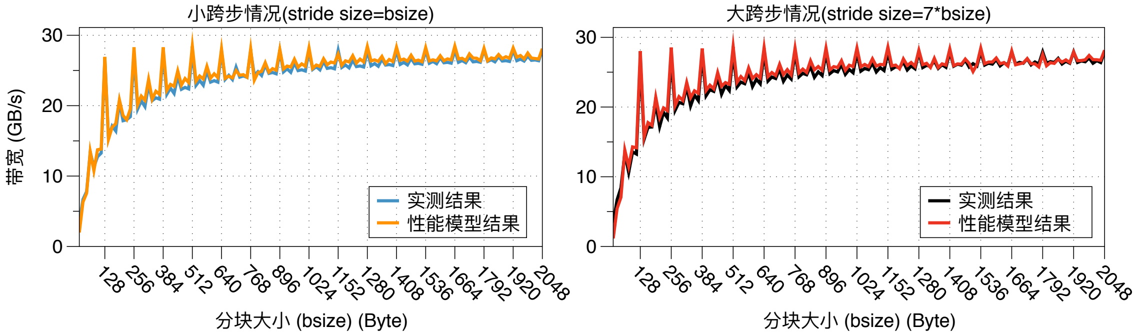


图 3.7　在不同跨步大小情况下，性能模型估计带宽和真实测量带宽之间比较。图例中跨步访存的数据块数目为 10，左图是小跨步情况，右图是大跨步情况。

描述。它能快速地反映出不同并行算法设计之间的优劣，而不需要给出精确的程序运行时间。

执行效率 EE（Execution Efficiency）描述由于指令发射部件空闲而造成的性能损失。它表示有效指令在总执行周期的占比，比如 $P0$ 执行部件的执行效率为: EE_{P0} = $\#P0\_valid/\#P0\_cycle$，这个指标可以通过分析程序核心段的汇编指令获得。如果充分利用指令发射部件，则意味着每个时钟周期都可以发射一条有效指令，此时 EE=1。如果汇编指令排布方式欠佳，存在指令间依赖关系或者发射了无效指令则会导致 EE 小于 1。

定性性能模型使用**需求带宽**和**实测带宽**两个性能指标来指示访存和计算的关系。需求带宽（Required Bandwidth），本文简称 RBW，是本层存储器访问的数据量和数据在本层存储器内的计算时间的比值。换而言之，它表示为了是并行算法能获得本层存储器计算的峰值性能，所需的对上层存储器访存的最低理论带宽。实测带宽（Measured Bandwidth），本文简称 MBW，是真实利用的带宽数值，它是本层存储器访问的数据量和真实计算时间的比值。需求带宽和实测带宽的比值是一个重要的指标。如公式3-4所示，可以用 LDM 和寄存器间的需求带宽 $RBW_{LDM->REG}$ 和实测带宽 $MBW_{LDM->REG}$ 的比值近似估计性能模型中 $\#P0\_cycle$, $\#P1\_cycle$ 的相对大小，公式中 $MBW_{LDM->REG}$ 表示从核 LDM 寄存器间的实测带宽，它是一个常数，数值等于主频 (1.45 GHz) × 256bit = 46.4 GB/s。如公式3-5所示，可以用内存 LDM 间的需求带宽 $RBW_{MEM->LDM}$ 和实测带宽 $MBW_{MEM->LDM}$ 的比值近似估计性能模型中 T_{DMA} 和 $T_{compute}$ 的相对大小，公式中 DMA 访存时间 T_{DMA} 可以通过上一节的公式 3-1和公式 3-2计算得到。

$$\frac{RBW_{LDM->REG}}{MBW_{LDM->REG}} = \frac{ldm\_data\_access/MBW_{LDM->REG}}{\#P1\_cycle/1.45GHz} = \frac{\#P0\_valid}{\#P1\_valid} \tag{3-4}$$

$$\frac{RBW_{MEM->LDM}}{MBW_{MEM->LDM}} = \frac{mem\_data\_access/MBW_{MEM->LDM}}{T_{compute}} = \frac{T_{DMA}}{T_{compute}} \tag{3-5}$$

定性性能模型是算法设计的重要理论工具，用来确定程序的优化瓶颈。RBW/MBW 用来分析程序是否受限于对上层存储器的访问效率，如果它小于 1，则说明程序受限于指令执行部件的效率，程序的整体效率理论上可以近似为 EE，此时应考察是否有更好指令排布方式来减少执行部件闲置时间，如果它大于 1，则说明程序受限于访存带宽使用率，此时应考察是否有更好的计算和访存方式可以来减少 RBW，或者是否有更好的访存模式可以增加 MBW。在本文后面部分（章节4.1.2，章节4.2.3，章节5.1.2等），定性性能模型被有效地用来指导并行算法的设计。

3.3 张量化编程模型

编程模型（Programming Model）是对底层硬件的更高层次抽象，不同于编程语言或者编程接口 API，它不关注具体实现细节，而是定义底层硬件系统对算法流程和数据结构的表达能力。和单核通用处理器架构不同，众核架构的编程模型要能够充分利用其强大的并行能力。对于 GPU 和 Xeon Phi 这种 SIMD 类众核架构，主要采用共享内存式的多线程向量化（Vectorization）的编程模型[156]。向量化编程模型的核心是利用向量化方式来进行计算和访存，相比标量化编程中一条指令只能处理一对操作数，向量化编程模型中一条指令可以同时处理多对操作数。比如，NVIDIA 提供了统一计算架构（Compute Unified Device Architecture，CUDA）技术以支持其众核架构的向量化编程模型，Intel Xeon Phi 系列则沿用 CPU 的向量化编程模型，用户可以在高级编程语言中嵌入向量化的 intrinsic 函数接口，或者使用 OpenMP 编译器提供的编译指导语句来设计向量化算法。

对于没有 Cache 层次数据共享能力的申威处理器，仅依靠向量化编程模型无法充分发挥其 MIMD 方式并行计算的能力。如同向量化之于 SIMD 众核架构，本文提出**张量化（Tensorization）编程模型**应该是核间互联特性众核架构上并行算法设计的指导原则。张量化编程中一条指令可以同时处理排列成张量形式的一块操作数，如图3.8所示，张量化运算操作的对象不再是一维数据而是二维甚至更高维的数据结构。

和向量化相比，张量化方式进行程序设计具有更高的难度。近年来，随着众核架构发展，能使用张量化方式进行算法设计的硬件结构越来越多，这些架构采用以阵列方式组织的计算部件，且核间可以进行高速数据传递，比如 NVIDIA 最新

的 Turing 架构 GPU、Google 的 TPU 所采用的 Tensor Core[157] 架构和本文研究的申威架构。不同于向量化编程模型中将一维连续数据通过 SIMD 指令映射到单个（或并列排布的）计算部件上，张量化编程模型需要将不连续多维数据映射到多个以阵列方式排布（或其他规则拓扑结构排列）的计算部件上，这些部件之间的协同计算需要通过阵列间通信网络来精细控制。

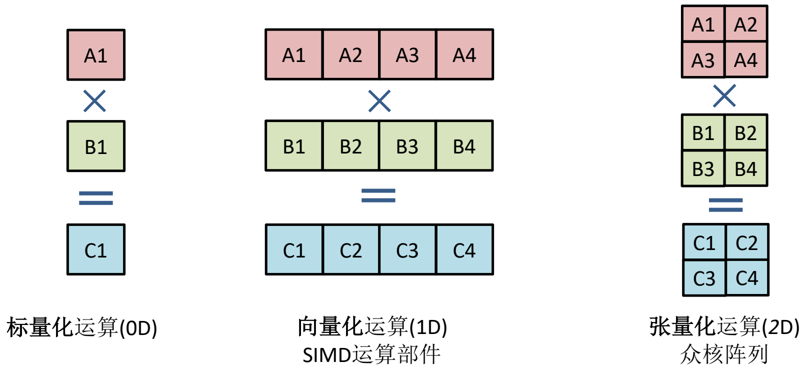


图 3.8　标量化、向量化和张量化乘法运算示意

本研究以申威 26010 处理器为例，提出了一套完整的张量化编程模型设计。本文设计的张量化编程模型提供了以张量数据结构的访存和计算为基本操作的接口，在此称这些接口为**张量化原语（Tensorized Primitives）**。图3.9中左边表示目前申威上已有的并行编程模型所提供的编程接口，包括 OpenACC、SW C intrinsics 和 SW-64 汇编指令集，采用这些接口来设计张量化并行算法是极其繁琐的。OpenACC 是一个简便的并行编程接口，使用时不需改变串行程序代码，只需加入编译指导语句即可实现并行算法设计，它的优化能力很低甚至无法充分利用寄存器通信、向量化等硬件特性。C intrinsics 提供了向量化、寄存器通信等汇编指令的 C 语言接口，但是由于编译器优化能力有限，即使使用 C intrinsics 接口编程也无法保证编译器生成的汇编指令排布方式是最优的，因此常常需要对核心计算部分使用汇编指令进行精细设计。图3.9中右边表示本文提出的张量化编程模型使用张量化原语为基本接口来表达并行算法流程，这些封装好的张量化原语使用 C intrinsics 和 SW-64 汇编指令来具体实现，由于张量化原语封装了硬件相关的优化细节，开发人员使用时只需关注于这些原语的最优调用方式，所以这种编程模型非常适合并行算法的灵活调优。

有了张量化编程模型，下面笔者分别从访存和计算两方面探究使用这种编程模型的程序优化原则。

图 3.9　和申威架构已有的并行编程模型不同，张量化编程模型使用张量化原语为接口来设计并行算法。

3.3.1　张量化访存优化

从核阵列应该以 DMA 方式传输数据，并避免使用 gld/gstd 方式。为了充分利用计算资源并发挥访存计算硬件流水并行能力，核组内 64 个从核应同时进行访存。根据 3.2性能模型，访存密集型的程序执行性能取决于 DMA 操作的效率。由于申威架构的软件 Cache 方式和低内存带宽特性，不加优化的 DMA 访存操作会严重拖累程序运行，本研究提出了如下两个通用的访存优化方法。

优化 1、设计良好的 DMA 访存模式：根据性能模型分析，**每次 DMA 访存应该传输大量连续存储的数据**来摊薄数百个周期启动延迟的开销，否则由于可利用的访存带宽将被延迟开销所拉低，小粒度的 DMA 访存是极其低效的。**DMA 操作最好以 128 位对齐的方式访存**来避免填充数据的出现。跨步内存访问常常不能满足对齐条件，**此时连续访问的数据分块大小应至少为 256 字节**来摊薄填充数据的访存带宽浪费。

优化 2、设计良好的数据排布：内存中数据排布方式对于访存模式有着巨大影响，因此非常有必要将其和算法流程进行联合设计。对于向量化算法设计，一种经典优化技巧[158-159] 是将数据排布从 AoB（Array-Of-Batch）变换为 BoA（Batch-Of-Array）形式①，它使向量化语句操作的一维数组能够被连续访问。申威架构不仅支持以连续一维的方式访存，还支持每个从核以跨步 DMA 方式进行二维数据切片（Tiles）的读取。推广向量化优化技巧，张量化优化采用 ToB（Tile-Of-Batch）变换为 BoT（Batch-Of-Tile）形式预处理数据。图 3.10展示了向量化和张量化中数据排布优化，对于张量化访问三维数据中切片块的问题，不能够通过一次跨步访存完成，如果 N, M 排布在低维，需要 K 次独立 DMA 操作完成内存访问，并且，

---

① 对于复杂结构体形式，通常被称为 AOS(Array-Of-Structures) 和 SOA (Structure-Of-Array)

由于只能跨步地访问 M 维度部分数据，连续访存的数据量非常小。如果将 K 排布在最低维度，连续访问的数据块大小增大了 K 倍，可以显著增加访存带宽。

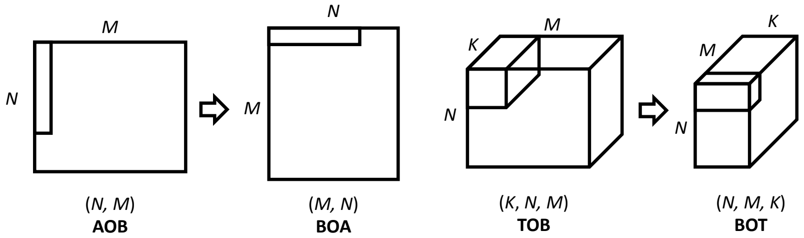


图 3.10　向量化和张量化中的数据排布优化

3.3.2　张量化计算优化

申威架构的从核配备 SIMD 指令执行部件，因此向量化优化技巧在单个从核内仍然适用。在向量化优化之上，张量化优化关注于如何协调 64 个从核共同完成计算任务。

优化 3、通过从核间的寄存器通信来共享数据：复杂的计算任务通常需要在不同从核之间频繁地共享数据，此时不能使用简单的数据并行。由于申威架构的从核间没有 Cache 层次的共享缓冲区，只能依赖基于从核阵列中的行和列通信总线的寄存器级数据共享方案，并行方式设计需要考虑从核阵列的 8×8 拓扑排布，尽量让存在依赖关系数据存储在同行同列的其他从核中。

优化 4、隐藏内存访问开销：通过重叠计算和访存操作来隐藏内存访问开销可以最大化提高访存和计算资源利用率，这种技巧在 SIMD 架构上被广泛采用。比如，KNL 采用内存延迟隐藏是通过同步多线程[160] 或硬件预取[161-162] 隐式地实现的，GPU 依赖于许多线程的快速上下文切换[163]。根据本文之前分析，寄存器通信相当于引入同步操作，这样会破坏了由于竞争访存造成的访存和计算流水线的天然重叠，换句话说，这样 $T_{overlap}$ 会很小。在申威架构上，通过利用 DMA 操作异步性来通过**软件预取**方式显式地隐藏访存开销。通过申请两块内存空间，一块用于计算另一块用于 DMA 访存，在循环方式的计算过程中，一次计算完毕后交换两块空间的索引地址来实现流水操作。图 3.11展示了双缓冲方式的软件预取优化效果。对于计算密集型程序（即 $T_{Compute} > T_{DMA}$ 情况），软件预取可以使 $T_{overlap} = T_{DMA}$，从而完全隐藏访存开销。对于访存密集型程序（即 $T_{DMA} > T_{Compute}$ 情况），软件预取仅减少了四条数据传输总线上 4 个从核 DMA 操作时间，也就是 $T_{DMA}/16$。因此，软件预取应该作为使用核间通信的计算密集型程序的必要手段。

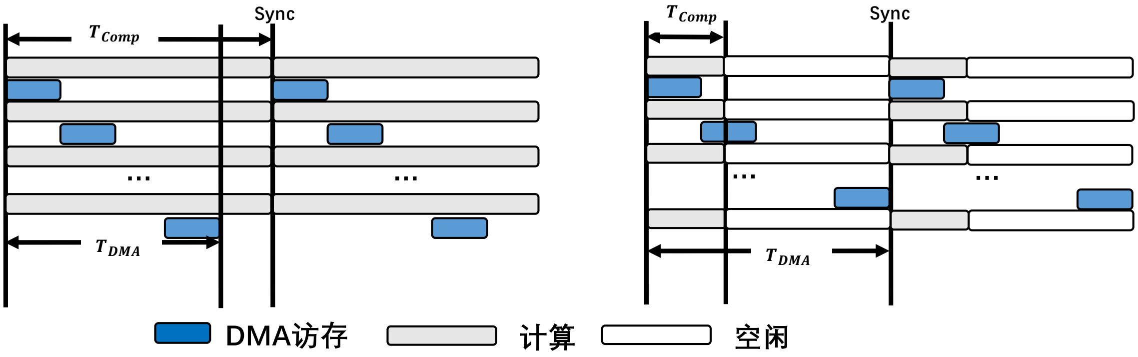
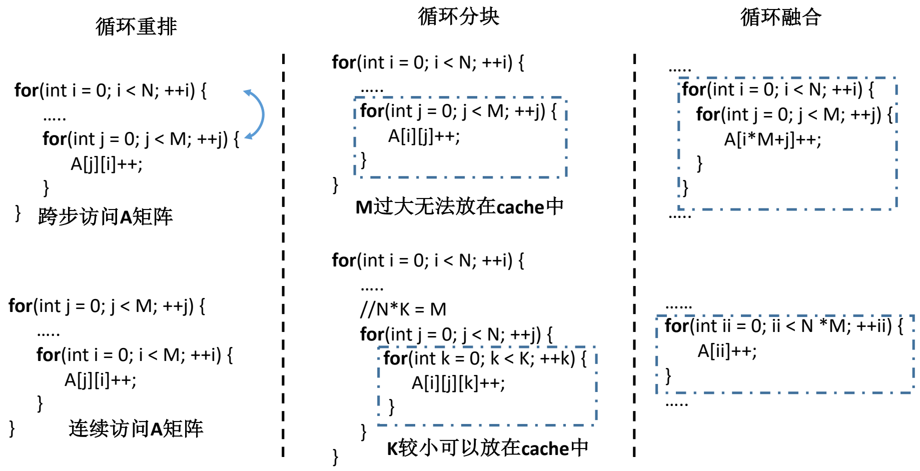


图 3.11　双缓冲软件预取优化效果


循环重排

```
for(int i = 0; i < N; ++i) {
    .....
    for(int j = 0; j < M; ++j) {
        A[j][i]++;
    }
}
```

```
for(int j = 0; j < M; ++j) {
    .....
    for(int i = 0; i < M; ++i) {
        A[j][i]++;
    }
}
```

循环分块

```
for(int i = 0; i < N; ++i) {
    .....
    for(int j = 0; j < M; ++j) {
        A[i][j]++;
    }
}
```

```
for(int i = 0; i < N; ++i) {
    .....
    //N*K = M
    for(int j = 0; j < N; ++j) {
        for(int k = 0; k < K; ++k) {
            A[i][j][k]++;
        }
    }
}
```

循环融合

```
.....
for(int i = 0; i < N; ++i) {
    for(int j = 0; j < M; ++j) {
        A[i*M+j]++;
    }
}
.....
```

```
......
for(int ii = 0; ii < N *M; ++ii) {
    A[ii]++;
}
.....
```

图 3.12　三种循环变换方式示意

3.3.3　张量化计算访存比优化

优化 5、对算法等价变换减少需求带宽：根据定性性能模型分析，高效程序设计需要尽可能减少算法需求带宽 *RBW*，张量化优化也不例外。使用张量化编程模型的深度学习计算核心绝大多数都是张量化计算访存原语的嵌套循环形式，对循环进行等价变换是减少需求带宽的必要手段。这些变换主要有循环分块（Loop Spliting），循环重排（Loop Reorder）和循环融合（Loop Fusion）三种。循环重排：如图3.12左所示，我们通过改变多重循环的执行顺序，从而改变内存的访问方式，以增加高速缓存中的数据复用。循环分块：如图3.12中所示，循环代码执行，由于内层循环太大导致需要的数据无法装载到高速缓存中，从而增加了上层存储器的读取。可以将一层循环拆解为多个循环，从而保证内层循环所需数据能够存储在高速缓存中。因为一层循环和多维张量的某一维度访问一一对应，因此循环的拆分也意味着原来应该连续访问的数据维度被分块。循环融合：如图3.12右所示，当两个循环遍历相同范围且不引用彼此数据时，可以用一个循环替换它们。

3.4　本章小结

本章对申威异构众核架构进行了介绍。申威架构具有不同于其他众核架构的硬件特征，它的 LDM（片上 Cache）是完全以软件方式工作的，它的从核阵列以 8×8 拓扑方式连接，同行同列从核可以进行寄存器级别数据传输。针对这些特点，本章设计了定量和定性的性能分析模型，用来指导申威架构上的并行程序优化。定量模型可以精确估计程序的运行时间，而定性模型可以快速分析程序的优化瓶颈。最后，本章提出了张量化编程模型及优化方法。张量化编程模型将算法表达为以张量为操作目标的访存和计算原语组合，这样有效地弥合了硬件使用方式和算法设计之间的鸿沟。本章是第 4章、第 5章、第 6章并行深度学习算法设计的理论基础。

第 4 章　swGEMM: 基于众核核间通信的矩阵乘法

上一章提出了申威架构上张量化并行编程模型。对于深度学习算子，它们的操作绝大多数都是浮点数乘加运算。矩阵乘法运算作为乘加操作的二维张量化形式，是张量化深度学习算子设计的基础。

本章首先介绍了一个基于众核核间通信方法的矩阵乘法并行优化方法。该方法充分利用寄存器通信的特点，和向量化等优化相结合，理论上可以达到芯片峰值运算速度。将它应用于 LDM 内矩阵数据并封装成标准接口，可以作为张量化编程模型的矩阵乘法原语使用。

然后，本章展示了如何将矩阵乘法原语应用于内存数据的 GEMM 操作，并形成了基于申威架构的矩阵乘法库 swGEMM。以此为例，提出了张量化编程模型使用时遇到问题的解决方案，比如分块大小选择，边缘补零问题，双缓冲优化等。最后，本章展示了使用性能模型对循环变换方案进行调优的效果。

本章实现的张量化矩阵乘法原语最优可达到芯片峰值性能的 97.3%。对于深度学习中常用的狭长形状的矩阵形式，在本章测试中，swGEMM 比神威提供的 BLAS 库平均加速 3.02x。

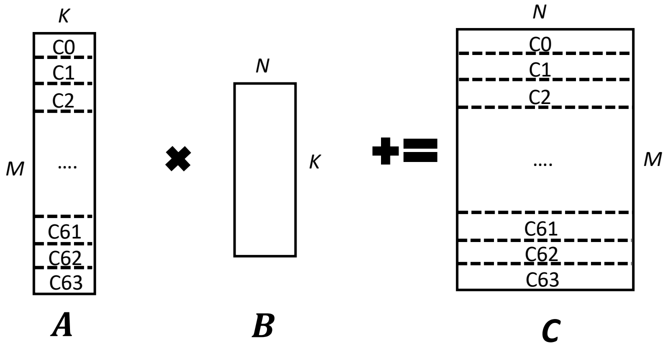
4.1　矩阵乘法原语优化

矩阵乘法原语对存储在 LDM 上的目标矩阵 A，B 和 C 执行 $C+=\alpha A\times\beta B$ 运算。本节将介绍基于申威架构众核核间通信的并行矩阵乘法优化方法，并以此为理论，实现高效的矩阵乘法原语，作为张量化编程模型的原语接口。

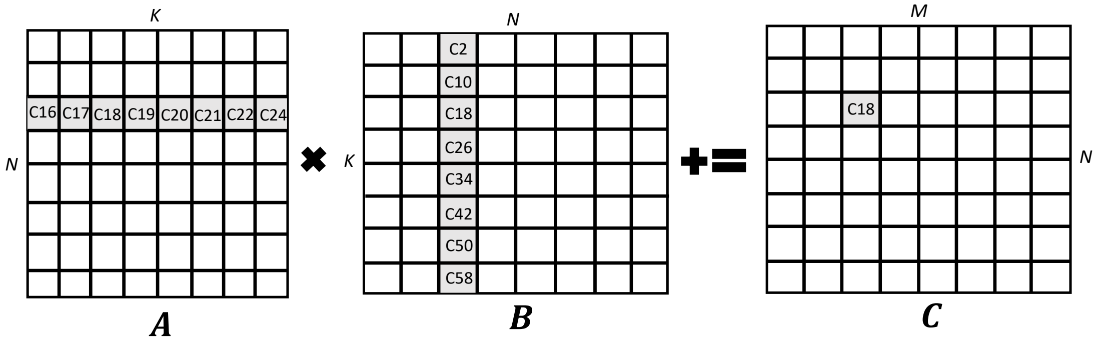
4.1.1　分布式矩阵存储与通信方式

矩阵 A, B, C 都需要分布式地存储于 64 个从核的 LDM 空间，因此设计一个分布式的存储形式是首要任务。如图4.1，申威架构上可以存在两种矩阵分布方案，在此称之为**无重叠矩阵分布方案**和**有重叠矩阵分布方案**。

有重叠矩阵分布方案应用于 A，B 其中一个非常小，以至于可以把它完整放置在一个从核的 LDM 中的情况。以 B 较小的情况为例，每个从核读取全部 B 矩阵，然后将 A 矩阵的 M 维度分割为 64 份，每个从核读取其中的 1/64。这种矩阵分布方案由于要重复存储 B 矩阵，一是浪费 LDM 的存储空间，二是每次计算都反复加载 B，导致严重浪费内存到 LDM 的带宽。鉴于有重叠方案应用范围狭窄且实现简单，本章将不做重点介绍。



有重叠矩阵分布方案



无重叠矩阵分布方案

图 4.1　下图，无重叠矩阵分布：从核阵列上进行矩阵乘法时，数据划分方案和从核数据依赖关系。灰色的数据块部分表示：计算 C 矩阵的 18 号从核结果，其需要的数据在 8×8 网格中的存放位置。上图，有重叠矩阵分布：K 较小时候矩阵的从核划分方案。其中 B 矩阵被固定在 LDM 中，将 A 和 C 按照行和列分别切分为 64 份。

无重叠矩阵分布方案将三个矩阵无重叠地存储在 64 个从核的 LDM 上。为了适应从核阵列 8× 8 网格状的拓扑结构，它将三个矩阵行列均分 8 块，总计 64 个矩阵块按顺序存放在 64 个从核上。这种数据划分方案下，任意两个从核上没有重复数据，因此可以保证没有内存到 LDM 数据传输带宽的浪费。但是，每个从核用本地的 1/64 数据进行矩阵乘法运算只能得到 1/8 的最终结果，如果想要获得全部最终结果，每个从核都需要寄存器通信从远端从核获取数据。

无重叠矩阵分布方案需要使用寄存器通信获取远程数据到本地通用寄存器，矩阵乘法的数据依赖关系恰好适用于同行同列间寄存器通信的特性。在此，笔者通过一个简化版本的 4×4 网格矩阵乘法，向读者展示矩阵乘法的寄存器通信策略。如图4.2所示，输入矩阵 A、B 和输出矩阵 C 被分成 4×4 块分布在 16 个从核上，整个计算将分 4 步完成：**第 0 步**，每一行第 0 列的从核通过行寄存器通信总线发送它本地的卷积核数据 $B(i,0)$ 到同行的第 1 ~ 3 列的其他从核；每一列的第 0 行的从核通过列通信总线发送它本地的输入数据 $A(0,j)$ 到同列的第 1 ~ 3 行的其他从核。

从核 (i, j) 收到信息后，用接收到的 $B(i, 0)$ 和 $A(0, j)$ 数据进行矩阵乘法运算，将结果加到结果矩阵 $C(i, j) += B(i, 0) \times A(0, j)$。**第 1 步**，每一行第 1 列的从核发送它本地的 $B(i, 1)$ 到同行的第 $0, 2, 3$ 列的其他从核；每一列的第 1 行的从核发送它本地的 $A(1, j)$ 到同列的第 $0, 2, 3$ 行的其他从核。从核 (i, j) 收到信息后，用接收到的 $B(i, 1)$ 和 $A(1, j)$ 数据进行矩阵乘法运算，将结果加到结果矩阵 $C(i, j) += B(i, 1) \times A(1, j)$。**第 2 步**，每一行第 2 列的从核发送它本地数据 $B(i, 2)$ 到同行的第 $0, 1, 3$ 列的其他从核；每一列的第 2 行的从核发送它本地的 $A(2, j)$ 到同列的第 $0, 1, 3$ 行的其他从核。从核 (i, j) 收到信息后，用接收到的 $B(i, 2)$ 和 $A(2, j)$ 数据进行矩阵乘法运算，将结果加到结果矩阵 $C(i, j) += B(i, 2) \times A(2, j)$。**第 3 步**，每一行第 3 列的从核发送它本地的 $B(i, 3)$ 到同行的第 $0, 1, 2$ 列的其他从核；每一列的第 3 行的从核发送它本地的输入数据 $A(3, j)$ 到同列的第 $0 \sim 2$ 行的其他从核。从核 (i, j) 收到信息后，用接收到的 $B(i, 3)$ 和 $A(3, j)$ 数据进行矩阵乘法运算，将结果加到结果矩阵 $C(i, j) += B(i, 3) \times A(3, j)$。图4.2展示了，以位于 2 行 1 列（下标从 0 开始）的从核为视角，获得最终结果 $C(2, 1)$ 的过程。

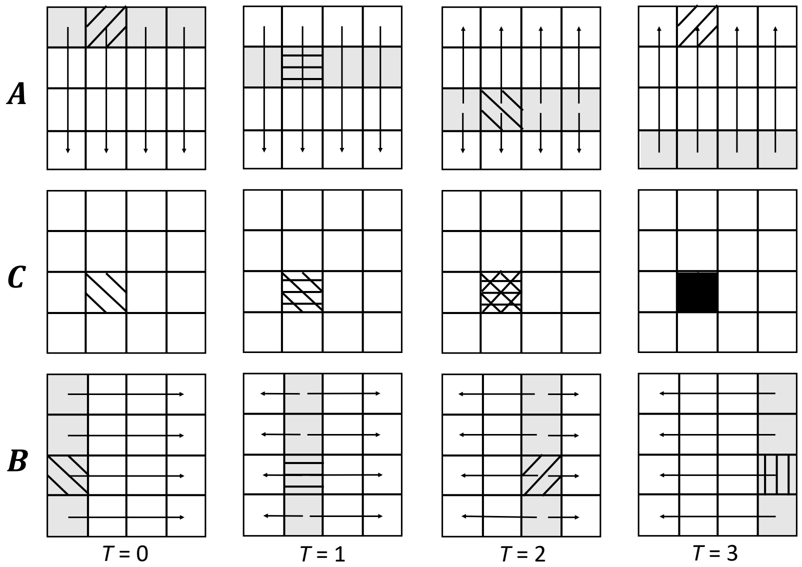


图 4.2　核间通信矩阵乘法的寄存器通信机制

4.1.2　增加寄存器数据局部性优化

即使采用高效的数据分布和寄存器通信，从 LDM 读取数据到寄存器可能存在访存受限的情况从而影响计算性能，因此需要对循环进行等效变换。LDM 内部的矩阵乘法计算可以表示成三层循环的实现方式，假设循环长度分别为 M，N 和 K。本研究要设计一个循环调度和分块方案来增加寄存器内部的数据复用，从而减少

对其上层存储器（LDM）的访问次数。类似章节3.3.3所述，增加寄存器数据复用采用对三层矩阵计算循环进行调度和分块来实现。本研究的变换方案如算法2描述所示，它对长度为 M 的循环进行分块，分块大小为 B_M；对长度为 N 的循环进行分块，分块大小为 B_N。B_M 和 B_N 是一个较小值，可以被调度到循环内层，它们对应的数据可以随之缓存在寄存器内反复参与运算。

Algorithm 2 基于寄存器通信共享数据的 LDM 内矩阵乘法的循环变换方案

```
INPUT: id ← 本从核的编号
 1: $cid \leftarrow id\%8; rid \leftarrow id8$
 2: for $T = 0 : 7$ do
 3:     for $cM = 0 : B_M : M/8 - 1$ do
 4:         for $cN = 0 : B_N : N/8 - 1$ do
 5:             从 LDM 读取 $C(cM : cM + B_M, cN : cN + B_N)$ 数据到寄存器数组 $R_c$
 6:             for $cK = 0 : K/8 - 1$ do
 7:                 if $cCore == cid$ then
 8:                     从本地 LDM 读取 $A(cK, cM : cM + B_M)$ 数据到寄存器数组 $R_a$，并广播到同
                        行其他从核的寄存器
 9:                 else
10:                     接收行寄存器广播数据到寄存器数组 $R_a$
11:                 end if
12:                 if $cCore == rid$ then
13:                     从本地 LDM 读取 $B(cN : cN + B_N, cK)$ 数据到寄存器数组 $R_b$，并广播到同列
                        其他从核的寄存器
14:                 else
15:                     接收列寄存器广播数据到寄存器数组 $R_b$
16:                 end if
17:                 for $cBm = 0 : B_M - 1$ do
18:                     for $cBn = 0 : B_N - 1$ do
19:                         $R_c(cBm, cBn) += R_a(cBm) \times R_b(cBn)$
20:                     end for
21:                 end for
22:             end for
23:             将寄存器数据 $R_o$ 存储到 LDM $C(cM : cM + B_M, cN : cN + B_N)$
24:         end for
25:     end for
26: end for
```

算法2还需要进行向量化优化来增加执行效率。为了达到 LDM 和寄存器通信的理论带宽 46.4 GB/s，每个时钟周期内，从核需要完成 32 Byte 数据的 LDM 访问。若以一个 8 Byte 标量浮点数为单位访存，那只能利用 LDM 到寄存器有效带宽的 1/4。因此，利用向量化的指令访问 LDM 也是充分利用 LDM 到寄存器带宽的关键。通过申威指令集的特殊向量指令，可以实现 LDM 读取数据或者寄存器通信接收数据。具体来说，算法2第 8 行和第 13 行可以通过 `VLDR`/`VLDC` 和 `LDDER`/`LDDEC` 指令实现。`VLDR(VLDC)` 将 32 Byte 长度的数据从连续内存区域中装入到一个向

量寄存器中，并进行同行（同列）广播。`LLDER(LLDEC)` 从内存装载一个双精度浮点数，同时写到向量寄存器的四个双精度浮点向量元素的位置，并进行向（列向）广播。申威指令集并没有对单精度浮点数提供相应功能的指令，它们只能通过两条指令实现。使用 `VLDS`(向量化读入四个单精度浮点数) 和 `PUTR/PUTC` 实现 `VLDR/VLDC` 的功能。使用 `LDSE`(向量化读入一个单精度浮点数并扩展成向量) 和 `PUTR/PUTC` 实现 `LLDER/LLDEC` 的功能。算法2第 10 和第 15 行，可以使用 `GETR/GETC` 获取同行/列其他从核寄存器的通信数据到本地寄存器单元。具体指令情况可以参考章节3.1.4。

算法2有四种向量化实现方式，它们有两点区别。第一个区别在于是对 M 还是 N 维度进行向量化读取，为了能够使用 `VMAD` 进行四个浮点数的加乘运算，本研究只能对 M 或 N 进行向量化读取，而另一个维度只能读取一个元素并扩展成向量参与运算。第二个区别在于从核矩阵是按照行优先还是列优先存储。表格4.2列举了四种向量化实现的区别，图4.3展示了方案 1 的向量化方案。

表 4.1　四种向量化实现方案

	方案 1	**方案 2**	**方案 3**	**方案 4**
A 访存指令	`VLDC`	`VLDR`	`LLDEC`	`LLDER`
A 存储方式	行优先	列优先	行优先	列优先
B 访存指令	`LLDER`	`LLDEC`	`VLDR`	`VLDC`
B 存储方式	列优先	行优先	列优先	行优先

章节 3.2.3提出的定性性能模型可以用来寻找最优的分块策略。对于向量化方案 1，LDM 与寄存器数据传输的需求带宽 $RBW_{LDM->REG}$ 计算如公式4-1所示，其它方案以此类推。它表示在寄存器使用数量小于 32 的限制下，最大化 $RBW_{LDM->REG}$，其中 B_N 项前的系数为 4，是因为向量化读取并扩展指令相当于读取 4 倍的内存数据。求得 $B_M = 4, B_N = 16$ 是此优化问题的最优解，此时 $RBW_{LDM->REG} = \frac{(16+4\times4)\times8Byte}{(2\times16\times4)/(1.45GHz*8)} = 23.2GB/s < 46.4GB/s$。这种情况下，输出结果缓存在 4×4 个寄存器中，每次执行 8 次数据加载操作和 16 次浮点加乘运算。换而言之，$RBW_{LDM->Reg}/MBW_{LDM->Reg} = \#P0\_valid/\#P1\_valid$=1/2，8 次数据加载操作可以和 16 次浮点加乘运算同时进行。综上所述，通过本研究提出的循环变换方法，可以保证 LDM 和寄存器之间不存在访存受限的问题。

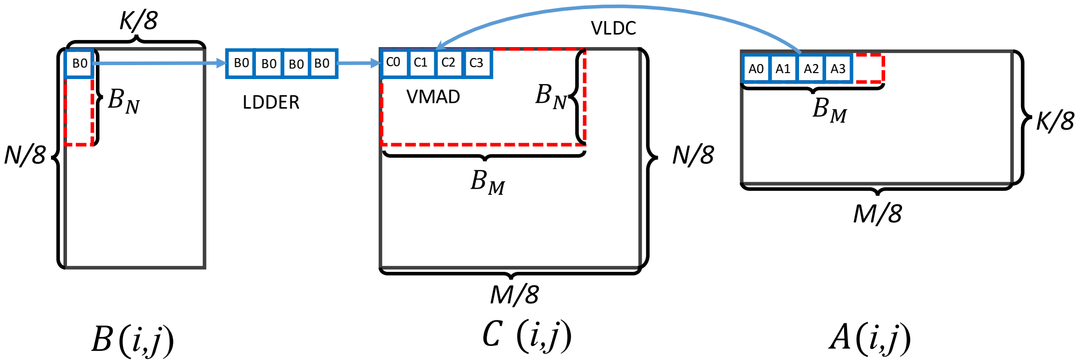


图 4.3　从核内寄存器分块方案。最大的黑色方块表示存储在 LDM 内数据。红色虚线表示数据存储在寄存器内。蓝色方框内是一个 256-bit 寄存器中内数据。

$$RBW_{LDM\rightarrow REG} = \frac{(B_M + 4B_N)DS}{2B_M \times B_N/T} \quad s.t.\ B_M + B_N/4 + B_M \times B_N/4 < 32 \tag{4-1}$$

寄存器分块方案在增加寄存器内数据局部性的同时，也带来一些对矩阵分块限制。比如 N 的大小需要是 $16\times 8 = 128$，M 大小需要是 $4\times 8 = 32$ 的倍数。但这并不影响张量化原语适用性，后面会介绍如何进行快速补零来避免这个问题。

4.1.3　增加计算单元效率优化

采用增加寄存器局部性的策略后，从 LDM 加载数据到寄存器将不再是性能瓶颈，此时浮点运算执行时间直接影响最终性能。根据上一章节性能模型的描述，计算时间 $T_{compute}$ =max$(\#P0_cycle, \#P1_cycle)/1.45GHz = \#P0_cycle = \#P0_valid + \#P0_idle$。为了追求极致的性能，本小节目标在于彻底消除计算流水线闲置周期 $\#P0\_idle$。

由于编译工具限制，需要手动汇编排布来设计最优的指令发射方式。算法2的最内层 *cK* 循环完成 16 次向量化浮点乘加（VMAD）计算。理想情况下，用 *P0* 执行部件进行 16 个向量加乘运算，*P1* 执行部件同时并行地处理数据的加载、存储、控制传输和其他标量操作，这需要一个极其精巧的指令排布方式来保证每个时钟周期以流水线方式完成一条浮点运算指令的发射。但是仅仅使用申威的从核编译器无法获得一个最优的指令排布方案，sw5cc -O3 编译生成的汇编代码，核心段 16 个 VMAD 操作需要 25 个时钟周期才能完成，此时的执行效率为 $EE = 16/25 = 64\%$。本研究需要手动设计指令重新排序的优化，使内层核心循环计算效率最高。其核心优化原则有两点：一是增加双指令流水线的指令集并行，另一个是避免指令依赖造成的写后写（WAW）或写后读（RAW）依赖造成的流水线卡顿。

清单 4.1展示了本文设计的指令排布方案。根据当前从核寄存器通信的角色不同，理论上需要四段汇编代码实现行发送列发送、行发送列接收、行接收列发送、行接收列接收四种不同角色。为了方便表述，本研究定义 4 个输入矩阵 *A* 向量的寄存器为 $A0 \sim A3$，4 个 *B* 矩阵向量寄存器是 $B0 \sim B3$，存储 16 个输出 *C* 矩阵向量的寄存器为 $C0 \sim C15$。并且，LDM 中输入矩阵 *A* 的起始位置为 `MEM_A`，*B* 矩阵的起始位置为 `MEM_B`。汇编代码采用宏定义方式，`FETCHA/FETCHB` 根据向量化方案不同，被定义为相应寄存器通信的发送或接收指令。笔者这里列举了采用向量化方案 1 中某从核承担行发送和列接收角色的情况，这种情况是最复杂的，原因在于本研究需要 4 个 *B* 矩阵 LDM 偏移操作。它包含 8 个对 *A*、*B* 的向量化访存操作 (`FETCHA/FETCHB`)，5 个整型数据标量操作 (`ADD`) 去更新读取数据的 LDM 起始位置 (`MEM_A,MEM_B`)，16 个浮点数加乘指令 (`VMAD`) 进行核心运算，1 次 `SUBW` 来更新循环控制变量 *cK*，和一次转移判断指令 (`BGT`)。此时 *P1* 执行部件需要发射 15 条指令，*P0* 执行部件需要发射 16 条指令。

```
1  #Rescheduled Pipeline
2  .Kloop_SecPutBPutA:
3  VMAD (B0, A0, C0); ADD(LDM_A_P, K, LDM_A_P)
4  VMAD (B0, A1, C1); FETCHA (A3, LDM_A_P)
5  VMAD (B0, A2, C2); FETCHB (B3, 96(LDM_B))
6  VMAD (B1, A0, C4); ADD(LDM_A, 8, LDM_A)
7  VMAD (B1, A1, C5); ADD(LDM_A, K, LDM_A_P)
8  VMAD (B2, A1, C9); SUBW (cK, 1, cK)
9  VMAD (B0, A3, C3); NOP
10 VMAD (B1, A2, C6); ADD(LDM_B, M, LDM_B)
11 VMAD (B2, A3, C19) FETCHB (B0, LDM_B)
12 VMAD (B1, A3, C7); FETCHB (B0, 32(LDM_B))
13 VMAD (B3, A0, C12); FETCHA (A0, LDM_A)
14 VMAD (B3, A1, C13); FETCHA (A1, LDM_A_P)
15 VMAD (B2, A2, C10); ADD(LDM_A_P, K, LDM_A_P)
16 VMAD (B3, A2, C14); FETCHA (A2, LDM_A_P)
17 VMAD (B2, A3, C11); FETCHB (B2, 64(LDM_B))
18 VMAD (B3, A3, C15); BGT cK .Kloop_SecPutBPutA
```

Listing 4.1　A 矩阵转置且 B 矩阵不转置情况下，LDM 内 GEMM 运算内层指令排布方法。

不同指令的延迟和数据依赖性应该在指令发射排布方案中被充分考虑。清单 4.1中 *C* 寄存器的数据不作为 16 个 `VMAD` 指令的目标寄存器，所以不需要考虑 *C* 寄存器的影响，但需要精心设计加载 *A*, *B* 寄存器的位置。向量化访存指令执行完 4 个时钟周期之后，目标寄存器才会得到数据。因此，它应该在目标寄存器参与计算之前 4 个时钟周期被发射。同理，`VMAD` 指令发射的 7 个时钟周期之后，目标寄存器才能被继续使用。清单 4.1中 3-18 行用 16 个时钟周期完成了 16 个 VMAD 运算，**它的执行效率为 100%**。

除了最内层循环 16 个 VMAD 指令排布，在外层循环的汇编指令排布上还需要许多优化技巧。申威架构每个从核只有 32 个寄存器，已经分配了 24 个寄存器用于存放计算数据，31 号寄存器不可修改固定为 0，只剩下 7 个寄存器用于存储中间变量。计算需要的内存地址和循环迭代变量信息也要存储在寄存器中，因此，需要对循环变量和寄存器映射关系进行优化设计。向量整理指令（参见章节3.1.4）可以实现标量数据打包和解包。比如，实现过程中笔者把最常用的内层循环长度信息 `cK` 和 A，B，C 矩阵的起始地址信息一起打包存储，同样，N，M，cN，cM，T 等外层循环控制变量也被打包存储。

另外，申威指令集对于双精度和单精度浮点数运算支持程度存在差异。同时完成向量化访存和寄存器广播通信操作的一系列指令 `VLDR/VLRC/LLDER/LLDEC` 只支持双精度浮点数版本，而没有单精度浮点数对应的指令。而深度学习数据一般都以单精度形式表示，本研究有如下两种策略来支持单精度数据的运算。策略一，用向量化访存和广播两条指令实现。由于访存和寄存器通信指令都必须在 $P1$ 流水线发射，对于需要行列广播的从核，这种策略导致 $P1$ 流水线多出 8 条待发射指令，从而破坏笔者精心排布的指令发射设计，导致指令级并行度降低。策略二，在 LDM 中将单精度浮点数转换为双精度。之后，计算单元仍然使用双精度数据进行矩阵运算，再将运算结果转换回单精度形式。这样会引入输入输出 LDM 内部转换的开销，并且会消耗更多的 LDM 空间，但是由于执行部件利用更加充分，策略二性能往往显著优于策略一，本研究也采用策略二实现单精度浮点矩阵乘法原语。

4.2　原语使用示例：张量化 GEMM 运算

矩阵乘法原语的最简单的应用场景就是 GEMM 运算，本节将介绍张量化编程模型下的 GEMM 运算的优化方法。以此为例，笔者将介绍使用张量化原语使用时遇到的一些典型问题的解决方案，包括如何寻找最优循环变换方案、如何边界处理、如何隐藏访存开销。张量化优化可以对形状狭长矩阵的 GEMM 运算取得良好效果，这种形状矩阵广泛应用于深度学习中。

4.2.1　深度学习中 GEMM 运算的挑战

在深度学习中，矩阵乘法广泛应用在全连接算子和卷积算子中。表4.2总结了深度学习算子实现过程中对矩阵乘法的需求。应用场景和矩阵布局方式笔者在下一章内会详细介绍，此时请注意矩阵的类型和位置两项信息。“申威 26010”上配置了名叫 xMath 的针对众核优化的 BLAS 库，其中包括了 GEMM 的例程[164]。本文称内存中矩阵乘法运算为“GEMM 运算”或“GEMM 例程”以和张量化的“矩阵乘法

原语”作区分。

xMath 的 GEMM 例程不能满足深度学习算子库搭建的需要。首先，xMath 是闭源的软件，仅提供封装好的二进制静态库供用户使用。xMath 只提供单个核组内存数据的 GEMM 运算，因此无法提供本研究需要的矩阵乘法原语。其次，xMath 无法满足深度学习应用中的 GEMM 运算的性能需求。GEMM 运算可以表示为 $C+=\alpha A\times B+\beta C$ 的形式。其中，矩阵 A 大小为 (M,K)，矩阵 B 大小为 (K,N)，矩阵 C 大小为 (M,N)。Goto 等人[165] 根据矩阵长宽两个维度大小不同情况将 GEMM 运算细分为如下几类。两个维度都很大的被称为 M（Matrix）类矩阵；其中一个维度很小为长条形状矩阵被称为 P（Panel）类矩阵；两个维度都很小成块状的矩阵被称为 B（Block）类矩阵。根据参与 GEMM 操作的矩阵形状，可以更加精确地表示为 GEMM、GEMP、GEPB 操作之一。由于申威架构特点导致小粒度的 DMA 访存操作很容易造成带宽不能被充分利用。因此，将 GEMM 形式的分块方式套用到 GEMP、GEPB 上，会导致 DMA 带宽利用率很低，从而无法获得令人满意性能，这是导致 xMath 对这些 GEMM 运算性能不佳的原因。BLAS 库中 GEMM 例程和深度学习矩阵运算之间的需求不匹配的情况也普遍存在于其它架构上，在 2019 年最新研究中，Masliah 等[166] 指出了现有 BLAS 库中尺寸较小矩阵的 GEMM 运算并没有充分优化。

表 4.2 深度学习对矩阵乘法的需求

应用场景	位置	A 布局	B 布局	C 布局	类型
1. 显式 CONV (正)[5.1.1节]	内存	$(K_rK_cN_i,N_o)$	$(K_rK_cN_i,C_oR_o)$	(N_o,C_oR_o)	GEMM,GEMP GEPB
2.Winograd [5.1.3]	内存	(BT,N_i)	(N_i,N_o)	(BT,N_o)	GEMP, GEPB
3. 隐式 CONV (batch)[5.1.2节]	LDM	(N_i,B)	(N_i,N_o)	(B,N_o)	矩阵乘法原语
4. 隐式 CONV (image)[5.1.2节]	LDM	(K_cN_i,N_o)	$(K_cN_i,64C_o)$	$(N_o,64C_o)$	矩阵乘法原语
5.LSTM [5.2节]	内存	(PB,N_i)	(N_i,N_o)	(PB,N_o)	GEPB
6. 全连接层 [5.2节]	内存	(B,N_i)	(N_i,N_o)	(B,N_o)	GEMM

综上，本章希望设计一个矩阵乘法库，使用上一节优化的矩阵乘法原语，来高效支持 GEMP、GEPB 形式非典型矩阵的 GEMM 运算，弥补 xMath 的不足。

4.2.2　张量化优化方法

因为 LDM 的空间有限，必须对内存中的矩阵进行分块，以满足矩阵乘法原语调用要求。图4.4展示了不同情况下矩阵乘法操作的内存分块方案。对 GEMM 形式的矩阵乘法，M、N、K 三个维度要足够大以进行分块，本研究定义它们对应的分块大小为 B_M,、B_N、B_K。对 GEMP 形式的矩阵乘法，只有两个维度可以分块。本研究在此只介绍对 M、K 分块的情况。N、K 分块情况可以以此类推。对 GEPB 形式的矩阵乘法，只有一个维度可以进行分块，本研究在此考虑 M 维分块的情况。

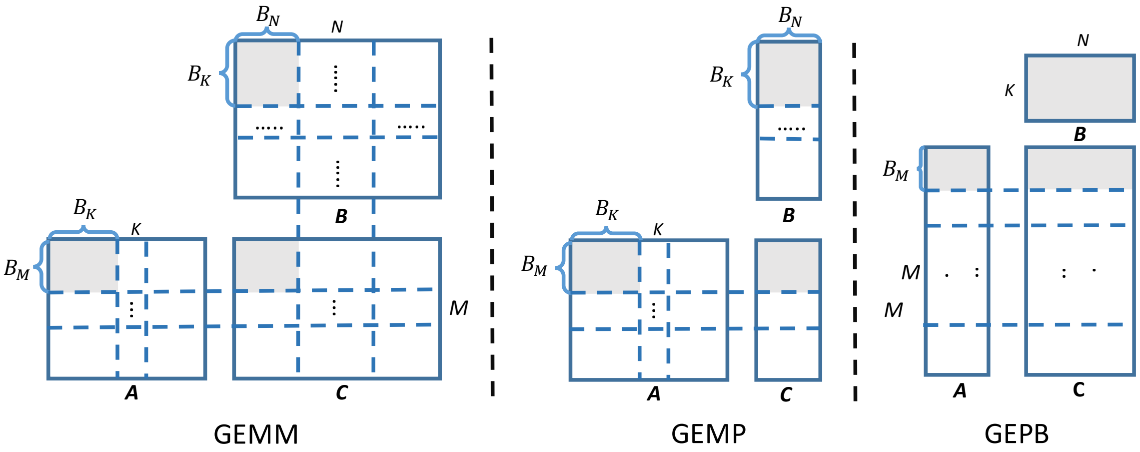


图 4.4　内存矩阵乘法针对不同参数特点设计的三种分块方案

对于最复杂的 GEMM 形式，本研究展示三种循环调度方式和它们的需求带宽。如图4.5左图所示，第一种方式将 K 循环调度到最内层，内层循环计算前固定大小为 (B_M, B_N) 的 C 矩阵块在 LDM 内，然后每次读取 (B_M, B_K) 大小的 A 矩阵分块，和 (B_M, B_K) 大小的 B 矩阵分块进行矩阵乘法计算以更新 C 矩阵的分块。这种方式的需求带宽与 B_N 和 B_M 成反比，而与 B_K 无关。如图4.5中图和右图所示，本研究也可以将 K 循环调度到倒数第二层，内层循环计算前固定 A/B 矩阵块在 LDM 内，然后每次读取 B/A 矩阵分块和 C 矩阵分块，进行矩阵计算更新 C 矩阵的分块后写回 LDM。这两种循环调度方式的算法需求带宽与 B_K 和 B_M/B_N 成反比。由于内层循环需要既读又写 C 矩阵块，因此 $\frac{1}{B_K}$ 项前的比例系数为 2。对于 GEMP 形式的矩阵乘法计算，可以得到和 GEMM 方式类似的结论，区别在于 GEMP 只对 M，K 两个维度分块。根据是否将 K 循环调度到循环最内层，可以有两种循环调度方式，K 在最内层时的需求带宽只与 B_M 成反比，而 K 在外层的需求带宽与 B_K 成反比，比例系数同样为 2。

对于 K 被放置在内层循环和倒数第二层循环两种情况，需求带宽公式性质明显不同。K 在内层时，增大 B_N 和 B_M 有助于减少算法需求带宽。K 在倒数第二层

```
for (cM = 0; cM < M; cM+= $B_M$)
 for (cN = 0; cN < N; cN += $B_N$) {
  load C block into LDM as $C_{blk}$
  for (cK = 0; cK < K; cK += $B_K$) {
   load A block into LDM as $A_{blk}$
   load B block into LDM as $B_{blk}$
   GEMM($A_{blk}$, $B_{blk}$, $C_{blk}$)
  }
  store $C_{blk}$ into memory
}
```

$$RBW_{LDM \to REG} = \frac{(B_K B_M + B_N B_K)DS}{2B_M B_K B_N / T} = (\frac{1}{B_N} + \frac{1}{B_M})\frac{DS}{2/T}$$

```
for (cM = 0; cM < M; cM+= $B_M$)
 for (cK = 0; cK < K; cK += $B_K$) {
  load A block into LDM as $A_{blk}$
  for (cN = 0; cN < N; cN += $B_N$) {
   load C block into LDM as $C_{blk}$
   load B block into LDM as $B_{blk}$
   GEMM($A_{blk}$, $B_{blk}$, $C_{blk}$)
   store $C_{blk}$ into memory
  }
}
```

$$RBW_{LDM \to REG} = \frac{(B_K B_N + 2B_N B_M)DS}{2B_M B_K B_N / T} = (\frac{1}{B_M} + \frac{2}{B_K})\frac{DS}{2/T}$$

```
for (cN = 0; cN < N; cN+= $B_N$)
 for (cK = 0; cK < K; cK += $B_K$) {
  load B block into LDM as $B_{blk}$
  for (cM = 0; cM < M; cM += $B_M$) {
   load C block into LDM as $C_{blk}$
   load A block into LDM as $A_{blk}$
   GEMM($A_{blk}$, $B_{blk}$, $C_{blk}$)
   store $C_{blk}$ into memory
  }
}
```

$$RBW_{LDM \to REG} = \frac{(B_K B_M + 2B_N B_M)DS}{2B_M B_K B_N / T} = (\frac{1}{B_N} + \frac{2}{B_K})\frac{DS}{2/T}$$

图 4.5　GEMM 例程外外三层循环的调度方式，图左中右分别对应图4.4的 GEMM、GEMP 和 GEPB 情况。

循环时，增大 B_K 和 B_N、B_M 其中之一有助于减少需求带宽。之前分析可知增大 B_K 也有助于提升矩阵乘法原语执行效率。但是，这种方式内层循环需要读入一次 C_{blk}，会造成额外的需求带宽，对大多数情况，反而弊大于利。因此，本研究优先选择 K 在内层的循环调度方案。

上述循环变换有助于减少算法需求带宽，一些技巧可以增加 DMA 实测带宽。如章节3.1.2描述的 DMA 访存时间模型所示，内存对齐方式对实测带宽提高至关重要。因此，在内存分配过程中将内存起始地址按照 128 Byte 进行对齐对提升访存带宽大有裨益。另外，由于矩阵乘法存在寄存器通信会减少 DMA 和计算的重叠时间，需要使用双缓冲区策略来实现软件流水线以达到通信计算重叠的效果。

4.2.3　自动调优分块大小

循环分块的参数 B_M、B_N、B_K 对矩阵乘法的性能影响很大。图 4.6展示了对以 $M = N = K = 1024$ 为参数的矩阵乘法不同分块大小可以获得的性能。在 81 种可行的分块方案中，最差的分块方案所能获得的性能仅为 66.83 GFLOPS，而最优的方案可以获得 363.59 GFLOPS，二者相差 5 倍多。由此可见，设计一种快速准确的自动分块调优方案对于 GEMM 运算的性能至关重要。

性能估计以第 3.2章的性能模型为依据。对于访存时间，GEMM 操作的跨步 DMA 时间可以采用公式 3-2进行精确估计。而且，张量化的算法设计有利于申威架构上计算时间估计。因为张量化原语的参数空间变化是离散且有限的，可以设计性能函数来表示搜索空间。矩阵乘法是一个内层计算指令数和外层循环开销的线性函数，鉴于核心段代码是本研究可控的模板，矩阵乘法计算时间可以通过公式 4-2来拟合，其中 a、b、c、d 是可学习的参数。

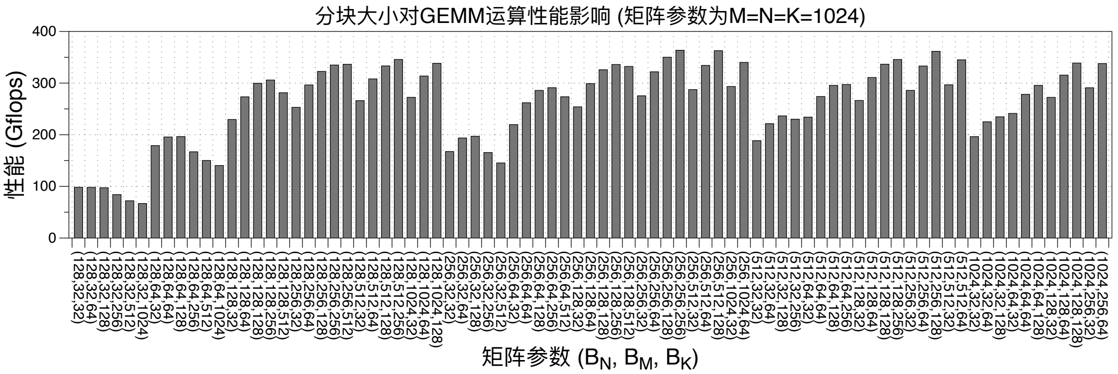


图 4.6　循环分块方案对矩阵乘法的性能影响示例

$$T_{GEMM\_Primitive} = aK + b\frac{KM}{vecM \times 4} + c\frac{KMN}{4} + d. \tag{4-2}$$

笔者采集了 1505 组测试参数进行拟合，拟合结果如图4.7所示，本研究的线性模型可以准确表达出计算时间。按照本研究的性能模型分析最内层循环 c 的系数，$cN_M B_N B_K = 8 * \frac{B_M}{8*16}\frac{B_N}{8*4}\frac{B_K}{8} * \frac{17}{1e9}$，由此推算出 $c = 2.86e - 12$，这和本研究的拟合结果 $c = 3.10e - 12$ 相比，误差非常小，还不到 8%。

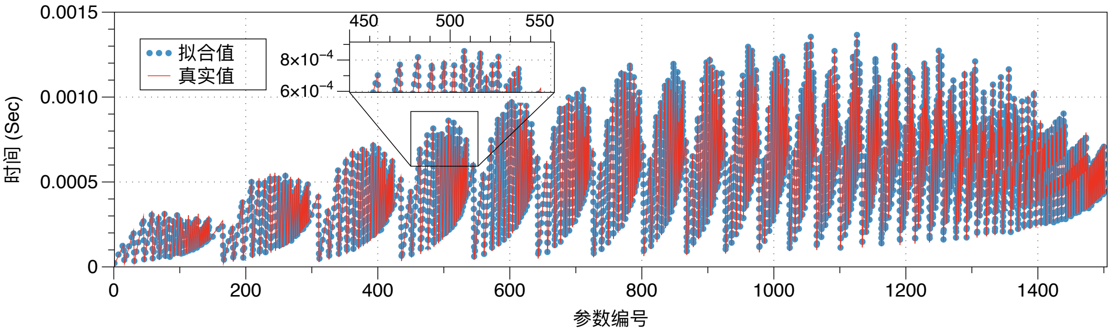


图 4.7　使用最小二乘法对采集数据进行拟合的结果

4.2.4　边界处理

在张量化原语使用时，调用参数需要满足一定的限制条件，因不满足限制条件导致的边界问题在张量化算法设计中普遍存在。具体到 GEMM 运算来说，其中一种实现的分块大小 B_M、B_N、B_K 必须是 128、32、8 的倍数，如果某一维度不是该维度分块大小的整数倍，需要使用补零后矩阵进行计算，计算结果再去零。通常的做法是将原矩阵拷贝到一块更大的、按照对齐要求分配的内存地址空间中去，

这种方案被称为“全拷贝”方式。这样做一方面由于内存带宽有限，拷贝全部矩阵大小仍可能是非常耗时的。另一方面，新分配矩阵本身也会占用大量内存空间。

为了减少对矩阵整体进行内存拷贝的巨大开销，本文提出了一种轻量的补零和去零操作。对于不满足分块大小的部分，可以先分配一段初始置零的辅助内存空间，并将原矩阵有效的部分拷贝到辅助内存空间。在 GEMM 操作过程中，本文将本该访问原矩阵边界的内存地址重定位到辅助空间的内存地址上去。去零操作和补零操作类似，在 LDM 内矩阵乘法原语计算过程中使用辅助内存，计算结束后将保存边界信息的辅助数组再拷贝到原矩阵中去。和原来全部内存拷贝相比，使用本研究的轻量补零去零操作会显著减少内存消耗和带宽需求。内存消耗从原来的 $O(M*K)$ 降低到 $O(M+K)$，内存带宽需求从原来的 $O(M*K)$ 降低到 $O(M+K)$。

4.3　实验结果

4.3.1　矩阵乘法原语性能

本章使用 1505 组参数对矩阵乘法原语进行测试，它们几乎覆盖了 LDM 空间范围允许的所有可行参数空间。图 4.8展示了测试的性能结果，横轴表示 LDM 矩阵尺寸参数，由于篇幅限制，本章只随机抽取 1/25 的结果进行展示。使用本章提出的众核核间通信矩阵乘法优化方法，原语的最优性能可达 722.95 GFLOPS，达到**峰值性能的 97.3%**。在 1505 组参数里，76.1% 的情况下矩阵乘法性能超过 500 GFLOPS（峰值性能的 67%）。那些性能比较差都是 K 维度比较小的算例，因为 K 维长度对性能影响最为直接，它是算法2的最内层循环，增加它的长度可以减少循环启动和终止时间开销的比例。

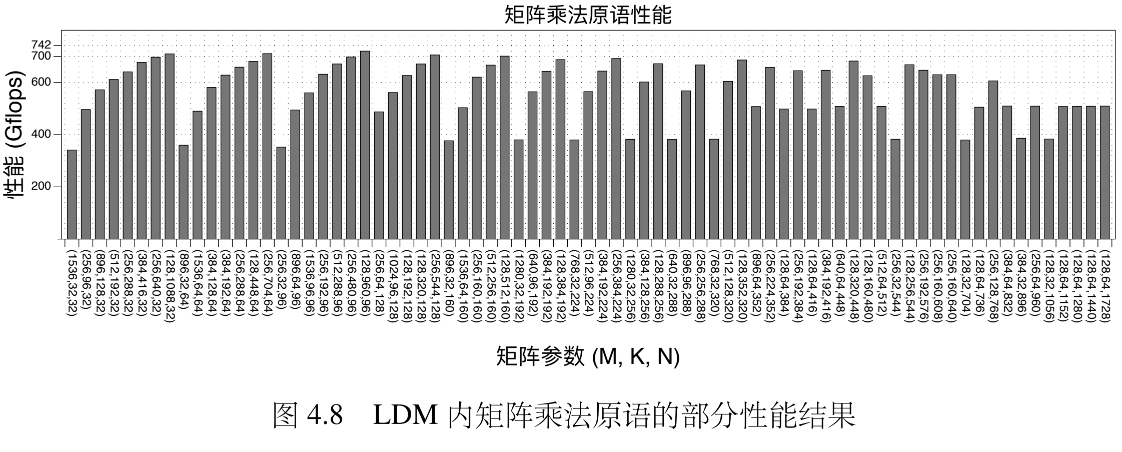


图 4.8　LDM 内矩阵乘法原语的部分性能结果

4.3.2　GEMM 运算性能

章节4.2提出的 GEMM 操作优化被用来实现成一个名叫 swGEMM 的函数库。本章以 xMath（机器自带的 BLAS 库）的 GEMM 例程为比较对象，展示了 swGEMM 的性能。根据测试矩阵的形状特点，本章的实验方式分为两类。第一类采用典型 GEMM 运算参数进行测试，第二类对特别针对 GEMP、GEPB 类的特殊 GEMM 运算参数进行测试。

4.3.2.1　通用矩阵乘法测试

本研究对清单 4.2 中 216 组参数和清单 4.3 中 343 组参数矩阵乘法进行实验。后者参数都满足 128 倍数，因此不需要补零操作。而前者参数都是 100 倍数，因此使用矩阵原语时存在边界问题，需要对 M、N、K 都进行补零操作，使其满足张量化原语使用的限制条件。图4.10展示本章轻量补零优化的效果，它对所有参数非对齐情况都显著降低补零开销时间比例。图中测试用例挑选了全拷贝开销最大的前 12 个加以展示，轻量级补零优化可以将补零时间占比降低到 5% 以下。

表4.3展示了通用矩阵乘法测试中 swGEMM 和 xMath 中 GEMM 例程性能对比结果。在矩阵尺寸对齐情况下，73% 组（250/343）取得了平均 31.6% 的加速，而其它组的减速效果仅为 6.6%。在矩阵尺寸非对齐情况下，96% 组（207/216）取得了平均 49.8% 的加速，而其它组减速效果仅为 4.3%。非对齐情况下更优的表现，要归功于本章提出的轻量级补零优化。

表 4.3　通用矩阵乘法测试中 swGEMM 和 xMath 的性能对比

	矩阵尺寸对齐		矩阵尺寸非对齐	
	数目	平均加速比	数目	平均加速比
加速情况	250	+31.6%	207	+49.8%
减速情况	93	-6.6%	9	-4.3%

为了给读者更直观的印象，图4.9展示了 swGEMM 和 xMath 性能对比的柱状图。横轴的矩阵参数通过间隔抽取清单 4.2 中的参数获得，并按照从大向小排列。对于 M、N、K 都很大的情况，swGEMM 和 xMath 性能相当。而对于参数比较小的情况，xMath 性能直线下降，而 swGEMM 此时则表现相对良好。

4.3.2.2　特殊矩阵乘法测试

本章使用清单 4.4 的脚本生成 180 组特殊形状矩阵的 GEMM 运算方案，并对它们进行测试。参与这些 GEMM 运算的矩阵，有两个维度相对较小，属于 GEMP 类

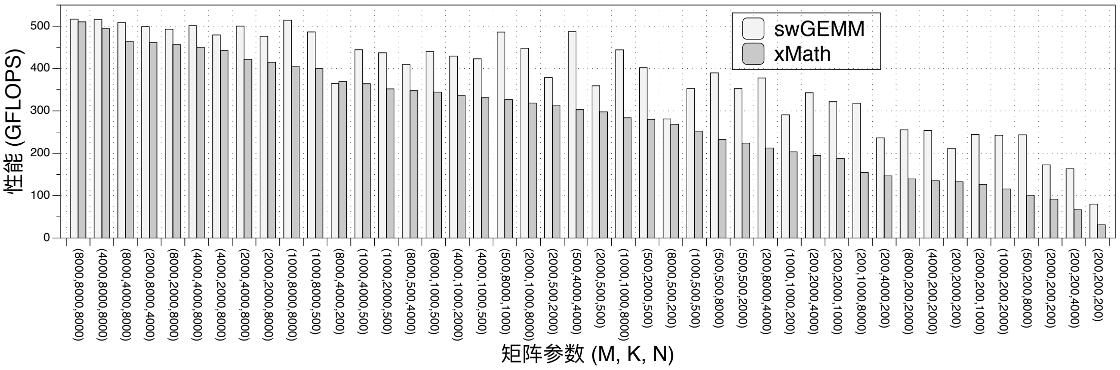


图 4.9　swGEMM 和 xMath 在典型 GEMM 运算测试实验中的性能对比结果

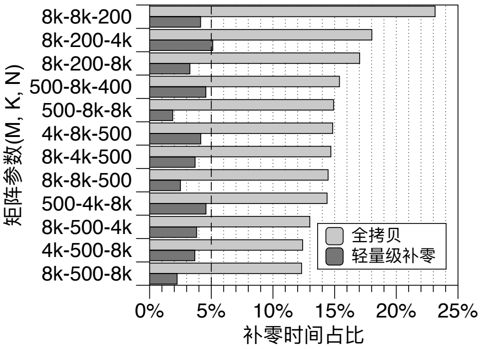


图 4.10　轻量级补零优化效果

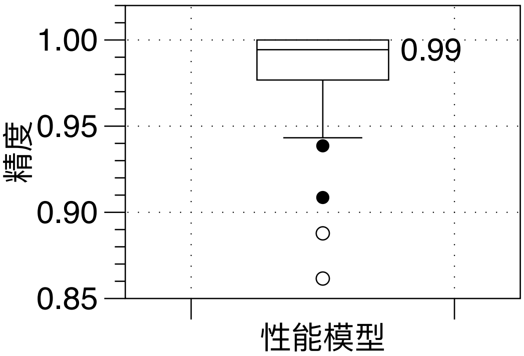


图 4.11　性能模型找出最优结果和实际最优结果比值的箱线图

或 GEPB 类的 GEMM 运算。在其中 162 组 (占总比例的 90%) 的测试上，swGEMM 相对 xMath 取得了加速，有加速效果的测试中，平均加速比 **3.02x**，在有性能损失的测试中，平均性能损失仅为 **9.0%**。

为了更直观展示 swGEMM 的效果，图4.12展示了随机抽取近 1/4 测试用例的结果。对于那些 xMath 性能极差的 GEPB 情况，swGEMM 加速效果非常明显。对于三个维度参数都比较大 GEMP 情况，比如 (256,2048,819)，(512,2048,32768) 等，xMath 性能稍微更好一些。这是因为 xMath 采用单独开辟一块双精度矩阵方式预先完成单双精度转换，因此计算过程中不存在精度转换开销，而 swGEMM 并没有选择这种浪费内存空间的实现方式，它需要在每次调用 GEMM 原语前后进行精度转换操作，因此引入一定额外开销导致在某些参数情况下效果不如 xMath。

4.3.2.3　性能模型调优效果

本研究比较了暴力枚举找到的实际最优循环分块方案性能和本研究性能模型推导出的循环分块方案的性能。图4.13展示了性能模型推导出的最优结果和暴力

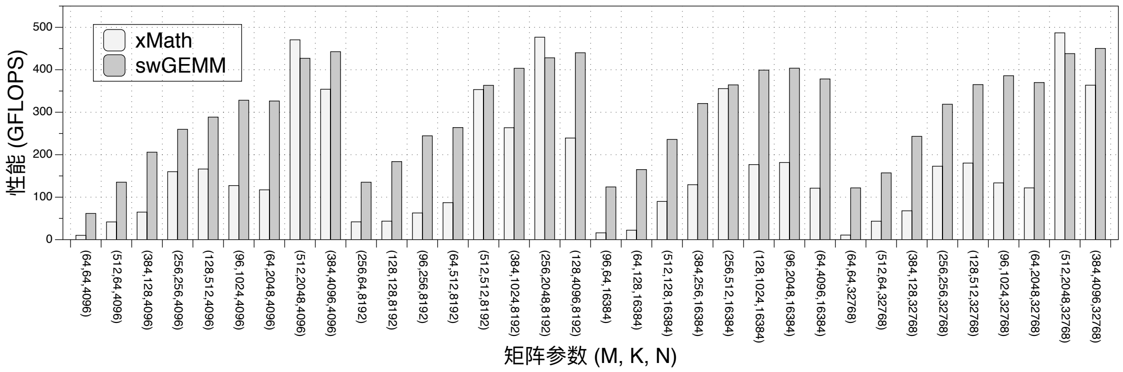


图 4.12　swGEMM 和 xMath 在 GEMP/GEPB 特殊 GEMM 运算测试实验中的性能对比结果

```
for M in 200 500 1000 2000 4000 8000;
 for N in 200 500 1000 2000 4000 8000;
  for K in 200 500 1000 2000 4000 8000;
    test_swGEMM $M $N $K
```

清单4.2 典型 GEMM 运算非对齐参数生成脚本。

```
for M in 256 512 768 1024 2048 4096 8192;
 for N in 256 512 768 1024 2048 4096 8192;
  for K in 256 512 768 1024 2048 4096 8192;
    test_swGEMM $M $N $K
```

清单 4.3 典型 GEMM 运算对齐参数生成脚本。

```
for M in 4096 8192 16384 32768
 for N in 64 128 256 512 1024 4096;
  for K in 64 96 128 256 384 512 ;
    test_swGEMM $M $N $K
```

清单4.4 GEMP/GEPB类特殊GEMM 操作参数生成脚本。

枚举找到的实际最优结果部分比较，此处抽取了 60 组做展示，可以观察到它们性能结果之间差别极小。箱线图4.11展示了图4.13的统计信息，性能模型得到的最优结果平均误差**不超过 1%**。暴力枚举需要遍历运行所有可行分块方案空间，而性能模型只需要进行一次四则运算几乎没有时间开销，因此性能模型方法相比暴力枚举可以节省成百上千倍的调试时间。

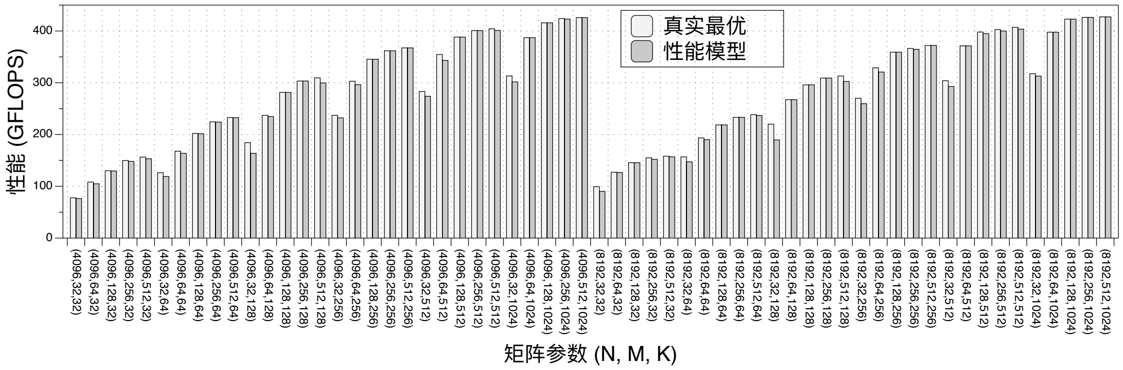


图 4.13　性能模型找出最优结果和实际最优结果部分比较

4.4　本章小结

本章提出了基于众核核间通信的矩阵乘法并行算法，并使用它实现了一个名为 swGEMM 的矩阵乘法库。该方法应用于申威众核处理器上，实现了操作 LDM 内数据的矩阵乘法原语，它的最优性能可以达到峰值的的 97.3%。矩阵乘法原语可以作为张量化算法设计的基础接口，并将它应用于实现 BLAS 库中的 GEMM 运算。为了解决张量化编程模型使用时遇到的边界处理和访存开销等问题，本研究提出了一系列操作优化方法，用于优化 swGEMM 矩阵乘法库。swGEMM 对深度学习中常见的形状狭长矩阵的 GEMM 运算中，相对机器自带的 BLAS（xMath）平均加速比达到 **3.02x**。

第 5 章　swDNN: 深度学习算子的张量化

深度学习算子是深度学习的计算核心。由于硬件特性上的显著差异，在 NVIDIA GPU 和 Intel Xeon Phi 等商用众核架构上的算子优化无法应用于国产申威架构。在上一章的矩阵乘法原语基础上，本章使用张量化编程模型对深度学习算子进行设计。一个实用高效的算子库设计面临多方面挑战。

首先，算子库设计需要对不同访存计算特性的计算核心进行各个击破，因此任务量巨大。根据 Roofline[154] 模型中计算密度（Arithmetic Intensity）指标划分，深度学习中的计算核心可以分为访存密集型和计算密集型两大类。卷积、全连接、LSTM 层的算子的计算密度为 $O(n)$ 量级的，它们属于计算密集的操作。如章节 3.3所论述，申威架构很容易存在访存带宽和计算能力不匹配的情况。这些算子的张量化优化重点在于如何尽可能增加计算访存比。池化、激活函数、批量归一化等操作是计算密度小于 $O(1)$ 量级的访存密集型的操作，申威架构的事务型内存特点给访存模式造成一定的限制，这些算子优化的核心在于设计良好的访存模式，以提升 DMA 带宽的使用率。

其次，算子库设计需要兼顾效率和通用性。深度学习算子的运算对象为多维张量，这意味算子设计需要足够健壮，才能够承受多个输入参数的变动。DMA 访存模式，从核缓存空间，向量化长度等硬件特性会带来算法设计的限制。这些限制导致特定算子的设计方案仅仅在某些参数范围是高效的，而超特定范围后变得低效。因此，设计通用解决方案以满足多种应用场景的需要也是算子库设计重要考量。

本章主要贡献包括:

- 本章实现一个名为 swDNN 的算子库，可以高效地支持目前深度学习使用的主流算子。
- 对于深度学习中最复杂的卷积算子，swDNN 设计了三种张量化卷积算子实现，它们分别是基于显式矩阵乘法的优化、基于隐式矩阵乘法的优化和基于 Winograd 的优化。卷积算子可以达到 60% 左右的运算效率，并且在非常广泛的输入参数空间上表现稳定。
- 对于 LSTM 算子，相比传统方法，张量化编程模型可以发掘更大的优化潜力。在 LSTM 的张量化优化中，通过控制张量化访存位置来更加高效缓存共享参数，从而极大缓解了访存受限程度。相比 cuDNN 中使用的优化方法，swDNN 获得了平均 2.6x 左右加速效果。

5.1　卷积算子

卷积层是卷积神经网络 (CNN) 的核心，它使 CNN 具有提取图片复杂特征的能力。卷积算子占整个 CNN 计算时间的 90% 以上，因此设计它的高效实现方法是所有深度学习算子库工作的重点内容[82]。笔者先简单介绍卷积算子工作原理，然后给出申威架构上张量化卷积优化方法。

和 2D 卷积操作不同，深度学习卷积层使用的卷积操作被称为“多通道卷积”。它的输入特征图为 $x^l \in \mathbb{R}^{B\times N_i\times R_i\times C_i}$，它表示 minibatch 大小为 B，通道数为 N_i，图片大小为 $R_i \times C_i$ 的四维张量。输出特征图为 $x^{l+1} \in \mathbb{R}^{B\times N_o\times R_o\times C_o}$，它表示 minibatch 大小为 B，通道数为 N_o，每张图片大小为 $R_o \times C_o$ 的四维张量。定义卷积核为 $W \in \mathbb{R}^{N_i\times N_o\times K_r\times K_c}$，它表示输出通道为 N_o（某些文献中称为卷积核数目），输入通道数为 N_i（某些文献称为通道数），每个卷积核大小 $K_r \times K_c$。在本文中，本研究规定变量的描述如表5.1所示。

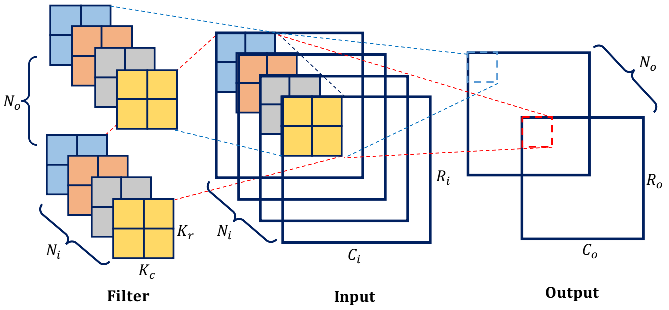
表 5.1　卷积层的参数表示

N_i	输入通道数	N_o	输出通道数
R_i	输入特征图图片高度 (行数)	C_i	输入特征图图片宽度（列数）
R_o	输出特征图图片高度 (行数)	C_o	输出特征图图片宽度（列数）
K_r	卷积核的高度	K_c	卷积核的宽度

如公式5-1所示，卷积层的正向传播可以使用多通道卷积操作实现。多通道卷积操作，本文表示为 $CONV$，它的计算方式如图5.1所示。需要首先将二维卷积核翻转 180 度，然后以相等的步幅逐步滑过输入图片的每个位置，并使用卷积核的元素与输入特征图对应位置的元素进行乘积，将结果相加以获得该当前位置处的输出。

$$x^{l+1} = CONV(x^l, W^l) \tag{5-1}$$

如公式5-2所示，反向传播过程需要两次操作分别计算本卷积层参数梯度和输入特征图的敏感度。参数梯度可以表示为损失函数 L 对 W^l 的偏导数 $\frac{\partial L}{\partial W^l}$。敏感度是和 x^l 尺寸一致的四维张量 $\delta^l \in \mathbb{R}^{B\times N_i\times R_i\times C_i}$，表示为损失函数 L 对 l 层特征图 x^l 的偏导数 $\delta^l = \frac{\partial L}{\partial x^l}$。反向过程的多通道卷积操作和正向略有区别，一方面，计算 δ^l 时，需要把 δ^{l+1} 周围补零再进行多通道卷积，补零大小为 $(K_r - 1, K_c - 1)$。另一方



```
for (cB = 0; cB < B; cB++)
 for(cCo = 0; cCo < Co; ++cCo)
  for(cRo = 0; cRo < Ro; ++cRo)
   for(cNi = 0; cNi < Ni; ++cNi)
    for(cNo = 0; cNo < No; ++cNo)
     for(cKr = 0; cKr < Kr; ++cKr)
      for(cKc = 0; cKc < Kc; ++cKc) {
        cRi = cRo + cKr; cCi = cCo + cKc;
        output(cB,cNo,cRo,cCo) +=
      input(cB, cNi, cRi, cCi)*
   Filter(cNo, cNi, Kr−1−cKr, Kc−1−cKc);
       }
```

图 5.1　左图：多通道卷积操作示意。右图：对应的一个朴素的循环结构代码实现。

面，计算 $\delta^l_{n,c}$ 时，不需要对卷积核旋转 180 度。更准确地说，这种形式的多通道卷积操作被称为“互相关操作”，本文表示为 $CORR$。

$$\delta^l = CORR(add\_pad(\delta^{l+1}), W^l) \qquad \frac{\partial L}{\partial W^l} = delete\_pad(CORR(\delta^{l+1}, x^{l+1})) \qquad (5\text{-}2)$$

除了需要将卷积核翻转，卷积操作和互相关操作计算方式完全相同，这里统称这两种操作为卷积算子。如章节 2.2.1介绍，卷积算子可以按照空间域方法、频率域方法和 Winograd 方法实现。对比这三类方法，以 FFT 为核心的频域方法可以减少浮点运算次数，但不适合在申威架构上实现。首先，它对内存带宽和存储空间有更高的需求，相较于申威架构提供的超过 3 TFlops 计算能力，32 GB 的 DDR3 存储空间和 128GB/s 的峰值访存带宽并不占优势。另外，基于频域的方法对于大卷积核更有效，但是目前主流网络大多采用 3×3 卷积核，所以在适用性上它并没有比基于空间域的方法更有优势。最后，FFT 方法不如矩阵乘法更适合张量化编程模型。综合考虑，本文选取空间域方法和 Winograd 方法作为申威架构上卷积算子的优化方法。

本章将介绍三种张量化卷积算子设计方法。如图 5.2和图 5.3，它们分别是**基于隐式矩阵乘法**的优化（Implicit-GEMM-CONV）、**基于显式矩阵乘法**的优化（Explicit-GEMM-CONV）和**基于 Winograd** 的优化（Winograd CONV）。三种优化方案可以适应不同算法变种和参数空间，满足目前各种主流 CNN 结构对卷积算子的要求。

5.1.1　基于显式矩阵乘法的卷积优化

在基于显式矩阵乘法的卷积（Explicit-GEMM-CONV），卷积操作可以显式转化为矩阵乘法操作，使用显式矩阵乘法实现卷积正向传播公式如5-3所示。输入张量首先通过 $im2col$ 操作变换成 Toeplitz 矩阵形式。为了方便，在此命名大小为 (N_i, R_i, C_i) 的输入特征图 x^l 为 im，输出大小为 $(N_iK_rK_c, R_oC_o)$ 的 Toeplitz 矩阵为 col。变换过程如图5.4所示，首先从 im 拷贝一块 $K_r \times K_c$ 的图片切片，然后将它们展开

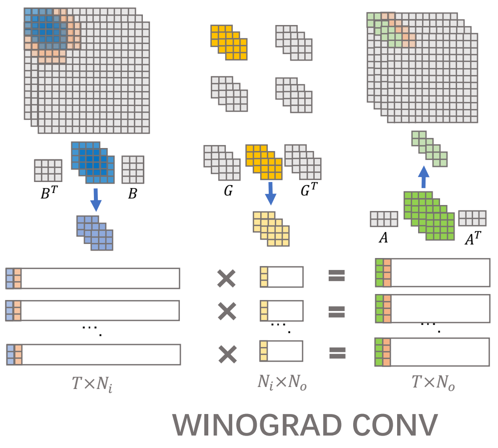
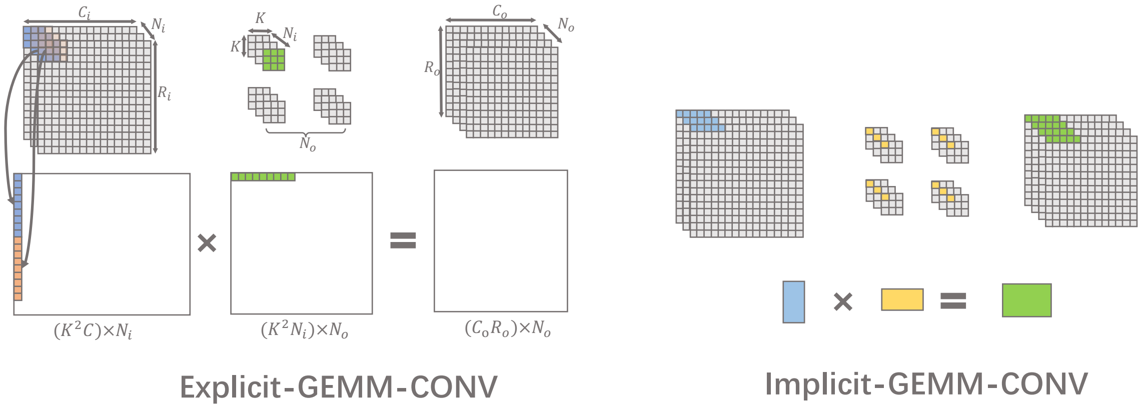


图 5.2　Winograd-CONV 张量化卷积实现方式



Explicit-GEMM-CONV

Implicit-GEMM-CONV

图 5.3　Implicit-GEMM-CONV 和 Explicit-GEMM-CONV 张量化卷积实现方式

成 col 矩阵一列中的部分元素。将卷积核沿着 N_o 维度展开，可以视为 $(N_iK_rK_c, N_o)$ 大小的矩阵 W。通过对 col 矩阵和 W 矩阵执行 GEMM 运算，可得到输出特征图结果。

使用 Explicit-GEMM-CONV 实现反向传播公式如5-4所示。卷积核的梯度计算仍需要对输入特征图 x^l 进行 $im2col$ 变换。计算敏感度 δ^l 需要对权重和后一层敏感度 δ^{l+1} 进行矩阵乘法，然后使用 $col2im$ 变换成 im 形式。$col2im$ 的计算过程是 $im2col$ 的逆过程，生成的 im 图片的元素为 $im2col$ 对应 col 矩阵列元素的加和结果。

$$u^l = im2col(x^l) \quad x^{l+1} = u^l \times W + b \tag{5-3}$$

$$u^l = im2col(x^l) \quad \frac{\partial E}{\partial W} = \delta^{l+1} \times u^l \quad \delta^l = col2im(W \times \delta^{l+1}) \tag{5-4}$$

GEMM 操作可以使用笔者上一章实现的矩阵乘法库 swGEMM 完成，但是 $im2col/col2im$ 很容易成为性能的瓶颈。如果 AlexNet 网络卷积层全部使用 Explicit-GEMM-CONV 实现，使用主核进行 $im2col/col2im$ 时间占 AlexNet 训练总计算时间 70% 以上。根据章节3.3总结的申威架构优化原则，必须要对 $im2col/col2im$ 过程进行张量化访存优化。

在众核上为 $im2col/col2im$ 设计能得到良好访存模式的并行任务划分并不简单。一个简单的方法可以采用数据并行方式，按照 im 一个图片切片为独立任务单位划分，每个从核读入一个 $K_r \times K_c$ 大小的图片切片到 LDM 并展开成一维数组，然后拷贝到 col 矩阵某列的对应位置。读取图片切片需要大量跨步 DMA 访存，且访存的连续数据块大小为 K_c 个元素，由于卷积核的行数 K_c 很小，因此这种实现将无法充分利用从核阵列的访存带宽，更严重的是，相邻图片切片有大量重叠数据，重复读取会严重浪费内存带宽。所以，需要结合算法特点精心设计一个访存模式更高效地进行内存的读写。

根据章节3.3.1提出的张量化访存优化原则，本研究设计的高效的张量化访存方案如图5.4所示。它以输入特征图图片的一行元素处理任务为基本任务单位进行从核间的并行划分。对于 $im2col$ 操作，每个从核读入 im 中连续 C_i 个元素的一行图片元素，然后复制到 col 矩阵 $K_r \times K_c$ 行的对应位置。这样划分后，相邻任务完全独立，不存在读写内容重叠的情况。对于 $col2im$ 操作，每次从 col 矩阵读入 $K_r \times K_c$ 行连续的 C_o 个元素，这些元素会一一对应 im 中图片的一个切片。然后，将应该写入相同 col 位置的元素进行加和，形成一个长度为 C_i 的数组，最后，使用 DMA 写该数组到 im 的对应位置。

这种方案还很容易实现一些卷积使用时的额外技巧，比如卷积补零（Zero-Padding）和卷积跨步（Strided Convolutions）。对于卷积补零，只需要事先申请长宽分别为 $C_o = C_i + K_c - 1, R_o = R_i + K_r - 1$ 的 LDM 内存，并将其初始化为零。将 im 的 C_i 元素复制到长度为 $C_i + 2 * P_r - 1$ 的 LDM 空间中起始地址偏移为 P_r 的位置，对于需要补零的行位置不进行 DMA 输出，即可自动完成行补零操作。对于卷积跨步，只需要在 LDM 中跨步地复制 im 数据到 col 缓冲中即可。

上述方法在输入输出特征图片非常大的情况可以获得极佳的 DMA 访存带宽，但对于 C_i，C_o 维度较小的情况，该方法无法达到最优的内存带宽使用率。举例来说，对于 C_i=8 的 $im2col$ 任务，DMA 连续读入内存块大小仅为 32 Byte，DMA 带宽利用率将很低。针对这种情况，可以采用章节3.3.1中提到的数据排布优化方法来改进。可以将 im 的数据排布为 (R_i, C_i, N_i, B) 形式，此时 B 放在四维张量的最低维，这样，$imcol/col2im$ 过程可以连续读/写 $B \times C_i$ 个元素，写/读 $B \times C_o$ 个元素，

从而大大增加内存带宽利用率。

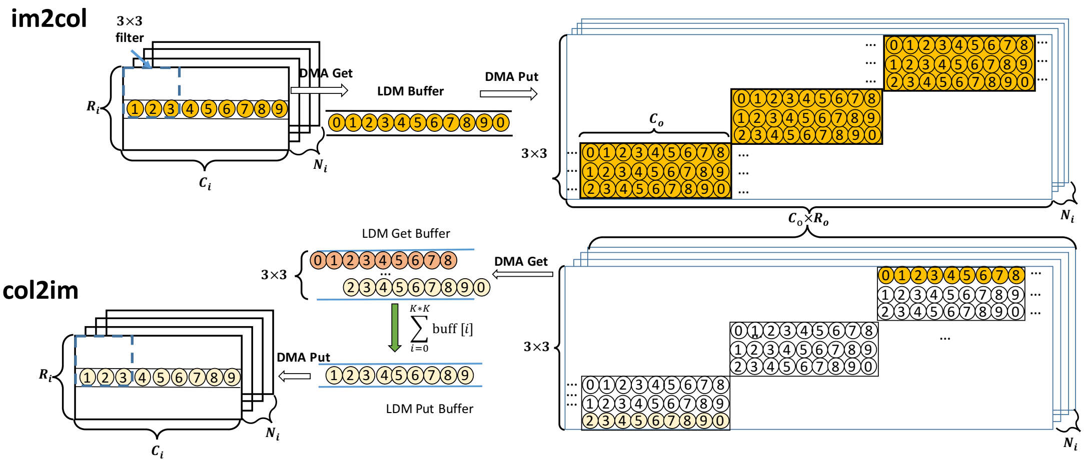


图 5.4　每个从核对图片一行进行 imcol/col2im 变换。为了便于读者理解，在本例中卷积核为 3×3，输入图片宽度为 $C_o = 9$。

对 *col* 和 *W* 矩阵进行 GEMM 运算可以调用笔者上一章 swGEMM 矩阵乘法库来实现。GEMM 操作也会出现边界问题，章节4.2.4已经设计了轻量级的补零操作来优化它。在深度学习应用场景中，可以优化 *col* 和 *W* 矩阵的分配方式来完全避免这部分开销，章节7.1会详细介绍它在深度学习框架层面的解决方案。

5.1.2　基于隐式矩阵乘法的卷积优化

基于显式矩阵乘法的卷积优点在于可以适配各种参数的卷积层，但是这种方法缺点也十分明显。第一，*im2col* 和 *col2im* 为 DMA 访存操作，此时计算部件完全处于闲置状态，这是对计算资源的浪费。第二，中间结果矩阵 *col* 大小是输入张量大小的 $K_c \times K_r$ 倍，由于存在大量冗余信息，会造成存储资源的额外开销。

本小节提出基于隐式矩阵乘法的卷积方法（Implicit-GEMM-CONV）。如图5.3右边所示，它每次按需读取小块数据进行矩阵乘法，从而充分重叠计算和访存时间，且不引入额外内存开销。从循环角度重新审视卷积算法，它的实现可以写成如图5.1的七层循环形式。按照张量化编程模型，以标量乘加操作为核心的七层嵌套循环可以等价变换为以矩阵乘法原语实现的嵌套循环。使用章节3.2描述的性能模型分析，隐式矩阵乘法卷积优化过程的关键在于：如何设计循环变换方法以最大限度减少需求带宽（RBW）和实测带宽（MBW）的比值，（即章节 3.2.3的 RBW/MBW）。

5.1.2.1　Batch-Size-Aware 循环变换方案

算法3展示了一种可以减小算法需求带宽的循环调度方案。这种方案对原始七层循环进行重排，将 N_i，B，N_o 维度对应循环放置在最内层，从而构成了一个张量化矩阵乘法运算。原语操作的目标通过 DMA 读取到 LDM 中，其中 W 的大小为 (N_o, N_i)，D_i 的大小为 (N_i, B)，D_o 大小为 (N_o, B)。由此可知，内层循环访存量为 $N_i \times B + N_i \times N_o$ 个浮点数，矩阵原语计算量为 $2N_i \times N_o \times B$。根据这两项指标，公式5-5推导了内存到 LDM（MEM→LDM）最小需求带宽 $RBW_{MEM->LDM}$。T 是数据在 LDM 中进行计算理想情况下可获得的最高性能，DS 是数据类型的大小。由公式5-5可知，这种循环变换方案的 $RBW_{MEM->LDM}$ 与 B 和 N_o 呈反比，鉴于实际应用中 N_o 都很大，所以它的性能受 B 大小影响更加严重，因此被称之为 Batch-Size-Aware 版本。

Algorithm 3 Batch-Size-Aware 版本张量化卷积实现方法

```
for $cR_o = range(0, R_o)$ do
  for $cC_o = range(0, C_o)$ do
    for $cK_r = range(0, K_r)$ do
      for $cKc = range(0, K_c)$ do
        $cR_i = cR_o + cK_r; cC_i = cC_o + cK_c$
        DMA 读 $D_i \leftarrow N_i \times B$ 个通道的 $\mathbf{in}_{(cC_i, cR_i)}$
        DMA 读 $W \leftarrow N_i \times N_o$ 个通道的 $\mathbf{weight}_{(cK_c, cK_r)}$
        矩阵乘法原语: $D_o += W \times D_i$
      end for
    end for
    DMA 写 $D_o \rightarrow N_o \times B$ 通道数的 $\mathbf{out}_{(cC_o, cR_o)}$
  end for
end for
```

$$RBW_{MEM\rightarrow LDM}^{Batch-Size-Aware} = \frac{(N_iB + N_iN_o)DS}{2BN_iN_o/T} = \frac{(B + N_o)DS}{2BN_o/T} = \frac{(\frac{1}{N_o} + \frac{1}{B})DS}{2/T} \tag{5-5}$$

5.1.2.2　Image-Size-Aware 循环变换方案

算法4展示了另一种可以减小算法需求带宽的循环调度方案。在这种方案中，对 C_o 维度和 B 维度对应的循环进行分块。其中 b_B 和 b_{C_o} 是长度为 B 和 C_o 循环

的分块大小。cK_c 作为计数器的循环（第 5 行）内部包含 $2 \times N_i \times N_o \times b_B \times b_{C_o}$ 个浮点运算，需要读入 $N_i \times b_B \times b_{C_o} + N_i \times N_o$ 个浮点类型的内存数据。根据这两项指标，公式5-6推导了内存到 LDM 的最小需求带宽 $RBW_{MEM->LDM}$。同样，T 是假设数据在 LDM 中进行计算理想情况下可获得的最高性能，DS 是数据类型的大小。由公式5-6可知，这种循环变换方案的 $RBW_{MEM->LDM}$ 与 b_{C_o}，b_B 和 N_o 呈反比，并且连续访存长度与 $b_{C_o} \times b_B$，N_o 有关。因此，即使长度 B 循环提供分块大小有限，也可以通过增大图片行维度分块 b_{C_o} 大小来补充。因为这种循环变换方案会受图片长度影响，因此被称为 Image-Size-Aware 版本。

$$RBW_{MEM\to LDM}^{Image-Size-Aware} = \frac{(N_iN_o + N_ib_{C_o}b_B)DS}{2b_{C_o}b_BN_oN_i/T} = \frac{(\frac{1}{b_{C_o}b_B} + \frac{1}{N_o})DS}{2/T} \tag{5-6}$$

Algorithm 4 Image-Size-Aware 版本

输出: 输入、输出和卷积核四维张量 **in, weight, out**

```
 1: for $cB = range(0, b_B, B)$ do
 2:     for $cR_o = range(0, R_o)$ do
 3:         for $cC_o = range(0, b_{C_o}, C_o)$ do
 4:             for $cK_r = range(0, K_r)$ do
 5:                 for $cK_c = range(0, K_c)$ do
 6:                     $cR_i = cR_o + cK_r; cC_i = cC_o + cK_c$
 7:                     $D_i \leftarrow N_i \times b_B$ 通道的 $\mathbf{in}_{(cB:cB+b_B, cC_i:cC_i+b_{C_o}, cR_i)}$
 8:                     $W \leftarrow N_i \times N_o$ 通道的 $\mathbf{weight}_{(cK_c, cK_r)}$
 9:                     矩阵乘法原语：$D_o += D_i \times W$
10:                 end for
11:             end for
12:             $D_o \rightarrow b_B \times N_o$ 通道的 $\mathbf{out}_{(cB:cB+b_B, cC_o:cC_o+b_{C_o}, cR_o)}$
13:         end for
14:     end for
15: end for
```

根据公式5-6和公式5-5可知，如果 minibatch 比较大时，可以采用 Batch-Size-Aware 的版本，如果图片尺寸比较大时，可以使用 Image-Size-Aware 的版本进行循环变换。对于这两个版本，一个大的输出通道参数 N_o 都会极大减小 RBW，同时增加了 DMA 对 W 的实测带宽。

5.1.2.3　其它技巧

由于卷积运算的计算时间大于内存访问时间，根据章节3.3.2中的张量化优化方法，需要采用双缓冲区策略隐藏访存时间。当数据在一个 LDM 缓冲区中计算时，下一次迭代时使用的数据通过 DMA 加载到另一个 LDM 缓冲区中。

另外，如果 LDM 空间足够，可以将 DMA 操作提升到外部循环，以进一步减少 $RBW_{MEM->LDM}$。这种情况对于参数比较小的情况会有效果，此时访存时间可能会超过计算时间。对于算法3，可以将第7行的 DMA 操作提升到第3行的循环下。对于算法4，可以将第8行的 DMA 操作提升到第4行的循环下。这两种情况，需要读取大小为 $N_i \times N_o \times K_c$ 的数据 $\mathbf{weight}_{(cK_r,:)}$。

5.1.3　基于 Winograd 的卷积优化

根据最小滤波算法（Minial Filtering Algorithm）[167] 可知，对于任意维度的卷积运算的最少乘法运算次数等于输入数据的元素个数。以此为基础，Winograd 算法[75] 通过算法变换减少乘法次数来加速卷积运算速度。它的流程如算法 5所描述，$F(m \times m, r \times r)$ 表示计算输出大小为 $m \times m$ 的 Winograd 卷积操作。本小节以最常见的 $F(2 \times 2, 3 \times 3)$Winograd 算法优化为例介绍笔者的优化方法，其他尺寸可以以此类推。

图 5.2展示了 Winograd 卷积算法流程，主要分四个阶段，它们分别是输入特征图预处理阶段（算法 13），卷积核预处理阶段（算法 8），批量矩阵乘法阶段（算法第18行）和输出特征图后处理阶段（算法 23行）。

5.1.3.1　输入/输出图片预处理:

该阶段对特征图的每一个输入图片 4×4 大小的切片 $\mathbf{in}_{c,b}$ 完成 $B^T\mathbf{in}_{c,b}B$ 变换的计算，相邻区块长宽维度有长度为 2 的边缘重叠。由于区块的变换相互独立，因此 $R_i \times C_i$ 维度以切片为单位在从核间进行数据并行任务划分，让每个从核独立进行 m 个 4×4 图片块的变换。为了高效的张量化访存，按照章节3.3.1张量化访存优化方法，设计最佳的数据排布为：输入数据排布为 (B, R_i, C_i, N_i)，输出数据排布为 $(16, B, T, N_i)$。此时，可以使用跨步 DMA 完成变换，跨步 DMA 的分块大小为输入通道数 N_i，每个从核每次读入大小为 $(2m + 2, 2, N_i)$ 的 m 个输入图片的切片。输入预处理的直观理解可以参考图5.5。另外，可以复用输入数据缓存和输出数据的缓存空间，来节省 LDM 开销。

虽然 Winograd 输入预处理也包含计算操作，但使用章节 3.2.3的定性性能模型分析可知，DMA 时间仍是影响预处理阶段性能的唯一因素。这个过程需要向量化

Algorithm 5 Winograd 最小滤波算法运算卷积 $F(m \times m, r \times r)$

```
图像切片的数量 $P = BN_i\lceil R_i/m\rceil\lceil C_i/m\rceil$，$\alpha = m + r - 1$ 输入图片切片大小.
$\mathbf{in}_{c,b} \in \mathbb{R}^{\alpha\times\alpha}$ 输入张量 $c$ 通道上图片切片 $b$
$\mathbf{W}_{k,c} \in \mathbb{R}^{r\times r}$ 是输出通道 $k$ 输入通道 $c$ 的卷积核
$G, B^T$ 和 $A^T$ 分别是卷积核、输入和输出的变换矩阵，它们的配置由[167] 而来。
$\mathbf{out}_{k,b} \in \mathbb{R}^{m\times m}$ 输出通道 $k$ 上的图片切片 $b$
for $k = 0$ to $N_o$ do
    for $c = 0$ to $N_i$ do
        卷积核预处理：$u = G\mathbf{W}_{k,c}G^T \in \mathbb{R}^{\alpha\times\alpha}$，图片块 $u$ 拼接成矩阵 U: $U_{k,c}^{(\xi,\nu)} = u_{\xi,\nu}$
    end for
end for
for $b = 0$ to $P$ do
    for $c = 0$ to $N_i$ do
        输入图片预处理：$v = B^T\mathbf{in}_{c,b}B \in \mathbb{R}^{\alpha\times\alpha}$，图片块 $v$ 拼接成矩阵 V: $V_{c,b}^{(\xi,\nu)} = v_{\xi,\nu}$
    end for
end for
for $\xi = 0$ to $\alpha$ do
    for $\nu = 0$ to $\alpha$ do
        $M^{(\xi,\nu)} = U^{(\xi,\nu)}V^{(\xi,\nu)}$
    end for
end for
for $k = 0$ to $N_o$ do
    for $b = 0$ to $P$ do
        输出图片预处理：从矩阵 M 中收集出图片块 $m$: $m_{\xi,\nu} = M_{k,b}^{(\xi,\nu)}$，$\mathbf{out}_{k,b} = A^T m A$
    end for
end for
```

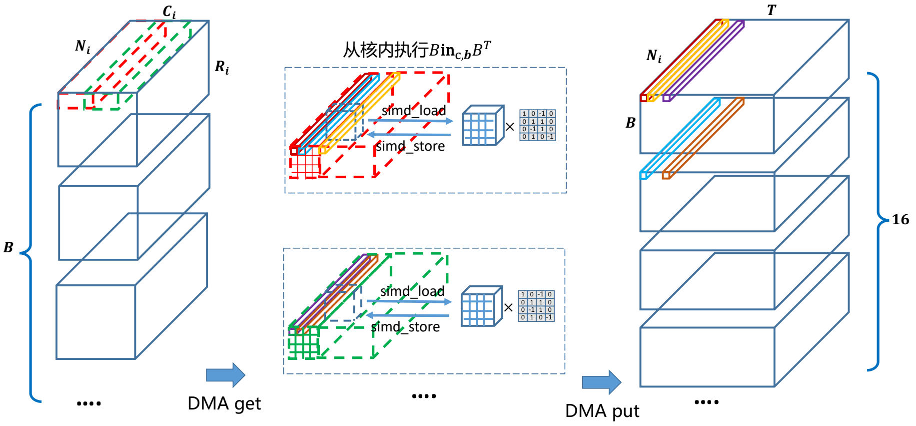


图 5.5　Winograd 输入预处理的张量化优化

读指令来将 (4×4) 大小图片块读入寄存器，然后使用向量化的四则运算进行变换操作，最后再使用向量化指令写数据到 LDM。完成这些操作需要 $P1$ 执行部件发射 16 个 VLDS 和 16 个 VSTS 指令进行读写，需要 $P0$ 执行部件发射 48 次加减运算指令以进行变换。由此可知，$P0\_valid = 48 > P1\_valid = 32$。由于变换后输入图片的排布的变化，处理后的数据需要以大小为 $m \times N_i$ 的数据块跨步写入内存。公式5-7展示了 DMA 访存时间和计算时间比值，通过计算可以得知 T_{DMA} 远远超过

$T_{compute}$。读入数据需要跨步 DMA 操作来完成，为了获得较高的 $MBW_{MEM->DMA}$，要尽可能保证每次 DMA 访问足够多的数据，且跨步读取的数据块大小足够大。鉴于实际应用中 N_i 在 3–512 相对较小的区间变化，可以不对 N_i 切分以保证它可以被连续访存。在满足 LDM 使用限制的条件下，尽可能增大 $m \times N_i$ 的大小，以使 DMA 操作达到良好带宽。

$$
\begin{aligned}
T_{DMA} &= \frac{64 \times N_i((2m+2) \times 4 + 16m)}{MBW_{MEM->LDM}} \\
T_{compute} &= \frac{mN_i/4 \times max(\#P0\_valid + \#P0\_idle, \#P1\_valid + \#P1_idle)}{1.45GHz} \\
\frac{T_{DMA}}{T_{compute}} &= \frac{185.6}{MBW_{MEM->LDM}} >> 1
\end{aligned}
\tag{5-7}
$$

由于输出后处理过程和输入类似，笔者在此不赘述。

5.1.3.2　卷积核变换阶段:

该阶段对每个 3×3 卷积核 $\mathbf{W}_{k,c}$ 完成 $G\mathbf{W}_{k,c}G^T$ 变换。这里以 3×3 大小的变换操作为并行任务单位，对输入张量的 $N_i \times N_o$ 维度进行划分来进行数据并行。每个从核读入若干通道的卷积核进行变换，LDM 和寄存器间数据传递同样可以使用向量化的指令完成。对于卷积核的变换如图5.6所示。为了增大连续读取数据量，卷积核预处理前的数据排布为 $(3, 3, N_o, N_i)$，预处理后的数据排布为 $(16, N_o, N_i)$。

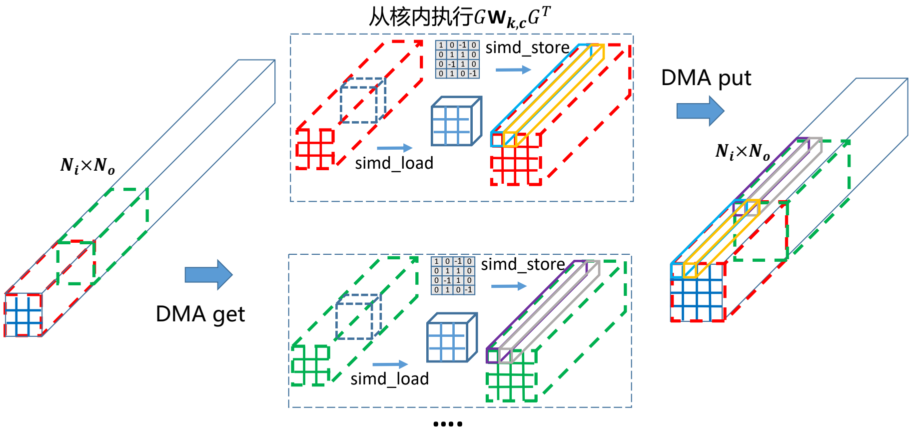


图 5.6　张量化优化的卷积核变换阶段

5.1.3.3　批量 GEMM 运算

当输入特征图和卷积核处理完毕后，需要 16 次 GEMM 操作代替 Winograd 的点乘操作，每次 GEMM 运算完成 $(B \times T, N_i)$ 大小的输入矩阵和 (N_o, Ni) 大小的权

重矩阵的乘法。N_o, N_i 相对较小，而 T 一般相对较大，此时 GEMM 运算按照章节4.2.1的分类，属于 GEMP 类型。而且，对于绝大多数卷积层，(N_o, N_i) 大小的矩阵完全可以分布在从核阵列的 LDM 中。章节5.4.1的实验结果表明，使用本研究的矩阵乘法库 swGEMM 相对系统提供的 xMath 有显著的性能提升。和显式矩阵乘法卷积类似，同样可以将补零操作和预处理阶段融合，通过 swGEMM 中的性能模型搜索到的最佳分块为 B_T，在输入特征图变换后拷贝到 $(\frac{\lceil B\times T\rceil}{B_T} \times B_T, N_i)$ 大小的空间，从而避免了补零操作。

5.2　全连接和 LSTM 算子

全连接和 LSTM 算子都是以 GEMM 操作为主，因此笔者在此一并介绍。

全连接层的输入输出特征图都是二维矩阵的形式。当输入特征图是卷积层输出的四维张量时，需要将表示图片通道、长度、宽度的三个维度 (N, R, C) 摊平成一维数据。全连接层正向传播的计算为 GEMM 运算：$x^{l+1} = W \times x^l$。同样可以使用链式法则计算本层的参数的梯度，反向传播的敏感度计算方式为：$\delta^l = W^T \times \delta^{l+1}$，梯度计算的方式为：$\frac{\partial L}{\partial W} = \delta^{l+1} \times (x^l)^T$。无论正向反向，全连接算子都可调用 swGEMM 实现。

LSTM 算子是一个包含矩阵乘法和激活函数的集合，其结构如图5.7所示。公式5-8展示它的正向传播计算过程，其中 $\odot$ 代表点乘（Element-wise Product），× 仍代表矩阵乘法或称内积，σ 是 sigmoid 函数。

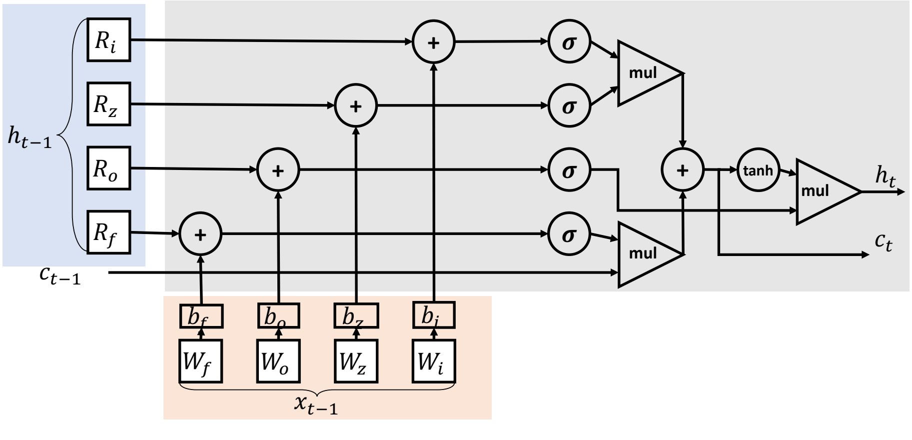


图 5.7　LSTM 算子的内部结构

对于 LSTM 算子，使用张量化编程模型进行并行算法设计可以带来更大的优化空间。cuDNN 采用 appleyard 等人的方法[80]，它以 cuBLAS 的基本例程为基础，在其上进行了矩阵乘法、点乘融合优化。这种方法的优化粒度并不精细，比如对

相邻时间步 R 矩阵的内存访问有时是可以避免的，但是由于 BLAS 只能操纵内存中的矩阵，并无法对 Cache 层次数据的生存时间进行更精细的控制。在张量化编程模型中，利用张量化访存和计算原语可以更精细控制 LDM 数据的生存时间，从而细粒度优化 LSTM 算子。算法6展示了使用张量化编程模型优化 LSTM 算子的方法。

矩阵乘法融合：首先，x_t、h_{t-1} 分别是公式5-8中的四个单独 GEMM 运算的共同输入。四个矩阵可以被组合成一个更大的矩阵参与 GEMM 运算，此时，仅需要 2 个 GEMM 运算而不是 8 个，而每个输入矩阵的大小扩大 4 倍。增大输入数据矩阵的维度意味着可以更充分地重叠通信和计算，从而摊薄 DMA 初始化开销并减少函数启动次数，对于 GEMM 运算有很大的提升。

其次，尽管 RNN 多个时间片的 LSTM 算子之间对 h_t 和 c_t 张量存在互相依赖关系。但是，所有输入 x_t 都在 RNN 计算开始时可以使用，因此，可以立即启动处理这些输入的矩阵运算。如算法6第2行所示，可以在 T 个时间步迭代前完成一次对内存中大矩阵的 GEMM 运算。同样，这个矩阵乘法属于 GEPB 形式，使用 swGEMM 相对 BLAS 会有显著性能提升。

LDM 数据复用：LSTM 网络结构的多个时间步计算共享算子的参数，这给 LDM 内张量数据的复用带来可能，可以减少 DMA 访存次数来减少算法的需求带宽。使用笔者上一章的 LDM 矩阵乘法原语接口，可以完成细粒度的缓存优化。由于 h_t 在下一个时间步计算会变成 h_{t-1}，对于某些参数尺寸并非十分巨大的情况，可以将它固定在从核的 LDM 中减少 DMA 带宽开销。将 $[i_t, f_t, o_t, c'_t]$，h_t 和 c_t 缓存在 LDM 中，点乘操作也可以直接使用 LDM 数据开始运算。

$$
\begin{aligned}
&i_t = \sigma(W_i \times x_t + R_i \times h_{t-1} + b_i) \quad f_t = \sigma(W_f \times x_t + R_f \times h_{t-1} + b_f) \\
&o_t = \sigma(W_o \times x_t + R_o \times out_{t-1} + b_o) \quad c'_t = \sigma(W_c \times x_t + R_c \times h_{t-1} + b_c) \\
&c_t = f_t \odot c_{t-1} + i_t \odot c'_t \quad h_t = tanh(o_t) \odot o_t
\end{aligned}
\tag{5-8}
$$

5.3　其它算子

除了复杂的卷积、全连接和 LSTM 算子，其它访存密集型算子，如池化、批量归一化等的优化方法均可采用数据并行方式，让每个从核处理若干图片元素为输入的独立任务，并不需要利用申威架构的寄存器通信特性。由于这些算子计算时间都远小于访存时间，DMA 可用带宽是主要瓶颈。可以利用章节3.3.1张量化访存优化指导，尽量增大单从核访存量，在必要时采用批处理的方式来增加连续访

Algorithm 6 LSTM 算子张量化优化方法

```
输入: 总时间片 $T$，输入数据 $[x_0, x_1, ..., x_T]$
$[R_i,R_z,R_o,R_f]$ 组成矩阵的数组 $R$，$[W_f,W_o,W_z,W_i]$ 融合成矩阵 $W$，$[b_f,b_o,b_z,b_i]$ 融合成为向量 $b$
内存调用 swGEMM 操作：$[I_0, I_1, ..., I_T] = W \times [x_0, x_1, ..., x_T] + b$
LDM 内存缓存 $h = h_0$
for $t = 1$ to $T$ do
    DMA 读取 $I_{t-1}$
    for $j = 0$ to $3$ do
        DMA 读取 $R[j]$
        LDM 内调用 GEMM 原语运算: $I_{t-1}[j] \mathrel{+}= \sigma(R[j] \times h)$
    end for
    $[i_t, f_t, o_t, c_t'] = I_{t-1}$
    LDM 内点乘：$c_t = f_t \odot c_{t-1} + i_t \odot c_t'$，$h = tanh(o_t) \odot o_t$
end for
```

存数据量，以增大 DMA 带宽利用率。

5.4　实验结果

5.4.1　卷积算子

卷积算子性能测试分为两部分。首先在目前主流的经典 CNN 模型上验证本章提出的三种张量化卷积算法，基于显式矩阵乘法的卷积（Explicit-GEMM-CONV）、基于隐式矩阵乘法的卷积（Implicit-GEMM-CONV）和基于 Winograd 的卷积（Winograd-CONV），在申威 26010 上实现后的性能表现。然后再遍历常用的卷积参数空间，测试三种实现的通用性，并汇总了卷积算子的整体性能。

5.4.1.1　经典用例测试

本小节以使用 ImageNet 数据集训练四种经典 CNN 时需要的卷积算子作为测试用例，评估本章卷积算子设计性能。这些 CNN 包括 AlexNet[7]，VGG16[168]，ResNet[54] 和 Yolo[169]。测试了 minibatch 大小（本节图中都使用符号 B 来表示）为 1、32、128 的三种情况。其中 B=32、128 应用于深度学习的训练过程，B=1 应用在深度学习的推理过程。对于每种 B 的取值，四种 CNN 总共包括 48 组不同参数的卷积层。本节展示了芯片四个核组的整体性能，本文使用多线程技术对 R_o 维度

切分以在四个核组上进行数据并行。

Explicit-GEMM-CONV：图 5.8展示了 Explicit-GEMM-CONV 的性能，横轴按照它们在网络模型中的相对顺序进行排布，笔者标出了每种网络模型第一层的起始位置。不同 B 参数条件下的 48 组测试都可以使用 Explicit-GEMM-CONV 方法实现。可以观察到，对于不同 B 大小，卷积算子性能基本保持稳定，并没有出现 B=1 效果明显变差的情况，这说明章节3.3.1中的张量访存优化技巧 2 应用于 *im*2*col* 中取得了不错的效果。

为了考察笔者前一章设计的 swGEMM 函数库在卷积算子中的应用效果，图5.9上半部分展示了以 swGEMM 和 xMath 两种方式实现 Explicit-GEMM-CONV 的性能对比。在 B=1 的 45 组测试、B=32 的 34 组测试和 B=128 的 37 组测试中，swGEMM 都相对 xMath 取得了加速，有些情况甚至获得超过一个数量级的性能提升。

图 5.9下图还展示了 *im*2*col* 在算子运算整体时间中的占比。在那些 Explicit-GEMM-CONV 表现整体不佳的测试用例中，绝大多数是因为 *im*2*col* 带来额外开销而造成的。比如，ResNet50 中有许多性能低于 800 GFLOPS，*im*2*col* 时间占比超过 40% 的情况。如以 $(R_i = 28, N_i = 512, N_o = 128)$ 为参数的算例，它在 $B = 128$ 时性能仅能达到 731.2 GFLOPS，*im*2*col* 时间占比高达 44%。

Implicit-GEMM-CONV：Implicit-GEMM-CONV 应用于 48 组参数中的 40 组，其它情况卷积层的通道数小于 64，无法使用矩阵乘法原语，因而 Implicit-GEMM-CONV 无法使用。本章设计的两种算法 Image-Size-Aware 版本和 Batch-Size-Aware 版本分别覆盖了 B 为 32 和 128 的情况，下一章节将通过自动化的方式生成 $B = 1$ 的代码。图 5.10展示了 Implicit-GEMM-CONV 在 B 为 32 和 128 时的性能。对于输入输出通道数较大 (>128) 的情况下，Implicit-GEMM-CONV 的性能在 1500~2100 GFLOPS 之间，只有 ResNet50 和 Yolo 前几层性能相对较差，这是由于这些层的 N_i, N_o 较小，比如 ResNet50 的前四个卷积层的参数 N_i 或者 N_o 小于等于 64，这种情况下可以利用的带宽 MBW 很小但是算法需求带宽 RBW 很大，根据章节3.2.3提出的性能模型可知，此时算子会严重受限于访存操作，导致无法充分利用计算部件。对于 Implicit-GEMM-CONV，输入输出的通道数越大，性能表现越好。

Winograd-CONV：四种经典 CNN 中有 14 个卷积层可以使用 $F(2\times2, 3\times3)$ 方式的 Winogard 卷积算子实现。图 5.11左半部分展示了这些卷积层上使用 Winograd-CONV 算子的性能。笔者采用 Winograd 原始论文[75] 使用的方法统计性能结果①，它的浮点运算次数采用直接卷积方式统计，而非真实浮点计算次数。除了第 2、3、4

---

① 这种性能统计方法也是学术界通用的方式，它方便和其他方式卷积实现性能在同一参照系下对比

算例，B=32、128 情况下算例的性能都达到或超过 2400 GFLOPS。这三层的 N_i 参数分别是 64、128、128，如之前所述，如果 N_i 参数比较小时，swGEMM 的性能无法充分发挥。即使这样第 3、4 算例在 B=32、128 情况下性能仍然超过了 1600 GFLOPS。图 5.11右图展示了使用 swGEMM 相对于 xMath 中 GEMM 例程对 Winograd-CONV 带来的整体加速效果。可见，对于 N_i，N_o 相对较小的算例，swGEMM 的加速效果是最突出的。

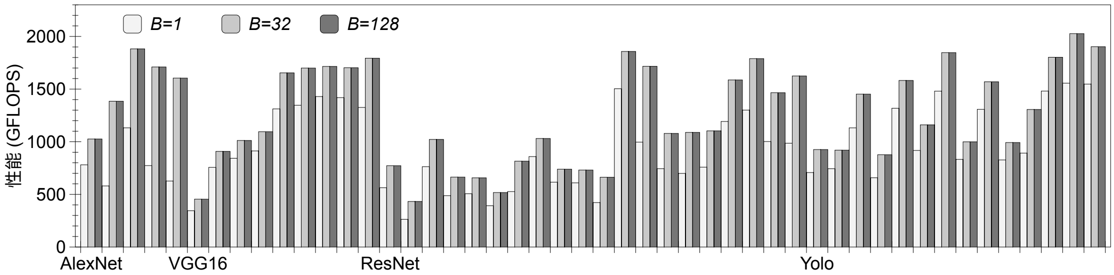


图 5.8　四种经典网络模型的卷积层使用 Explicit-GEMM-CONV 算法的实现性能

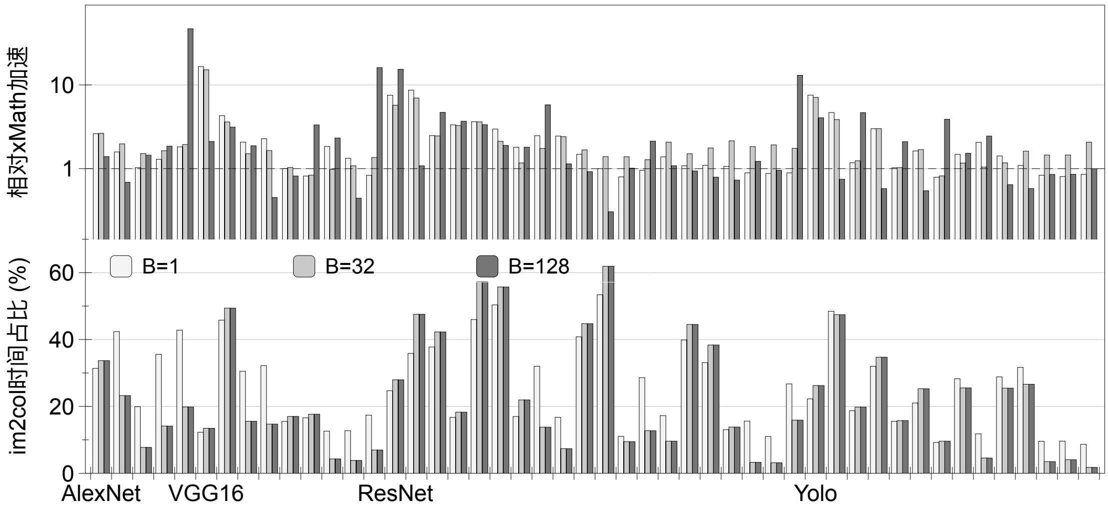


图 5.9　四种经典网络模型的卷积层使用 Explicit-GEMM-CONV 算法中 GEMM 运算和 im2col 运算效果，上图：使用 swGEMM 方法相对于 xMath 的加速比，下图：整体时间中 *im2col* 操作的时间占比。

5.4.1.2　通用性测试

为了更全面评估 swDNN 卷积算子的效率，本节使用清单 5.1生成不同输入通道数、图片尺寸和 minibatch 情况下的测试用例，在 $B = 1, 32, 128$ 时，各有 225 组不同的测试用例。

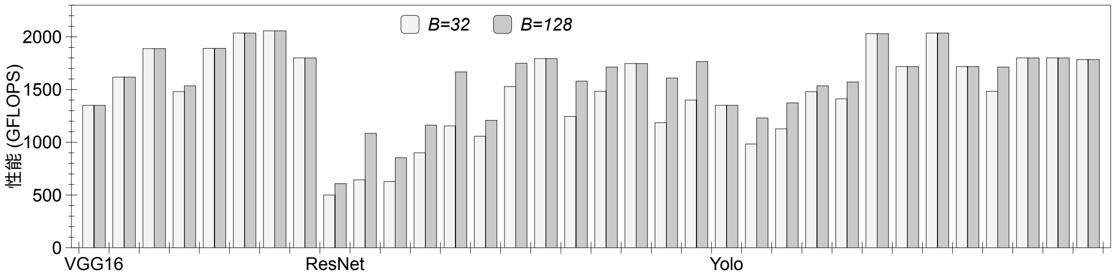


图 5.10　三种经典网络模型的卷积层使用 Implicit-GEMM-CONV 算法的实现性能

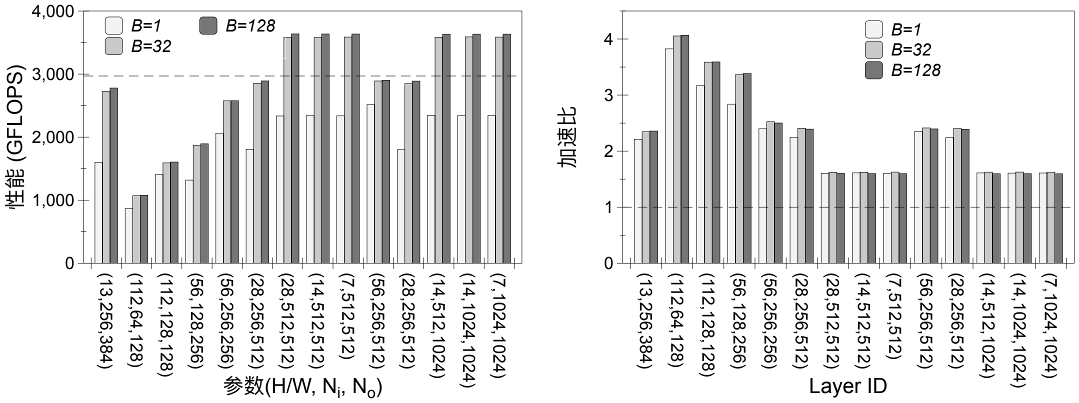
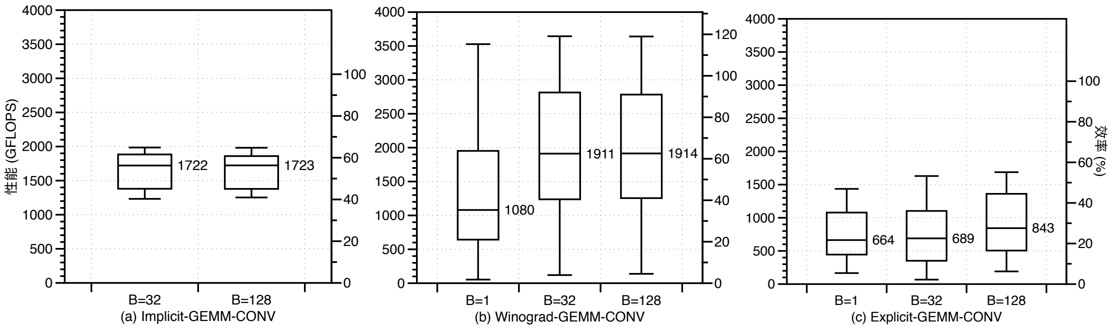


图 5.11　左图：四种经典网络模型的卷积层使用 Winograd-CONV 算法的实现性能，右图：在四种经典网络模型中，在 Winograd-CONV 中使用 swGEMM 的加速效果。

图 5.12使用箱线图（Box Plot）的形式对总体性能测试结果进行展示。图中分别展示了最小值、第一四分位数、中位数、第三四分位数与最大值信息。图的左边以吞吐量（GFLOPS）为单位表示性能，图的右边以百分比（%）为单位表示效率。

通过观察可以看出，Implicit-GEMM-CONV 算法的性能最为稳定，它的平均性能在峰值性能的 57% 左右。Winograd-CONV 方法的平均性能表现最好，在 B=32、128 时运算效率略超过峰值的 60%。但是，Winograd-CONV 的箱线图的第一四分位数和第三四分位数的差距较大，这说明它的表现并不稳定，有些参数效果非常好，有些则非常差。这是因为 Winograd 的输入输出预处理操作是访存受限的，对于 N_i, N_o 比较小的情况，Winograd 的 GEMM 计算部分占比减小，而访存部分占比增加，整体性能会变差。但是，对于 N_i, N_o 比较大的算例，则非常适合使用 Winogard-CONV 进行卷积计算，可以观察到，箱线图中最高点的运算效率接近峰值性能的 120%。Explicit-GEMM-CONV 算法性能相对较差，平均运算效率不到 30%，但是，它是通用性最好的方法，可以适应所有卷积的变种。

笔者比较了上一章实现的 swGEMM 函数库在通用测试中的加速效果。如表 5.2所示，在以 GEMM 运算为核心的 Explicit-GEMM-CONV 和 Winogard-CONV



(a) Implicit-GEMM-CONV　(b) Winograd-GEMM-CONV　(c) Explicit-GEMM-CONV

图 5.12　通用性测试中，三种张量化卷积算法在整个申威 26010 芯片上的表现。

实现中，swGEMM 在大部分情况下相对 xMath 都获得了加速效果。对于 Explicit-GEMM-CONV 中绝大多数情况，swGEMM 带来了平均 21%~26% 的整体加速效果；对于 Winograd 优化，swGEMM 带来了 295%~316% 的平均加速效果；这再次印证了笔者设计的 swGEMM 函数库成功解决了 xMath 对 GEMP 类型矩阵乘法（狭长形状矩阵）效果不佳的问题。

```
for cB in 1 32 128;
 for cNi in 64 128 256 384 512;
  for cNo in 64 128 256 384 512;
   for cRo in 32 64 128 256;
    if[$cNi >= $cNo] ./test_swDNN conv B=$cB Ni=$cNi No=$cNo Ro/Co=$cRo K=3
```

Listing 5.1　卷积算子通用测试的参数生成脚本

表 5.2　通用测试的 225 组参数中，swGEMM 相对 xMath 的加速情况。

Minibatch	**1**		**32**		**128**	
	加速	减速	加速	减速	加速	减速
Explicit	54(+21%)	21(-17%)	59(+23%)	16(-%22)	55(+26%)	20(-22%)
Winograd	75(+316%)	0	75(+295%)	0	75(+306%)	0

总的来说，本研究的张量化卷积算法非常健壮，可以覆盖测试中全部参数使用情况，对于大通道卷积性能表现尤为可观。另外，在卷积算子实现中应用笔者前一章设计的张量化原语和 swGEMM 函数库取得了良好效果。在实际使用中可以采用动态调整策略，在构建深度学习网络前通过遍历三种算法寻找最优的卷积算子。图5.13展示了 swDNN 卷积算子的整体效果，每种参数的结果是使用三种张量化优化中的最优方案测试得到。对于 $B = 32, 128$ 的情况，平均的运算效率达到峰值性能的 61% 左右，对于 $B = 1$ 的情况，平均运算效率达到峰值性能的 57%。另外，swAutoDNN 优化后算子库非常健壮。图5.13中第一四分位数的位置和最小值非常接近，都接近 50% 的运行效率，这说明 swDNN 中并没有对某种参数性能支

持非常差的情况。而第三四分位数在 80% 之上，这说明对于测试中很多算例，算子库可以提供非常不错的性能。

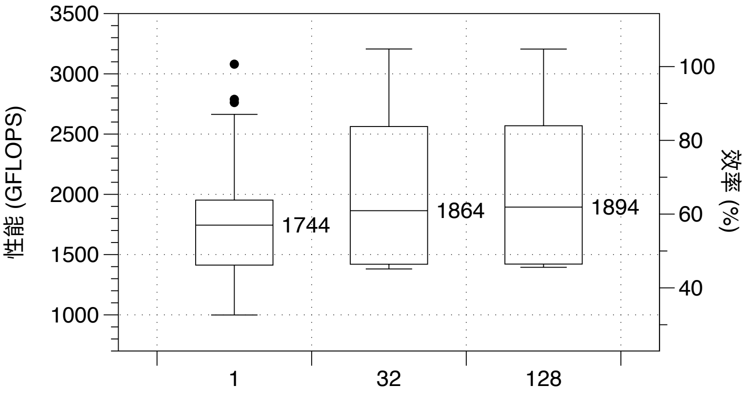


图 5.13　swDNN 中卷积算子整体性能箱线图

5.4.2　LSTM 算子

本小节测试 LSTM 算子张量化优化的效果，测试用例使用清单5.2生成的 72 组参数。笔者按照 cuDNN 中使用的 appleyard 等人的方法[80] 在申威上实现了一个基础版本作为对比。

图5.14展示了申威单核组上本章提出的张量化优化相对基础版本的整体加速效果，坐标轴按照脚本生成顺序排列。在所有测试用例上，张量化优化均取得了加速。本章张量化优化来带的加速效果非常明显，平均在 2.6x 左右。对于隐藏层数目 H 大小比较小的情况，最高可以达到 6.2x。另外，对于 H 非 64 倍数情况，加速效果较 64 的倍数更加明显，这是由于基础版本点乘操作 DMA 数据划分不均匀引起的，这种现象在张量化优化中被完全消除。

LSTM 算子中的 GEMM 运算是计算密集的，而点乘运算则是访存密集的，本节将它们拆解分别观察优化效果。图5.15展示了算法6的第 6-9 行的 LDM 数据复用对矩阵乘法优化的效果。其中“基础版本”表示调用 swGEMM 实现的基础版本，而“张量化优化”表示使用张量化矩阵乘法原语实现精细缓存控制的优化版本。对于隐藏层数目 H 比较小的小矩阵乘法情况，“张量化优化”效果明显优于 BLAS 接口调用。而对于 H 比较大的情况，由于计算访存比相应增加，swGEMM 的计算可以和访存重叠，因而“张化优化”加速效果不显著。这是因为算法6第 7 行的 DMA 操作由于受内存限制，无法使用双缓冲技术使它和 GEMM 运算重叠。另外单双精度转化操作也有额外开销，对于小尺寸矩阵 GEMM 运算这部分开销相对明显，此时“张量化优化”相对直接调用 swGEMM 加速效果最为明显。

```
for cT in 64 128 256 512 1024 2048;
 for cB in 64 96 128;
  for cH in 64 128 256 384;
    ./test_lstm B=$cB N=$cN H=$cH
```

Listing 5.2　LSTM 算子测试参数生成脚本

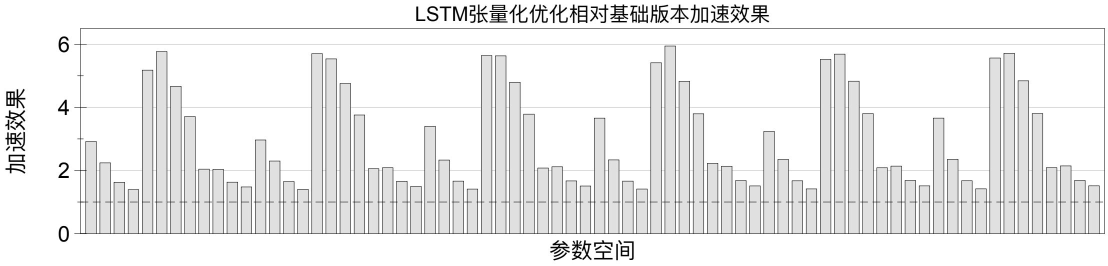


图 5.14　使用张量化优化对 LSTM 的整体加速效果

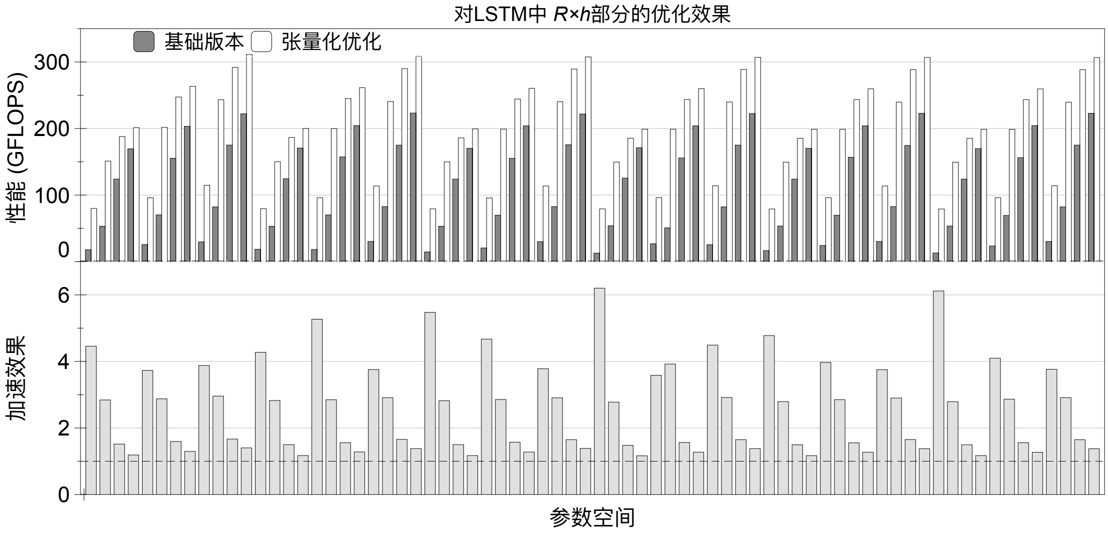


图 5.15　使用张量化原语对 LSTM 中 $R \times h$ 部分矩阵乘法加速效果

图5.16展示了算法6中第 11 行点乘操作计算时间。点乘部分涉及 sigmoid 函数，由于从核运算 exp 指令非常慢，swDNN 采用泰勒展开方式实现了定制的 exp 函数，并保证不影响数值精度。可以观察到点乘部分占比随着 H 增大而减少，这是算法本身所表现的特性决定的，由于点乘操作随 H 线性增长，而 GEMM 运算随 H 二次方增长，所以 H 越大点乘时间占比越低。

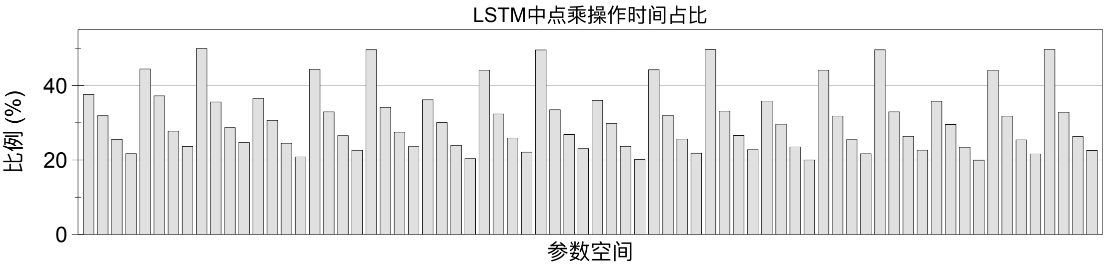


图 5.16　点乘操作在张量化优化的 LSTM 算子中占比

5.5　本章小结

深度学习的主要计算由不同的算子操作组成，本章使用第3章提出的张量化编程模型，对这些主要算子在申威架构上的优化障碍进行各个击破。

对于深度学习中最复杂的卷积算子，本章提出了三种张量化设计方法，分别是基于显式矩阵乘法的优化、基于隐式矩阵乘法的优化和基于 Winograd 的优化。大量测试表示，卷积算子的平均性能可以达到芯片峰值计算速度的 60%，而且它们在大范围的参数变化情况下表现非常稳定。对于 LSTM 算子，使用张量化原语来精细控制缓存使用，本章的实现相对目前主流优化方法带来了平均 2.6x 左右加速效果。在两种算子优化过程中，本章使用 swGEMM 代替 xMath 取得了显著加速效果，再次证明了 swGEMM 的实用效果。对于基于显式矩阵乘法卷积中绝大多数情况，swGEMM 带来了平均 21%~26% 的整体加速效果，对于 Winograd 优化，swGEMM 带来了 295%~316% 的平均加速效果。

本章的工作充分验证了第3章提出的张量化编程模型及其优化方法的实用性。对于未来的具有全新功能的算子，可以遵循张量化编程模型的优化原则进行高效的并行算法设计来对 swDNN 进行扩展。本章是第6的基础，届时笔者将针对本章的张量化优化方法设计自动调优工具。

第 6 章　swAutoDNN: 深度学习算子张量化自动调优

前一章介绍了如何使用张量化编程模型在申威架构上设计深度学习算子。然而，手工进行张量化优化需要专家知识和巨大的人力成本。本章将介绍一个端到端的自动化工具，来为深度学习算子进行自动张量化优化。开发人员使用时只需定义深度学习运算符的计算描述和若干调度策略，自动化工具就可自动生成近乎最优的可执行代码。自动化工具还可以自动处理张量化编程模型使用时遇到的问题，如使用张量原语引起的边界问题，以及隐藏内存访问延迟等。本章节的主要工作如下。

- 本章发现张量化编程模型天然适合进行自动优化，它已经将硬件相关的优化技巧封装在张量化原语中，自动优化只需专注于寻找最优的算法流程。
- 本章提出了一套名为 swAutoDNN 的自动优化工具，它由调度器、IR 优化器、自动调优器和代码生成器组合而成，用户只需定义深度学习运算符 DSL 形式的计算描述和若干调度策略，swAutoDNN 可以自动生成调优后的可执行代码。
- 本章将 swAutoDNN 用来调优 swDNN 中最复杂的卷积算子，将运算效率从 60% 提升到了 70% 以上，并且有效覆盖了手动优化代码不能处理的参数情况。
- 本章利用章节3.2.2设计的性能模型来比较不同候选代码的优劣。与黑盒自动调优相比，使用性能模型能够减少超过两个数量级的时间成本，将自动调优时间从几天缩短到几分钟。即使在最差的情况下，它也只会带来不到 8% 的性能损失。

6.1　张量化自动优化动机

根据张量化编程模型指导，可以使用张量化原语为最基本操作实现深度学习算子，但是针对它们的手动优化却是十分困难的。首先，输入输出张量每个维度的参数存在很大选择范围，又因为同一个算法在不同输入输出参数下最优的循环书写形式应该是不同的，理论上需要多套不同的实现来适配不同参数情况。其次，新硬件平台上编程人员缺少相应的专家知识，编写调优这些算子最优实现的任务是繁杂且困难的。开发一个最优的算子库需要巨大的人力成本和漫长的开发周期，即使 swDNN 算子库对于复杂算法手动设计张量化的实现也仅覆盖了有限的参数空间。以章节5.1.2中基于隐式矩阵乘法的卷积算法为例，swDNN 中虽然手动设计

了 Batch-Size-Aware 和 Image-Size-Aware 两种张量化优化实现，但是这两个版本实现对于小 minibatch 或小尺寸的图片仍无法胜任。

嵌套循环形式的计算核心广泛存在于深度学习算子中，但是借助基础编译器很难将一个七层循环逆向工程成为一个张量化卷积操作。为解决以上问题，需要构建一个更高层次的自动优化工具，来跨越算子的算法描述和最佳张量化实现之间存在的鸿沟，它能够使开发人员不需了解申威架构的优化细节，只需输入一个计算逻辑描述并加入若干优化指导，就可以自动完成代码优化和代码自动生成工作。最近，有许多工作致力于面向不同硬件对深度学习算子进行自动调优，例如 TVM[87]、Glow[170] 和 Tensor Comprehensions（TC）[85] 等。但是，针对如“申威 26010”这种具有特殊创新特性的硬件架构，这些方法仍然有一些局限。首先，这些工具无法自动利用精细的架构功能，比如核间寄存器通信机制。而且，即使在 GPU 上，在大多数情况下它们仅被视为手工实现的补充而不是替代品。其次，现有工具严重依赖于一些高级编译优化工具，比如 LLVM[171]，而它们尚不在“申威 26010”的相关系统工具支持范围内。第三，它们的自动调优方法过分强调对不同硬件的普适性，因此忽略了有助于缩小搜索空间的性能模型的应用。

这就需要针对申威架构特点，提出一套更实用的自动调优和代码生成方法。上文提到的 TVM、Glow，TC 等工具的设计思想都脱胎于 Halide[83]——一个为图像处理算子设计的自动调优工具。它们都遵循硬件无关优化与硬件有关优化分离的思想，而张量化算法设计思路和这种思想天然契合。硬件相关的优化细节已经封装在手工精细设计的张量化原语中，硬件无关优化的目的在于寻找张量化原语的最佳组织方式。这个过程等同于在初始算法描述的等价变换空间中寻找最优解的过程，可以借助本研究之前对硬件建模的知识来加速寻优过程。本文接下来就采用这种思想来设计自动优化工具，以解决申威架构自动调优的问题。

6.2 swAutoDNN 设计方法

6.2.1 概观

图6.1展示了 swAutoDNN 自动优化框架的设计概览，它由功能模块**调度器**、**IR 优化器**、**自动调优器**和**代码生成器**组合而成。通过读入以 DSL（Domain Specific Language）方式定义的计算描述和调优指导，swAutoDNN 完成自动建立搜索空间、寻找最优方案、应用张量化原语生成可执行代码。整个过程中各个功能模块协同配合：调度器生成搜索空间，它是所有等价的张量化算法实现方案组成的集合，每种实现方案被翻译成中间表示（Intermediate representation，IR）的形式；IR 优化器在 IR 中加入可以提升生成代码性能的必要优化；自动调优器使用性能模型以极

低的成本在搜索空间中定位最优的实现方案；代码生成器使用之前章节优化过的张量化原语生成可执行的代码。本节其他部分将详细介绍各部分的设计细节。

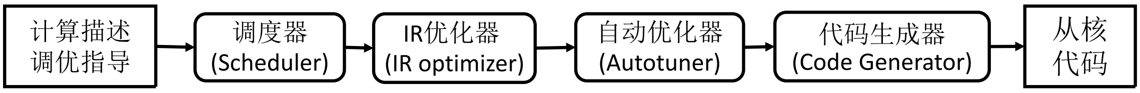


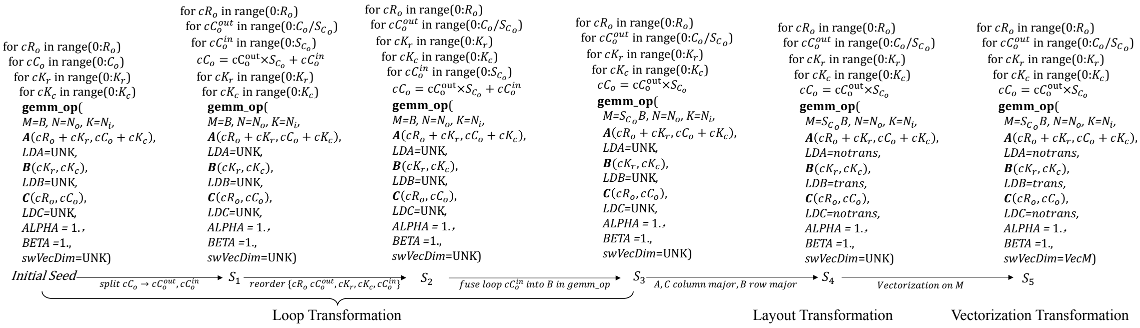
图 6.1　swAutoDNN 中各功能模块

6.2.2　计算描述 DSL

本研究设计了一个 DSL 来定义计算描述和变换组合，用户通过这种 DSL 来定义**调度种子**和基于它的一系列等价变换方式作为调优指导。调度种子是只定义循环结构和计算位置的模糊张量化实现，等价变换方式在下一小节介绍调度器时会详细介绍。为了降低开发门槛，用户不需关心具体的 DMA 操作发起时间、访存位置、数据布局和向量化方式等细节，这些具体信息 swAutoDNN 后期会自动推理补全。图 6.3展示了一个 DSL 描述计算的例子，并在下方展示了对应图 6.2中描述的一种变换方式。在 DSL 定义中，变量和常量被定义为 `Var/ConstVCar`，张量的定义包括它的维度和名字。比如，图中张量 `B` 被定义为一个四维张量，但是它的数据排布是不确定的，既可以是 (K_r,K_c,N_i,N_o) 也可以是 (K_c,K_r,N_o,N_i)。

6.2.3　调度器

调度器寻找等价于调度种子的所有可行的张量化实现的候选方案。一个包含数据排布、向量化方式并在逻辑上等同于调度种子的候选方案称为一种**调度策略**。通过对**循环变换**、**数据排布变换**和**向量化变换**进行相互组合，可以从调度种子构建出整个调度搜索空间。



```
for $cR_o$ in range(0:$R_o$)
 for $cC_o$ in range(0:$C_o$)
  for $cK_r$ in range(0:$K_r$)
   for $cK_c$ in range(0:$K_c$)
    gemm_op(
    M=B, N=$N_o$, K=$N_i$,
    A($cR_o + cK_r, cC_o + cK_c$),
    LDA=UNK,
    B($cK_r, cK_c$),
    LDB=UNK,
    C($cR_o, cC_o$),
    LDC=UNK,
    ALPHA = 1.,
    BETA =1.,
    swVecDim=UNK)
```

```
for $cR_o$ in range(0:$R_o$)
 for $cC_o^{out}$ in range(0:$C_o/S_{C_o}$)
  for $cC_o^{in}$ in range(0:$S_{C_o}$)
   $cC_o = cC_o^{out} \times S_{C_o} + cC_o^{in}$
   for $cK_r$ in range(0:$K_r$)
    for $cK_c$ in range(0:$K_c$)
     gemm_op(
     M=B, N=$N_o$, K=$N_i$,
     A($cR_o + cK_r, cC_o + cK_c$),
     LDA=UNK,
     B($cK_r, cK_c$),
     LDB=UNK,
     C($cR_o, cC_o$),
     LDC=UNK,
     ALPHA = 1.,
     BETA =1.,
     swVecDim=UNK)
```

```
for $cR_o$ in range(0:$R_o$)
 for $cC_o^{out}$ in range(0:$C_o/S_{C_o}$)
  for $cK_r$ in range(0:$K_r$)
   for $cK_c$ in range(0:$K_c$)
    for $cC_o^{in}$ in range(0:$S_{C_o}$)
     $cC_o = cC_o^{out} \times S_{C_o} + cC_o^{in}$
     gemm_op(
     M=B, N=$N_o$, K=$N_i$,
     A($cR_o + cK_r, cC_o + cK_c$),
     LDA=UNK,
     B($cK_r, cK_c$),
     LDB=UNK,
     C($cR_o, cC_o$),
     LDC=UNK,
     ALPHA = 1.,
     BETA =1.,
     swVecDim=UNK)
```

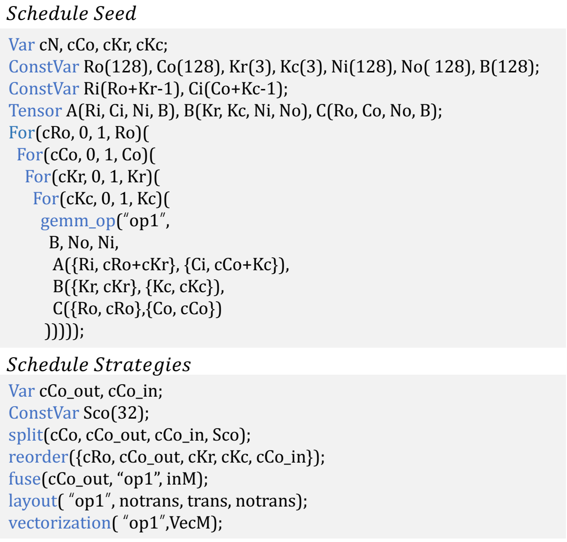
```
for $cR_o$ in range(0:$R_o$)
 for $cC_o^{out}$ in range(0:$C_o/S_{C_o}$)
  for $cK_r$ in range(0:$K_r$)
   for $cK_c$ in range(0:$K_c$)
    $cC_o = cC_o^{out} \times S_{C_o}$
    gemm_op(
     M=$S_{C_o}$B, N=$N_o$, K=$N_i$,
     A($cR_o + cK_r, cC_o + cK_c$),
     LDA=UNK,
     B($cK_r, cK_c$),
     LDB=UNK,
     C($cR_o, cC_o$),
     LDC=UNK,
     ALPHA = 1.,
     BETA =1.,
     swVecDim=UNK)
```

```
for $cR_o$ in range(0:$R_o$)
 for $cC_o^{out}$ in range(0:$C_o/S_{C_o}$)
  for $cK_r$ in range(0:$K_r$)
   for $cK_c$ in range(0:$K_c$)
    $cC_o = cC_o^{out} \times S_{C_o}$
    gemm_op(
     M=$S_{C_o}$B, N=$N_o$, K=$N_i$,
     A($cR_o + cK_r, cC_o + cK_c$),
     LDA=notrans,
     B($cK_r, cK_c$),
     LDB=trans,
     C($cR_o, cC_o$),
     LDC=notrans,
     ALPHA = 1.,
     BETA =1.,
     swVecDim=UNK)
```

```
for $cR_o$ in range(0:$R_o$)
 for $cC_o^{out}$ in range(0:$C_o/S_{C_o}$)
  for $cK_r$ in range(0:$K_r$)
   for $cK_c$ in range(0:$K_c$)
    $cC_o = cC_o^{out} \times S_{C_o}$
    gemm_op(
     M=$S_{C_o}$B, N=$N_o$, K=$N_i$,
     A($cR_o + cK_r, cC_o + cK_c$),
     LDA=notrans,
     B($cK_r, cK_c$),
     LDB=trans,
     C($cR_o, cC_o$),
     LDC=notrans,
     ALPHA = 1.,
     BETA =1.,
     swVecDim=VecM)
```

图 6.2　一个调度器工作示例

循环变换：循环排布方式对算子实现的性能有着巨大的影响，循环变换的方



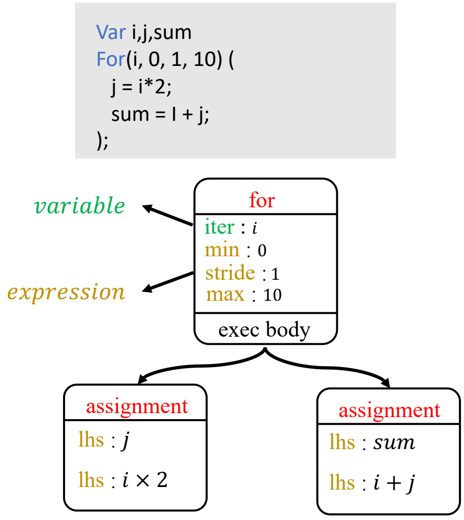
Schedule Seed

```
Var cN, cCo, cKr, cKc;
ConstVar Ro(128), Co(128), Kr(3), Kc(3), Ni(128), No( 128), B(128);
ConstVar Ri(Ro+Kr-1), Ci(Co+Kc-1);
Tensor A(Ri, Ci, Ni, B), B(Kr, Kc, Ni, No), C(Ro, Co, No, B);
For(cRo, 0, 1, Ro)(
 For(cCo, 0, 1, Co)(
  For(cKr, 0, 1, Kr)(
   For(cKc, 0, 1, Kc)(
    gemm_op(“op1”,
     B, No, Ni,
      A({Ri, cRo+cKr}, {Ci, cCo+Kc}),
      B({Kr, cKr}, {Kc, cKc}),
      C({Ro, cRo},{Co, cCo})
    )))));
```

Schedule Strategies

```
Var cCo_out, cCo_in;
ConstVar Sco(32);
split(cCo, cCo_out, cCo_in, Sco);
reorder({cRo, cCo_out, cKr, cKc, cCo_in});
fuse(cCo_out, “op1”, inM);
layout( “op1”, notrans, trans, notrans);
vectorization( “op1”,VecM);
```

图 6.3　一种调度策略的 DSL 书写示例



```
Var i,j,sum
For(i, 0, 1, 10) (
   j = i*2;
   sum = I + j;
);
```

图 6.4　一个简单的 IR 表示示例

式包括循环切分、循环重新排序和循环融合，章节 3.3已经详细介绍了它们在张量化优化中的应用，在此不赘述。

数据排布变换：确定了循环变换方式后，还可以使用不同数据布局方式来实现调度种子。数据在内存中的排布会在两个方面影响算子性能。一方面，它通过更改连续访问的内存块大小、跨步长度等参数来影响 DMA 内存访问的性能。另一方面，LDM 中的数据布局方式会影响张量化原语的调用，比如矩阵乘法原语的转置方式，从而影响计算的性能。为了达到高效地进行张量化计算和访存效果，内存数据排布转换应遵循一些规则。外层循环迭代变量对应的维度应该布局在高维，对应张量化原语参数信息的维度可以任意重新排序。

向量化变换：应用不同的向量化方案还可以进一步扩展搜索空间。以章节 4.1.2中矩阵乘法原语的向量化方式为例：对以 M,N,K 为参数的矩阵乘法有两种向量化方式，既可以对 M 循环，也可以对 N 循环进行向量化。向量化带来更高的计算性能的同时，也会对循环长度产生一些限制。比如对 M 维度进行向量化则要求 M 长度必须满足 128 的倍数限制。因此，应该根据循环参数的具体情况选择可行的向量化方案。

swAutoDNN 还提供了以组合方式定义调度策略，例如，循环切分的因子可以是形如 $\{2,4,8,16\}$ 的整数数组，最终的调度搜索空间是所有可能情况的排列组合。所有有效的调度策略构成**调度搜索空间**。

6.2.4　IR 优化器

调度策略首先被翻译成中间表示 IR，以便灵活地对最终代码实现进行优化。IR 是由一系列语句节点组成的抽象语法树（AST），通过对此 AST 变换结构和修改节点属性来实现相应的调度策略。本节首先介绍 IR 构建过程，然后介绍三种针对张量化算法的性能优化技巧，它们分别完成 DMA 信息推理、访存延迟和边界处理。

6.2.4.1　IR 的构建

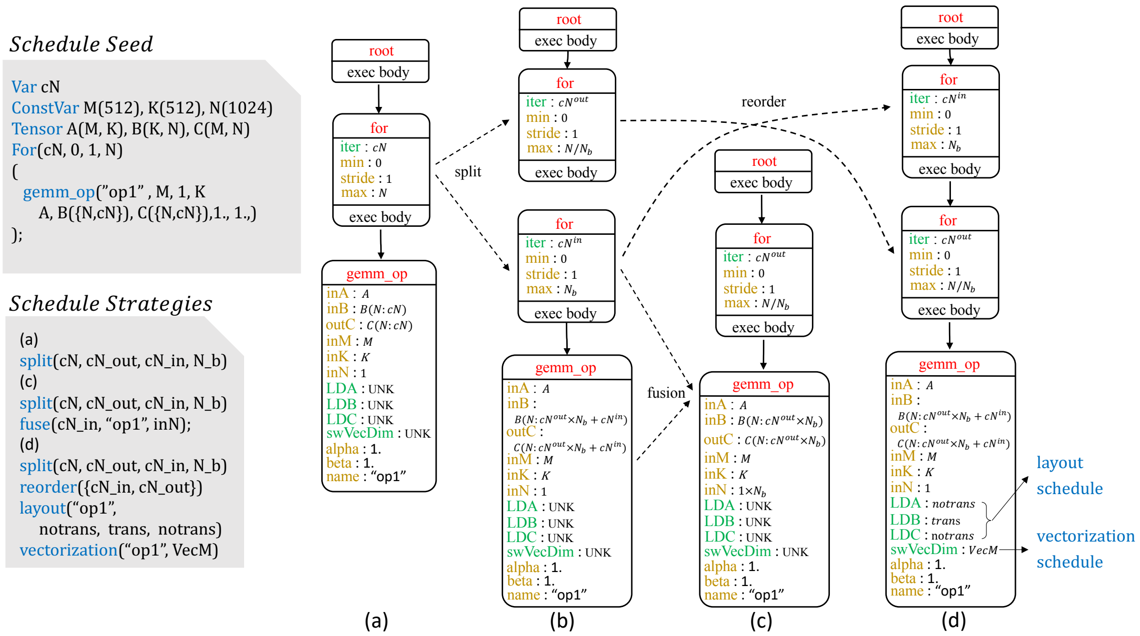
AST 中的节点被称为语句节点，是 IR 的最基本语法单元。一个语句节点表示 `for`、`if-then-else`、`dma`、`gemm_op` 等语句。每个语句节点包含多个属性，属性为变量符号、常量符号或是由两种符号经过算术运算组成的表达式。比如，`for` 语句节点就包含迭代符号变量 `iter` 和 `min`、`stride`、`max` 三个表达式变量。`for` 和 `if-then-else` 语句节点还包含“执行体”，它是一个指向其他语句节点的指针，表示 `for` 和 `if then else` 内具体执行内容。AST 中，子节点为父节点的“执行体”。兄弟节点之间，左兄弟在右兄弟之前执行（或者说，同一个父节点的孩子，由左向右依次执行）。图6.4展示了一个简单的 IR 表示，左边为一个 `for` 循环计算描述，右边为其对应的中间表示语法树。DSL 定义的调度种子可以用来递归地构建 IR。调度策略中使用的信息，比如全局定义的变量和张量，被登记在一个全局的数据结构中。

6.2.4.2　调度策略和 IR 的映射

调度策略可以通过变换 IR 的结构和修改节点属性来实现，下面对三种调度方式进行分类讨论。

循环拆分：首先将一个 IR 的 `for` 语句节点拆分为两个 `for` 语句节点。然后，修改与循环的终止条件相关的属性，并在最内层循环的执行体添加一个表达式，它使用两个新的迭代变量计算原始迭代变量。例如，以 i 作为迭代器变量的循环被分为 i_{out}，i_{in}，添加表达式 $i = i_{out} * factor + i_{in}$ 到循环 i_{in} 中，其中 $factor$ 为切分因子。**循环重排**：它通过重新排序 AST 中的 `for` 语句节点来实现。**循环融合**：这种变换方式有两种情况需要分别处理。如果融合目标是两个循环，声明新的 `for` 节点，并修改对应属性和表达式来替换原始 AST 对应部分，swAutoDNN 使用内部数据结构跟踪修改。如果融合目标是循环和张量计算原语，例如，融合内层循环和矩阵乘法原语的 M 维度，需要消除循环，并修改张量化原语的属性。

图6.5展示了一个对 IR 应用循环变换的例子。通过改变张量原语节点的维度信



Schedule Seed

```
Var cN
ConstVar M(512), K(512), N(1024)
Tensor A(M, K), B(K, N), C(M, N)
For(cN, 0, 1, N)
(
  gemm_op(”op1” , M, 1, K
     A, B({N,cN}), C({N,cN}),1., 1.,)
);
```

Schedule Strategies

```
(a)
split(cN, cN_out, cN_in, N_b)
(c)
split(cN, cN_out, cN_in, N_b)
fuse(cN_in, “op1”, inN);
(d)
split(cN, cN_out, cN_in, N_b)
reorder({cN_in, cN_out})
layout(“op1”,
    notrans,  trans,  notrans)
vectorization(“op1”, VecM)
```

图 6.5　对 IR 应用循环变换示例

息属性和向量化属性，可以实现数据布局转换和向量化变换。

6.2.4.3　推理 DMA 位置

swAutoDNN 在 IR 优化过程中自动推断所需 DMA 原语的位置、属性信息，用户不需要在描述调度种子的 DSL 中对其进行明确定义。IR 优化器构建 DMA 语句节点，推断 DMA 节点的源和目标内存地址、传输数据大小、跨步大小和数据复制方向等属性，并将其插入到 IR 中正确位置。

算法7展示了在 IR 中定位 DMA 节点的方法。DMA 节点的内存地址信息只和 `for` 语句节点的 `iter` 属性有关，所以 swAutoDNN 自动将 DMA 节点插入到所有依赖变量都满足的 `for` 语句执行体中。swAutoDNN 寻找包含计算内存地址需要的 `iter` 属性信息的最内层 `for` 语句节点，如果找到目标 `for` 节点，则 DMA 节点将被插入为其最左边的子节点，否则将成为 IR 的 `root` 节点的最左边的子节点。

6.2.4.4　隐藏访存延迟

如章节 3.3.2介绍，通过软件预取实现访存开销隐藏是张量化编程模型中的关键优化之一。但是，手动实现软件预取是十分繁琐的，尤其对于多层嵌套循环，需要添加复杂的边界判断来精细地计算下一次 DMA 访问的内存地址。在 IR 优化中，swAutoDNN 使用算法8自动推断下一次计算的访存索引。它的基本原理是：找到与

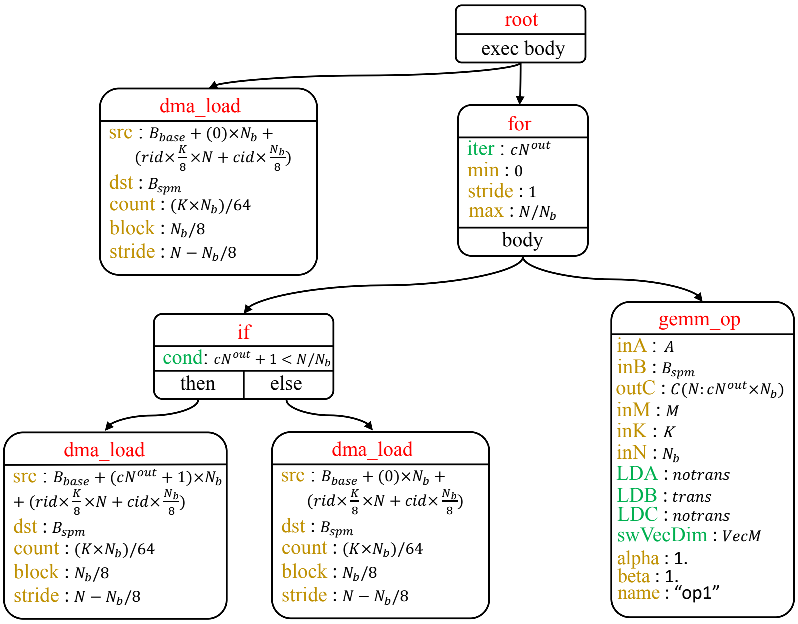


图 6.6　为图6.5（c）IR 结构的输入张量 B 添加 DMA 推理和内存访问延迟隐藏优化之后的 IR 结构

当前需要隐藏 DMA 节点的地址运算表达式相关的最内层循环，如果它的循环迭代指示变量不等于终止条件 end，则对迭代指示变量 $+stride$。如果它的循环迭代指示变量等于终止条件 end，则找到上一层相关的循环，再递归地操作。算法8执行过程会修改循环的属性信息，并在在 IR 中插入 if-then-else 节点和使用 $next\_addr$ 构造的 DMA 节点。算法最后在根节点下添加首次 DMA 访存作为访存重叠流水线的初始化操作。

6.2.4.5　边界自动处理

边界处理问题在使用张量化编程模型时广泛存在。当循环的长度不能除以分割因子时，就会出现边界问题，不能使用原始的张量原语处理边界数据。边界问题使得手动编写高效且正确的程序变得更加困难。一方面，为了处理边界问题，有必要引入一些 if-then-else 语句，从而大大增加了代码的复杂性。另一方面，边界处理倾向于产生冗余计算和额外的存储空间，因此降低了整体性能。

swAutoDNN IR 优化器提供了自动边界处理的功能。当张量原语的参数确定后，如果出现数据大小不能被参数整除的情况，此时采用如下两种策略自动处理边界。

自动原语切换：边界虽然不满足原语参数，无法用既定的原语进行计算，但是依然满足张量原语的限制条件，可以采用新的原语计算。swAutoDNN 会生成在边界处自动切换原语的代码（同时相应的调整 DMA 原语的参数）完成边界计算。

Algorithm 7 在 IR 中定位 DMA 节点位置

```
Input: $node_{dma}$ 需要定位的 DMA 节点; $root$ IR 的根节点.
$\mathcal{L} \leftarrow \{loop_k, loop_{k-1}, ..., loop_0\}$ // 将 for 节点按照从内到外循环顺序排序, $loop_k$ 是最内层的 for 节点
$\mathcal{V} \leftarrow \{var_0, var_1, ...\}$ // 计算 $node_{dma}$ 中地址属性所需要的变量
$node_{for} \leftarrow NULL$
for $loop_i \in \mathcal{L}$ do
    if $loop_i.iter \in \mathcal{V}$ then
        $node_{for} \leftarrow loop_i$ // the target for node
        break
    end if
end for
if $node_{for} == NULL$ then
    让 $node_{dma}$ 成为 $root$ 的最左端子节点
else
    让 $node_{dma}$ 成为 $node_{for}$ 的最左端的子节点
end if
```

自动边界补零：如果边界无法采用自动原语切换的方式处理，则采用补零方式。swAutoDNN 沿用章节 4.2.4提出的轻量级的补零优化。在自动生成的代码中，仅仅拷贝边界数据到一个辅助数组中，并在调用位置加入自动切换数据访问地址的代码。在处理边界数据时，自动切换到边界缓存，从而减小拷贝开销保证程序计算性能。

6.2.5 自动调优器

自动调优器的目标是从调度空间中找到最优的调度策略。一种基本的设计方式是使用黑盒调优，它为所有调度的 IR 生成代码，并通过收集实际执行时间来选择最佳的 IR。但是，由于调度空间可能包含有数千个不同的调度，因此使用黑盒自动调优将需要很长时间。2018 年 Chen 等人的最新工作[172] 通过采集程序特征使用机器学习的方式对 GPU 上深度学习算子进行调优，但是这种方式仍然需要大量的测试。因为申威架构从核阵列是基于延迟的并行方式，所以可以进行相对精确的性能建模，这是基于吞吐量的 GPU 架构所不具备的。使用性能模型方式可以大大加速申威架构张量化算法调优速度。

深度学习算子执行时间由两部分来估算，DMA 的时间 T_{DMA}，指令执行单

Algorithm 8 IR 优化器推断下一次访存地址算法

```
Input: $addr$ 内存地址表达式
Output: $next\_addr \leftarrow addr$
初始化下次次迭代内存地址表达式
$\mathcal{L} \leftarrow \{loop_k, loop_{k-1}, ..., loop_0\}$ //将 for 节点按照从内到外循环顺序排序，$loop_k$ 是最内层的 for 节点
$\mathcal{V} \leftarrow \{var_0, var_1, ...\}$ // 计算 $addr$ 使用的变量
for $loop_i \in \mathcal{L}$ do
    if $loop_i.iter \in \mathcal{V}$ then
        if $loop_i.iter + loop_i.stride < loop_i.max$ then
            使用 $(loop_i.iter + loop_i.stride)$ 替换 $next\_addr$ 中的 $loop_i.iter$
            break
        else
            使用 $(loop_i.min)$ 代替 $next\_addr$ 中的 $loop_i.iter$
        end if
    end if
end for
```

元的时间 $T_{compute}$。T_{DMA} 的估算方法采用章节 3.2.2方法来估计跨步 DMA 时间，$T_{compute}$ 可以采用章节 4.2.3来估计张量化原语计算时间。因为 IR 优化器已经成功重叠 DMA 和计算时间，因此总执行时间 $T_{Overall}$ 是 $T_{DMA}, T_{compute}$ 的最大值。

6.2.6　代码生成器

添加了 DMA 节点、经过访存隐藏、边界处理优化后的 IR 已经包含生成可执行代码的全部信息，swAutoDNN 按照如下步骤将语法树翻译为从核代码。第一步，处理变量声明，将变量符号翻译为变量名，表达式递归翻译为表达式字符串。第二步，根据 DMA 节点信息添加分配和释放从核空间的语句。第三部，从根节点开始递归地翻译语法树生成执行逻辑。对于语句节点，首先根据节点的类型，将节点属性翻译并组成字符串，如果有执行体，则再对它递归地进行翻译。

6.3 实验结果

6.3.1 相对手动优化性能提升

本章以基于隐式矩阵乘法的卷积算法（Implicit-GEMM-CONV）实现的卷积算子为例，它是 swDNN 中最复杂的算法，展示 swAutoDNN 的优化效果。本章沿用章节5.4.1中使用的经典 CNN 测试用例和通用测试用例。

和 swDNN 手动优化相比较，图6.7展示了三种经典 CNN（VGG16，ResNet50，Yolo）中 40 组测试用例上 swAutoDNN 生成代码的加速效果。swAutoDNN 对所有参数都取得了加速效果，平均获得 1.44x 的加速比。大部分测试用例中，swAutoDNN 自动生成的代码都超过了 1500 GFLOPS，考虑到矩阵乘法原语的单双精度转换开销（章节4.1.3），这已经基本等同于矩阵乘法原语的极限性能。对于输入输出通道数 N_i, N_o 比较小的情况，比如 ResNet50 和 Yolo 网络的前几层，swAutoDNN 性能提升效果尤为明显。

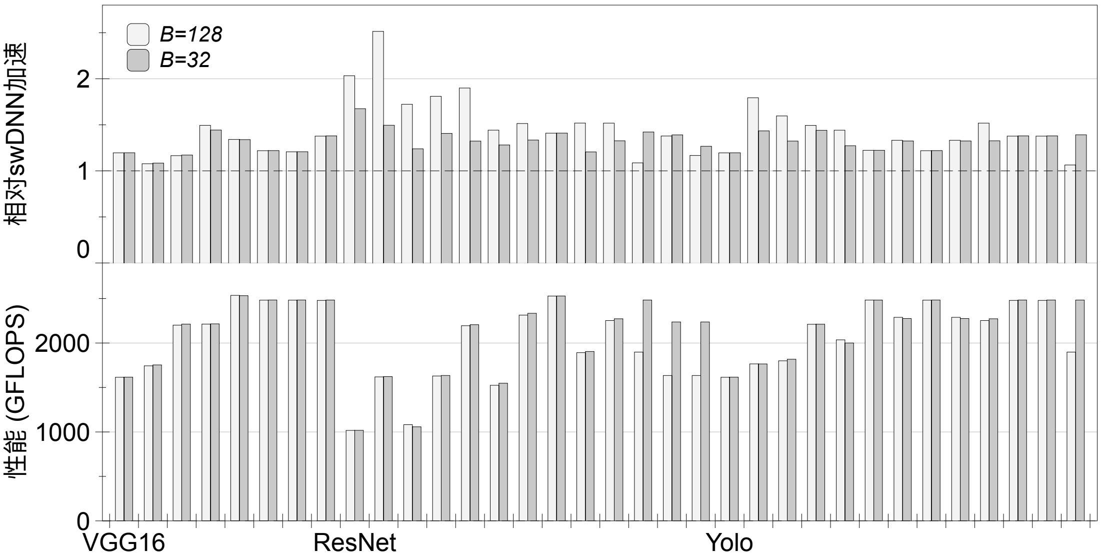
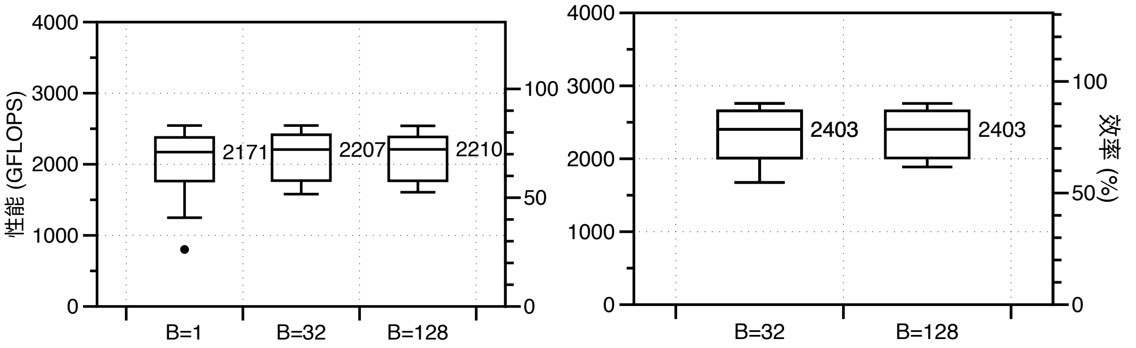


图 6.7　对于基于隐式矩阵乘法的卷积优化，上图展示 swAutoDNN 相对 swDNN 的性能提升，下图展示 swAutoDNN 获得的性能。

图6.8展示了通用测试的性能，测试用例使用上一章实验部分的清单 5.1生成。和 swDNN 中手动优化的实现相比，swAutoDNN 在 B=32、128 情况下的 150 组算例中都取得加速效果。在 B=32 情况下，平均获得了 21% 的加速效果。在 B=128 情况下，平均获得了 28% 加速效果。

swAutoDNN 不仅覆盖了 swDNN 支持的 B=32、128 时算例的全部算子，还有效覆盖了 swDNN 中 Batch-Size-Aware 和 Image-Size-Aware 两个 Implicit-GEMM-



(a) Implicit-GEMM CONV (小卷积核)　(b) Implicit-GEMM-CONV (大卷积核)

图 6.8　使用 swAutoDNN 优化的 Implicit-GEMM-CONV 算子在不同卷积核和不同 B 大小情况下的性能和效率的箱线图

CONV 版本尚未覆盖的参数空间。对于“小卷积核”(卷积核为 3×3）情况，swAutoDNN 新增了 $B=1$ 情况的支持。并且，swAutoDNN 还新增了“大卷积核”且输出特征图图片大小非常小（输出图片为 3×3）的情况。这种情况下，由于无法分块 C_o 维度，Image-Size-Aware 版本失效，只能使用 swDNN 的 Batch-Size-Aware 版本。此时，swAutoDNN 则可以找到替代 Batch-Size-Aware 版本的更优循环变换方案。

图 6.8使用箱线图的方式展示了 swAutoDNN 实现的 Implicit-GEMM-CONV 算子在参数空间上测试的性能结果。测试用例使用脚本6.1生成。对于 $B=1,32,128$ 的小卷积核情况，Implicit-GEMM-CONV 表现稳定，平均获得了超过 2.1 TFLOPS，达到 70% 的峰值计算效率。对于 $B=32,128$ 的大卷积核情况，Implicit-GEMM-CONV 表现相对小卷积核更加优异，平均获得了超过 2.4 TFLOPS，达到 78.5% 的峰值计算效率。

```
for cB in 32 128;
 for cNi in 64 128 256 384 512;
  for cNo in 64 128 256 384 512;
   for cK in 32 64 128 256;
    if[$cNi >= $cNo] ./test_swDNN B=$cB Ni=$cNi No=$cNo Co=3 K=cK
```

Listing 6.1　大卷积核测试的参数生成脚本

6.3.2　自动调优性能和效果

以 Implicit-GEMM-CONV 调优为例，笔者在此将基于性能模型自动调优结果和基于暴力枚举的黑盒调优结果做对比。自动调优时间可以通过将执行时间乘以实际运行次数来获得。对于特定算子，黑盒遍历调度空间的每个调度策略以发现最佳代码，基于性能模型的自动调优方法使用性能模型通过简单的四则运算来评估最佳的循环变换策略。

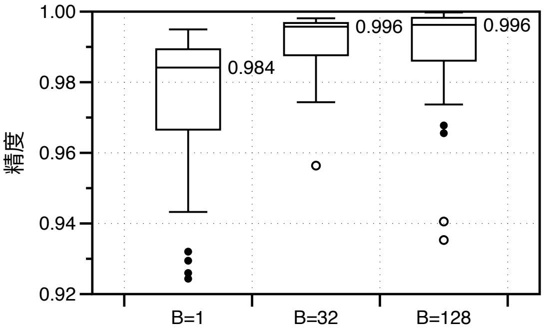


图 6.9　对于脚本 5.1产生的 225 不同参数的测试用例，swAutoDNN 使用基于性能模型方法自动调优性能和黑盒搜索的最佳性能的比值。

如表格 6.1所示，黑盒调优一般花费 2~3 天时间去调优一个完整的 CNN，而本研究提出的基于性能模型的自动调优策略只花费几分钟。对于某个特定卷积算子，黑盒调优需要花费几个小时，而基于性能模型的自动调优花费时间不到 1 分钟。对于 VGG16，ResNet50 和 Yolo 网络的调优速度的平均加速比分别是 **454x, 353x, 365x**。

图6.9展示了 swAutoDNN 的自动调优器找到的最佳版本与通过黑盒暴力搜索找到的真实最佳版本之间的性能差异。这里以箱线图的形式展示二者之间的比值的分布。平均来说，性能模型方式自动调优引起**不到 2%** 的性能损失，即便对于最差的情况，性能损失也不到 8%。

表 6.1　三种经典 CNN 中 implicit-GEMM CONV 性能调优时间

	搜索空间大小		**黑盒调优时间**		**性能模型调优时间**	
	总计	平均	总计	平均	总计	平均
VGG16	4068	454.2	47h 50m	5h 20m	6m 21s	< 1m
ResNet50	7064	353.7	83h 6m	4h 9m	14m 7s	< 1m
Yolo	5112	365.1	60h 10m	4h 18m	9m 53s	< 1m

6.3.3　应用 swAutoDNN 到 swDNN

将 swAutoDNN 优化工具应用于 swDNN 中，本节重新测试卷积算子整体性能。图6.11左图展示了现在的卷积算子在通用测试上的效果，如图所示对于 B=1、32、128 三种情况，卷积算子平均性能可以达到 2180 GFLOSP，2213 GFLOSP，2212 GFLOSP，对应峰值运算效率分别是 73%、74% 和 74%。现在 swDNN 的卷积算子相对章节5.4.1.2中三种情况的运算效率 57%、61%、61%，分别有 28%，21%，21% 的性能提升。

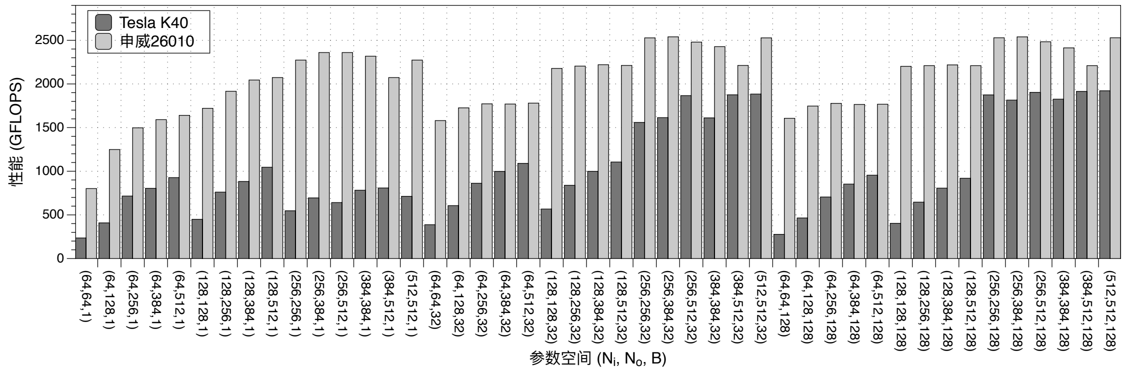


图 6.10 脚本5.1中前 45 组参数的参数空间内申威 26010 和 NVIDIA K40 GPU 卷积性能对比。

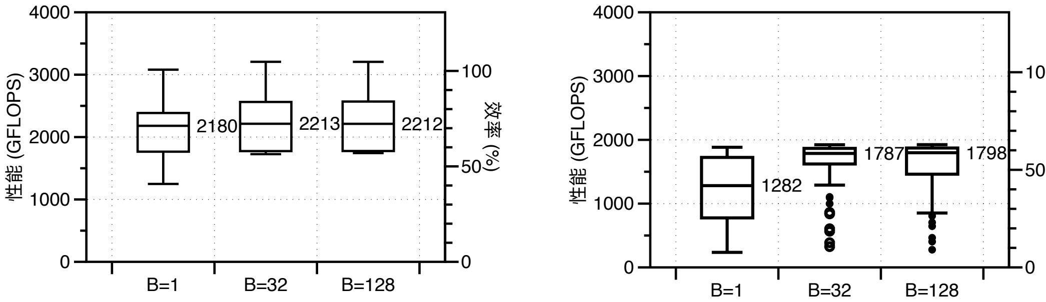
6.3.4 和 GPU 性能对比

本节还比较了 swAutoDNN 优化后的 swDNN 性能和 GPU 上最新版本深度学习算子库 cuDNNv7.5 的性能对比。公平起见，比较对象采用同世代的 NVIDIA Tesla K40 GPU，K40 的单精度浮点峰值性能为 4290 GFLOPS，申威 26010 上从核阵列的峰值性能为 2969.6 GFLOPS。测试用例使用脚本5.1产生的除了 $R_o = 256$ 之外的 178 组算例，之所以刨除大图片测试是因为 Tesla K40 显存限制，无法运行参数太大的算例。其中，cuDNN 无法处理其中较大的 10 组参数，而 swAutoDNN 可以处理整个的参数空间。159 组算例上 swAutoDNN 更优，相比 cuDNN 的**最大加速比为 5.8x，平均加速比 1.71x**。在 9 组没有取得加速的算例中，swAutoDNN 的减速效果也微乎其微，**最大性能损失仅为 1.7%**。图6.10展示了二者性能比较，出于空间限制，笔者展示了前 45 组性能结果（图片 16×16 的情况，其它图片尺寸的性能变化趋势类似）。

图6.11右边展示了 cuDNNv7.5 的整体性能和效率，图6.11左边展示了 swDNN 的整体性能和效率。对比来看，swDNN 生成算子性能更加稳定，从图6.11右边可以看出，B=32、128 情况下最小值位置之下还有大量异常点，说明有些情况 cuDNN 的性能的很差，反观 swDNN 的箱线图中异常点则屈指可数。平均来说，在 $B = 1$ 情况下 swDNN 相比 cuDNN 快 **2.26x**。在 B=32、128 情况下，swDNN 相比 cuDNN 分别快 **1.24x** 和 **1.22x**。

6.4 本章小结

采用将硬件无关与硬件相关部分分离调优的思想，本章为张量化编程模型设计了自动优化方法。本章实现一个名为 swAutoDNN 的自动优化工具，它可以减少手工优化的人力成本，降低张量化算法的开发难度。它由调度器、IR 优化器、自



(a) swDNN (经过swAutoDNN优化)　(b) cuDNNv7.5

图 6.11　在包含 178 组参数的参数空间内，左图 swAutoDNN 优化后的 swDNN 的性能和效率，右图 cuDNNv7.5 的性能和效率。

动调优器和代码生成器四个功能模块组成。

相比 swDNN 中的手动优化，swAutoDNN 生成的代码的性能表现更佳，并且可以生成手动优化设计中尚未覆盖的实现方案。本章以最复杂的基于隐式矩阵乘法卷积算法为测试目标，swAutoDNN 将 swDNN 中卷积算子的效率从 60% 左右提升到 74% 左右。另外，利用 swAutoDNN 优化结果，swDNN 中卷积算子实现的效率远优于 Tesla K40 上 cuDNNv7.5 的效率，而且表现更加稳定。

本章再次验证了章节 3.2.2 设计的性能模型的实用性。与黑盒自动调优相比，使用性能模型能够减少超过两个数量级的时间成本，将自动调优时间从几天缩短到几分钟，只造成了不到 2% 的性能损失。

第 7 章　swCaffe: 基于“神威 · 太湖之光”的并行深度学习框架

以经过张量化优化的算子库为基础，本章设计了一个名为 swCaffe 的深度学习框架，它可以使用“神威 · 太湖之光”多节点进行高效并行训练。目前主流深度学习框架，如 Caffe[72]，Tensorflow[89]，MXNet[91] 等，通常是针对 CPU 和 GPU 组成的异构系统而设计的，在“神威 · 太湖之光”上需要进行定制化的重构。首先，章节5提出的 swDNN 算子库在内存排布、参数范围、调用方式等方面具有自己的特点，所以需要对训练流程进行相应的修改。其次，传统框架的通信和 I/O 模块都是在小规模网络连接、简单磁盘组织方式情况下部署，在超算上进行大规模扩展，需要针对“神威 · 太湖之光”的网络连接和 I/O 方式做针对性的设计。

为了解决传统深度学习框架设计方式和“神威 · 太湖之光”芯片、网络、磁盘等硬件特点不匹配的矛盾，本章设计了基于“神威 · 太湖之光”的并行框架 swCaffe。swCaffe 保持了与目前最流行开源深度学习框架 Caffe 相同的接口，同时可以在“神威 · 太湖之光”系统上高效地部署。本章主要工作如下：

- swCaffe 采用静态图方式构建深度学习训练过程的计算图。它在网络构建前加入定制优化：对齐分配内存和预先补零处理；插入张量转换操作以便高效地调用算子；整理梯度到连续的内存空间来减少网络通信延迟。
- 为了扩展训练任务到多个节点上，本章提出了两层次的数据并行通信方法和高效并行 I/O 策略。单芯片的四个核组使用多线程任务时使用内存作为数据同步媒介，跨芯片间使用互联网络进行通信。
- 为了弥补太湖之光 MPI Allreduce 性能缺陷，本章设计了充分考虑“神威”网络连接拓扑结构的 Topawa-Allreduce 方法，它相比系统自带的 MPI 接口有一个数量级的加速效果。
- 使用目前主流的 CNN 对 swCaffe 单节点和多节点并行训练性能进行评估。在 minibatch 大小为 32K 的情况下，使用 ImageNet 数据训练 CNN 的扩展规模达到 1024 个节点。

7.1　单核组计算性能优化

swCaffe 采用“静态图”方式构造计算图。如章节2.2.2介绍，深度学习训练的计算流程可以表示成以算子为节点，张量数据为边的计算图。深度学习框架的目标是让用户使用配置文件或者脚本语言像搭积木一样轻松构建训练网络的计算图。

swCaffe 是通过对学术界和工业界广泛采用的深度学习软件框架 Caffe[72] 重构而成，采用“定义并运行”的“静态图”方式。“静态图”方式在模型运行之前定义好计算方式，这样可以在计算图层次加入面向“神威 · 太湖之光”的定制优化。

swCaffe 由**内存管理模块**，**算子模块**，**模型构造模块**和**求解器模块**四个功能模块协同工作完成单节点训练任务。内存管理模块负责分配和管理计算图内张量的存储，包括特征图、模型参数和反向传播得到的敏感度、梯度。算子模块是采用本文第5章所优化的各种算子执行网络层的功能。模型构造模块定义了深度学习模型的网络结构，并对算子的调用方式进行定义来完成正反向传播过程。求解器模块负责实现训练参数更新，比如选择使用 SGD 或是 Adam 进行梯度更新，此模块是硬件无关的。针对申威架构的定制优化集中在内存管理模块和网络模块内。

内存管理模块优化：内存分配方式会对 DMA 访存带宽造成影响，影响算子运行性能，因此有必要在计算图层次对内存管理进行优化。如章节 3.3.2所述，以对齐方式访问内存可以减少内存事务的数目有利于提高 DMA 的访存带宽。为此，swCaffe 中定制了 aligned_malloc 接口为所有数据分配内存空间，它能使分配内存的起始地址都是 128 字节对齐的。

swCaffe 在计算图的内存分配过程中，预先分配了边界处理的空间，以避免使用张量化原语时带来的边界补零开销。以 GEMM 运算为例，swGEMM 矩阵乘法库的使用时，如果输入矩阵尺寸没有符合矩阵乘法原语要求，则需要进行补零操作。章节4.2.4提出了轻量级补零的优化技巧来减少内存拷贝开销，这种开销通过使用计算图层次的内存管理被完全消除。swCaffe 先对网络初始计算图进行一遍扫描，预先分配好补零后的内存。比如，基于显式矩阵乘法的卷积实现中为 col 矩阵按照补零后大小分配内存，这样 $im2col$ 过程后，无需补零开销就可以立刻执行计算。

算子模块优化：swCaffe 中，为了实现网络层的功能，算子的调用方式需要精心设计。针对申威架构的各种深度学习算子的具体优化方式已经在第5章进行了介绍。但是，swCaffe 在搭建网络后，每一个网络层可选的算子类型有多种，每个算子又针对不同参数有多种实现方式。比如 swDNN 提供三种卷积算子的实现方式，如基于隐式矩阵乘法、基于显式矩阵乘法和基于 Winograd 的方案。具体到基于隐式矩阵乘法方式，它可以使用 swAutoDNN 进行调优，因此有不同的代码实现方案。swCaffe 对计算图进行一次扫描，通过搜索算子的候选空间来确定性能最好的算子使用方式。在之后的训练迭代过程中，使用最优的实现进行反复迭代训练。

模型构造模块优化：如章节3.3.2描述，必须对多维张量的数据排布进行优化，才能获得最优的访存和计算效率。基于 CPU-GPU 异构架构的深度学习框架采用通

道滞后（Channel-Last）①，或者通道优先（Channel-First）②的布局方式存储深度学习计算使用的张量。而在申威上存储形式更为复杂，比如 swAutoDNN 的调度器会将某个维度拆分为两个维度分布在不同位置，这就会导致训练任务所需的张量无法采用一种统一的排布方式。swCaffe 扫描计算图加入必要的张量变换算子，对于连续多个层采用相同的数据排布进行计算的情况，中间的张量变换可以省略。值得注意的是，反向传播时每一层敏感度计算和梯度计算采用的算子排布可能不同，swCaffe 采用贪心的方式优先适配相对更耗时的算子。

7.2　多节点并行性能优化

在对单核组内的网络训练过程进行优化后，swCaffe 需要将训练任务扩展到超算上多个计算节点来加快训练速度。目前开源 Caffe 框架专为使用一台高性能服务器进行独立训练任务而设计，仅支持单 CPU 多 GPU 的计算平台。为了有效地将框架部署到“神威 · 太湖之光”超级计算机上，在单节点功能模块之上添加了**并行通信模块**和**并行 I/O 的优化**。本研究为并行通信模块设计一个定制化的集合通信方法，它也可以拓展到“神威 · 太湖之光”上其他并行应用程序中。

7.2.1　并行通信模块

7.2.1.1　“神威 · 太湖之光” 网络硬件特点

在介绍 swCaffe 通信方式和优化之前，先介绍一下“神威 · 太湖之光”的网络硬件特点。如图7.1所示，“神威 · 太湖之光”的计算节点是通过两级的 InfiniBand 网络连接。在底层是一系列通过交换机连接而成的超节点（Supernode），太湖之光总共由 160 个超节点组成，每个超节点包含 256 个节点，交换机对所有节点提供全分割的双向通信带宽。再上一层，160 个超节点由一个胖树网络 (Fat-Tree)[173] 连接。如图7.1所示，每个超节点和每个中央交换节点有两条链路连接，总共 32 个中央交换节点，这样每个超节点和中央交换网络只有 64 根物理链路相连接，这导致超节点间通信存在 1/4 的带宽裁剪。

本研究对“神威 · 太湖之光”网络的传输带宽和延迟特性进行评估。测试方式使用 OSU-Micro-Benchmarks③提供的基准测试例程。图7.2展示了“神威 · 太湖之光”网络与 Infiniband FDR 网络进行比较的结果。图7.2左半部分展示了带宽测试结果，单向带宽（bandwidth）和双向带宽（bidirect-bandwidth）通过两个节点间多次无

---

① 对应本文第5章中（B,R,C,N）布局

② 对应本文第5章中（B,N,R,C）布局

③ http://mvapich.cse.ohio-state.edu/benchmarks/

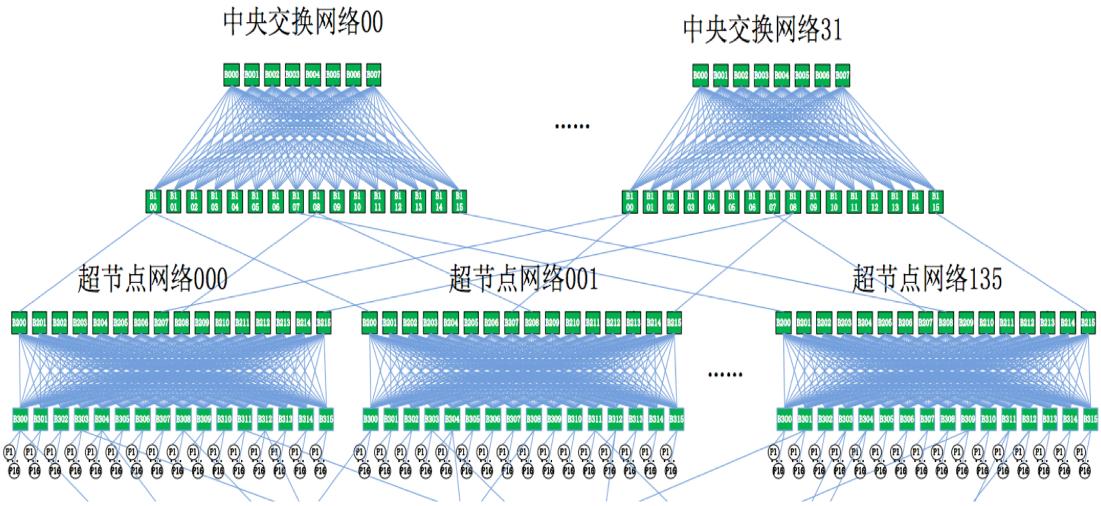


图 7.1　“神威 · 太湖之光”节点间网络连接方式示意

阻塞发送接收操作测试得到，剪裁带宽（Over-subscription bdw）和剪裁双向带宽（Over-subscription bi-bdw）通过超节点 A 的全部 256 个节点和超节点 B 的全部 256 个节点同时进行多次无阻塞发送接收操作测试得到，结果显示两个超节点之间的剪裁带宽大约为满带宽的 1/4。图7.2右半部分展示了延迟测试结果，它通过乒乓测试方式获取。发送方向接收方发送具有特定数据大小的消息，并等待接收方的回复，接收方从发送方接收消息并发回具有相同数据大小的回复，多次执行此乒乓测试并获得平均单向延迟数。由图可知，对于小尺寸消息传递，二者延迟相似。但是，当消息大小大于 2 KB 时，太湖之光网络明显具有更高的延迟。总结来说，尽管“神威 · 太湖之光”网络拥有和 Infiniband 类似的高带宽，但“神威 · 太湖之光”网络明显具有更高的延迟。

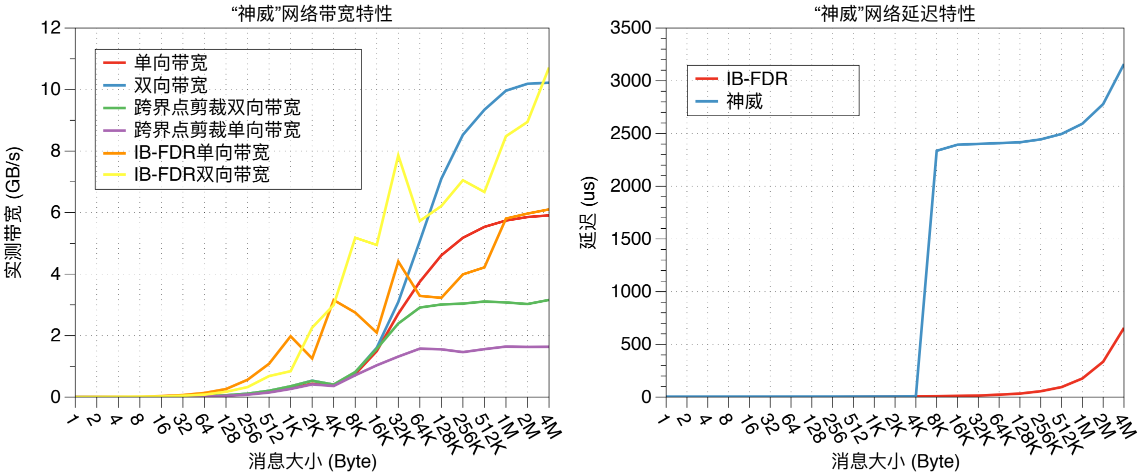


图 7.2　“神威 · 太湖之光”网络和 Infiniband FDR 网络通信的带宽和延迟性能比较

7.2.1.2　两层次数据并行算法设计

swCaffe 采用数据并行方式将计算扩展到多个节点。章节2.3中比较过数据并行、模型并行和流水线并行三种并行策略，并且讨论了异步 SGD 和同步 SGD 的优劣。同步 SGD 方式的数据并行是目前应用最广泛和有效的，它在高性能计算机集群和超级计算机系统中被广泛采用[110] [150]。数据并行对一个 minibatch 数据实例进行划分，然后分配到不同的节点上。每个节点使用本地的数据对模型按照优化方法进行更新，然后通过网络对更新后的模型进行全局同步。swCaffe 采用两层次的数据并行策略，具体流程如算法9描述。

swCaffe 使用多线程技术在单个“申威 26010”处理器内的四个核组并行计算梯度。线程 $i(\in \{0,1,2,3\})$ 处理 1/4 的 minibatch 的数据 x_i，使用共享的参数计算自己的梯度信息。相较多核 Intel CPU 而言，申威架构启动线程的延迟开销很大，为了避免在每个算子计算时都开启一次线程，swCaffe 训练过程只启动一次线程。swCaffe 仅在训练迭代开始前调用一次 pthread_create() 在四个核组上启动四个线程，每个线程调用 swDNN 算子库使用 x_i 数据执行正向传播计算。一次迭代计算完毕后，四个核组并行执行规约操作，以得到该 minibatch 计算的梯度。swCaffe 定制了自己的同步函数 customized_sync()，它通过存储在内存中的信号量实现，整个训练过程终止后使用 pthread_spawn() 释放线程资源。

在单个节点获得梯度之后，swCaffe 通过互联网络通信来同步所有节点的梯度，同步机制可以采用参数服务器或 Allreduce 两种方案。在此，本研究使用 Thakur 等人[131] 所采用的经典通信性能分析模型对二者进行比较。此模型中总传输时间由延迟和带宽两项开销组成，在任意两个节点之间发送消息所花费的时间可以建模为 $\alpha + \beta n$，其中延迟项 α 代表每条消息的延迟（或启动时间），与消息大小无关。带宽项 β 代表每个字节的传输时间，n 是传输的字节数。下面，本研究使用此模型分析“神威 · 太湖之光”两种同步方案的性能表现。

参数服务器方案: 此方案将一个或多个节点作为“服务器”来存储参数，计算节点将梯度传送给这些“服务器”，执行规约操作后，再传回计算节点。假设有 p 个计算节点，q 个参数“服务器”，每个“服务器”维护 $1/q$ 全局参数，全局梯度大小为 n。这种方式中，每个计算节点将本地的 $\frac{n}{q}$ 大小梯度信息发送给对应的“服务器”。处理器只有一个网络端口，最多可以发送一个消息，并且可以同时接收一个消息，可得通信开销为 $2(q\alpha + pq\frac{n}{q}\beta) = 2q\alpha + 2pn\beta$。由此可见，它的通信带宽项和节点数呈正比，同时从大量计算节点接收梯度信息可能成为参数服务器通信的瓶颈。

Allreduce 方案: 此方案采用调用 MPI Allreduce 方式同步梯度信息，它的实现分为 Reduce-Scatter 和 Allgather 两个步骤。Reduce-Scatter 过程使每个节点分别得

到不同的 $\frac{1}{p}$ 全局梯度。Allgather 让每个节点收集其他 $p-1$ 个节点的部分梯度，然后拼接起来构成全局结果。Reduce-Scatter 实现有多重方式，如环状和树状算法等，具体算法细节在下一小节会详细介绍。这些实现中，带宽项最大为 $2\frac{p-1}{p}n\beta$，延迟项最大为 2α。Allreduce 的带宽项不再和节点数成正比，因此传输时间远小于参数服务器方案，原因在于 Allreduce 方案可以充分利用网络节点互相连接所带来的聚合带宽的潜力，而参数服务器中计算节点间通信带宽没有被利用。

Algorithm 9 节点 k 上进行两级数据并行

```
输入: 数据集 $\chi$，每个节点上 minibatch 大小 $b$，节点数 $N$，可学习参数 $w = w[0],...,w[M]$
在 4 个核组上分别使用 pthread_create() 启动四个线程
for $t = 0, 1, ...max\_iter$ do
    for 对每个线程 $i$ do
        从 $\chi$ 中抽样 $\frac{1}{4}$ minibatch （$\frac{b}{4}$ 个实例）作为 $x_i$
        使用正向和反向传播计算 $\nabla f(x_i; w_t)$
    end for
    customized_sync()
    $G_t^k = \frac{1}{4}\sum_{i=1}^{4}\nabla f(x_i; w_t)$
    跨节点同步：$G_t^k : G_t \leftarrow \frac{1}{N}\sum_{k=1}^{N} G_t^k$
    $w_{t+1} \leftarrow SGD(w_t, G_t)$
end for
pthread_spawn() 结束四个线程
```

7.2.1.3　网络连接拓扑方式友好型 Allreduce

笔者发现“神威 · 太湖之光”提供的 MPI Allreduce 例程并不能满足 swCaffe 同步性能需求，考虑到“神威”网络硬件高带宽、高延迟、跨超节点带宽剪裁特点，本研究设计了适合其连接拓扑的 Allreduce 方案来代替申威 MPI 编译器提供的默认方案。

太湖之光提供的 `MPI_Allreduce` 例程是采用 MVAPICH①中的经典算法而实现的，并没有考虑“神威 · 太湖之光”网络连接特点。首先，它的通信模式并没有充分考虑网络分层连接的拓扑结构，从而造成带宽剪裁的情况发生。另外，它的

---

① http://mvapich.cse.ohio-state.edu/

数据规约操作是在主核上执行的，在数据传输量很大情况下，规约计算反而成为了性能瓶颈。

在介绍本研究优化后的 Allreduce 之前，笔者先使用通信模型来分析经典 Allreduce 算法在“神威 · 太湖之光”上的实现性能。考虑“神威 · 太湖之光”网络的复杂性，本研究添加一些必要参数来完善通信性能模型。本小节假设 p 是为 Allreduce 操作中的节点总数，q 是一个超节点中的节点数，β_1 是一个超节点内进行满带宽的传输时间，$\beta_2(\approx \frac{1}{4}\beta_1)$ 是跨超节点通信带宽剪裁后的时间，γ 是对一个字节的数据执行规约操作的时间成本。

Thakur 等人[131] 在 Mpich2 的设计中总结了两种实现 Allreduce 操作的经典算法，它们分别被称为树状算法和环状算法。

树状算法的一种是 Rabenseifner 算法[174]。图7.3左边展示了经典 Rabenseifner 算法实现方式，通信路径呈二叉树状（Binomial Tree）。Reduce-Scatter 阶段采用 Recursive Halving 算法，执行完毕后，所有节点之间分布式地存储 $1/p$ 大小的规约结果。在第一步中，每个节点与距离自己 $p/2$ 远的节点交换 $n/2$ 大小的数据，并对接收的数据执行规约操作。在第二步中，每个节点与一个距离为 $p/4$ 的节点交换 $n/4$ 数据，并对接收的数据执行规约操作。此过程以递归方式执行 $\log p$ 步，每个步骤传送的数据量减半，通信节点距离也同时减半。

Rabenseifner 算法使用 Recursive Doubling 算法实现的 Allgather。每个节点收集来自其他节点的部分结果，方法类似于 Reduce-Scatter 算法。在第一步中，距离为 1 的节点交换其 n/p 规约后数据。在第二步中，距离为 2 的节点交换它们自己的数据以及它们在上一步中收接收到的数据，其总大小为 $2n/p$。在第三步中，距离为 4 的节点交换它们自己的数据以及它们在前两个步骤中接收到的数据，其总大小为 $4n/p$。最后一步中，节点交换消息大小为 $n/2$，节点距离 $p/2$。

经典 Rabenseifner 节点逻辑编号和物理编号按照递增顺序一一对应。如图7.3所示，在 MPI 启动时物理编号 0，1，2，3... 分配的逻辑编号为 0，1，2，3...。经典 Rabenseifner 算法的通信开销在公式7-1，公式7-2和公式7-3进行了推导。最后两个方程是通过对每个时间步长的成本求和得到的，可以将其视为几何级数求和问题。

$$t_{tree-allreduce} = t_{tree-reduce-scatter} + t_{tree-allgather} \tag{7-1}$$

$$t_{tree-reduce-scatter} = \log p\alpha + \frac{n(q-1)}{p}\beta_1 + \frac{n(p-q)}{p}\beta_2 + \frac{p-1}{p}n\gamma_{tree} \tag{7-2}$$

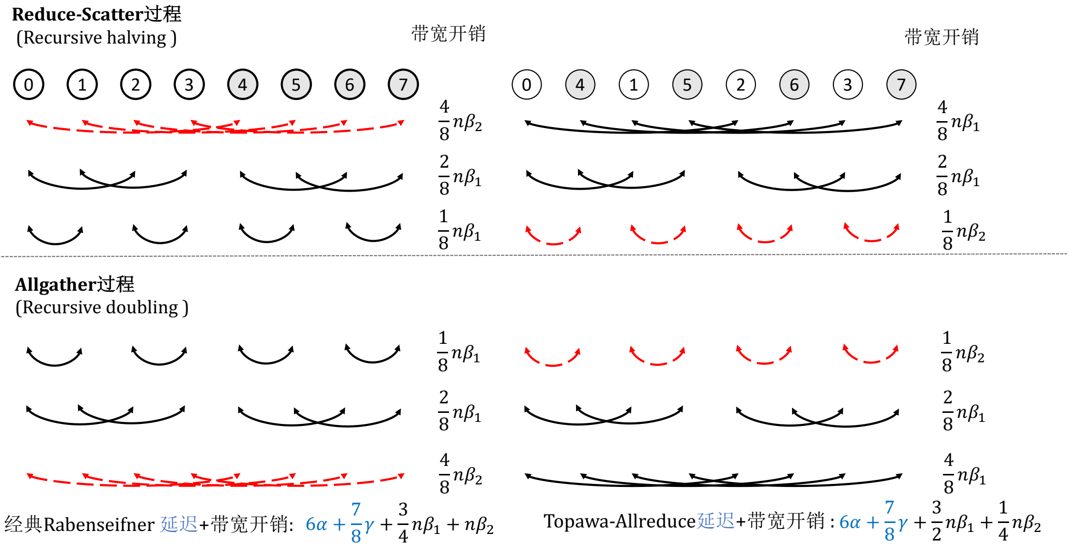


图 7.3　一个简单的例子说明 Topawa-Allreduce 算法相对经典算法的改进。左图是经典 Rabenseifner 算法实现；右图是本研究改进后的 Topawa-Allreduce 算法实现。图中 8 个节点分布在 2 个超节点中。节点 0-3 在一个超节点中，节点 4-7 在另一个超节点中。通信成本标记在右侧，红色虚线表示存在带宽剪裁的跨超节点通信。本研究的方法可以在很大程度上减少跨超节点的流量，从而提升 Allreduce 整体性能。

$$t_{tree-allgather} = \log p\alpha + \frac{n(q-1)}{p}\beta_1 + \frac{n(p-q)}{p}\beta_2 \tag{7-3}$$

环状算法中，每个节点将本地消息均分为 p 份数据块。此方法的通信路径呈环状，节点 $i \in \{0,1,\ldots,\text{p-1}\}$ 只和自己的左节点（编号为 $(i-1)\%p$）和右节点（编号为 $(i+1)\%p$）进行通信。Reduce-Scatter 阶段的第一步，节点 i 发送第 i 份数据到右节点的对应位置，同时接收左节点上的第 $(i-1)\%p$ 份数据块，并将接收的数据规约到本地对应位置的数据块。第二步，节点 i 发送第 $(i-1)\%p$ 份部分规约得到的数据块到右节点上的对应位置，同时接收左节点上的第 $(i-2)\%p$ 份数据块，并将接收的数据规约到本地对应位置的数据块上。以此类推，直到第 $p-1$ 步之后，节点 i 获得了第 $(i+1)\%p$ 份数据块消息的最终规约结果。

环状算法的 Allgather 阶段，所有节点都共享它们之间的部分规约结果来获得完整的结果。可以通过重复不执行规约操作的环状 Reduce-Scatter 过程实现。即每个节点中将接收到的数据块覆盖为相应的本地数据块。$p-1$ 步之后，每个节点获得最终消息的规约结果后 Allgahter 操作完成。性能模型推导出树状通信算法开销如公式7-4所示。

$$t_{ring-allreduce} = t_{ring-reduce-scatter} + t_{ring-allgather} \tag{7-4}$$

$$t_{ring-reduce-scatter} = (p-1)\alpha + \frac{n(p-1)}{p}\beta_1 + \frac{p-1}{p}n\gamma_{ring} \tag{7-5}$$

$$t_{ring-allgather} = (p-1)\alpha + \frac{n(p-1)}{p}\beta_1 \tag{7-6}$$

本研究对环状和树状算法在延迟、带宽和规约操作三个方面性能进行比较。首先，环状算法中的延迟项为 $p\alpha$，和节点数目成正比。树状 Rabenseifner 算法[131]的延迟项是 $2\log p\alpha$。考虑到“神威 · 太湖之光”网络硬件点对点通信高延迟的特点，树状算法在减少延迟开销上更加有效。另外，环状算法在规约操作上也不具优势。环状每次对 $\frac{n}{p}$ 大小数据进行规约，而树状算法规约对象大小从 $\frac{n}{p}$ 不断增大到 $\frac{n}{2}$。因此，每个字节的平均规约开销 $\gamma_{tree} < \gamma_{ring}$。最后，环状算法中，每个节点只和相邻节点进行通信，因此可以灵活适应各种拓扑结构。树状结构，节点的通信目标每步都在变化，对网络拓扑结构更加敏感。对于经典 Rabenseifner 算法的性能开销公式7-2和公式7-3来说，如果 p 远大于 q，则 $(p-q)\beta_2\frac{n}{p}$ 大小的带宽项将是很大的开销。因此，根据拓扑结构设计最优的树状网络通信形式是本文研究的优化重点。

针对“神威 · 太湖之光”两层连接的网络结构，本研究设计了一种被称为 Topawa-Allreduce(取自英文 Topology-Aware Allreduce 之意) 高效的 Allreduce 集合通信方法。它可以克服经典 Rabenseifner 算法无法感知“神威 · 太湖之光”拓扑结构的缺陷。

由图7.3观察，Rabenseifner 算法中不同步骤中的通信流量不均衡。Recursive Halving 算法逐渐减少流量，Recursive Double 算法逐渐增加流量。在经典 Rabenseifner 算法实现中，由于物理编号相近的节点都被分配在一个超节点内，相同超节点内的节点被分配相邻的逻辑节点号。在 Recursive Halving 的前几个步骤和 Recursive Doubling 的最后几个步骤中，每个节点必须与远距离节点进行通信。这将导致通信带宽剪裁发生，仅利用全双向带宽的 1/4。而且，这些步骤的通信量是所有步骤中最多的。

考虑到“神威 · 太湖之光”网络的拓扑结构特性，更好的 Allreduce 实现应该将通信量大的步骤放置在一个超节点内，通信量少的步骤放置在超节点间。Topawa-Allreduce 重新设计使用的物理距离和逻辑距离之间的关系，通过以轮询的方式将相邻逻辑编号分配给不同超节点的节点。图7.3展示以 4 个超节点为例 Topawa-Allreduce 的理论加速效果。本研究设计的节点逻辑编号分配方案如下：将物理编号为 0,4,8,... 的节点分配给超节点 0；将物理编号为 1,5,9,... 分配给超节点 1，依此类推。新的 Allreduce 在 Reduce-Scatter 阶段最后 $\log\frac{p}{q}$ 步骤和 Allgather 阶段的前 $\log\frac{p}{q}$ 步骤进行跨越超节点通信。对于这些步骤中只需要交换相对少量的消息。公式7-7和公式7-8展示了新的时间开销。新的方法将 β_2 项前的稀疏从 $p-q$ 减少到

$\frac{p}{q}-1$，从而大大减少由于剪裁造成的带宽下降的损失。

$$t_{topawa-reduce-scatter}=\log p\alpha+(p-\frac{p}{q})\frac{n}{p}\beta_1+(\frac{p}{q}-1)\frac{n}{p}\beta_2+\frac{p-1}{p}n\gamma_{tree} \tag{7-7}$$

$$t_{topawa-allgather}=\log p\alpha+(p-\frac{p}{q})\frac{n}{p}\beta_1+(\frac{p}{q}-1)\frac{n}{p}\beta_2 \tag{7-8}$$

7.2.1.4　梯度打包优化

深度学习参数同步对象包括多个网络层权重的梯度信息，使用梯度打包将它们聚合在一块连续内存中可以提高 Allreduce 使用效率。如图7.4所示，上图是 Caffe 中梯度内存分配方式，下图是 swCaffe 梯度打包后的内存分配方式。

通过将梯度信息打包在一起有两方面性能优势。一方面可以缩减延迟开销。对于每一个参数都启动一次 Allreduce 操作会带来巨大的延迟开销，发送许多小尺寸消息不仅无法充分利用带宽也增加了网络延迟开销。另一方面可以增加规约操作执行效率。深度神经网络的每一层参数的大小各不相同，比如，在 VGG16 中第一个全连接层参数的尺寸为 102 MB，而第一个卷积层参数的尺寸只有 1.7 KB。规约操作是访存密集型的，因此 DMA 带宽利用率直接决定规约操作的效率。对内存小尺寸消息进行访问无法充分利用从核的 DMA 带宽，打包后可以保证梯度足够大，从而提升 DMA 带宽利用率。

Jia 等人[106] 的工作也采用了类似的方案，不过它的实现是在每次反向传播过程中进行数据拷贝完成打包操作，而 swCaffe 在网络初始化静态图过程中完成梯度的连续分配。

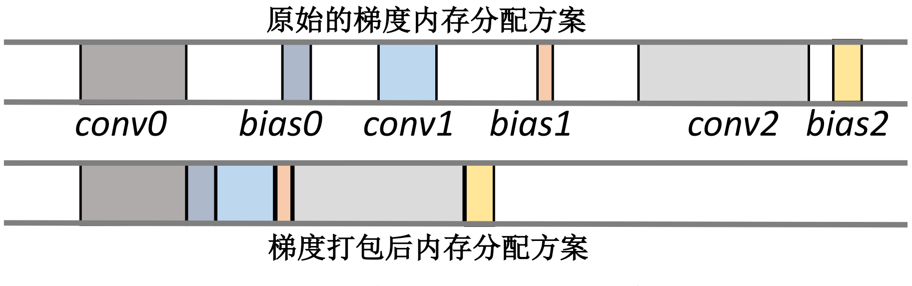


图 7.4　网络构建阶段打包分配梯度内存空间

7.2.2　并行 I/O 模块

“神威 · 太湖之光”的计算节点采用共享文件系统。并行深度学习训练任务的每个进程，使用 I/O 线程在每次迭代之前通过随机采样预取一个小批量数据。“神威 ·

太湖之光”的文件系统默认采用单条带模式（Single-split Mode）进行数据分发，这意味着一个文件连续地被存储在磁盘阵列上。如图7.5左边所示，由于每次训练不同进程访问的数据是相邻的，每个磁盘需要服务大量进程，几乎在串行地被访问。在这种情况下，如果同时读取文件，随着进程数量的增加，多个并发进程的聚合读取带宽很容易就达到单个磁盘阵列的上限，这将导致每个进程 I/O 时间变长。

如图7.5右边所示，swCaffe 通过增加条带数的方式改善 I/O 的性能。swCaffe 采用轮询策略，将数据分为 256 MB 的数据块，在 32 个磁盘阵列上分布式存储。采用优化后的 I/O 策略，不同进程访问磁盘位置被分散，每个磁盘阵列服务的进程数大大减少。笔者发现 32 个条带数和 256 MB 分割大小是最佳设置。如果拆分太小，每个进程都需要访问大量磁盘阵列，在这种情况下，每个磁盘阵列都需要为所有进程提供服务，这会增加对它的搜索开销，从而降低磁盘访问效率。如果拆分太大则造成并行度降低，从而导致性能下降。

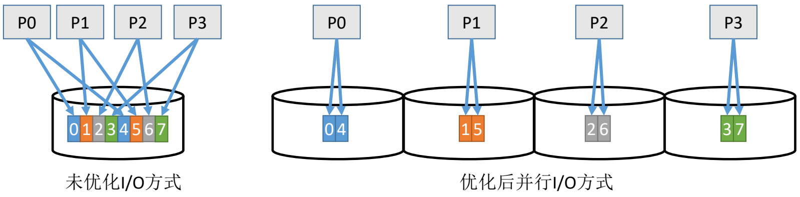


图 7.5　并行 I/O 优化示意，优化后可以避免竞争同一个磁盘阵列 I/O 资源，I/O 聚合带宽提高了四倍。

7.3　实验结果

7.3.1　单节点性能效果

本节以 CNN 训练为目标，将 swCaffe 和 GPU 版本 Caffe，CPU 版本 Caffe 的性能进行比较。目标硬件采用和 SW26010 同世代的 GPU 和 Intel CPU 芯片，其具体性能指标如表 7.1所示。部署在 NVIDIA K40m GPU 上版本的 Caffe 经过 g++-4.8.0 和 CUDA-8.0 编译，并使用 cuDNNv7.5 作为算子库。部署在 Intel E52680 V3 12-核 CPU 上版本的 Caffe 经过经过 g++-4.8.0 编译，并采用 OpenBLAS 优化后的算子库。本章在 ImageNet 数据①上测试了多种经典 CNN，包括 AlexNet，VGG，Google 和 ResNet。

表格7.2展示了 CNN 训练过程的整体性能效果。相对于 Intel 多核 CPU，申威 26010 的浮点运算峰值性能是它的 2.35x，内存带宽是它的 1.88x。申威相比 CPU

---

① http://www.image-net.org/

表 7.1　目标架构的性能指标

	申威 26010	Tesla K40 GPU	E52680 V3 CPU
制造时间	2014 年	2013 年	2014 年
内存带宽	128 GB/s	288 GB/s	68 GB/s
单精度浮点性能	3.02 TFLOPS	4.29 TFLOPS	1.28 TFLOPS

对 CNN 训练的加速效果在 7.17x~9.37x 之间。与硬件的优势相比，申威 26010 上运行 swCaffe 比 CPU 上运行 Caffe 的效率更高，这展示了众核架构在深度学习上的优势。相对于 Tesla K40，申威 26010”的单精度浮点运算峰值性能是它的 0.70x，内存带宽是他的 0.44x。尽管在这两项指标上处于劣势，申威 26010 在 AlexNet 训练任务上仍然相对 K40 取得了 1.19x 的加速。主要原因是由于 GPU 训练 AlexNet 需要将数据从 CPU 内存通过 PCI-E 总线搬移到 GPU 显存，由于 AlexNet 计算速度太快，无法使用计算隐藏 I/O 开销。而申威架构采用片上异构的方式，成功避免了 PCI-E 造成的瓶颈，因此取得加速。对于卷积层计算量比较大的 VGG 系列网络结果，“申威 26010”获得了和 Tesla K40 相似的运行效率，但对于 ResNet50 和 GoogleNet，申威架构效率处于劣势，仅获得的了 0.53x 和 0.66x 的 GPU 性能表现。这是因为这两个网络中访存运算占比较大，申威处理器的带宽劣势使它相较 GPU 较差。

图7.6展示了 swCaffe 训练 AlexNet，VGG 和 ResNet 各部分时间占比。首先，对于 AlexNet 和 VGG 网络卷积层占比相似，都接近 75%，网络整体表现仍以计算密集型为主。而对于 ResNet50，其卷积层运算时间占比不到 60%，访存密集型算子比例明显增多。其次，图标中 Trans 代表张量转置操作的时间占比，三种网络中都不到 10%，可见 swCaffe 数据排布优化是行之有效的，并未引起过多开销。图7.6也解释了 ResNet50 和 GoogleNet 性能相对较差的原因。这两种网络卷积层的通道数比较小，不利于申威架构计算能力的发挥。而且，它的访存密集型算子（主要以 BN 层，ReLU 层为主）占比较大，申威处理的带宽也处于劣势。

7.3.2　多节点性能效果

7.3.2.1　Topawa-Allreduce 性能

本章实现了三种 Allreduce 实现方案，分别是 Topawa-Allreduce、优化后的环状 Allreduce 和“神威 · 太湖之光”的 MPI 编译器 SWMPI 2.2 (基于开源 Mvapich 2.2) 版本的 Allreduce 实现。测试时，随机启动不同超节点中的物理节点。图 7.7展示了

表 7.2　swCaffe 单节点测试和 CPU 和 GPU 的性能对比结果 (img/sec)

	12-Core CPU	**Tesla K40**	**申威 26010**	**申威/GPU**	**申威/CPU**
AlexNet（B=256）	12.01	79.25	94.17	1.19x	7.84x
VGG-16 （B=64）	1.06	13.79	9.93	0.72x	9.37x
VGG-19 （B=64）	1.07	11.2	8.83	0.78x	8.25x
ResNet50 （B=64）	1.99	25.45	14.28	0.53x	7.17x
GoogleNet （B=64）	4.92	66.09	43.62	0.66x	8.86x

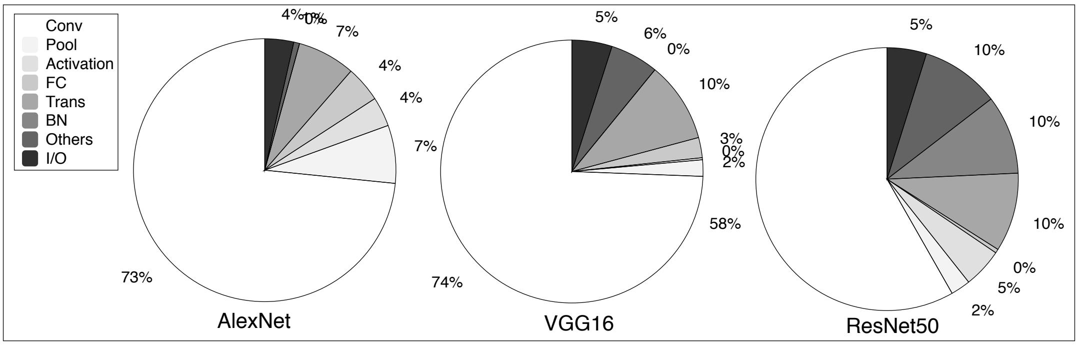


图 7.6　swCaffe 训练过程中各部分占比

它们利用的带宽的对比，纵轴测试带宽（Measured Bandwidth）是通过传输总数据量 $2\frac{(p-1)n}{p}$ 和通信总时间比值得到。

由图可知，在 256、512 和 1024 节点大规模情况下 Topawa-Allreduce 性能都是最优的。相对于 SWMPI 的加速效果在 20x 左右，相对于优化后的环状实现，Topawa-Allreduce 由于延迟开销和规约操作开销的优势，性能也明显占优。Topawa-Allreduce 和环状 Allreduce 随着消息长度增加性能表现变好，这主要因为规约操作利用 DMA 带宽增加的缘故，环状 Allreduce 需要大于 512MB 才能充分利用 DMA 带宽，而 Topawa-Allreduce 超过 32MB 就可以充分利用 DMA 带宽。

7.3.2.2　swCaffe 扩展性效果

采用优化后的 Allreduce，swCaffe 的扩展性测试如图 7.8所示。本研究采用 You 等人[109] 提出的 LARS（Layer-wise Adaptive Rate Scaling）方法，它可以成功将 AlexNet 和 ResNet50 两种网络训练的 minibatch 大小增加到 32K，仍可以保证正确的收敛速度。目前，单次迭代使用 minibatch=32K 数据进行计算是对 ImageNet 数据集训练任务的极限规模。本章扩展性测试中，每个计算节点的 minibatch 被固定为 32。ResNet50 的扩展性接近线性，使用 1024 节点训练速度是单节点训练的

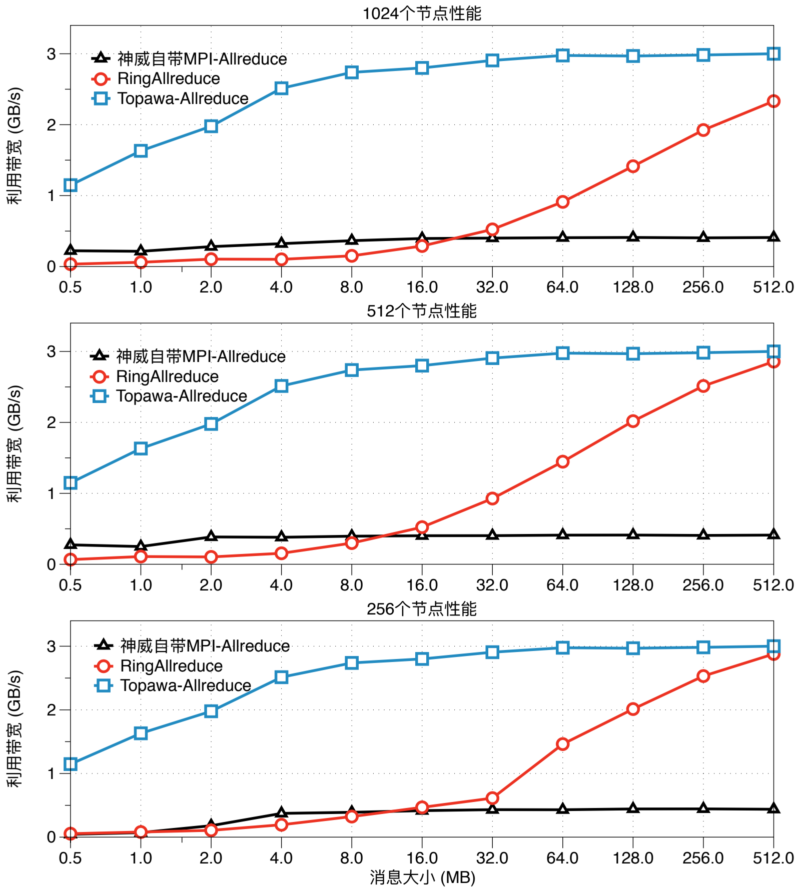


图 7.7 Topawa-Allreduce，RingAllreduce 和“神威 · 太湖之光”MPI 提供的 Allreduce 在不同扩展规模和不同消息长度下利用网络通信带宽能力对比。

1004.4x，弱可扩展性达到 98.0%。对于 AlexNet，1024 节点速度是单节点的 657.7x，弱可扩展性达到 64.2%。AlexNet 的通信参数为 233 MB，而 ResNet50 为 103 MB，是 AlexNet 的 44%，如表 7.2所示，AlexNet 计算时间仅是 ResNet50 计算时间的 15%。AlexNet 的计算时间更短而通信时间更长，扩展性本身就不如 ResNet50，因此在“神威 · 太湖之光”上更不易进行扩展。

图 8.10展示了扩展训练过程到多节点中通信时间占总体训练时间的比例。相比“神威 · 太湖之光”自带的 MPI Allreduce 实现，本研究的梯度打包和 Topawa-Allreduce 优化联合起来可以显著降低通信时间占比。对 ResNet50 在大于 8 个节点规模训练中，通信时间占比从 10% 降低到不到 1%。对 AlexNet 在大于 2 个节点规模训练中，通信时间占比从超过 70% 降低到平均 30% 左右。

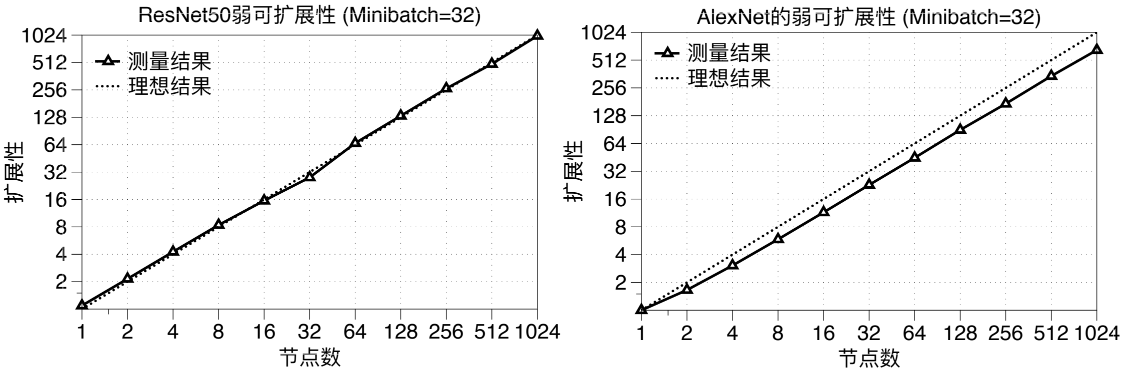


图 7.8　swCaffe 的扩展性测试

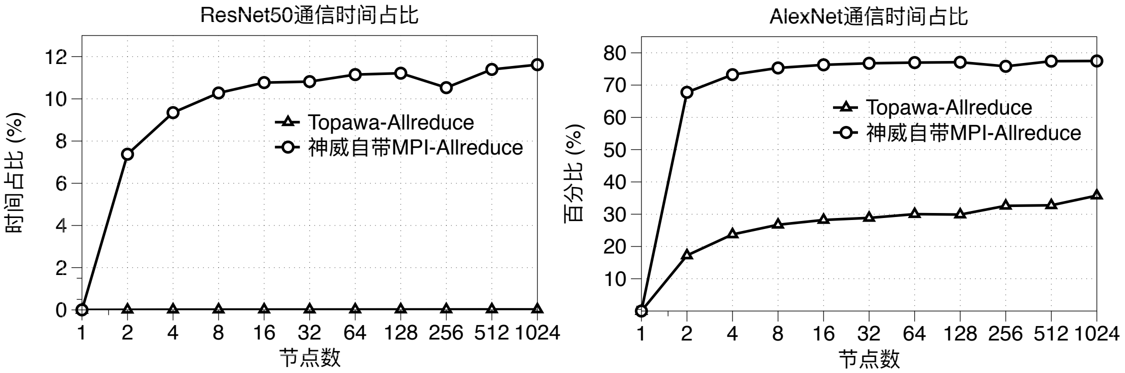


图 7.9　swCaffe 的通信时间占比

7.4　本章小结

本章提出了深度学习训练任务在“神威 · 太湖之光”上多节点并行优化方法，用以实现成名为 swCaffe 的框架。swCaffe 采用静态图的形式生成训练过程计算图。在单节点中，通过在计算图构建前加入面向“申威 26010”定制化的优化，可以保证框架以最优方式调用 swDNN 算子库。这些优化包括：内存管理模块进行数据对齐分配和边界处理，算子模块动态选择最优算子实现向计算图插入必要张量变换操作。本章采用片内多线程和片间 Allreduce 方式的两层次数据并行，可以将训练任务扩展到在多个计算节点上。节点间并行采用一系列优化，包括 Topawa-Allreduce，一个可以感知“神威 · 太湖之光”网络连接拓扑结构的定制的 Allreduce 方案，梯度打包和并行 I/O。在不同网络结构的单节点测试中，swCaffe 获得相对 K40 GPU 上的 Caffe 的 53% 到 119% 的性能。在 ImageNet 数据集上训练 AlexNet、ResNet5 的扩展性测试中，任务规模扩展到 minibatch=32K 规模，使用 1024 节点部署 swCaffe 分别获得 64.2% 和 98.0% 的弱可扩展性。

第 8 章　RedSync: 深度学习数据并行通信压缩方法

数据并行已成为在多个计算节点并行训练深度学习模型的主要方法。随着未来 E 级超算所配备处理器的运算速度显著提升，网络通信将成为深度学习并行训练的主要性能瓶颈。因此，压缩通信数据来减少通信带宽需求的研究受到了广泛关注，在最近提出的几种压缩算法中，**残差梯度压缩**（Residual Gradient Compression，RGC）是最成功的方法之一，它可以将每个节点传输消息压缩为原本大小的 0.1 %，并不损失训练得到模型的精度。然而，关于 RGC 方法的现有文献几乎都专注于理论预研，其在真实并行系统的应用表现还未被充分研究。本章将设计一个名为 RedSync（取自 **Re**ducing **Sync**hronization Bandwidth 之意）的数据并行方案，它利用 RGC 方法可以在超算系统上加快深度学习训练速度。不只考虑理论压缩率，RedSync 以系统运行整体性能提升为目标，在减少通信带宽的同时引入尽可能少的额外开销。本章主要工作如下：

- 在最新的 RGC 研究进展基础上，本章提出了名为交替符号量化法对稀疏化的通信集合进行量化压缩，可以进一步减少一半传输带宽。
- 本章提出了两类并行友好型的 top-0.1% 算法作为通信集合选择算法，它们可以快速实现通信数据压缩的操作，在 GPU 上比目前最快的 top-k 实现快一个数量级。
- 本章提出了基于 Allgather 的稀疏参数同步方案，并使用性能模型推导了 RGC 方法理论性能提升极限和系统实现的瓶颈。使用性能模型，本章澄清了目前学术界在研究 RGC 方法时存在的“认为模型的压缩率等于并行通信带宽的压缩率”的误区。
- RedSync 能够在没有精度损失情况下加速一系列深度神经网络的训练过程。在 GPU 超算 Piz Daint（目前世界排名第 5）上的实验结果表明，扩展到 128 GPU 规模时，RedSync 给那些长期以来被认为扩展性较差的深度神经网络的训练过程（如 VGG，AlexNet 和一些 LSTM）带来了显着的性能提升。另外，本工作为下一代“神威”超算上大规模深度学习系统设计提供了预研。

8.1　研究动机

如本文章节2.3介绍，由于其简单有效的特点，数据并行是目前在多个计算节点上并行训练深度神经网络最普遍的选择。但是，网络通信带宽正逐渐成为限制数据并行性能的瓶颈。一方面，深度神经网络的模型已经包含数十到数百层，拥

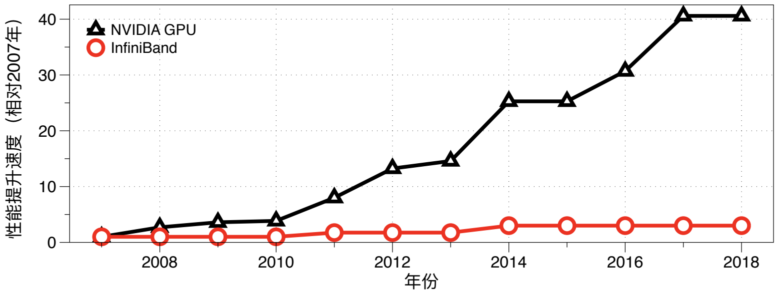


图 8.1　GPU 的单精度浮点运算能力提升速度和 Infiniband 双向互联带宽提升速度的对比

有上千万个训练参数，而且会继续变得越来越大[145]。另一方面，由于互连网络带宽的增速远不及处理器计算速度的增速，同步开销已成为使用最新计算硬件的并行系统上数据并行的瓶颈。如图8.1所示，近十年来以 GPU 为代表的计算硬件的浮点运算能力提升速度已经远远超过以 Infiniband 为代表的高速网络互连带宽的提升速度。2019 年最新的 NVIDIA V100 型号 GPU 的单精度浮点运算性能是 2007 年 G8800 型号性能的 40 倍，相比之下，2019 年的 EDR InifiniBand 的带宽指标只是 2008 年 QDR 带宽的 3 倍。另外，像 InfiniBand 这样用于超算的高质量网络硬件十分昂贵，无法用于每个数据中心，而公有云的计算资源仍采用低质量网络连接。例如，Amazon EC2 现在提供 25 Gbps 的最大带宽，远低于 InfiniBand EDR 4-Link 速度的 96 Gbps。

最近的许多研究都集中在通过减小需要传输的梯度的大小来降低节点之间的通信成本。这些研究可分为两类。其中一类工作[175-177] 使用低精度数据格式来对通信的梯度进行量化（Quantization）。考虑通过量化实现的压缩比（压缩后梯度大小与其原始梯度大小的比率）是有限的，与量化相辅相成的另一类研究是稀疏化通信梯度，并将权重更新限制在一小部分参数上。残差梯度压缩（本章简称 RGC）方法[123-127] 是目前最有希望的梯度稀疏化方法。该方法在获得良好的压缩比的同时，也能确保最终训练出的模型不会有精度损失。它仅传输一小部分梯度，并将剩余的梯度部分保存为残差用以添加到下一次迭代计算出的梯度之上。RGC 的核心思路由 Strom[123] 在 2015 年提出，最初实现版本使用基于阈值的方法对全连接层进行压缩，它仅仅发送数值大于预定义的某阈值的梯度。经过一些改进工作[124] [125] [127]，目前最新的 RGC 方法是 Lin 等人[126] 提出的方案，他们设计了各种算法层面的优化技巧，增加了 RGC 方法的泛化能力，能够在本地梯度上实现 0.1% 的压缩比，同时确保在各种主流深度神经网络上几乎不损失模型精度。

尽管目前最新成果[127] [125] [126] 通过模拟多节点训练实验获得了良好的模型训练精度，但它们没有讨论将最新残差梯度压缩方法集成到实际分布式训练系统后的潜在性能提升效果。将残差梯度压缩应用于超算规模并行系统的挑战来自两个

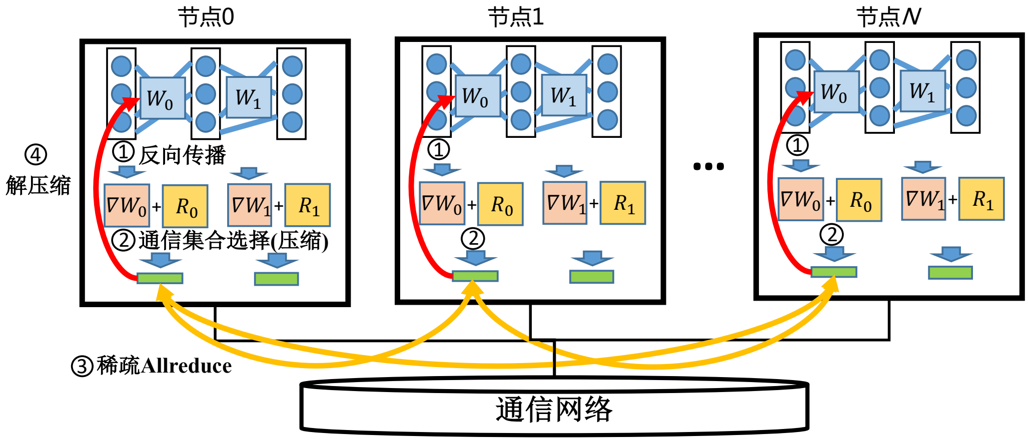


图 8.2　RedSync 系统的设计概览

方面。首先，没有针对 RGC 提出有效的压缩算法实现。根据本章第8.2.1节的实验结果，使用最先进的基于 GPU 的 top-k 算法选择 top-0.1% 元素的开销是很大的，以至于压缩的开销远远高于减少网络带宽的性能增益。其次，缺少对压缩后稀疏数据结构同步方式的研究。超算上实现网络通信的系统工具，比如 MPI 接口，都是针对稠密数据结构设计的。因此必须为 RGC 方法设计有效的稀疏数据同步方式。

8.2　RedSync 系统设计方法

算法10展示了 RedSync 中使用的 RGC 算法基本流程。深度神经网络模型表示为 $f(\mathbf{w})$，其中 $\mathbf{w}$ 将所有参数表达成一维向量形式。假设一个系统有 N 个计算节点，每个计算节点都保存全局权重 $\mathbf{w}$ 的本地副本。算法中 χ_k^t 表示第 k 号计算节点在迭代步 t 的数据，minibatch 大小为 b。RedSync 采用同步 SGD 方法，在每次迭代时，节点 k 使用本地模型副本计算模型的全部梯度 G^k，本章使用 G_j^k 表示层 j 的梯度信息。每个节点还保留一个名为 V^k 的残差，它被初始化为 0，用于累积先前迭代的未传输梯度信息。在添加最新梯度之后，选择残差子集作为**通信集合**（Communication-Set），并将其压缩为稀疏表示的数据结构进行通信。算法10中的 `select` 操作根据数值大小选择更重要的元素。Masks 表示一个 0/1 矩阵，0 表示对应位置元素没有被选择，1 表示该位置元素已经被选择作为通信集合。使用稀疏 Allreduce 操作在所有节点之间同步通信集合。通信集外的剩余元素被指定为下一次迭代的新残差。

Algorithm 10 RGC 方法的基本流程

```
Input: 节点的 id $k$, 总结点数 $N$, 训练数据 $\chi$, 每个节点 minibatch 大小 $b$
Input: 初始模型参数 $w = w[0],...,w[\#layer]$, 压缩率 $D$
  $V^k \leftarrow 0$
  for $t = 0, 1, ...max\_iter$ do
    从 $\chi$ 中抽样 $b$ 个数据 $\rightarrow \chi_k^t$
    使用正反向传播算法计算模型梯度：$G^k \leftarrow \nabla f(\chi_k^t ; \mathbf{w})$
    for $j = \#layer, \#layer - 1, ..., 0$ do
      $V_j^k += G_j^k$
      Masks $\leftarrow$ select $(V_j^k, D)$
      $G_j^k \leftarrow$ Sparse_Allreduce(compress($V_j^k \cdot$ Masks))
      $V_j^k \leftarrow V_j^k \odot$ (1 - Masks)
    end for
    $\mathbf{w} \leftarrow$ SGD($\mathbf{w}$, decompress($G^k$))
  end for
```

图8.2展示了多节点上 RedSync 系统设计概览。第一步，单节点独立执行正反向传播得到梯度，并将它累加到残差中。第二步，通过通信集合选择来进行压缩。第三步，在计算节点间使用稀疏 Allreduce 方式同步残差。第四步，使用解压缩操作累加到权重参数上。在算法改进基础上，在下文中将详细介绍 RedSync 中通信集合选择，稀疏 Allreduce 和解压缩操作等方面的系统设计。

8.2.1　并行友好型通信集合选择算法

从残差中选择通信集合操作的效率对系统的整体性能至关重要，目前仍缺乏高效众核架构实现方法，这也是制约 RGC 方法应用部署的主要障碍。Lin 等人[126-127] 建议从每个网络层的残差中选择绝对值最大的前 0.1% 元素作为通信集合。top-0.1% 操作可以看成 top-k 算法的一种特殊情况，应用 Quickselect 算法[178]，串行地从 n 个元素的数组中选取 top-k 的时间复杂度为 $O(N)$。然而，top-0.1% 选择过程在众核架构上实现却没有单核 CPU 架构上那么简单。目前，众核 GPU 架构上最优的实现是 **radixSelect 算法**[179]，它采用类似基数排序算法。假设排序数组存储为 32-bit 表示的单精度浮点数形式，与基数排序一样，完整的 top-k 过程由若干个相同的步骤组成，每个步骤处理每个元素 32-bit 中 d-bit 的部分，每次执行相同的直方图统计操作和前缀和（Perfix-Sum）操作。radixSelect 和基数排序不同之

处在于，它并不需要将每个元素都正确归置到对应的直方图的桶中，只需使用直方图找到包含第 k 大元素的桶 B，下一步只对上一步寻找到的 B 的元素递归地操作，然后继续检查下一个 d-bit 匹配桶中的元素数字。

然而，radixSelect 前缀和操作的核心——扫描（Scan）运算[180] 和分散（Scatter）操作非常耗时。如图 8.3所示，Titan X GPU 上 radixSelect 的 top-0.1% 的计算时间有时甚至略高于通过带宽为 3.5 GBps 网络同步这些参数的时间。为避免对大量参数执行 top-0.1% 操作，本章提出了两种通信集选择算法，称为**剪枝 top-*k* 选择**和**基于阈值的二分搜索 top-*k***，它们更适合在众核架构上实现。

Algorithm 11 剪枝 top-k 选择

Input: 目标张量 X，通信集合大小 k

Output: 通信集合 $< indice, values >$

```
$mean \leftarrow \texttt{mean}(\mathrm{abs}(X)); max \leftarrow \texttt{max}(\mathrm{abs}(X))$
$\epsilon \leftarrow 0.2; ratio \leftarrow (1 - \epsilon)$
$nnz = \texttt{count\_nonzero}(\mathrm{abs}(X) > threshold)$
while $nnz > k$ do
    $threshold \leftarrow mean + ratio \times (max - mean)$
    $nnz = \texttt{count\_nonzero}(\mathrm{abs}(X) > threshold)$
    $ratio = ratio - \epsilon$
end while
$indice \leftarrow \texttt{nonzero\_indices}(\mathrm{abs}(X) > threshold))$
$values \leftarrow X[indice]$
```

剪枝 top-*k* 选择：累计梯度后的残差分布类似于正态分布，可以使用统计特征去除大多数较小的元素，并在相对较小的数据子集上使用 radixSelect 操作。如算法11所示，首先计算该层的残差绝对值的平均值和最大值。根据平均值和最大值选择一个相对较大的阈值，例如，$0.8 \times (max - mean) + mean$。操作 `count_nonzero` 用来获取绝对值大于阈值的元素个数。如果个数小于 k（top-0.1% 元素的数量），动态降低阈值，直到残差中绝对值高于阈值的参数数量大于 k。然后去掉所有小于阈值的元素，并使用 radixSelect 对其余元素执行 top-k 选择操作。操作 `mean`,`max` 和 `count_nonzero` 都可以通过一次归约操作有效地实现。`nonzero_indices` 是典型的 *Stream Compaction* 问题，它只使用一次扫描操作[181]。

基于阈值的二分搜索算法：对于参数数目巨大的网络层，即使在一小部分残差元素上进行 radixSelect 操作仍然是非常耗时的操作。为了完全避免在 GPU 上使

Algorithm 12 基于阈值的二分搜索 top-k

```
Input: 目标张量 $X$，通信集合大小 $k$，终止条件参数 $\epsilon$
Output: 通信集合 $< indice, values >$
mean ← mean(abs(X)); max ← max(abs(X))
$l \leftarrow 0.0$; $r \leftarrow 1.0$; $threshold = 0.0$
while $r - l > \epsilon$ do
    $ratio = l + (r - l)/2$
    $threshold \leftarrow mean + ratio \times (max - mean)$
    $nnz$ = count_nonzero(abs(X) > threshold)
    if $nnz > k$ and $2k > nnz$ then
        break
    else if $nnz < k/2$ then
        $r = threshold$
    else
        $l = threshold$
    end if
end while
$indice$ ← nonzero_indices(abs(X) > threshold))
$values \leftarrow X[indice]$
```

用 radixSelect 操作，本章提出二分阈值搜索算法来选择最大 0.1%~0.15% 元素作为通信集合。如算法12所示，它不再去准确寻找第 k（最高 0.1%）大元素，而是去搜索一个阈值，使其数值在第 k 大到第 $1.5k$ 大元素之间，然后选择大于该阈值的所有元素作为通信集合。在这种情况下，通信集合中包括至少前 0.1% 的最大元素，因此算法的收敛率也不会被影响。可以使用二分搜索算法来寻找这样的阈值，为了避免过度搜索，当左边界和右边界的差小于一个非常小的值 ϵ 时，算法将自动终止。

对于参数元素非常多的网络层，例如 VGG16 中的第一个全连接层或 LSTM 中的 Softmax 层，`count_nonzero` 操作的时间仍然不容忽视。可以通过减少 `count_nonzero` 操作的数量来进一步提高选择算法的效率。在对该层进行基于阈值的二分搜索操作之后，可以在接下来的几次迭代中重用阈值元素。搜索间隔根据经验设置为 5，选择算法平均只引入一个 `nonzero_count` 开销。这里称这种方案为抽样的基于阈值的二分搜索 top-0.1%。

图 8.3对比了 radixSelect、剪枝 top-0.1%、基于阈值的二分搜索 top-0.1%、抽

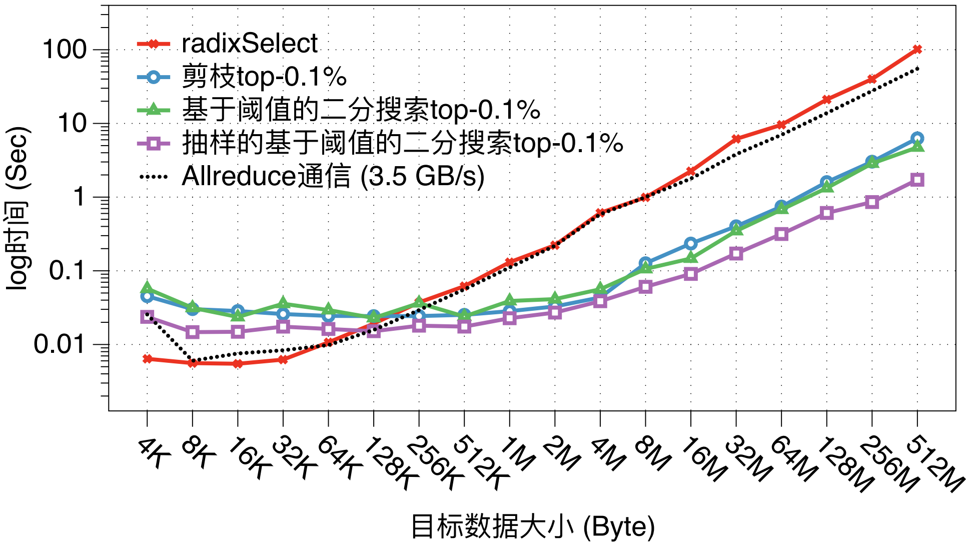


图 8.3　四种通信集选择方法在不同目标数据大小情况下的效果。测试数据使用标准均匀分布随机生成，纵轴是 100 次独立操作的总时间。

样的基于阈值的二分搜索 top-0.1% 和使用实测峰值带宽 3.5GB/s 的网络进行通信的时间，横轴是目标数据的大小。可以观察到，radixSelect 运行时间甚至和通信传输时间相当。抽样的基于阈值的二分搜索-0.1% 效果最好，剪枝 top-0.1% 和基于阈值的二分搜索 top-0.1% 二者时间消耗相当。当目标数据大于 256KB 时，本章提出的三种方法明显优于 radixSelect 的表现，在 64MB 情况下相比 radixSelect 更是有 30 多倍的加速效果。

8.2.2　通信集合的量化方法

压缩后的残差可以存储在包括 k 个索引和 k 个数值的稀疏数据结构中。RedSync 进一步对 k 个数值项进行量化：将通信集中相同符号元素的值设置为它们的均值，可以仅使用一个浮点数而不是 k 个来表示数值信息，从而进一步消除传输值信息的通信带宽要求。Strom 等人的早期工作[123] 就使用了量化和压缩结合的方法。但是，由于通信集合中存在正数和负数，需要分别计算它们的均值，并需要一个 bitmap 存储它们的符号信息，这不仅增加了压缩和解压开销，还增加了带宽需求。

为了便于量化压缩，本章提出了**交替符号量化法**（Alternating Symbol Quantization，ASQ）。它在相邻迭代中交替选择最大的 k 个元素和最小的 k 个元素作为通信集合，以便进行量化压缩。换句话说，如果 ASQ 选择了此次迭代的最大 k 个元素（都是正数）作为通信集合，下一次迭代将选择最小 k 个元素（都是负数）作为通信集合。实验章节展示的结果表明，这种方法仍然可以保证正确收敛速度和

模型精度。因为残差中的梯度信息已经累加了数百个迭代步，晚一个迭代步更新不会对梯度方向产生太大影响。

ASQ 方法实现不会引入任何额外计算，只需要稍微修改 top-k 算法中 `count_nonzero` 操作对象为张量的真实数值而非绝对值，就可以确保通信集中的元素都具有相同的符号。值得注意的是，基于采样的二分阈值搜索 top-k 算法不能与量化优化一起使用。另外，不可以量化 DNN 的输出层，以便区分正确的分类信息。所以，尽管 LSTM 中的 Softmax 层很大，也不要对其进行量化。

8.2.3　稀疏 Allreduce 方法

在传统并行深度学习系统中，使用稠密矩阵为数据结构存储需要同步的信息，不同节点间同步操作通过 Allreduce 操作完成。虽然，Allreduce 在多 GPU 系统上的优化已经被充分研究[149]，但是，在多节点针对稀疏数据结构的 Allreduce 方法设计并不那么简单，因为每个计算节点可以在其压缩残差中贡献不同的非零索引。根据观察，不同节点的通信集分布中相互重叠的索引非常少。例如，使用 16 个 GPU 在 Cifar10 数据集上训练 VGG16，每个节点的压缩比为 0.1 %，所有节点同步残差的平均压缩比为 1.55%。

RedSync 选择利用 Allgather 操作实现针对稀疏残差的 Allreduce 操作。每个节点收集所有节点的通信集合。使用基于阈值二分搜索 top-0.1% 时，每个节点的消息的长度是不同的。因此，打包的消息还应包括一个初始元素，它指示压缩元素的长度。另外，为了避免对索引和数值数据分别使用两次独立 Allgather 操作，RedSync 将它们打包到一个消息包中以减少启动延迟。

完成 Allgather 操作后，每个节点从所有其他节点收集网络层的 N 个压缩后的残差。在使用学习速率进行缩放后，将压缩后的残差累加到本地模型中的相应权重位置，这个操作可以看作是一个将稀疏数组添加到密集数组的操作，可以调用 GPU 上的 cuSparse 库的 1 级函数 axpyi() 实现。

为了分析稀疏同步带来的潜在性能增益，本章沿用章节7.2.1.2使用的通信性能模型[131] 来估计延迟和带宽方面的通信成本。假设在任意两个节点之间发送消息所花费的时间可以建模为 $\alpha + n\beta$，其中 α 是每条消息的延迟（或启动时间），与消息大小无关，β 是每个字节的传输时间，n 是传输的字节数。M 是当前网络层残差中的元素数。D 是压缩率，表示压缩后数据大小与原始数据大小的比率。对于规约操作来说，假设 γ_2 是执行大小为 M 的稠密数据结构的规约成本，和 γ_1 是解压缩 Allgather 收集到的大小为 M 的稀疏数据结构的规约成本。对于每个节点的压缩比不同的情况，比如采用基于阈值二分搜索 top-k 方法时，D 表示所有节点的平

均压缩比。

本章沿用 Rabenseifner 算法[131] 提到的 Recusive-Doubling 算法来实现 Allgather 过程。使用 RGC+ASQ 方法后的量化稀疏数据结构和稠密数据结构的同步时间成本分别用公式8-1和8-2表示。

$$T_{sparse} = T_{select} + \log(p)\alpha + (p-1)(MD)\beta + p\gamma_1 \tag{8-1}$$

$$T_{dense} = 2\log(p)\alpha + 2\frac{p-1}{p}M\beta + \frac{p-1}{p}\gamma_2 \tag{8-2}$$

在此，笔者简要介绍公式8-1的推导过程。图8.4左边部分展示了如何使用 Recursive Doubling 算法来实现 Allgather，从而实现稀疏 Allreduce 过程的。在第一步中，距离为 1 的节点交换其压缩后的数据，其大小为 $M \times D$。在第二步中，距离为 2 的节点交换它们自己的数据以及它们在上一步中接收的数据，大小为 $2M \times D$。在第三步中，距离为 4 的节点交换它们自己的数据及它们在前两个步骤中接收的数据，大小为 $4M \times D$。通过这种方式，对于节点规模为 2 的幂次的扩展规模，所有节点都在 $\lg p$ 个步骤之后获得所有数据。每个节点交换的数据量在第一步为 $M \times D$，在第二步为 $2M \times D$，依此类推，最多为 $2^{lg(p)-1}M \times D$。因此，此算法的消息传输时间为 $T_{transfer} = lg(p)\alpha + (p-1)M \times D\beta$。加上收集的 p 个不同的压缩后残差的解压缩开销 γ 和通信集合选择的开销 T_{select} 得到公式8-1。

章节7.2.1.3已经介绍了公式8-2的推导过程，在此不做赘述。如图8.4右侧部分所示，采用 Rabenseifner 算法对消息进行 Allreduce 操作。Rabenseifner 算法所花费的时间是 Reduce-Scatter、Allgather 和规约操作所花费的时间之和。

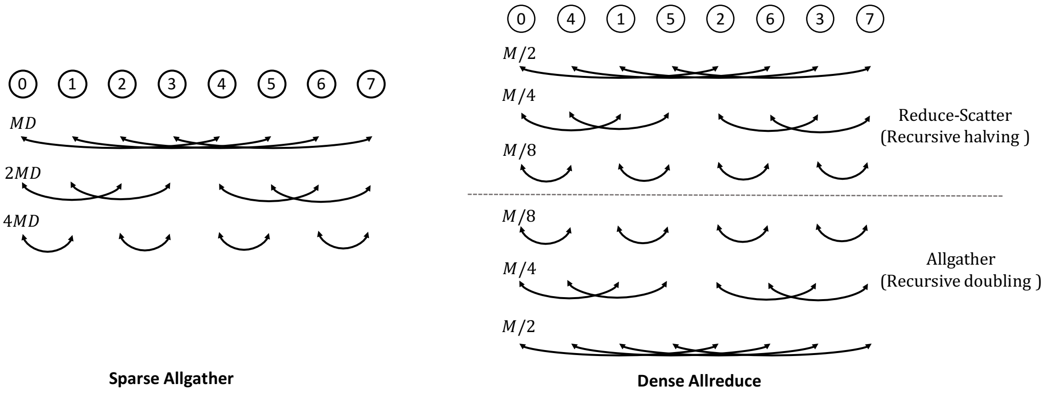


图 8.4　左图：使用 Allgather 的稀疏同步，右图：使用 Allreduce 的稠密同步的通信模式。

分析公式8-1和公式8-2可以得到两个有关 RGC 算法重要结论。第一，因为稀疏同步的带宽项是 $(p-1)(MD)\beta$，其与节点数 p 成比例，所以**模型的压缩率不等于通**

信带宽的压缩率。但是，目前研究 RGC 算法的文章[124-127] 通常将模型的压缩率等价于通信带宽的压缩率进行分析。即使所有 p 个节点残差的压缩率 D 为 0.1%，当 p 为 128 时，稀疏 Allreduce 同步的通信带宽需求将是稠密 Allreduce 同步的 12.8% 而不是压缩率的 0.1%。第二，在将 RedSync 扩展到更大规模时，规约开销可能是一个新的瓶颈。稀疏 Allreduce 性能分析公式8-1的最后一个项 $p\gamma_1$，它表示规约操作开销也随节点数 p 线性增加。但是，稠密 Allreduce 的性能分析公式8-2中规约操作几乎不随节点数增加。

8.2.4　通信计算重叠

通过流水线方式将每一个网络层的反向传播计算和梯度通信重叠起来可以提高数据并行整体效率。在将同步后的梯度更新到权重之前，通常采用梯度修剪（Gradient Clipping）来避免梯度爆炸。当所有梯度的二范式总和超过某个阈值时，它会被重新缩放到一个合理的范围。对于 RGC 方法，Lin 等人[126] 采用局部梯度剪裁（Local Gradient Clipping）技术来达到和原始 SGD 的梯度剪裁同等效果。在将当前梯度添加到先前残差之前，每个计算节点需要独立使用新阈值（原始值的 $N^{-1/2}$ 倍）进行梯度剪裁。经典数据并行算法可以在完成所有层的通信之后进行梯度剪裁，而 RGC 算法需要在通信之前进行剪裁。在这种情况下，则需要等待整个反向传播的完成才能获得所有层的梯度，然后再对梯度进行剪裁、压缩和同步通信操作。因此，它能在计算和通信之间引入同步，从而消除了通信隐藏的可能性。

RedSync 针对 CNN 和 RNN 采用不同的方案来重叠计算和通信时间。如图8.5所示，因为深层 CNN 很少有梯度爆炸的问题，RedSync 选择不对 CNN 进行梯度剪切从而可以将通信和计算进行重叠。去除掉剪裁操作后，压缩完成后立刻开始该层传输残差的通信，此层的反向传播计算可以与后一层的通信重叠。对于 RNN，需要使用 Back Propagation Through Time（BPTT）方法进行反向传播，当完成最后一层的最后一个时间步的反向传播后，RedSync 使用所有层的梯度来进行局部梯度修剪。在这种情况下，通信时间只能与压缩计算重叠。但是，即使采用经典的数据并行方法，LSTM 中也只有最后一个时间步可以重叠计算和通信时间。相比而言，RedSync 实现 RGC 的方式并没有引入太多的开销。

8.2.5　其它技巧

RedSync 实现了 Lin 等人[126] 工作中提出的一系列可以增加 RGC 算法稳定性的优化，并对它们在真实并行环境中因地制宜地进行了取舍。RedSync 采用了 Lin 等人为 Momentum SGD 设计的 Momentum Correctness 和 Momentum Masking 方法。

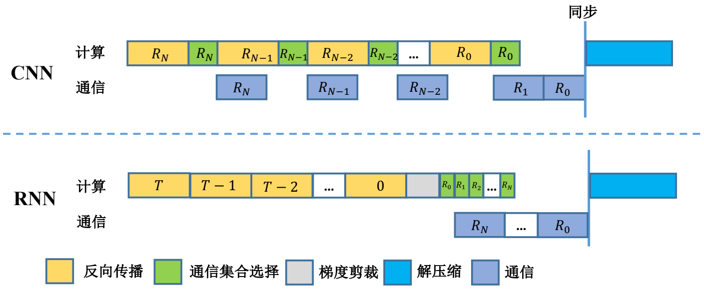


图 8.5　CNN 和 RNN 采用不同的方案来重叠与计算的通信

Lin 等人建议以预热（Warm-up Training）的方式从高到低降低残差压缩比，比如：25%，6.25%，1.5625%，0.4%，0.1%，以避免产生无法收敛的情况。但是，由本章的通信性能模型分析可知，它在大规模并行情况下效率低下。如前一节所述，即使压缩率为 1.5625% 的压缩残差同步，在 64 个节点上使用 RGC+ASQ 方法也需要 100% 的 Allreduce 带宽需求。所以，对于必须进行预热的训练过程，RedSync 在最初几个迭代由 Allreduce 同步原始 SGD 优化器，而不是采用高压缩比的 RGC 方法进行预热训练。

8.3　实验结果

8.3.1　软硬件配置

本章在多 GPU 节点的计算平台上测试了 RedSync 的精度和性能，它们包括世界顶级的 GPU 超级计算机和单 CPU 多 GPU 的服务器。它们使用的 GPU 处理器性能远高于“申威 26010”(3-5 倍增速)，和下一代申威架构芯片目标性能相当，因此可以为下一代“神威”系列超算并行系统设计提供有效预研结果。

Muradin：配备一个 Intel Xeon E5-2640 v4 CPU 和 8 个 TITAN V GPU（单精度峰值性能 14.90 TFLOPS），GPU 通过 PCI-E 3.0 连接到 CPU。

Piz Daint：是目前世界排名第 5 的基于 GPU 组建的超级计算机。它的每个节点包括两个 Intel Xeon E5-2690 v3 CPU 和一个 NVIDIA Tesla P100 GPU（单精度峰值性能 10.6 TFLOPS），总共 5320 个节点通过 Aries 网络采用 Dragonfly 的拓扑结构连接。

本章以 Pytorch（v4.0 版本）深度学习框架搭建 RedSync 系统，并采用 horovod 作为通信接口，用于提供包括 Allreduce 和 Allgather 集合通信操作。在两个系统上部署的 horovod 都是使用 CUDA-Aware OpenMPI v3.1 编译的。

本章测试了两种主流深度学习应用的性能。对于**图片分类**任务，本章在 ImageNet[21] 和 Cifar10[182] 数据集上，对 AlexNet、VGG16、ResNet-50、ResNet-44 和 VGG16 五种 CNN 进行测试。所有 CNN 都使用 Nesterov’s Momentum SGD 作为求解器，它被证明可以获得目前最优训练的精度结果。实验过程对 RGC 方法和 SGD 方法使用相同的学习率调整策略。预热（Warm-up）技术应用于 SGD 和 RGC 的 ResNet50 和 VGG16 的前 5 个 epoch。

针对**语言建模**（Language Modeling）任务，本章选择了两个数据集对 RedSync 进行评估。Penn Treebank 语料库（PTB 数据集）[183] 包括 923,000 个单词用于训练，73,000 个单词用于验证和 82,000 个单词用于测试。WikiText 语言建模数据集[184] 是从 Wikipedia 文章中提取的，它拥有超过 1 亿个单词，包括 2,088,628 个单词用于训练，217,646 个单词用于验证和 245,569 个单词用于测试。在这两个数据集上，本章采用了经典的两层 LSTM 语言模型[185] 评估 RedSync，它的每层有 1500 个隐藏单元，并且编码器和解码器的权重被绑定起来，并使用了梯度剪裁的原始 SGD 方法作为求解器。训练过程中，如果验证集上的 Loss 没有改善时则缩小学习率。

8.3.2　模型精度测试

本章在四个数据集上检验了 RedSync 方案在多种经典网络结构上的收敛性。RedSync 中通信集合选取方式为：当采用剪枝 top-k 作为通信集合选择方法时，所有网络层的压缩比为 0.1%；采用基于阈值的二分搜索 top-k 的压缩比介于 0.1% 和 0.15% 之间。在 Cifar10 数据集，本章使用两个经典 CNN，即 ResNet44 和 VGG16，作为测试用例。在 ImageNet 数据集上，本章测试了 AlexNet，ResNet50 和 VGG16 的训练结果在测试集上的 top-1 错误率。在 PTB 和 Wiki2 数据集上，本章测试了前面提到的两层 LSTM 的训练结果在测试集上的 perplexity 指标。

表8.1展示了全面的模型精度测试结果。表格中的模型大小以 MB 为单位，GFlop 表示使用单个输入样本进行正向传递所需的浮点运算次数。RGC 表示使用 RedSync 实现一种最新 RGC 方法[126] 得到的结果。RGC+ASQ 表示使用 ASQ 对 RGC 方法进一步量化优化得到的结果。CNN 训练模型的精度用验证集上 top-1 的错误率来评估，LSTM 训练模型的精度用验证数据集的 Perplexity 来评估。Cifar10 数据集上的结果是使用 4 个节点，每个节点的 minibatch 大小为 64 训练得到的。ImageNet 数据集上的结果是使用 6 个节点，每个节点的 minibatch 大小为 32 训练得到的。LSTM 的结果是使用 4 个节点，每个节点的 minibatch 大小为 4 训练得到的。

通过表8.1可以观察到，使用 RedSync 实现的 RGC 和 RGC+ASQ 方法训练得到的模型，其准确率结果和 SGD 方法得到结果相似，相差不超过 1%。只有一种情况

（ImageNet-AlexNet）中，SGD 取得了最佳的精度结果，在其余六种情况中，RGC 和 RGC+ASQ 甚至略好于 SGD。在三种情况（Cifar10-VGG16、ImageNet-ResNet50、Wiki2-LSTM）中，RGC+ASQ 略好于 RGC，这说明本章提出的 ASQ 量化方法是非常可靠的，对训练精度不会产生影响。

		模型大小	Gflop	SGD	RGC	RGC+ASQ
Cifar10	ResNet44	2.65	0.20	7.48%	**7.17%**	7.87%
	VGG16	59	0.31	8.31%	8.45%	**8.13%**
ImageNet	AlexNet	233	0.72	**44.73%**	44.91%	44.80%
	ResNet50	103	8.22	24.07%	23.98%	**23.85%**
	VGG16	528	15.5	29.5%	**29.1%**	29.3%
PTB	LSTM	204	2.52	75.86	**75.14**	74.69
Wiki2	LSTM	344	2.52	88.23	88.01	**87.84**

表 8.1　在多种典型的深度学习模型和不同训练集上，本文提出的方法与传统数据并行方法的训练精度结果比较。

	minibatch 大小	128	256	512	1024	2048
ResNet44	SGD	7.09	7.48	8.18	**10.02**	16.8
	RGC	**6.40**	**7.17**	**7.471**	10.13	10.87
	RGC+ASQ	7.06	7.87	7.62	11.86	**10.83**
VGG16	SGD	7.74	8.31	**9.06**	**9.49**	10.09
	RGC	**7.43**	8.45	9.31	9.90	11.12
	RGC+ASQ	8.17	**8.13**	9.09	9.97	**9.81**

表 8.2　在 Cifar10 上不同批量大小的 RGC 和 SGD 方法的测试误差

不仅仅关注于最终结果，本章也对训练收敛速度进行了考察，图8.6展示了使用 RedSync 的收敛曲线情况。左图是验证集上 top-1 准确度 vs Cifar10 上训练 VGG16 的 epoch 数（配置：4 个 GPU，batch size = 256）。中图是验证集上 top-1 准确度 vs ImageNet 上训练 ResNet50 的 epoch 数（配置：8 个 GPU，batch size = 256）。右图是 Perplexity vs PTB 上训练 LSTM 的 epoch 数量（配置：4 个 GPU，batch size = 20）。可以观察到，与原始训练方法 SGD 相比，RGC 方法和 RGC+ASQ 的收敛速度没有太大区别。在使用 ResNet50 训练 ImageNet 的任务中，后两者收敛速度甚至略快于原本的方法。

本章还测试了 RGC 方法对训练参数 minibatch 大小的敏感性。如表格8.2所示，当 minibatch 大小增加到 2048 时，RGC 和 RGC+ASQ 与原始 SGD 相比完全没有精

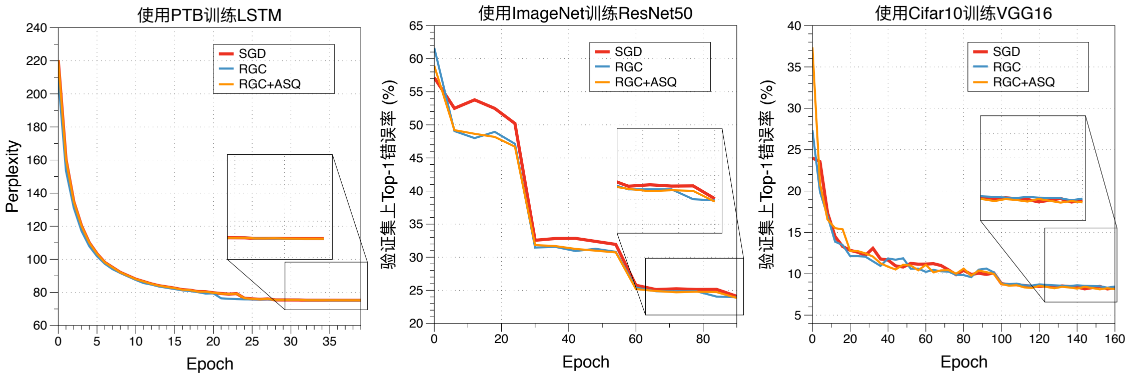


图 8.6　RedSync 的收敛性结果

度损失。

8.3.3　扩展性测试

接下来，本研究测试了 RedSync 系统在不同节点规模上的扩展性。图8.9用了四个测试用例展示了 RedSync 在 Piz Daint 超级计算机上的扩展性。图8.7和图8.8的六个测试用例显示了 RedSync 在 Muradin 上的扩展性，两幅图中都展示了使用 RedSync 实现的 RGC 方案和 RGC+ASQ 方案的结果，并与 horovod 提供的基本数据并行方案做了对比。扩展性通过测试 1000 次训练迭代的平均训练时间归一化得到。RedSync 使用剪枝 top-*k* 算法来压缩大于 128 KB 的卷积层，并且对 LSTM 的 Softmax 层和全连接层使用基于阈值二分搜索 top-*k* 算法。图8.10展示了在 Piz Daint 上将训练规模扩展到 128 个 GPU 时，RedSync 不同部分的时间占比。通过观察，得到如下结论：

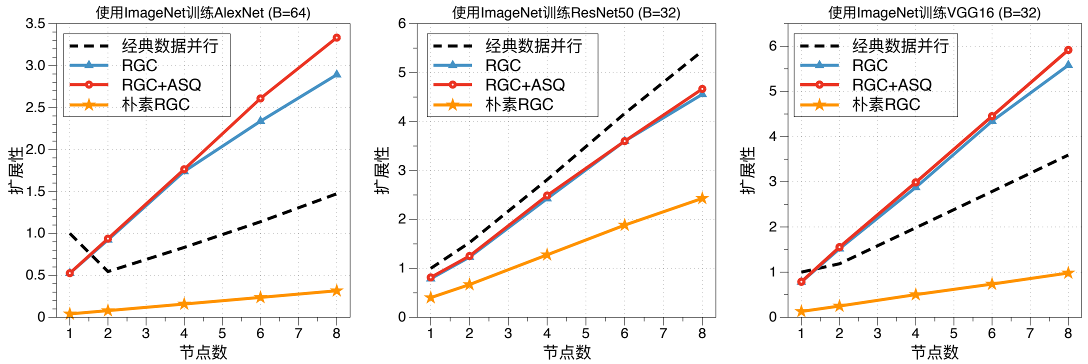


图 8.7　使用 RedSync 在 Muradin 上训练卷积神经网络的扩展性

第一，使用本章提出的并行友好型压缩选择方法对系统整体性能至关重要。在图8.7和图8.8中，本章添加了一个名为“朴素 RGC”的实现，它使用 radixSelect 选择

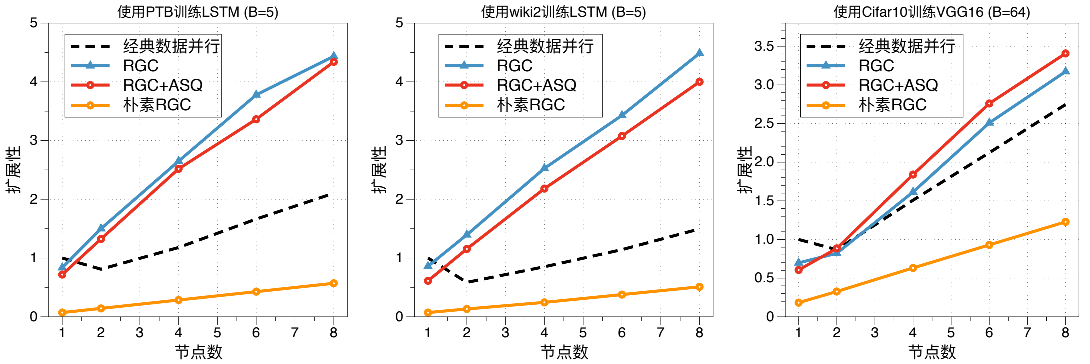


图 8.8　在 Muradin 上使用 LSTM 训练 PTB 和 Wiki2 数据集和使用 VGG 训练 Cifar10 数据集的扩展性

top-0.1% 元素作为通信集。因为压缩时间太长，“朴素 RGC”实现的性能甚至比经典数据并行方案还慢。而只有使用本文提出的两种并行友好型 top-0.1% 算法才能真正带来加速效果。

第二，使用本章提出的 ASQ 量化技巧的 RGC+ASQ 方案在大部分情况会带来性能收益。对于三种 CNN 来说，RGC+ASQ 性能总是比 RGC 性能好。如图8.9所示，在 128 个节点规模训练 AlexNet、ResNet50 和 VGG16 上 RGC+ASQ 相对 RGC 有 1.24x、1.10x 和 1.21x 的整体性能加速。但是，对于 LSTM 来说，在小规模扩展时 RGC+ASQ 的扩展性却比 RGC 扩展性差。这是由通信和压缩开销的相对比例变化引起的，CNN 采用剪枝 top-*k* 作为通信集选择方法，RGC+ASQ 方案和 RGC 方案具有相似的通信集合选择开销。如图8.10所示，在 CNN 训练过程中，RGC+ASQ 和 RGC 方案的通信集合选择的时间成本并没有显著差异。因此，通过量化降低通信成本可以提高系统的整体性能。对于 LSTM，RedSync 使用采样的基于阈值的二分搜索 top-0.1% 算法来加速 RGC 方案的通信集合选择过程，但是，对于 RGC+ASQ 方案，选择通信集合的过程无法使用抽样技巧来优化。使用采样的阈值二分搜索要快得多，因此在小规模扩展性测试时，由于通信开销较小而压缩开销占比较大，RGC 性能此时优于 RGC+ASQ。当扩展规模到超过 16 个 GPU 时，通信开销占比变大，此时 RGC+ASQ 可以通过减少通信时间来补偿通信集选择的成本，所以此时性能优于 RGC。

第三，RedSync 适用于加速通信计算比高的网络模型的并行训练。对于 VGG16，AlexNet 和 LSTM，虽然由于需要压缩和解压缩开销，单个 GPU 上的 RedSync 性能不如经典数据并行方案好，但在两个以上的 GPU 上，使用 RedSync 进行数据并行可以获得显著的加速。唯一例外的是，在 Piz Daint 和 Muradin 上使用 RedSync 并没有对 ResNet50 有性能提升。如表8.1所示，ResNet50 网络的通信与计算量比值

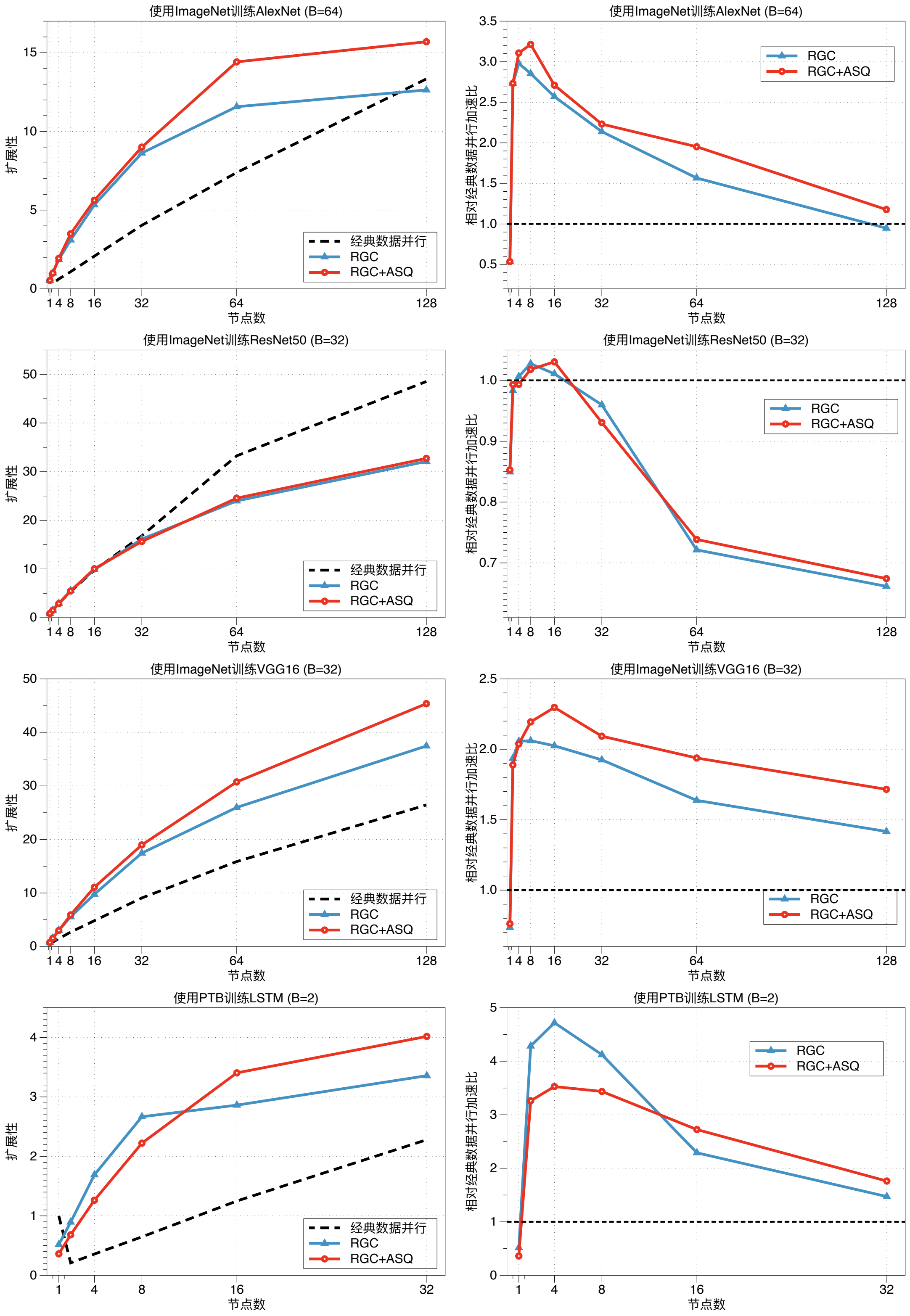


图 8.9　在 Piz Daint 上使用 RedSync 训练 CNN 和 LSTM 的扩展性和相对经典数据并行的加速比

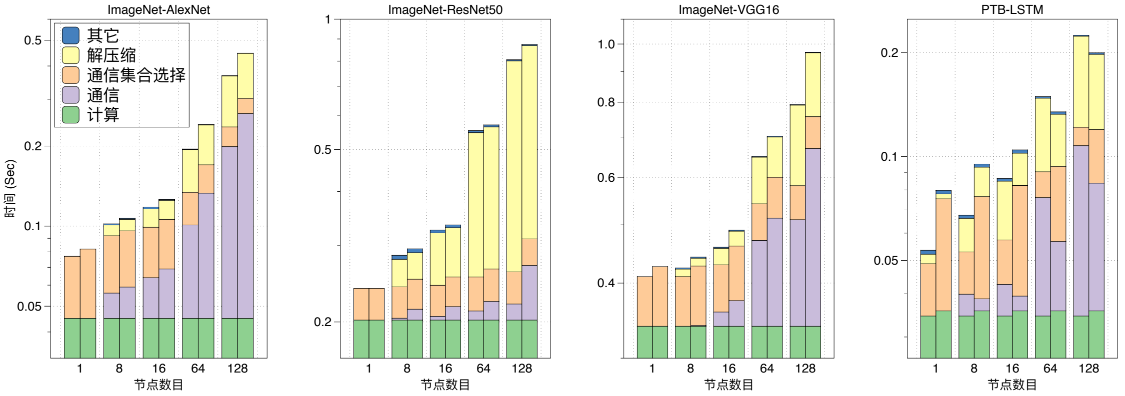


图 8.10　Piz Daint 上 RedSync 运行中不同部分的时间成本，时间是平均 10 次迭代成本。对于每个双列组，左列是 RedSync 实现 RGC+ASQ 的时间分解，右列是 RedSync 实现 RCG 的时间分解。

在本实验涉及的所有网络结构中是最低的。另如图8.10所示，在大扩展规模时，使用 RedSync 实现 RGC 和 RGC+ASQ 对 ResNet50 进行训练的大部分时间都浪费在解压缩阶段，这掩盖了通信带宽减少的好处。

第四，如图8.9右所示，Piz Daint 的实验结果表明中等扩展规模时，RedSync 加速效果最为明显。对于 AlexNet 网络，RedSync 在 4-16 个节点规模时加速效果最明显，在 3x 左右。对于 ResNet50 网络，RedSync 在 4-16 个节点规模时加速效果最明显，在 1.2x 左右。对于 VGG16 网络，RedSync 在 4-64 个节点规模时加速效果最明显，在 2x 左右。对于 LSTM 网络，RedSync 在 2-16 个节点规模时加速效果最明显，在 3x 左右。这是因为通信带宽要求和解压缩开销都随着使用中的 GPU 数量线性增长。这种现象恰好印证了本章第 8.2.3节提出的通信性能模型。

8.4　本章小结

本章提出了一种名为 RedSync 的数据并行设计方法，它通过利用一种称为残差梯度压缩方法来减少同步通信数据量。本章解决了在多节点系统上实现 RGC 方法的两个主要障碍：高压缩开销以及缺乏对稀疏数据结构 Allreduce 方法的支持。在算法层面，本章提出了交替符号量化法可以在稀疏化基础上进一步以量化的方式减少通信带宽需求。在系统层面，设计了两种并行友好的压缩方法，它们分别是剪枝 top-*k* 和基于阈值二分搜索 top-*k* 方法。在充分考虑数据分布特性基础上，RedSync 采用 Allgather 操作在不同节点交换压缩的稀疏数据结构。本章在超级计算机系统和多 GPU 服务器两种平台上测试了 RedSync 的性能。对于 AlexNet，VGG16 和 LSTM 训练任务，RedSync 在 128 个节点规模有 1.24x、1.10x 和 1.21x 左右加速效果。这些网络由于自身的高通信计算比值，在经典数据并行时常常被

认为是难以扩展的。另外，本章发现“模型的压缩率不等于通信带宽的压缩率”，这澄清了目前学术界研究 RGC 方法时的误区，将有助于 RGC 算法系统层面的进一步发展。最后，RedSync 可以为下一代“神威”系列超算上并行算法设计提供参考。

第 9 章　总结与展望

9.1　本文总结

深度学习的发展是由计算和数据共同驱动的，随着物联网、互联网快速发展导致全球数据爆炸式增长，计算能力逐渐成为限制深度学习发展的最关键因素。因此，使用世界上最快的计算设备—超级计算机来满足深度学习日益增长的计算需求已经成为近年来计算机领域的研究热点。

随着国家之间科技竞争加剧，为了突破美国对我国超算技术的封锁，采用国产众核处理器的国产超算将是未来我国超算发展的主旋律。如何利用我国世界领先的超算系统来提升我国人工智能发展水平是超算系统软件研究领域的重要问题。本文以“神威 · 太湖之光”超级计算机为目标，填补了国产超算平台上深度学习系统软件研究的空白。

在对**国产异构众核架构创新特性使用方法**上，本研究得出如下结论：

第一，面对国产众核架构的创新特性，建立性能分析模型可以极大地简化了并行算法开发过程。“兵马未动，粮草先行”，在真正着手编写程序之前，本研究对目标硬件“申威 26010”众核处理器的硬件特性进行了深入研究，在第3章提出了定性和定量的性能模型将体系结构特性进行了量化的描述。它们在系统设计的各个环节都发挥了至关重要的作用，定量的性能模型能够相对精确地估计程序执行时间，成功用于深度学习算子、矩阵乘法运算的自动调优过程；定性的性能模型能快速地反映出不同设计之间优劣，成功指导了深度学习算子张量化算法设计过程。

第二，由于申威架构创新性的硬件特点使算法设计更加复杂，本文提出的“张量化”编程模型可以有力地弥合硬件使用方式和算法设计之间的鸿沟，并且非常适合进行自动调优。“张量化”编程模型以可以存储在片上缓存的张量为基本访存和计算操作单元，非常适合表达深度学习算子的计算流程。通过将硬件相关的优化技巧封装在张量化原语中，自动调优过程得以专注于硬件无关优化。调优过程中，由于张量化原语具有离散的参数空间，也更方便使用性能模型对运行时间进行准确估计，给调优提供了极大的便利。

在**“神威 · 太湖之光”系统软件设计方法**上，本研究形成了如下成果：

第一，针对申威架构配备的核间通信网络特点，本研究设计了相应的矩阵乘法并行优化方法。该方法充分利用核间数据传输能力，有效减少了对内存访问的需求，结合经典向量化优化方法，理论上可以充分利用芯片峰值计算能力。使用这种方法实现了运算矩阵乘法的张量化原语，它的最优性能可达到芯片峰值性能

的 97.3%。

第二，本研究利用张量化编程模型开发了矩阵乘法库 swGEMM，它使用矩阵乘法原语，利用性能模型调优循环变换方式。对于深度学习中常用的狭长形状矩阵的乘法操作，swGEMM 比机器自带的 BLAS 库的 GEMM 例程平均加速 3.02x，有利地弥补了申威架构系统软件的不足。在深度学习系统中 swGEMM 展示了良好的实用效果：对于基于显式矩阵乘法的卷积算子，swGEMM 带来平均 21%~26% 的整体加速效果；对于基于 Winograd 方法的卷积算子，swGEMM 带来 295%~316% 的平均加速效果。

第三，本研究使用张量化编程模型开发了深度学习算子库 swDNN，它高效实现了所有主流深度学习算子，包括卷积、LSTM、全连接等。对于最复杂的卷积算子，本研究提出了三种适应不同参数情况的张量化优化算法，它们分别是基于显式矩阵乘法的优化、基于隐式矩阵乘法的优化和基于 Winograd 的优化。大量测试结果表明，卷积算子的运行效率达到峰值性能的 60% 左右，而且它的性能非常稳定，不会出现在某种参数下性能特别不好的情况。对于 LSTM 算子，使用张量化原语精细控制缓存使用，本研究的实现相对 GPU 算子库使用的方法带来了平均 2.6x 左右加速效果。

第四，本研究开发了为深度学习算子进行张量化自动优化的工具 swAutoDNN。因为张量化编程模型中硬件有关优化细节可以封装在张量化原语中，所以它天然符合硬件有关和无关优化分离调优的思想。它由调度器、IR 优化器、自动调优器和代码生成器组合而成，用户只需定义深度学习运算符 DSL 形式的计算描述和若干调度策略，swAutoDNN 可以自动生成调优后的可执行代码。对于 swDNN 中最复杂的基于隐式矩阵乘法的卷积算子，swAutoDNN 将手工实现的效率从 60% 左右提升到 70% 左右，另外还有效实现了 swDNN 没有支持的参数情况。另外，使用 swAutoDNN 的算子运行效率显著高于 GPU 上 cuDNN 库的实现。与黑盒自动调优相比，采用本研究设计的性能模型的自动调优器能够减少超过两个数量级的时间成本，将自动调优时间从几天缩短到几分钟。即使在最坏的情况下，它也只会带来**不到 8%** 的性能损失。

第五，本研究在 swDNN 算子库基础上开发了并行深度学习框架 swCaffe。单节点优化中，它采用“静态图”方式，在计算图构造前，加入了内存分配和管理、算子调用选择等方面的优化。申威 26010 在 AlexNet，VGG16，ResNet50，GoogleNet 训练测试上，分别获得相对 Tesla K40 GPU 上 Caffe 运行性能的 1.19x，0.72x，0.78x，0.53x，0.66x。多节点优化中，swCaffe 设计了充分考虑网络拓扑结构特点的 Topawa-Allreduce 方法，相比调用系统自带的 MPI 接口，可以有一个数量级的加速效果。

在梯度打包、并行 I/O 两方面进行定制优化后，swCaffe 可以成功扩展深度学习训练到 1024 个节点规模。对 AlexNet 和 ResNet，将网络训练规模扩展到目前极限的数据并行的任务规模（minibatch=32K），swCaffe 在 1024 个节点上相对单节点分别获得 1004.4x 和 657.7x 加速效果，达到了 64.2% 和 98.0% 的弱可扩展性。

第六，本研究开发一种名为 RedSync 的数据并行通信压缩系统，它通过利用目前最高效的压缩方法—残差梯度压缩方法（RGC）来减少同步通信数据量，提高并行可扩展性。在算法层面，本研究提出了交替符号量化法可以在稀疏化压缩基础上进一步以量化的方式减少通信带宽需求。在系统层面，本研究清除了在多节点系统上实现 RGC 方法的两个主要障碍：高压缩开销以及缺乏对稀疏数据结构 Allreduce 方法的支持。对于 AlexNet，VGG16 和 LSTM 模型，RedSync 在 128 个节点规模训练带来 2x 左右整体加速效果。本研究澄清了残差梯度压缩方法有关的一些误解，比如梯度的压缩率和通信带宽压缩率并不等价，这有助于 RGC 方法在未来的继续改进。另外，RedSync 可以作为下一代“神威”系列超算上并行算法设计的参考方案。

9.2　未来展望

深度学习训练系统软件工程浩大，本研究工作虽力求在国产超算上取得突破，但笔者深知目前仍无法面面俱到，在研究的广度和深度上尚有诸多不足之处。

第一，本文提出的性能模型尚无法对寄存器通信行为进行准确的预测。由于缺乏对核组内寄存器通信机制充分了解，从核进行寄存器通信的具体顺序是无法获知的，以往的性能模型工作[155] 甚至对此避而不谈。尽管本研究发现了寄存器通信存在类似同步操作的作用，但是对除矩阵乘法中使用的通信模式之外的情况，仍缺乏更通用的研究。

第二，“神威 · 太湖之光”上还缺乏一个计算图层次的自动调优工具。本研究提出的 swAutoDNN 目前还仅限于算子级别的自动调优任务，而在计算图角度对算子融合、删减等优化还并未涉及。比如，swCaffe 在构建计算图时，选择每一层网络算子实现方式采用简单的贪心策略，尚无法得到理论最优的性能表现。图级别自动优化还可以通过更加智能方式快速找到更优的排布方式，并带来理论最优的性能表现。

第三，“神威 · 太湖之光”还缺乏对模型并行、流水线并行方式的研究。随着深度学习技术的发展，未来更大更深的网络会相继出现，这时内存的限制会导致模型并行、甚至流水线并行在某些情况下或许是替代数据并行的更佳选择。

随着深度学习已经逐渐成为超级计算机上的“杀手”级应用，在我国下一代 E

级超算规划中，明确指出将深度学习作为重点支持目标[45]。本研究从硬件体系结构和系统软件层面也对下一代超算系统设计提供一些参考。

第一，增加更广泛的浮点运算支持。一方面，深度学习训练过程的计算以单精度浮点数为主要格式，目前有些工作甚至已经拓展到半精度（IEEE-754 FP16）。如今，最新的 NVIDIA V100 GPU 所采用的 Turing 架构已经增加了半精度运算的浮点部件。目前，“申威 26010”处理器的单精度和双精度浮点运算峰值相同，而其它众核处理器架构上单精度峰值性能一般是双精度的两倍。另一方面，申威指令集甚至并没有对单精度浮点进行全面的支持。如章节4.1.3所述，一些寄存器通信指令只支持双精度浮点数，而没有对应的单精度浮点数指令。这导致如果单精度运算进行相关寄存器通信操作，必须在 LDM 内显式转换成双精度浮点数才能完成运算，引入了额外开销。因此，希望下一代“神威”超算释放单精度甚至半精度的计算潜力，这将直接翻倍深度学习运算能力。

第二，增加处理器的内存访问带宽。如章节3.1.5所述，和其他同世代主流众核处理器相比，申威架构的计算访存比相对较低，这意味着更难以使应用程序利用到芯片的峰值运算性能。swCaffe 在 ResNet50 上相对 GPU 较差的表现，也正是由于大多数运算受限于内存带宽。而且，随着 NVIDIA 和 Intel 众核处理器采用更新的内存配置，比如 Xeon Phi 采用的 MCDRAM 和 GPU 采用的 HBM2，内存带宽已经达到“申威 26010”的五倍之多。如不在访存带宽上取得突破，未来国产超算对深度学习计算的支持能力将会陷入落后的局面。

第三，提高编译和自动优化工具能力。在本研究的深度学习系统软件构建过程中，花费了大量精力进行手写汇编排布工作，这给软件的快速开发造成了不小的困难。编译软件是应用程序通往底层硬件的最后一道桥梁，高效的编译工具可以让几乎全部国产超算使用人员受益。另外，下一代国产超算如果继续采用核间通信机制和分布式 Cache 特性，那么参考本文章节 3.3的张量化编程模型来设计自动优化工具，将会给国产超算平台开发以极大的便利。

第四，配备更加高效的系统库接口。在本研究对深度学习系统软件的设计过程中，对制造商提供的 BLAS GEMM 接口（章节 4.2）、MPI Allreduce 接口（章节 7.2.1.3）进行了重新优化设计，并取得了相对原始系统显著的加速效果。鉴于这些系统接口使用的广泛性，本研究的实现方案的支持范围不仅限于深度学习应用。在下一代“神威”系列超算设计过程中，可以参考利用这些研究成果。

参考文献

[1] Russell S J, Norvig P. Artificial intelligence: a modern approach[M]. [S.l.]: Malaysia; Pearson Education Limited,, 2016

[2] Nilsson N J. Principles of artificial intelligence[M]. [S.l.]: Morgan Kaufmann, 2014

[3] LeCun Y, Bengio Y, Hinton G. Deep learning[J]. nature, 2015, 521(7553): 436.

[4] Bishop C M. Pattern recognition and machine learning[M]. [S.l.]: springer, 2006

[5] Bengio Y, Courville A, Vincent P. Representation learning: A review and new perspectives[J]. IEEE transactions on pattern analysis and machine intelligence, 2013, 35(8): 1798-1828.

[6] Haykin S. Neural networks: volume 2[M]. [S.l.]: Prentice hall New York, 1994

[7] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[C]//Advances in neural information processing systems. [S.l.: s.n.], 2012: 1097-1105.

[8] Farabet C, Couprie C, Najman L, et al. Learning hierarchical features for scene labeling[J]. IEEE transactions on pattern analysis and machine intelligence, 2013, 35(8): 1915-1929.

[9] Tompson J J, Jain A, LeCun Y, et al. Joint training of a convolutional network and a graphical model for human pose estimation[C]//Advances in neural information processing systems. [S.l.: s.n.], 2014: 1799-1807.

[10] Mikolov T, Deoras A, Povey D, et al. Strategies for training large scale neural network language models[C]//2011 IEEE Workshop on Automatic Speech Recognition & Understanding. [S.l.]: IEEE, 2011: 196-201.

[11] Hinton G, Deng L, Yu D, et al. Deep neural networks for acoustic modeling in speech recognition [J]. IEEE Signal processing magazine, 2012, 29.

[12] Ma J, Sheridan R P, Liaw A, et al. Deep neural nets as a method for quantitative structure–activity relationships[J]. Journal of chemical information and modeling, 2015, 55(2): 263-274.

[13] Ciodaro T, Deva D, De Seixas J, et al. Online particle detection with neural networks based on topological calorimetry information[C]//Journal of physics: conference series: volume 368. [S.l.]: IOP Publishing, 2012: 012030.

[14] Helmstaedter M, Briggman K L, Turaga S C, et al. Connectomic reconstruction of the inner plexiform layer in the mouse retina[J]. Nature, 2013, 500(7461): 168.

[15] Lu C, Tang X. Surpassing human-level face verification performance on lfw with gaussianface [C]//Twenty-Ninth AAAI Conference on Artificial Intelligence. [S.l.: s.n.], 2015.

[16] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of go without human knowledge [J]. Nature, 2017, 550(7676): 354.

[17] Esteva A, Kuprel B, Novoa R A, et al. Dermatologist-level classification of skin cancer with deep neural networks[J]. Nature, 2017, 542(7639): 115.

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

[19] Rumelhart D E, Hinton G E, Williams R J, et al. Learning representations by back-propagating errors[J]. Cognitive modeling, 1988, 5(3): 1.

[20] Mack C A. Fifty years of moore's law[J]. IEEE Transactions on semiconductor manufacturing, 2011, 24(2): 202-207.

[21] Deng J, Dong W, Socher R, et al. Imagenet: A large-scale hierarchical image database[Z]. [S.l.]: Ieee, 2009.

[22] Sun C, Shrivastava A, Singh S, et al. Revisiting unreasonable effectiveness of data in deep learning era[C]//Proceedings of the IEEE international conference on computer vision. [S.l.: s.n.], 2017: 843-852.

[23] Alain G, Bengio Y. Understanding intermediate layers using linear classifier probes[J]. arXiv preprint arXiv:1610.01644, 2016.

[24] Shwartz-Ziv R, Tishby N. Opening the black box of deep neural networks via information[J]. arXiv preprint arXiv:1703.00810, 2017.

[25] Joulin A, Grave E, Bojanowski P, et al. Bag of tricks for efficient text classification[J]. arXiv preprint arXiv:1607.01759, 2016.

[26] Zoph B, Le Q V. Neural architecture search with reinforcement learning[J]. arXiv preprint arXiv:1611.01578, 2016.

[27] Thornton J E. The cdc 6600 project[J]. Annals of the History of Computing, 1980, 2(4): 338-348.

[28] Russell R M. The cray-1 computer system[J]. Communications of the ACM, 1978, 21(1): 63-72.

[29] Amdahl G M, Blaauw G A, Brooks F. Architecture of the ibm system/360[J]. IBM Journal of Research and Development, 1964, 8(2): 87-101.

[30] Stephan M, Docter J. Juqueen: Ibm blue gene/q® supercomputer system at the jülich supercomputing centre[J]. Journal of large-scale research facilities JLSRF, 2015, 1: 1.

[31] Villa O, Johnson D R, O'Connor M, et al. Scaling the power wall: a path to exascale[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE Press, 2014: 830-841.

[32] The 52nd edition of the top500 list (november 2018).[EB/OL]. 2018. https://www.top500.org/lists/2018/11/.

[33] He L, An H, Yang C, et al. Peps++: Towards extreme-scale simulations of strongly correlated quantum many-particle models on sunway taihulight[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(12): 2838-2848.

[34] Johnsen P, Straka M, Shapiro M, et al. Petascale wrf simulation of hurricane sandy: Deployment of ncsa's cray xe6 blue waters[C]//SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE, 2013: 1-7.

[35] Yang C, Xue W, Fu H, et al. 10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE Press, 2016: 6.

[36] Ao Y, Yang C, Wang X, et al. 26 pflops stencil computations for atmospheric modeling on sunway taihulight[C]//2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [S.l.]: IEEE, 2017: 535-544.

[37] Fu H, He C, Chen B, et al. 18.9-pflops nonlinear earthquake simulation on sunway taihulight: enabling depiction of 18-hz and 8-meter scenarios[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: ACM, 2017: 2.

[38] Qiao F, Zhao W, Yin X, et al. A highly effective global surface wave numerical simulation with ultra-high resolution[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE Press, 2016: 5.

[39] Duan X, Xu K, Chan Y, et al. S-aligner: Ultrascalable read mapping on sunway taihu light[C]// 2017 IEEE International Conference on Cluster Computing (CLUSTER). [S.l.]: IEEE, 2017: 36-46.

[40] Groen D, Zwart S P, Ishiyama T, et al. High-performance gravitational n-body simulations on a planet-wide-distributed supercomputer[J]. Computational Science & Discovery, 2011, 4(1): 015001.

[41] Mathuriya A, Bard D, Mendygral P, et al. Cosmoflow: using deep learning to learn the universe at scale[C]//SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE, 2018: 819-829.

[42] Li L, Fang J, Jiang J, et al. Sw-aes: Accelerating aes algorithm on the sunway taihulight [C]//2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC). [S.l.]: IEEE, 2017: 1204-1211.

[43] Kurth T, Zhang J, Satish N, et al. Deep learning at 15pf: supervised and semi-supervised classification for scientific data[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: ACM, 2017: 7.

[44] Kurth T, Treichler S, Romero J, et al. Exascale deep learning for climate analytics[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. [S.l.]: IEEE Press, 2018: 51.

[45] Lu Y, Qian D, Fu H, et al. Will supercomputers be super-data and super-ai machines?[J]. Communications of the ACM, 2018, 61(11): 82-87.

[46] 国务院关于印发新一代人工智能发展规划的通知[EB/OL]. 2017. http://www.gov.cn/zhengce/content/2017-07/20/content_5211996.htm.

[47] Davis Z S. Artificial intelligence on the battlefield[Z]. [S.l.: s.n.], 2019.

[48] Li L. China’s manufacturing locus in 2025: With a comparison of “made-in-china 2025”and “industry 4.0”[J]. Technological Forecasting and Social Change, 2018, 135: 66-74.

[49] Ding J. Deciphering china’s ai dream[J]. Future of Humanity Institute and University of Oxford. URL: https://www. fhi. ox. ac. uk/wp-content/uploads/Deciphering_Chinas_AI-Dream. pdf, 2018.

[50] Xu Z, Lin J, Matsuoka S. Benchmarking sw26010 many-core processor[C]//2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). [S.l.]: IEEE, 2017: 743-752.

[51] Fu H, Liao J, Ding N, et al. Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: ACM, 2017: 1.

[52] Fu H, Liao J, Xue W, et al. Refactoring and optimizing the community atmosphere model (cam) on the sunway taihulight supercomputer[C]//SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: IEEE, 2016: 969-980.

[53] 姚敏. 下一代超算或比现世界冠军快八倍[J]. 计算机与网络, 2017, 43(14): 14-14.

[54] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2016: 770-778.

[55] Huang G, Liu Z, Van Der Maaten L, et al. Densely connected convolutional networks[C]// Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2017: 4700-4708.

[56] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2016: 2818-2826.

[57] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2015: 1-9.

[58] Pascanu R, Mikolov T, Bengio Y. Understanding the exploding gradient problem[J]. CoRR, abs/1211.5063, 2012, 2.

[59] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

[60] Cho K, Van Merriënboer B, Bahdanau D, et al. On the properties of neural machine translation: Encoder-decoder approaches[J]. arXiv preprint arXiv:1409.1259, 2014.

[61] Vincent P, Larochelle H, Lajoie I, et al. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion[J]. Journal of machine learning research, 2010, 11(Dec): 3371-3408.

[62] Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[C]//Advances in neural information processing systems. [S.l.: s.n.], 2014: 2672-2680.

[63] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518(7540): 529.

[64] Mnih V, Badia A P, Mirza M, et al. Asynchronous methods for deep reinforcement learning [C]//International conference on machine learning. [S.l.: s.n.], 2016: 1928-1937.

[65] Qian N. On the momentum term in gradient descent learning algorithms[J]. Neural networks, 1999, 12(1): 145-151.

[66] Nesterov Y E. A method for solving the convex programming problem with convergence rate o (1/kˆ 2)[C]//Dokl. akad. nauk Sssr: volume 269. [S.l.: s.n.], 1983: 543-547.

[67] Duchi J, Hazan E, Singer Y. Adaptive subgradient methods for online learning and stochastic optimization[J]. Journal of Machine Learning Research, 2011, 12(Jul): 2121-2159.

[68] Hinton G, Srivastava N, Swersky K. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent[J]. Cited on, 2012, 14.

[69] Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[70] Werbos P J, et al. Backpropagation through time: what it does and how to do it[J]. Proceedings of the IEEE, 1990, 78(10): 1550-1560.

[71] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing[C]//Tenth International Workshop on Frontiers in Handwriting Recognition. [S.l.]: Suvisoft, 2006.

[72] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. [S.l.]: ACM, 2014: 675-678.

[73] Chetlur S, Woolley C, Vandermersch P, et al. cudnn: Efficient primitives for deep learning[J]. arXiv preprint arXiv:1410.0759, 2014.

[74] Lavin A. maxdnn: an efficient convolution kernel for deep learning with maxwell gpus[J]. arXiv preprint arXiv:1501.06633, 2015.

[75] Lavin A, Gray S. Fast algorithms for convolutional neural networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], 2016: 4013-4021.

[76] Vasilache N, Johnson J, Mathieu M, et al. Fast convolutional nets with fbfft: A gpu performance evaluation[J]. arXiv preprint arXiv:1412.7580, 2014.

[77] Zlateski A, Lee K, Seung H S. Znni-maximizing the inference throughput of 3d convolutional networks on multi-core cpus and gpus[J]. arXiv preprint arXiv:1606.05688, 2016.

[78] NVIDIA. Cublas library documentation[C]//https://docs.nvidia.com/cuda/cublas/index.html. [S.l.: s.n.], 2019.

[79] Intel. Mkl library documentation[C]//https://software.intel.com/en-us/mkl. [S.l.: s.n.], 2019.

[80] Appleyard J, Kocisky T, Blunsom P. Optimizing performance of recurrent neural networks on gpus[J]. arXiv preprint arXiv:1604.01946, 2016.

[81] Haidar A, Abdelfattah A, Zounon M, et al. A guide for achieving high performance with very small matrices on gpu: A case study of batched lu and cholesky factorizations[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(5): 973-984.

[82] NVIDIA. cudnn developer guide, cudnnv7.5.0[C]// https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html. [S.l.: s.n.], 2019.

[83] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. ACM SIGPLAN Notices, 2013, 48(6): 519-530.

[84] Tensorflow xla overview.[EB/OL]. 2017. https://www.tensorflow.org/performance/xla.

[85] Vasilache N, Zinenko O, Theodoridis T, et al. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions[J]. arXiv preprint arXiv:1802.04730, 2018.

[86] Truong L, Barik R, Totoni E, et al. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks[J]. ACM SIGPLAN Notices, 2016, 51(6): 209-223.

[87] Chen T, Moreau T, Jiang Z, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). [S.l.: s.n.], 2018: 578-594.

[88] Bergstra J, Breuleux O, Bastien F, et al. Theano: a cpu and gpu math expression compiler [C]//Proceedings of the Python for scientific computing conference (SciPy): volume 4. [S.l.]: Austin, TX, 2010.

[89] Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning[C]// 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). [S.l.: s.n.], 2016: 265-283.

[90] Ketkar N. Introduction to pytorch[M]//Deep learning with python. [S.l.]: Springer, 2017: 195-208

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

[92] Seide F, Agarwal A. Cntk: Microsoft's open-source deep-learning toolkit[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. [S.l.]: ACM, 2016: 2135-2135.

[93] Tokui S, Oono K, Hido S, et al. Chainer: a next-generation open source framework for deep learning[C]//Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS): volume 5. [S.l.: s.n.], 2015: 1-6.

[94] Neubig G, Dyer C, Goldberg Y, et al. Dynet: The dynamic neural network toolkit[J]. arXiv preprint arXiv:1701.03980, 2017.

[95] Liang X, Shen X, Feng J, et al. Semantic object parsing with graph lstm[C]//European Conference on Computer Vision. [S.l.]: Springer, 2016: 125-143.

[96] Tai K S, Socher R, Manning C D. Improved semantic representations from tree-structured long short-term memory networks[J]. arXiv preprint arXiv:1503.00075, 2015.

[97] Looks M, Herreshoff M, Hutchins D, et al. Deep learning with dynamic computation graphs[J]. arXiv preprint arXiv:1702.02181, 2017.

[98] Chen T, Xu B, Zhang C, et al. Training deep nets with sublinear memory cost[J]. arXiv preprint arXiv:1604.06174, 2016.

[99] Gruslys A, Munos R, Danihelka I, et al. Memory-efficient backpropagation through time[C]// Advances in Neural Information Processing Systems. [S.l.: s.n.], 2016: 4125-4133.

[100] Rhu M, Gimelshein N, Clemons J, et al. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design[C]//The 49th Annual IEEE/ACM International Symposium on Microarchitecture. [S.l.]: IEEE Press, 2016: 18.

[101] Wang L, Ye J, Zhao Y, et al. Superneurons: Dynamic gpu memory management for training deep neural networks[C]//ACM SIGPLAN Notices: volume 53. [S.l.]: ACM, 2018: 41-53.

[102] Zhang X, Mckenna M, Mesirov J P, et al. An efficient implementation of the back-propagation algorithm on the connection machine cm-2[C]//Advances in neural information processing systems. [S.l.: s.n.], 1990: 801-809.

[103] Farber P, Asanovic K. Parallel neural network training on multi-spert[C]//Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing. [S.l.]: IEEE, 1997: 659-666.

[104] Raina R, Madhavan A, Ng A Y. Large-scale deep unsupervised learning using graphics processors[C]//Proceedings of the 26th annual international conference on machine learning. [S.l.]: ACM, 2009: 873-880.

[105] Zhang H, Zheng Z, Xu S, et al. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters[C]//2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). [S.l.: s.n.], 2017: 181-193.

[106] Jia X, Song S, He W, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes[J]. arXiv preprint arXiv:1807.11205, 2018.

[107] Seide F, Fu H, Droppo J, et al. On parallelizability of stochastic gradient descent for speech dnns[C]//2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). [S.l.]: IEEE, 2014: 235-239.

[108] Keskar N S, Mudigere D, Nocedal J, et al. On large-batch training for deep learning: Generalization gap and sharp minima[J]. arXiv preprint arXiv:1609.04836, 2016.

[109] You Y, Zhang Z, Hsieh C J, et al. Imagenet training in minutes[C]//Proceedings of the 47th International Conference on Parallel Processing. [S.l.]: ACM, 2018: 1.

[110] Goyal P, Dollár P, Girshick R, et al. Accurate, large minibatch sgd: Training imagenet in 1 hour [J]. arXiv preprint arXiv:1706.02677, 2017.

[111] Smith S L, Kindermans P J, Ying C, et al. Don't decay the learning rate, increase the batch size [J]. arXiv preprint arXiv:1711.00489, 2017.

[112] Recht B, Re C, Wright S, et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent[C]//Advances in neural information processing systems. [S.l.: s.n.], 2011: 693-701.

[113] Dean J, Corrado G, Monga R, et al. Large scale distributed deep networks[C]//Advances in neural information processing systems. [S.l.: s.n.], 2012: 1223-1231.

[114] Noel C, Osindero S. Dogwild!-distributed hogwild for cpu & gpu[C]//NIPS Workshop on Distributed Machine Learning and Matrix Computations. [S.l.: s.n.], 2014.

[115] Agarwal A, Duchi J C. Distributed delayed stochastic optimization[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2011: 873-881.

[116] Dekel O, Gilad-Bachrach R, Shamir O, et al. Optimal distributed online prediction using mini-batches[J]. Journal of Machine Learning Research, 2012, 13(Jan): 165-202.

[117] Lian X, Huang Y, Li Y, et al. Asynchronous parallel stochastic gradient for nonconvex optimization[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2015: 2737-2745.

[118] Ho Q, Cipar J, Cui H, et al. More effective distributed ml via a stale synchronous parallel parameter server[C]//Advances in neural information processing systems. [S.l.: s.n.], 2013: 1223-1231.

[119] Polyak B T, Juditsky A B. Acceleration of stochastic approximation by averaging[J]. SIAM Journal on Control and Optimization, 1992, 30(4): 838-855.

[120] Zhang S, Choromanska A E, LeCun Y. Deep learning with elastic averaging sgd[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2015: 685-693.

[121] Povey D, Zhang X, Khudanpur S. Parallel training of deep neural networks with natural gradient and parameter averaging[J]. arXiv preprint arXiv:1410.7455, 2014.

[122] Lee S, Agrawal A, Balaprakash P, et al. Communication-efficient parallelization strategy for deep convolutional neural network training[C]//2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC). [S.l.]: IEEE, 2018: 47-56.

[123] Strom N. Scalable distributed dnn training using commodity gpu cloud computing[C]//Sixteenth Annual Conference of the International Speech Communication Association. [S.l.: s.n.], 2015.

[124] Aji A F, Heafield K. Sparse communication for distributed gradient descent[J]. arXiv preprint arXiv:1704.05021, 2017.

[125] Chen C Y, Choi J, Brand D, et al. Adacomp: Adaptive residual gradient compression for data-parallel distributed training[J]. arXiv preprint arXiv:1712.02679, 2017.

[126] Lin Y, Han S, Mao H, et al. Deep gradient compression: Reducing the communication bandwidth for distributed training[J]. arXiv preprint arXiv:1712.01887, 2017.

[127] Sattler F, Wiedemann S, Müller K R, et al. Sparse binary compression: Towards distributed deep learning with minimal communication[J]. arXiv preprint arXiv:1805.08768, 2018.

[128] Li M, Andersen D G, Park J W, et al. Scaling distributed machine learning with the parameter server.[C]//OSDI: volume 14. [S.l.: s.n.], 2014: 583-598.

[129] Chilimbi T, Suzue Y, Apacible J, et al. Project adam: Building an efficient and scalable deep learning training system[C]//11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). [S.l.: s.n.], 2014: 571-582.

[130] Gropp W D, Gropp W, Lusk E, et al. Using mpi: portable parallel programming with the message-passing interface: volume 1[M]. [S.l.]: MIT press, 1999

[131] Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in mpich[J]. The International Journal of High Performance Computing Applications, 2005, 19 (1): 49-66.

[132] Chan E, Heimlich M, Purkayastha A, et al. Collective communication: theory, practice, and experience[J]. Concurrency and Computation: Practice and Experience, 2007, 19(13): 1749-1783.

[133] Hoefler T, Moor D. Energy, memory, and runtime tradeoffs for implementing collective communication operations[J]. Supercomputing frontiers and innovations, 2014, 1(2): 58-75.

[134] Nvidia collective communications library (nccl).[EB/OL]. 2019. https://developer.nvidia.com/nccl.

[135] Baidu-allreduce.[EB/OL]. 2019. https://github.com/baidu-research/baidu-allreduce.

[136] Gloo: Collective communications library with various primitives for multi-machine training. [EB/OL]. 2019. https://github.com/facebookincubator/gloo.

[137] Sergeev A, Del Balso M. Horovod: fast and easy distributed deep learning in tensorflow[J]. arXiv preprint arXiv:1802.05799, 2018.

[138] Intel(r) machine learning scaling library for linux* os.[EB/OL]. 2019. https://github.com/intel/MLSL.

[139] Lee S, Kim J K, Zheng X, et al. On model parallelization and scheduling strategies for distributed machine learning[C]//Advances in neural information processing systems. [S.l.: s.n.], 2014: 2834-2842.

[140] Krizhevsky A. One weird trick for parallelizing convolutional neural networks[J]. arXiv preprint arXiv:1404.5997, 2014.

[141] Gholami A, Azad A, Jin P, et al. Integrated model, batch and domain parallelism in training neural networks[J]. arXiv preprint arXiv:1712.04432, 2017.

[142] Petrowski A, Dreyfus G, Girault C. Performance analysis of a pipelined backpropagation parallel algorithm[J]. IEEE Transactions on Neural Networks, 1993, 4(6): 970-981.

[143] Harlap A, Narayanan D, Phanishayee A, et al. Pipedream: Fast and efficient pipeline parallel dnn training[J]. arXiv preprint arXiv:1806.03377, 2018.

[144] Chen C C, Yang C L, Cheng H Y. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform[J]. arXiv preprint arXiv:1809.02839, 2018.

[145] Huang Y, Cheng Y, Chen D, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism[J]. arXiv preprint arXiv:1811.06965, 2018.

[146] Real E, Aggarwal A, Huang Y, et al. Regularized evolution for image classifier architecture search[J]. arXiv preprint arXiv:1802.01548, 2018.

[147] Coates A, Huval B, Wang T, et al. Deep learning with cots hpc systems[C]//International conference on machine learning. [S.l.: s.n.], 2013: 1337-1345.

[148] Iandola F N, Moskewicz M W, Ashraf K, et al. Firecaffe: near-linear acceleration of deep neural network training on compute clusters[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], 2016: 2592-2600.

[149] Awan A A, Hamidouche K, Hashmi J M, et al. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters[C]//Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. [S.l.]: ACM, 2017: 193-205.

[150] You Y, Buluç A, Demmel J. Scaling deep learning on gpu and knights landing clusters[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.]: ACM, 2017: 9.

[151] 神威 太湖之光并行程序设计与优化[EB/OL]. 2017. http://www.nsccwx.cn/uploads/down/4653_1489643053.pdf.

[152] Lin J, Xu Z, Cai L, et al. Evaluating the sw26010 many-core processor with a micro-benchmark suite for performance optimizations[J]. Parallel Computing, 2018, 77: 128-143.

[153] Flynn M J. Some computer organizations and their effectiveness[J]. IEEE transactions on computers, 1972, 100(9): 948-960.

[154] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures[J]. Communications of the ACM, 2009, 52(4): 65-76.

[155] Xu S, Xu Y, Xue W, et al. Taming the" monster": Overcoming program optimization challenges on sw26010 through precise performance modeling[C]//2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [S.l.]: IEEE, 2018: 763-773.

[156] Hassaballah M, Omran S, Mahdy Y B. A review of simd multimedia extensions and their usage in scientific and engineering applications[J]. The Computer Journal, 2008, 51(6): 630-649.

[157] Markidis S, Der Chien S W, Laure E, et al. Nvidia tensor core programmability, performance & precision[J]. arXiv preprint arXiv:1803.04014, 2018.

[158] Deilmann M, et al. A guide to vectorization with intel c++ compilers[J]. Intel Corporation, April, 2012.

[159] Oster B. Advanced cuda tutorial[J]. NVIDIA Corporation, 2008.

[160] Eggers S J, Emer J S, Levy H M, et al. Simultaneous multithreading: A platform for next-generation processors[J]. IEEE micro, 1997(5): 12-19.

[161] Chen T F, Baer J L. Effective hardware-based data prefetching for high-performance processors [J]. IEEE transactions on computers, 1995, 44(5): 609-623.

[162] Jouppi N P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers[C]//ACM SIGARCH Computer Architecture News: volume 18. [S.l.]: ACM, 1990: 364-373.

[163] Volkov V. Understanding latency hiding on gpus[D]. [S.l.]: UC Berkeley, 2016.

[164] Jiang L, Yang C, Ao Y, et al. Towards highly efficient dgemm on the emerging sw26010 many-core processor[C]//Parallel Processing (ICPP), 2017 46th International Conference on. [S.l.]: IEEE, 2017: 422-431.

[165] Goto K, Geijn R A. Anatomy of high-performance matrix multiplication[J]. ACM Transactions on Mathematical Software (TOMS), 2008, 34(3): 12.

[166] Masliah I, Abdelfattah A, Haidar A, et al. Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices[J]. Parallel Computing, 2019, 81: 1-21.

[167] Winograd S. Arithmetic complexity of computations: volume 33[M]. [S.l.]: Siam, 1980

[168] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition [J]. arXiv preprint arXiv:1409.1556, 2014.

[169] Redmon J, Divvala S, Girshick R, et al. You only look once: Unified, real-time object detection [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2016: 779-788.

[170] Rotem N, Fix J, Abdulrasool S, et al. Glow: Graph lowering compiler techniques for neural networks[J]. arXiv preprint arXiv:1805.00907, 2018.

[171] Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation[C]//Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. [S.l.]: IEEE Computer Society, 2004: 75.

[172] Chen T, Zheng L, Yan E, et al. Learning to optimize tensor programs[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2018: 3393-3404.

[173] Leiserson C E. Fat-trees: universal networks for hardware-efficient supercomputing[J]. IEEE transactions on Computers, 1985, 100(10): 892-901.

[174] Rabenseifner R. Optimization of collective reduction operations[C]//International Conference on Computational Science. [S.l.]: Springer, 2004: 1-9.

[175] Seide F, Fu H, Droppo J, et al. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns[C]//Fifteenth Annual Conference of the International Speech Communication Association. [S.l.: s.n.], 2014.

[176] Alistarh D, Grubic D, Liu J, et al. Communication-efficient stochastic gradient descent, with applications to neural networks[Z]. [S.l.]: Curran Associates, Inc., 2017.

[177] Wen W, Xu C, Yan F, et al. Terngrad: Ternary gradients to reduce communication in distributed deep learning[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2017: 1508-1518.

[178] Hoare C A. Find (algorithm 65)[J]. Communications of the ACM, 1961, 4(7): 321-322.

[179] Alabi T, Blanchard J D, Gordon B, et al. Fast k-selection algorithms for graphics processing units[J]. Journal of Experimental Algorithmics (JEA), 2012, 17: 4-2.

[180] Sengupta S, Harris M, Zhang Y, et al. Scan primitives for gpu computing[C]//Graphics hardware: volume 2007. [S.l.: s.n.], 2007: 97-106.

[181] Sengupta S, Lefohn A E, Owens J D. A work-efficient step-efficient prefix sum algorithm[C]//Workshop on edge computing using new commodity architectures. [S.l.: s.n.], 2006: 26-27.

[182] Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images[R]. [S.l.]: Citeseer, 2009.

[183] Marcus M P, Marcinkiewicz M A, Santorini B. Building a large annotated corpus of english: The penn treebank[J]. Computational linguistics, 1993, 19(2): 313-330.

[184] Merity S, Xiong C, Bradbury J, et al. Pointer sentinel mixture models[J]. arXiv preprint arXiv:1609.07843, 2016.

[185] Press O, Wolf L. Using the output embedding to improve language models[J]. arXiv preprint arXiv:1608.05859, 2016.

致　谢

在本文即将付梓之际，我要由衷感谢科研道路上的两位领路人—杨广文教授和付昊桓教授。感谢杨教授长期以来对我的关怀与支持。杨教授对学术的严格要求，对生活的充沛热情，对祖国超算事业的执着追求都让我受益匪浅。感谢付教授对我的培养与指导，让我在科研方法、论文写作、演讲沟通、接人待物等方面受益良多。博士期间忝列门墙，得到二位教授的言传身教，实乃我之幸也。

本文工作的顺利完成承蒙许多人的协助。感谢江南计算所的袁欣辉老师启发了我在申威上进行矩阵乘法优化的思路，这是本工作得以生根发芽的种子。感谢赵文来师兄和陈炳炜师弟帮助我完成了繁重的矩阵乘法汇编指令重排工作。感谢赵祎、李连登、游心同学帮助我完成了 swCaffe 部分算子的优化工作。感谢高伟同学帮助我完成了 swAutoDNN 自动优化工具的具体实现。

在我的博士道路上，需要感谢的人还有很多。感谢加州大学洛杉矶分校的 Cho-Jui Heish 老师对我在美国访学期间的细心指导和无私帮助。感谢清华大学薛巍老师、江南计算所的刘鑫老师长期以来对我的关怀与鼓励。感谢王英侨、吕子鉦两位师兄，在我独立承担科研项目时传授给我的宝贵经验。感谢李唯嘉同学给予我参与地学系遥感图像处理相关研究的机会。感谢何聪辉、徐世真、季颖生、陈宇澍、甘霖、廖俊峰等师兄给我提供的宝贵科研建议。感谢三位室友王吴凡、黄逍和涂铜壁给带来了生活的欢乐。感谢伯克利的尤洋师兄、戴维斯的 Joe 和 Toni 夫妇对我在美国生活中的照顾。感谢清华大学、加州大学戴维斯分校的同组同学，国家超算无锡中心的共事同事，纸短情长，恕篇幅有限，不能在此将各位名字一一列举。

“关山难越，谁悲失路之人；萍水相逢，尽是他乡之客”。五年来，我曾辗转过许多地方，从美丽的清华校园到静谧的加州乡村，从炎热的雪浪山麓到湿润的太湖之滨。跨越过千山万水，经历了离合悲欢，更加感激求学期间经历的所有苦难和磨砺。正如一本描述博士生活的书《The Ph.D. Grind》所隐喻：唯有碾碎自己，才能获得新生。

感谢国家超级计算无锡中心、国家留学基金委、国家自然科学基金、邓峰基金、斯伦贝谢奖学金、宜信奖学金、董氏东方奖学金对本人的资助。感谢我的女朋友在我遇到困难时给予的鼓励和支持。本文最早的二位读者是我最亲爱的父母，感谢他们对我的无私奉献，让我在一个幸福的家庭成长，并鼓励我勇敢走上学术的道路。

最后，感谢答辩委员会专家和匿名评审专家的宝贵时间和建议。

声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：____________ 日 期：____________

个人简历、在学期间发表的学术论文与研究成果

个人简历

1992 年 4 月 19 日出生于辽宁省沈阳市。

2010 年 9 月考入北京邮电大学计算机系，2014 年 7 月本科毕业并获得工学学士学位。

2014 年 9 月免试进入清华大学计算机系攻读博士学位至今。

2017 年 8 月至 2018 年 8 月以访问学者身份在美国加州大学戴维斯分校学习。

发表的学术论文

[1] **Fang, Jiarui**, and Fu, Haohuan and Yang, Guangwen, and Cho-Jui Heish, RedSync : Reducing Synchronization Traffic for Distributed Deep Learning. Journal of Parallel and Distributed Computing (JPDC) (CCF 推荐 B 类期刊接收阶段).

[2] **Fang, Jiarui** and Fu, Haohuan and Zhao, Wenlai and Chen, Bingwei and Zheng, Weijie and Yang, Guangwen. swDNN: A Library for Accelerating Deep Learning Applications on Sunway TaihuLight Supercomputer, 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS 17’), Orlando, USA, 2017 (CCF 推荐 B 类会议发表)

[3] **Fang, Jiarui** and Fu, Haohuan and Yang, Guangwen: Cache-friendly Design for Complex Spatially-variable Coefficient Stencils on Many-core Architectures. IEEE 23rd International Conference on High Performance Computing, Data, and Analytics (HiPC 16’),p222-p231, Hyderabad, India, 2016.(CCF 推荐 C 类会议发表)

[4] **Fang, Jiarui** and Fu, Haohuan and Zhang, He and Wu, Wei and Dai, Nanxun and Gan, Lin and Yang, Guangwen. Optimizing Complex Spatially-Variant Coefficient Stencils for Seismic Modeling on GPU. IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS 15’), p641-p648 Melbourne, Australia, 2015.(CCF 推荐 C 类会议发表)

[5] **Fang, Jiarui**, and Fu, Haohuan and Yang, Guangwen. GPU-based explicit time evolution method. The 84th Society of Exploration Geophysicists Technical Program Expanded Abstracts (SEG 15’), p3549-p3553, New Orleans, USA, 2015

[6] Wei, Gao* and **Fang, Jiarui***, and Zhao, Wenlai and Jinzhe, Yang and Wang, Long and Lin, Gan and Fu, Haohuan and Yang, Guangwen. swATOP: Automatically

Optimizing Deep Learning Operators on SW26010 Many-core Processor, 48th International Conference on Parallel Processing (ICPP 19') ,August 5-8, Kyoto, Japan, 2019.(共同一作，CCF 推荐 B 类会议接收阶段)

[7] Li, Liandeng* and **Fang, Jiarui*** and Fu, Haohuan and Jiang, Jinlei and Zhao, Wenlai and He, Conghui and You, Xin and Yang, Guangwen. swCaffe: a Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight, IEEE Cluster (Cluster 18'), Belfast, UK, 2018.(共同一作，CCF 推荐 B 类会议发表)

[8] Li, Weijia and He, Conghui and **Fang, Jiarui** and Zheng, Juepeng and Fu, Haohuan and Yu, Le. Semantic Segmentation-Based Building Footprint Extraction Using Very High-Resolution Satellite Images and Multi-Source GIS Data[J]. Remote Sensing, 2019, 11(4): 403. (SCI 检索)

[9] Huang, Xiao and Yu, Chaoqing and **Fang, Jiarui** and Huang, Guorui and Ni, Shaoqiang and Hall, Jim and Zorn, Conrad and Huang, Xiaomeng and Zhang, WenyuanA dynamic agricultural prediction system for large-scale drought assessment on the Sunway TaihuLight supercomputer, Computers and Electronics in Agriculture Volume 154, November 2018, Pages 400-410. (SCI 检索)

[10] Li, Weijia and He, Conghui and **Fang, Jiarui** and Fu, Haohuan, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, Salt Lake City, UT, USA. 2018: 18-22. (SCI 检索)

[11] Zhao, Wenlai and Fu, Haohuan and **Fang, Jiarui** and Zheng, Weijie and Gan, Lin and Yang, Guangwen Optimizing Convolutional Neural Networks on Sunway TaihuLight Supercomputer, ACM Transactions on Architecture and Code Optimization (TACO), 2018, 15(1): 13. (CCF B 类期刊发表)

[12] Li, Liandeng and **Fang, Jiarui** and Jiang, Jinlei and Gan, Lin and Zheng, Weijie and Fu, Haohuan and Yang, Guangwen. SW-AES: Accelerating AES Algorithm on the Sunway TaihuLight, 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC). IEEE, 2017: 1204-1211. (CCF C 类会议发表)

[13] Li, Weijia and Fu, Haohuan and You, Yang and Yu, Le and **Fang, Jiarui**. Parallel Multi-class Support Vector Machine for Remote Sensing Data Classification on Multi-Core and Many-Core Architectures, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 10.1109/JSTARS.2017.2713126 (SCI 检索)

[14] Fu, Haohuan and Wang, Yingqiao and Um, Evan Schankee and **Fang, Jiarui** and Wei, Tengpeng and Huang, Xiaomeng and Yang, Guangwen. A parallel finite-element time-domain method for transient electromagnetic simulation. Geophysics, 80(4), E213-E224, 2015. (SCI 检索)