中图分类号：TP391.4

盲审编号：10406_085405_2016085400101_LW

南昌航空大学

硕士学位论文

题　目

基于张量代数的卷积神经网络性能优化

学科、专业　软件工程

专业代码　085405

学位类别　专业学位硕士

2023 年 4 月 10 日

学校代码：10406 学号：2016085400101
分类号：TP391.4

南昌航空大学
硕 士 学 位 论 文
(专业学位研究生)

基于张量代数的卷积神经网络性能优化

硕 士 研 究 生：卢帅
导 师：储珺教授
申请学位级别：硕士
学 科 、 专 业：软件工程
所 在 单 位：软件学院
答 辩 日 期：2023 年 6 月
授予学位单位：南昌航空大学

Performance optimization of convolutional neural networks based on tensor algebra

A Thesis

Submitted for the Degree of Master

On Software Engineering

by **Shuai Lu**

Under the Supervision of

Jun Chu

School of Software

Nanchang Hangkong University, Nanchang, China

June, 2023

摘要

卷积神经网络是深度学习中最重要的网络模型之一，在图像处理、语音识别、自然语言处理等领域具有广泛应用价值。在常见的卷积神经网络中，卷积操作占网络模型总推理执行时间的 90%以上，所以卷积操作一直是卷积神经网络性能优化的重点。常见的深度学习框架中实现卷积操作通常使用直接卷积算法、基于矩阵乘法的卷积算法、基于快速傅里叶变换的卷积算法等。然而，这些传统的卷积算法在处理高维数据、不同尺寸数据、不同形式数据时，往往会导致冗余计算、内存浪费和灵活性不足等问题。这些问题限制了卷积神经网络的性能，也影响了其在更多场景下的应用潜力。针对上述问题，本文从以下两方面入手展开研究：（1）基于张量代数改进卷积算法；（2）针对 CPU 和 GPU 平台特性实现高性能的张量卷积算法。主要工作如下：

（1）提出了一种新的基于张量代数的卷积算法，即基于窗口序的张量卷积算法（image to window, im2win），由内存高效的 im2win 变换和后续的张量乘法两部分组成。本文提出的 im2win 变换将输入图像按照点乘窗口运算访问顺序重新排列数据，相比于传统的 image to column（im2col）变换，能够显著减少内存开销，并保证点乘窗口内存访问连续性和数据重用性。通过 im2win 变换得到的 im2win 张量是一个四维张量，相比于传统的 im2col 变换得到的二维矩阵，能够保留更多高维隐含信息。后续使用张量乘法将 im2win 张量和卷积核张量进行点乘运算以实现卷积操作，可以避免重复计算和数据移动，降低计算复杂度和内存消耗，并使得卷积实现更加灵活适应不同硬件平台。

（2）在 CPU 平台上实现了高性能的基于 im2win 变换的张量卷积算法，并根据 CPU 平台特性采用了一系列优化技术，以提高运行效率。首先建立了一个分析优化空间与策略选择的数学分析模型，并结合 CPU 架构特性和张量卷积计算过程设计完善该模型；然后利用 SIMD 向量指令集来加速向量乘法，并通过多线程并行来充分利用 CPU 资源；最后采用循环展开等技术来减少分支判断和函数调用开销。实验表明，在计算性能方面，本文提出的基于 im2win 变换的张量卷积算法相比于基于 im2col 变换的卷积算法可以实现 2.2 倍到 5.8 倍的性能提升，相比于直接卷积算法可以实现 3.8 倍到 7.1 倍的性能提升；在内存占用方面，本文提出的张量卷积算法相比于基于 im2col 变换的卷积算法平均降低了 41.6%的内存占用。

（3）在 GPU 平台上实现了高性能的基于 im2win 变换的张量卷积算法，并根据 GPU 平台特性采用了一系列优化技术来进一步提高运行效率。首先根据

GPU 的 CUDA 架构模型，对张量卷积算法的索引进行重构，使得该算法可以更好地映射在 GPU 的架构上；然后利用 Tiling 策略来减少全局内存访问，并利用共享内存与寄存器来缓存局部数据；最后采用微内核设计、向量化加载/存储、双缓冲与数据预取等技术来提高计算吞吐率。实验表明，在计算性能方面，本文提出的张量卷积算法相比于基于 im2col 变换的卷积算法平均可以实现约为 3.5 倍的性能提升，相比于 cuDNN 库的卷积算法最高可以实现约为 1.8 倍的性能提升，平均性能提升为 1.1 倍；在内存占用方面，本文提出的张量卷积算法相比于基于 im2col 变换的卷积算法平均降低了 23.1% 的内存占用，相比于 cuDNN 库的卷积算法平均降低了 23.1% 的内存占用。
关键词：卷积算法，张量代数，卷积神经网络，并行计算，高性能计算

Abstract

Convolutional neural networks are one of the most important network models in deep learning, and have wide application value in fields such as image processing, speech recognition, and natural language processing. In common convolutional neural networks, the convolution operation accounts for more than 90% of the total inference execution time of the network model, so the optimization of the convolution operation has always been a focus of performance improvement for convolutional neural networks. Common deep learning frameworks typically implement convolution operations using direct convolution algorithms, matrix multiplication-based convolution algorithms, or fast Fourier transform-based convolution algorithms. However, these traditional convolution algorithms often lead to problems such as redundant calculations, memory waste, and lack of flexibility when dealing with high-dimensional data, data of different sizes, and data in different forms. These problems limit the performance of convolutional neural networks and also affect their potential for applications in more scenarios. To address these issues, this paper focuses on two aspects of research: (1) improving the convolution algorithm based on tensor algebra; (2) implementing high-performance tensor convolution algorithms for CPU and GPU platforms based on their respective characteristics. The main work is as follows:

(1) A new convolution algorithm based on tensor algebra, called image to window (im2win) tensor convolution algorithm, is proposed in this paper. It consists of two parts: an efficient im2win transformation and subsequent tensor multiplication. The im2win transformation reorders the input image data according to the point-wise window operation access order, which can significantly reduce memory overhead compared to the traditional image-to-column (im2col) transformation. Moreover, it ensures the memory access continuity and data reuse of the point-wise window. The resulting im2win tensor is a four-dimensional tensor that preserves more high-dimensional hidden information compared to the two-dimensional matrix obtained by the traditional im2col transformation. The subsequent use of tensor multiplication to perform dot product operations with the im2win tensor and convolution kernel tensor can avoid redundant calculations and data movement, reduce computational complexity and memory consumption, and make convolution implementations more flexible to different hardware platforms.

(2) A high-performance tensor convolution algorithm based on the im2win

transformation is implemented on the CPU platform, and a series of optimization techniques are adopted to improve its runtime efficiency based on the characteristics of the CPU platform. First, a mathematical analysis model is established to analyze the optimization space and strategy selection, and the model is further refined by combining CPU architecture characteristics and tensor convolution calculation processes. Then, SIMD vector instructions are used to accelerate vector multiplication, and multi-threading parallelism is utilized to fully exploit CPU resources. Finally, loop unrolling and other techniques are used to reduce branch prediction and function call overhead. Experimental results show that the proposed tensor convolution algorithm based on the im2win transformation can achieve 2.2 to 5.8 times performance improvement compared to the im2col-based convolution algorithm and 3.8 to 7.1 times performance improvement compared to the direct convolution algorithm, while reducing memory consumption by an average of 41.6% compared to the im2col-based convolution algorithm.

(3) A high-performance tensor convolution algorithm based on the im2win transformation is implemented on the GPU platform, and a series of optimization techniques are adopted to further improve its runtime efficiency based on the characteristics of the GPU platform. First, according to the CUDA architecture model of the GPU, the index of the tensor convolution algorithm is reconstructed to better map the algorithm onto the GPU architecture. Then, tiling strategy is used to reduce global memory access, and shared memory and registers are utilized to cache local data. Finally, microkernel design, vectorized loading/storing, double buffering, and data prefetching techniques are used to improve computational throughput. Experimental results show that the proposed tensor convolution algorithm can achieve an average of about 3.5 times performance improvement compared to the im2col-based convolution algorithm, and up to about 1.8 times performance improvement compared to the cuDNN library's convolution algorithm, with an average performance improvement of 1.1 times. Moreover, the proposed tensor convolution algorithm reduces memory consumption by an average of 23.1% compared to the im2col-based convolution algorithm and the cuDNN library's convolution algorithm.

Keywords: Convolution Algorithm, Tensor Algebra, Convolutional Neural Network, Parallel Computing, High Performance Computing

目录

第 1 章 绪论

1.1 研究背景及意义

随着计算机技术的发展和数据量的增加，人工智能及其相关应用逐渐兴起，并在各行各业中发挥着重要作用。人工智能是一门让计算机具有类似人类智能的技术科学，它可以利用数据和算法来解决各种复杂问题，并产生有价值的结果。人工智能的概念最早由 Turing 在 1950 年提出[1]，经历了多次兴衰和挑战，直到 2006 年深度学习算法被提出，才引发了人工智能领域的革命性变化[2]。神经网络是人工智能领域的一个重要分支，它是一种由多个节点相互连接形成的计算模型，其中每个节点都代表一个神经元，可以接受多个输入信号，通过对这些信号进行加权、求和并加入偏置量，输出一个经过非线性变换后的结果。神经网络最早源于 1940 年代和 1950 年代的生物学研究，研究者们试图通过研究大脑神经元之间的相互作用来理解人类思维和行为的本质。然而，直到 20 世纪 80 年代，神经网络才开始被广泛应用于计算机科学中。

深度学习是使用多层结构的神经网络来学习数据特征和抽象表示，并实现强大表达和推理能力的一种方法。深度学习的概念最早可以追溯到 1943 年，当时 Warren McCulloch 和 Walter Pitts[3]提出了第一个人工神经元模型。但是，直到 1986 年 Rumelhart 等人[4]提出了反向传播算法，才使得训练多层神经网络成为可能。深度学习源于人工神经网络的研究，但是由于梯度消失、计算资源、数据量等问题，长期受到限制。直到 21 世纪初，随着计算能力、数据量和算法的进步，深度学习开始在各种任务上取得突破性的成果，例如语音识别[5][6]、图像分类[7][8]、自然语言生成[9][10]等。在医疗方面，深度学习可以帮助医生诊断疾病，解决医疗资源不足的问题。例如，在肺癌筛查方面，美国 FDA 批准了基于深度学习的算法，可以更准确地识别肺癌[11]。而在金融领域，深度学习则可以提高风险控制能力，防止欺诈行为的发生。例如，支付宝采用了深度学习技术对用户行为进行精准识别，从而有效预防电信诈骗。深度学习技术的快速发展和广泛应用，提高了神经网络模型的准确性，同时也增加了神经网络的深度和复杂度，导致了模型的计算和访问内存的显著增加。

卷积神经网络（CNN）是一种受生物视觉系统启发而设计的深度学习模型。它通过卷积层从输入图像中提取有用的特征，如纹理、边缘、形状等，然后通过全连接层将这些特征映射到输出类别上。CNN 最初由 Yann LeCun[12]在 1980 年代末和 1990 年代初提出，并被用于手写数字识别任务。随着计算机硬件的不

表 1-1 常用卷积神经网络模型中各网络层的计算量

网络模型	卷积层计算量	全连接层计算量	总计算量
AlexNet[7]	1.37 GFLOP	0.06 GFLOP	1.43 GFLOP
VGG-16[27]	15.3 GFLOP	0.1 GFLOP	15.5 GFLOP
GoogLeNet-v1[28]	1.5 GFLOP	0.02 GFLOP	1.6 GFLOP
ResNet-50[29]	3.8 GFLOP	0.04 GFLOP	3.9 GFLOP
Inception-V3[30]	5.72 GFLOP	0.01 GFLOP	5.8 GFLOP

断进步和深度学习技术的不断发展，CNN 逐渐成为计算机视觉领域中最广泛使用和最有效果的模型之一。例如，在 ImageNet 图像分类竞赛中，CNN 模型连续多年获得了优异的成绩，并且超越了人类的表现水平[7]。近年来，CNN 已经被广泛应用于图像识别[13][14]、目标检测[15][16][17]、人脸识别[18][19]、图像超分辨率重建[20][21]、自然语言处理[22][23]等领域。CNN 是一种前向传播的人工神经网络，在训练过程中通过反向传播和梯度下降算法来学习最优的卷积核参数。CNN 通常由卷积层、池化层、全连接层和激活函数等组成，其中卷积层是其最核心和最耗时的部分。卷积层中每个卷积核都对应一个特定类型或尺度的特征，在输入图像上滑动并进行点乘运算来生成输出特征图。池化层则对输入特征图进行下采样操作来减少维度大小，并增加模型对位置变化或噪声干扰等因素的鲁棒性。全连接层则将卷积层和池化层提取的特征向量拼接起来，并通过线性变换和激活函数来输出最终的预测结果。在许多流行的 CNN 模型中，卷积层计算量占据了超过 90%的模型总计算量（如表 1-1 所示）。此外，卷积层中卷积操作在 CNN 推理中占据了 50%-90%[24]的总操作数和超过 90%[25][26]的总执行时间。因此，在提高 CNN 性能时优化卷积操作至关重要。

卷积运算是深度学习中的核心操作，它有三种主要的实现方式：直接卷积算法[31][32]、基于 im2col 变换和通用矩阵乘法（GEMM）的卷积算法[33]和基于快速傅里叶变换（FFT）的卷积算法[34]。这三种算法在现代深度学习框架中，如 PyTorch[35]，TensorFLow[36]，Caffe[37]等都有广泛的应用。然而，这些传统的卷积算法也存在着一些问题和局限性。例如，在处理高维数据时会产生大量冗余计算；在处理不同尺寸或稀疏数据时会造成内存浪费；在处理不同形式或结构化数据时会缺乏灵活性等。例如，对于一个三维图像输入，如果使用基于 im2col 和 GEMM 的方法进行卷积运算，则需要将其展开为一个二维矩阵，并与卷积核矩阵进行乘法运算。这样做会导致输入图像中相邻像素之间的空间关系丢失，并且增加了计算量和内存消耗。

近年来，基于张量代数的方法[38][39][40][41]已成为探索 CNN 性能优化的研究热点。张量代数[42][43]是一种描述多维数组运算和变换规则的数学工具，可以将

复杂抽象的运算简化为直观清晰的符号表示，并提供更高层次的抽象和优化空间。基于张量代数的方法可以将卷积操作转换为不同形式的张量乘积运算，如循环卷积[44]、低秩张量格式卷积[45]等。这些方法可以有效地减少冗余计算，提高运算速度，同时也可以适应不同形式或者结构化数据的处理，具有很强的灵活性和通用性。

目前传统的卷积算法在处理高维或者大规模数据时仍然面临着内存占用过大和性能不高等问题。为了解决这些问题，本文旨在探索基于张量代数方法对 CNN 中卷积操作进行性能优化，并分析其在不同平台和任务上的有效性和通用性。具体而言，本文主要包括以下三个方面：（1）对比分析不同形式的卷积算法在不同应用场景中的优缺点，并提出一种适合于 CNN 结构特点和计算需求的基于张量代数的卷积算法；（2）结合 CPU 平台特性实现基于该张量卷积形式的高效卷积算法，并利用多线程并行技术提高其计算速度；（3）在 GPU 平台上实现基于该张量卷积形式的高效卷积算法，并利用 CUDA 编程框架进行加速优化。

1.2 本课题的研究现状

深度学习模型在数据科学领域具有强大的表达能力和优异的精度，但同时也带来了巨大的计算开销和资源消耗。这一问题不仅存在于模型训练阶段，也影响了模型部署和应用阶段。因此，如何优化深度学习模型的性能是一个重要而紧迫的课题。卷积神经网络作为深度学习中目前最重要和最常用的模型，国内外学者对卷积神经网络性能优化进行了大量的研究，主要从卷积算法、网络结构、硬件平台等方面进行改进和创新。其中卷积算法作为卷积神经网络最核心的部分，直接决定了网络的计算效率和资源消耗。现有的深度学习神经网络框架实现卷积操作的方式各有优缺点，没有一种通用的解决方案适用于所有场景。本节将从卷积神经网络性能优化，卷积算法，深度学习神经网络框架介绍该领域的研究现状。

1.2.1 卷积神经网络性能优化的研究现状

深度学习是近年来人工智能领域最为活跃和前沿的研究方向之一。其中卷积神经网络（Convolutional Neural Network, CNN）作为深度学习中最重要和最常用的模型之一，在图像识别等任务上取得了突破性的进展[2]。2012 年，Krizhevsky 等人提出了 AlexNet[7]，这是第一个利用 GPU 加速训练并在 ImageNet 数据集上大幅提高识别精度的 CNN 模型。AlexNet 的成功引发了深度学习领域对 CNN 模型设计和优化的广泛关注。然而，随着 CNN 模型越来越复

杂和深层，其训练和推理所需的计算资源也越来越大。这不仅增加了模型部署和应用时所需时间、金钱成本，也限制了模型在移动设备等资源受限平台上运行时效率[46]。因此，在保证模型性能不降低或者降低很少情况下对 CNN 模型进行加速成为了当前深度学习领域面临并且迫切需要解决的问题。目前针对 CNN 模型加速的方法主要可以分为两类：硬件层面的方法和软件层面的方法。

硬件层面的方法主要是通过使用专用硬件加速器来实现 CNN 模型训练或推理时效率提升。目前使用最广泛的是 Nvidia 公司开发的异构图形处理单元（Graphics Processing Unit, GPU）以及配套的软件生态统一计算架构（Compute Unified Device Architecture, CUDA）[47]。GPU 相比于传统 CPU 在并行计算方面具有更高效率，并且 CUDA 提供了方便开发者编程使用 GPU 的接口。除此之外，还有一些比较出名的专用硬件加速器，例如 Google 公司设计开发张量处理单元（Tensor Processing Unit, TPU）[48]和华为公司设计开发晟腾（HUAWEI Ascend）[49]系列人工智能芯片。TPU 是一种定制集成电路（Application-Specific Integrated Circuit, ASIC），专为 Google 公司开发的深度学习框架 TensorFlow[36]优化而设计，能够在训练和推理时提供高效的性能。晟腾系列芯片是华为公司基于自主研发的达芬奇架构打造的人工智能计算平台，包括了面向云端、边缘端和终端设备的不同规格和型号的芯片。

软件层面的方法主要是通过对 CNN 模型结构进行压缩、剪枝或者变种等操作，以减少模型大小或者计算量。这类方法可以分为以下几种：张量分解、权重共享、网络剪枝、知识蒸馏等。

张量分解[50][51][52][53]的核心思想是将 CNN 模型中每一层的参数权重看成是单独的高阶张量，然后将高阶张量近似表示为若干秩一张量之和。Denton 等人[54]利用张量分解和低秩近似的技术，将卷积层或全连接层上权重张量分解成多个小矩阵，也就是将一个大的网络层分解成几个小的网络层，并且这几个小网络层的计算总量小于原来大网络层的计算总量。这样一来不仅可以大大减少模型所占的空间，同时也减少了模型的计算总量，从而实现对 CNN 模型的加速。Rigamonti 等人[55]则提出在卷积层上可以考虑使用低秩卷积核来实现加速。

权重共享的核心思想是将 CNN 模型中具有相同功能或者相似功能的参数权重设置为相同或者接近的值，从而减少模型中需要存储和更新的参数数量。这种方法可以有效地降低模型的存储空间和计算复杂度，同时也可以防止过拟合[56]。权重共享的一种常见形式是哈希化神经网络（Hashed Neural Network, HNN），它使用哈希函数将不同位置和不同层次的参数映射到一个有限大小的哈希表中，使得多个参数共享一个哈希表中的值[57][58][59][60]。

网络剪枝的核心思想是在训练或者训练后对 CNN 模型中一些不重要或者冗

余的参数进行删除，从而减少模型大小和计算量。这种方法可以根据不同的剪枝策略和剪枝粒度进行分类，例如基于权重大小、梯度大小、敏感度分析等指标进行剪枝，以及基于通道、卷积核、滤波器等单元进行剪枝[61][62][63][64]。

知识蒸馏的核心思想是利用一个已经训练好的大型 CNN 模型（称为教师模型）来指导一个较小规模的 CNN 模型（称为学生模型）的训练过程，从而使学生模型能够学习到教师模型的知识，并在保持较高精度的同时实现模型加速[65][66][67][68]。知识蒸馏的方法主要包括两个步骤：第一步是使用教师模型对数据集进行预测，并记录其输出的软标签（Soft Label），即每个类别的概率分布；第二步是使用学生模型对数据集进行训练，并使用教师模型输出的软标签作为额外的监督信息来优化学生模型的损失函数。

1.2.2 卷积算法的研究现状

卷积操作是 CNN 中最核心也最复杂的运算之一，它在 CNN 模型中起着关键的作用，同时也决定了 CNN 性能优化的上限。目前常用的卷积算法主要有三种：直接卷积算法、基于 im2col 变换和通用矩阵乘法（GEMM）的卷积算法和基于傅里叶变换（FFT）的卷积算法。这三种算法各有优缺点，在不同场景下表现不同。

直接卷积算法是最直观也最简单的一种卷积实现方式，它直接按照卷积的定义进行计算，即将输入特征图与滤波器在空间域上进行逐元素相乘并求和。这种方式虽然简单易懂，但是计算效率较低，并且难以利用现代 CPU 或 GPU 的并行计算能力，在实际应用中很少使用这种方式来实现卷积。Evangelos 等人[69]通过使用专门针对处理器的数据布局（即将数据按照特定的顺序或形式存储或访问），从而提高直接卷积的性能。然而，这种方法的实际应用需要对卷积的输入和输出进行代价昂贵的布局转换操作。因此，直接卷积算法虽然易于理解，但是在性能和并行计算方面存在很大的局限性。

基于 GEMM 的卷积算法是目前最常用也最高效的一种卷积实现方式，它通过对输入特征图和滤波器进行变换（例如 im2col 变换），将卷积运算转化为通用矩阵乘法（General Matrix Multiplication, GEMM）运算[70][71]。这种方式可以充分利用现代 CPU 或 GPU 的并行计算能力，并且可以调用高度优化过的 GEMM 库来执行计算[72][73][74]。然而，这种方式也存在一些缺点：首先，变换过程会增加额外的内存开销；其次，变换后得到的矩阵可能会存在冗余或者稀疏性；最后，在某些情况下（例如小尺寸滤波器或者大步长），变换后得到的 GEMM 运算可能不是最优的[75]。最近一些工作对 im2col 变换进行改进[76][77][78]，将单个矩阵乘法分成多个小矩阵乘法来减少存储开销，然后并行执行多个

GEMM 操作以完成卷积运算。这种方法可以在一定程度上缓解内存开销和优化问题，但是仍然存在数据冗余，需要额外的内存开销。综上所述，基于 GEMM 的卷积算法虽然具有较高的计算性能，但是也存在一些内存开销和优化问题，因此需要进一步改进。

基于 FFT 的卷积算法是另一种卷积实现方式，它通过对输入特征图和滤波器进行快速傅里叶变换（Fast Fourier Transform, FFT），将卷积运算从空间域转化到频域，然后在频域上进行逐元素相乘并求和，最后通过逆傅里叶变换得到输出特征图[79][80]。这种方式可以利用 FFT 算法降低卷积运算的计算复杂度，并且可以避免 GEMM 运算中的冗余或者稀疏性问题。然而，这种方式也存在一些缺点：首先，FFT 和逆 FFT 过程会增加额外的内存开销；其次，在某些情况下（例如大尺寸滤波器或者小步长），FFT 运算可能不是最优的；最后，在处理非方形或者非二次幂尺寸的输入特征图或滤波器时，需要进行补零操作，从而增加计算量和内存消耗[81][82][83]。

以上的传统卷积算法在不同场景下有各自的优缺点，但也存在着冗余计算、内存浪费和灵活性不足等问题。为了解决这些问题，本文工作提出了一种基于张量代数理论的张量卷积算法，它能够在不同硬件平台上实现高性能且低内存占用的卷积运算，并进行了相关实验验证。本文工作主要包括以下几个方面：针对基于 GEMM 的卷积算法中存在的数据冗余和额外内存开销问题，提出一种新的数据变换方法，减少数据重复和内存消耗；在 CPU 和 GPU 平台上实现了高性能且低内存占用的张量卷积算法，并与传统卷积算法进行了对比分析；通过实验验证了张量卷积算法在 CNN 模型中的有效性和优越性，并与其他卷积算法进行了比较。

1.2.3 深度学习神经网络框架的研究现状

深度学习框架是实现人工智能应用的重要工具，它为深度学习算法研究人员提供了简单好用、高效灵活地构建深度神经网络模型的编程接口和工具包。目前市场上有多种流行且成熟的深度学习框架，如 PyTorch[35]，TensorFlow[36]，Caffe[37]等。这些框架都需要对卷积计算操作进行高效的处理，因为卷积是深度神经网络中最常用和最耗时的算法之一。卷积可以通过不同的方法实现，例如将其转换成等效的矩阵乘法或者使用专门设计的算法。不同的方法有不同的优缺点，在性能、资源消耗、通用性等方面会有差异。在早期的深度学习框架中，对于卷积操作都采用的是将卷积转换成等效的矩阵乘法操作，然后根据所使用的 CPU 或 GPU 平台选择使用对应的基础线性代数库[84]，如 MKL[85]、openBLAS[86]或 cuBLAS[72]。这种实现卷积的方法虽然简单易用，但是效率低下，

并且不能充分利用硬件资源。而在现在的深度学习框架中，在 CPU 平台上可以使用 Intel 开发的 oneDNN 库[87]，在 GPU 平台上则可以使用 NVIDIA 开发的 cuDNN 库[73]。这些库中封装了多种卷积算法的实现，并进行了针对性的高度优化，它们能够根据输入数据和网络结构自动选择最合适的算法，并且利用并行计算和内存复用等技术提高运行速度和节省空间。但这些高度优化的深度学习库往往都是闭源的，其定义的卷积运算参数比较固定，在一些特殊的应用场景下达不到最佳性能，构造一些特殊的卷积时也可能因为参数不够灵活而受限。

1.3 本文主要工作

传统卷积算法在处理高维数据、不同尺寸数据、不同形式数据时，存在着冗余计算、内存浪费和灵活性不足等问题。为了解决这些问题，本文以降低卷积算法的内存占用和提升卷积算法的性能为目标，基于张量代数理论，在不同硬件平台上实现了高性能且低内存占用的张量卷积算法，并进行了相关实验验证。本文主要研究内容包括以下三个方面：

（1）针对传统卷积算法内存占用过高和内存访问不连续的问题，提出了一种新的基于张量代数的卷积算法，名为基于 im2win 变换的张量卷积。该算法由一种新颖高效的 im2win 变换和后续的张量乘法两部分组成。本文提出的 im2win 变换是一种将输入图像按照点乘窗口运算访问顺序重新排列数据的方法，相比于传统的 im2col 变换，它能够显著减少内存开销，并保证点乘窗口内存访问连续性和数据重用性。im2win 变换得到的 im2win 张量仍然是一个四维张量，相比于传统的 im2col 变换得到的二维矩阵，它能够保留更多高维隐含信息。利用张量乘法将输入图像和卷积核都视为四维张量，并进行点乘运算来实现卷积操作。这样就可以避免重复计算和数据移动，降低计算复杂度和内存消耗，并使得卷积实现更加灵活适应不同硬件平台和卷积场景。

（2）针对 CPU 平台上卷积算法内存占用过大和计算性能不高的问题，在 CPU 平台上实现了高性能的基于 im2win 变换的张量卷积算法，并采用了 SIMD 向量指令集、多线程并行、循环展开等优化技术来提高运行效率。首先建立了一个分析优化空间与策略选择的数学分析模型，并结合 CPU 架构特性和张量卷积计算过程设计完善该模型；然后利用 SIMD 向量指令集来加速张量乘法，并通过多线程并行来充分利用 CPU 资源；同时还采用循环展开等技术来减少分支判断和函数调用开销；最后与 CPU 平台上常见深度学习库进行对比，在减少内存消耗和提升计算性能方面都有较大提升。

（3）针对 GPU 平台上卷积算法在不同类型的卷积参数下性能不稳定和内存占用过大的问题，在 GPU 平台上实现了高性能的基于 im2win 变换的张量卷

积算法，并采用了 Tiling 策略、共享内存与寄存器、微内核设计、向量化加载/存储、双缓冲与数据预取等优化技术来进一步提高运行效率。首先根据 GPU 的 CUDA 架构模型，对张量卷积算法的索引进行重构，使得该算法可以更好地映射在 GPU 的架构上；然后利用 Tiling 策略来减少全局内存访问，并利用共享内存与寄存器来缓存局部数据；同时还采用微内核设计、向量化加载/存储、双缓冲与数据预取等技术来提高计算吞吐率；最后与 GPU 平台上常见的深度学习库对比，在减少内存消耗的同时，计算性能更优或相当。

1.4 本文的组织架构

本文根据研究课题及研究内容，总共包含五个章节，具体内容如下：

第 1 章：绪论。这一章首先介绍卷积神经网络的应用背景，以及卷积神经网络模型训练和推理中存在的性能问题，从而引出卷积神经网络性能优化的必要性和研究价值。然后，在研究现状部分主要介绍卷积神经网络性能优化和卷积操作性能优化上的国内外研究现状。最后，介绍本文的主要工作，以及本文各章节的内容安排。

第 2 章：一种新的基于张量代数的卷积算法。这一章首先介绍卷积操作的定义以及传统的卷积实现算法，并分析现有的传统卷积算法存在的不足和缺点。然后，为了解决传统卷积算法的内存占用过大和内存访问不连续的缺点，提出一种基于窗口序排列的张量数据转置算法来降低内存占用并保持内存访问连续。最后，将本文提出的基于窗口序排列的张量数据转置算法与传统的数据转置算法进行对比分析。

第 3 章：基于窗口序的张量卷积在 CPU 平台上的优化实现。这一章首先介绍 CPU 的硬件体系结构、SIMD 向量指令集以及 CPU 并行技术。然后，根据 CPU 的硬件体系结构特点和并行模型，在 CPU 上设计实现本文提出的基于窗口序的张量卷积算法，并采用了 SIMD 向量指令集、多线程并行、循环展开等优化技术来提高运行效率。最后，在 CPU 设备上使用一个完备的测试基准对实现的基于窗口序的张量卷积算法与现有方法之间进行性能对比，包括运行时间、每秒浮点运算次数和内存占用三个指标，并分析本文提出的算法的通用性和有效性。

第 4 章：基于窗口序的张量卷积在 GPU 平台上的优化实现。这一章首先介绍了 GPU 的硬件体系结构、软件编程模型以及计算统一设备架构 CUDA。然后，根据 GPU 的硬件体系结构特点和计算统一设备架构 CUDA，在 GPU 上设计实现本文提出的基于窗口序的张量卷积算法，并采用了 Tiling 策略、共享内存与寄存器、微内核设计、向量化加载/存储、双缓冲与数据预取等优化技术来进一

步提高运行效率。最后，在一个完备的测试基准对实现的基于窗口序的张量卷积算法与现有方法进行性能测试，指标包括运行时间、每秒浮点运算次数和内存占用，并且对本文实现的基于窗口序的张量卷积算法进行消融实验，以分析本文提出的算法的通用性和有效性。

第 5 章：总结与展望。这一章对本文的研究内容进行概括总结，提出本研究所存在的不足以及对后续工作的展望。

第 2 章 一种新的基于张量代数的卷积算法

第一章节中详细介绍了卷积神经网络性能优化的研究背景和意义，并回顾了卷积神经网络性能优化、卷积算法和深度学习框架国内外相关研究的现状和进展。本章首先介绍卷积操作的基本概念和原理，以及传统的卷积算法的计算过程和主要实现方式。然后指出了传统的卷积算法存在的问题和不足，如计算复杂度高、内存消耗大、实现方式不灵活等，并提出了本文的研究目标和创新点，即设计一种基于张量代数的卷积算法，该算法由 im2win 变换和张量乘法两部分组成。本文提出的 im2win 变换是一种新颖高效的数据重排方法，相比于常用的 im2col 变换，它能够显著减少内存开销，并保证点乘窗口内存访问连续性和数据重用性。张量乘法是一种将输入图像和卷积核都视为四维张量，并进行点乘运算来实现卷积操作的方法。这样可以避免重复计算和数据移动，降低计算复杂度和内存消耗，并使得卷积实现更加灵活适应不同硬件平台和卷积场景。

2.1 传统的卷积算法

卷积操作是卷积神经网络中的核心运算，它通过滑动一个固定大小的卷积核来对输入数据进行卷积操作。传统的卷积算法包括直接卷积算法以及基于 im2col 变换和通用矩阵乘法（GEMM）的卷积算法。直接卷积算法是最朴素的实现方式，它通过遍历输入数据的每个位置并执行卷积运算来得到输出结果。这种方法计算量大，并且存在大量重叠的计算，导致效率较低。基于 im2col 变换和 GEMM 的卷积算法将输入数据和卷积核都转化为二维矩阵形式，从而将卷积操作转化为矩阵乘法运算，进而减少计算量并提高效率。这种方法的关键在于将输入数据变换为二维矩阵，然后使用矩阵乘法进行卷积计算。这种方法虽然速度快于直接卷积算法，但是需要增加一些额外的计算开销来做数据变换。

2.1.1 卷积操作的定义

卷积操作是卷积神经网络中最基本的操作之一，它通过在输入数据上滑动一个固定大小的窗口，并计算窗口内输入数据与卷积核的点乘来提取特征信息。卷积是一种稀疏连接的线性运算，与全连接操作相比，卷积操作可以有效提取数据的局部特征，并且可以减少参数量和降低过拟合。在卷积操作中，一共有三个主要的张量，输入张量（$\mathcal{I}$）、卷积核（$\mathcal{F}$）和输出张量（$\mathcal{O}$）。本文默认使用的张量数据布局为 NCHW 格式，三个主要张量可以表示为输入张量

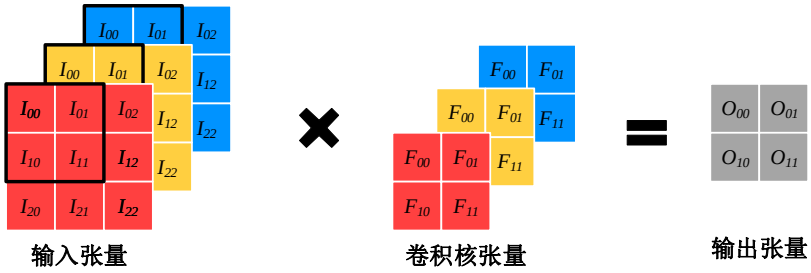

图 2-1 卷积计算过程

$\mathcal{I}[N_i][C_i][H_i][W_i]$、卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$ 和输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$。卷积操作的标准数学表达式为：

$$\mathcal{O}_{(i,j,m,n)}=\sum_{j=0}^{C_i-1}\sum_{m=0}^{H_f-1}\sum_{n=0}^{W_f-1}\left(\mathcal{I}_{(i,j,m\times s+u,n\times s+v)}\times\mathcal{F}_{(r,j,u,v)}\right) \tag{2-1}$$

其中各变量的取值范围如下：

$$i=0,1,..,N_i\text{-}1,\ j=0,1,..,C_o\text{-}1,\ m=0,1,..,H_o\text{-}1,n=0,1,..,W_o\text{-}1,$$
$$u=0,1,..,H_f\text{-}1,\ v=0,1,..,W_f\text{-}1,r=0,1,..,C_i\text{-}1.$$

N_i 表示批次大小维度，s 表示步长大小，C_i 和 C_o 表示输入张量和输出张量的通道维度，$H_{i/f/o}$ 和 $W_{i/f/o}$ 表示三个张量的宽和高两个维度。

2.1.2 直接卷积算法

直接卷积是最简单的卷积实现，直接使用原始的高维张量进行计算，具体的实现逻辑如算法 1 所示。一个基本的直接卷积算法由七个嵌套的循环组成，外部四个循环遍历输出批次 N_o、输出通道 C_o、输出高度 H_o 和输出宽度 W_o 四个维度以获得输出的索引值（如算法 1 中第 1 行至第 4 行）。内部的三个循环遍历

算法 1：直接卷积算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

1: **for** i = 0 **to** N_o − 1 **do**
2: **for** j = 0 **to** C_o − 1 **do**
3: **for** m = 0 **to** H_o − 1 **do**
4: **for** n = 0 **to** W_o − 1 **do**
5: **for** r = 0 **to** C_f − 1 **do**
6: **for** u = 0 **to** H_f − 1 **do**
7: **for** v = 0 **to** W_f − 1 **do**
8: $\mathcal{O}[i][j][m][n]+=\mathcal{I}[i][r][m\times s+u][n\times s+v]\times\mathcal{F}[j][r][u][v]$

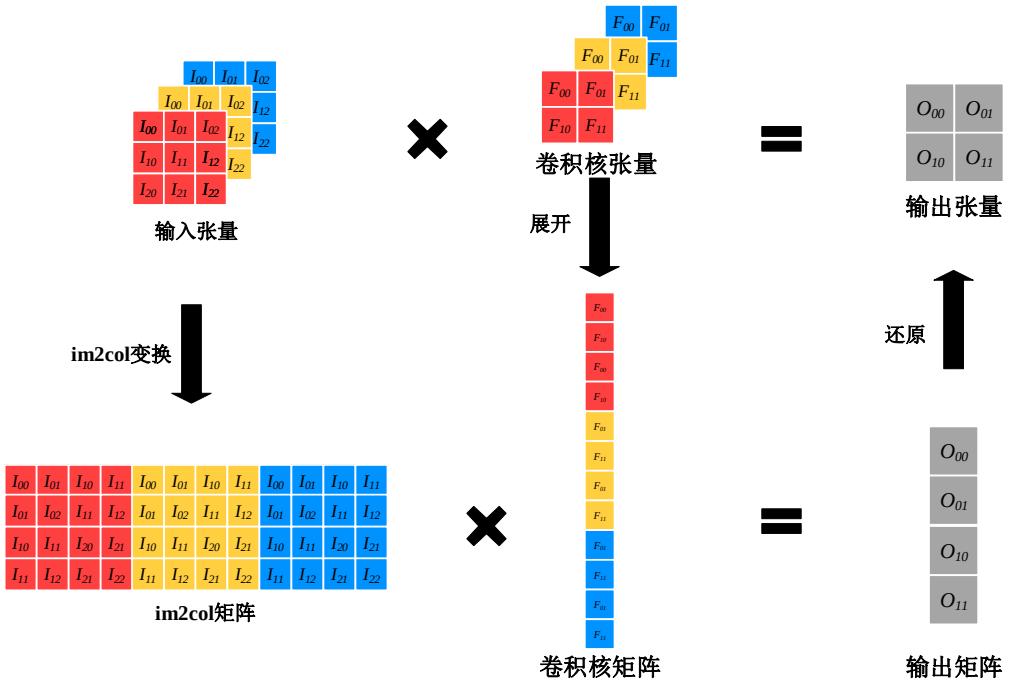

图 2-2 基于 im2col 变换的卷积算法

卷积核通道C_f、卷积核高度H_f、卷积核宽度W_f，并在最内部的循环中执行乘加操作来计算输出（如算法 1 中第 5 行至第 8 行）。虽然这种直接卷积的算法实现时较为简单，但这种实现可能会存在一些性能问题：（1）数据内存访问不连续。在算法 1 执行乘加操作时（如算法 1 中第 8 行），以输入张量和卷积核的对应的点乘窗口顺序访问，但每一个进行点乘窗口中的数据并不是全部连续的，非连续内存读取导致了数据加载的时延很高，性能表现很差；（2）乘加操作可能无法进行向量化。因为每一个进行点乘窗口中的数据（如图 2-1 中第 1 个窗口）并不是全部连续的，而向量化指令加载数据到向量寄存器中需要是内存连续的数据，所以算法 1 中的乘加操作可能无法进行向量化。

2.1.3 基于 im2col 变换和通用矩阵乘法的卷积算法

因为直接卷积算法在实际实现中性能表现往往不好，由 Chellapilla 等人[33]提出的基于 image to column（im2col）变换和通用矩阵乘法（GEMM）卷积（以下简称为基于 im2col 变换的卷积）是目前最常用的卷积算法，在现有的深度学习框架中（如 PyTorch[35]、TensorFlow[36]、Caffe[37]）得到了广泛应用。由于该卷积算法的实现较为简单且性能良好，它经常被用作与其他卷积算法性能比较的基准。

基于 im2col 变换的卷积本质上是将原本的卷积操作转换成一个 GEMM 操作。在该卷积算法中，大小为$N_i \times C_i \times H_i \times W_i$的输入张量$\mathcal{I}$被分成$N_i$个批次处理，每个批次包含大小为$C_i \times H_i \times W_i$（即单个图像）的数据$\mathcal{I}'$。如图 2-2 所示，

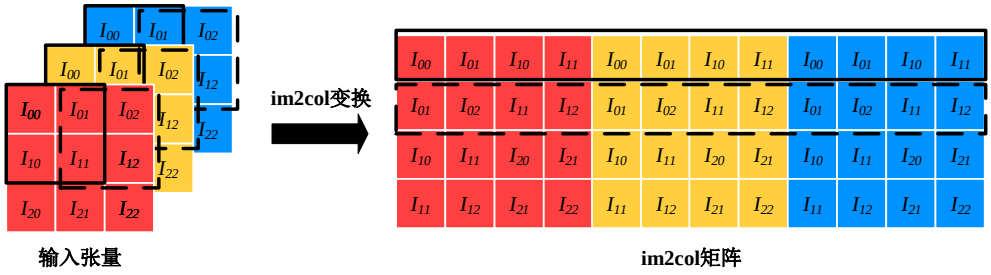

图 2-3 im2col 变换过程

im2col 变换将 $\mathcal{I}'$ 转换成二维矩阵，本文称这个二维矩阵为 im2col 矩阵，卷积核张量 $\mathcal{F}$ 则被展开成一个二维卷积核矩阵。

在 im2col 中，$\mathcal{I}'$ 的每个点乘窗口中的元素复制到矩阵的单行中（如图 2-3）。将 im2col 矩阵表示为 M，卷积核矩阵表示为 N，则 im2col 变换可以写成：$M(mW_o+n,(rH_f+u)W_f+v)=\mathcal{I}'(r,m+u,n+v),N((rH_f+u)W_f+v,j)=\mathcal{F}(j,r,u,v).$ 之后，使用基础线性代数库（BLAS）[84]中的 GEMM 函数将 im2col 变换得到的 im2col 矩阵和展开后的卷积核矩阵进行矩阵乘法操作得到输出矩阵：$R'=M\times N$。卷积的结果张量 R 由 R' 转置得到：$R(j,m,n)=R'(mW_o+n,j).$

得益于高度优化的 BLAS 库（如 OpenBLAS[86]、MKL[85]），基于 im2col 变换的卷积具有可靠的性能并支持各种输入张量大小。但是，这种卷积算法仍然存在一些缺陷：（1）im2col 变换操作需要大量的内存消耗和带宽占用。由于从原始高维张量到二维矩阵的 im2col 变换中，im2col 矩阵存储了每一个进行点乘窗口的数据。变换得到的 im2col 矩阵往往比原始张量大很多，需要消耗大量额外内存空间，这导致了在内存有限的设备上可能难以部署此算法。（2）im2col 矩阵与卷积核矩阵尺寸差距过大导致 GEMM 操作的性能下降。基于 im2col 变换的卷积本质上是将卷积操作转换为矩阵乘法操作，所以该卷积的性能在很大程度上取决于 GEMM 操作的性能。BLAS 库中的 GEMM 操作的性能在两个输入矩阵尺寸相近的情况下最佳，但 im2col 矩阵通常会比卷积核矩阵大很多，这导致了之后的 GEMM 操作性能有所下降[88]。

2.2 一种新的基于张量代数的卷积算法

为了解决基于 im2col 变换的卷积中内存占用过多和直接卷积中内存访问不连续的问题，本文提出了一种新的基于张量代数的卷积算法，名为基于 im2win 变换的张量卷积，该算法由一种新的 im2win 变换和后续的张量乘法两部分组成。

2.2.1 内存高效的 im2win 变换

本文提出一种名为 image to window (im2win) 变换的内存高效的数据变换算

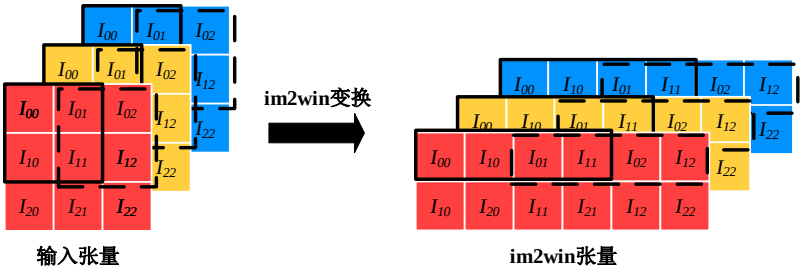

图 2-4 im2win 变换过程

法，该变换算法以点乘窗口运算访问顺序重新排列数据。相对于 im2col 变换，im2win 变换通过更紧凑的数据排列方式显著减少了内存开销。im2col 变换将原始输入张量的每一个点乘窗口都存储为了 im2col 矩阵的一行（如图 2-3 所示），所以在 im2col 矩阵的每一行之间存在大量的冗余重复元素，这导致了 im2col 矩阵的大小比原始输入张量的大小要大得多，需要花费大量额外内存储存 im2col 矩阵。与 im2col 变换将原始输入张量的每一个点乘窗口都重新存储不同，本文提出的 im2win 变换按照卷积操作中的点乘窗口顺序对元素进行重新排列转换。如图 2-4 所示，对于连续的窗口，im2win 变换得到的张量数据没有重复元素并且窗口中的元素在访问时可以保持连续性。im2win 变换除了能够降低内存消耗并保证点乘窗口内存访问连续外，还可以在后续卷积操作计算过程中提高数据重用。在连续的两个点乘窗口操作中，第一个操作中加载的大部分元素会在第二个操作中被重复使用，这可以极大地提高数据重用和高速缓存的命中率。

在 im2win 变换中，将单张输入图像 $\mathcal{I}'$ 的每个通道划分为大小为 $H_f \times W_f$ 的 $H_o \times W_o$ 个窗口，并将同一行中的 W_o 个窗口复制到 im2win 张量的一行中。为了按照点乘窗口存储数据，im2win 变换时采取了列主序的顺序存储输入图像 $\mathcal{I}'$ 中大小为 $H_f \times W_i$ 的元素。对 $\mathcal{I}'$ 中单个通道上的所有窗口执行上述操作后，将得到一个大小为 $(H_o, H_f \times W_i)$ 的张量（如图 2-4）。该张量按照点乘窗口顺序排列，并且比 im2col 矩阵具有更少的冗余元素。对输入张量 $\mathcal{I}$ 的批次和通道维度执行以上变换后，将得到一个大小为 $(N_i, C_i, H_o, W_i \times H_f)$ 的张量，并称此张量为 im2win 张量。将 im2win 张量表示为 $\hat{\mathcal{I}}$，im2win 变换如下式所示：

$$\hat{\mathcal{I}}(i, r, m, kH_f + u) = \mathcal{I}(i, r, m+u, n+v) \tag{2-2}$$

其中各变量的取值范围如下：

$$m = 0,1,..,H_o - 1, n = 0,1,..,W_o = 1, u = 0,1,..,H_f - 1, v = 0,1,..,W_f - 1,$$
$$i = 0,1,..,N_i - 1, r = 0,1,..,C_i - 1, k = 0,1,..,W_i - 1.$$

算法 2：im2win 变换算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：im2win 张量 $\hat{\mathcal{I}}[N_i][c_i][H_o][W_i \times H_f]$

1: $H_o = (H_i - H_f)/s + 1$
2: **if** $H_f > s$ **then**
3: 分配 $N_i \times C_i \times H_o \times W_i \times H_f$ 大小的空间给 $\hat{\mathcal{I}}$
4: **else**
5: 分配 $N_i \times C_i \times H_i \times W_i$ 大小的空间给 $\hat{\mathcal{I}}$
6: **for** $i = 0$ **to** $N_i - 1$ **do**
7: **for** $r = 0$ **to** $C_i - 1$ **do**
8: **for** $m = 0$ **to** $H_o - 1$ **do**
9: **for** $k = 0$ **to** $W_i - 1$ **do**
10: **for** $u = 0$ **to** $H_f - 1$ **do**
11: $\hat{\mathcal{I}}[i][r][m][k \times H_f + u] = \mathcal{I}[i][r][m \times s + u][k]$

根据式 2-2 实现的 im2win 变换算法如算法 2 所示，在本文中算法中的区分维度的索引变量 (i, j, m, n, r, u, v) 保持统一命名。在算法 2 中，首先根据 H_f 与 s 的大小关系来决定 im2win 张量 $\hat{\mathcal{I}}$ 的大小（算法 2 中第 2 行至第 5 行）。当 $H_f > s$ 时，意味着 im2win 变换过程中上一行的点乘窗口的下部分和当前行的点乘窗口的上部分有相同的元素，因此给 $\hat{\mathcal{I}}$ 分配大小为 $N_i \times C_i \times H_o \times W_i \times H_f$ 的内存空间（算法 2 中第 3 行）。例如在图 2-4 中，$H_f = 2$ 且 $s = 1$，因此 $H_f > s$。当 $H_f < s$ 时，意味着 im2win 变换过程中前一行的点乘窗口的下部分和当前行的点乘窗口的上部分没有相同的元素。在这种情况下，im2win 张量 $\hat{\mathcal{I}}$ 的大小与输入张量 $\mathcal{I}$ 相等，因此为 $\hat{\mathcal{I}}$ 分配大小为 $N_i \times C_i \times H_i \times W_i$ 的内存空间（算法 2 中第 5 行）。接下来，算法将 $\mathcal{I}$ 中的元素复制到 $\hat{\mathcal{I}}$ 中。需要注意的是，所有的数据的底层存储都是基于内存中的一维数组。如果张量数据以 $NCHW$ 格式的顺序存储，则在 W 维度上了连续两个元素之间的内存地址距离为 1，而在 H 维度上连续两个元素之间的内存地址距离为 W。在分层存储结构的设备中，由于连续的元素往往会被一起移动到较高层次的存储器中，因此元素之间内存地址距离越短，空间局部性就越好。所以，访问 N, C, H, W 四个维度的访问代价也相应地降低。因此，在将元素从输入张量 $\mathcal{I}$ 中复制到 im2win 张量 $\hat{\mathcal{I}}$ 时，需要根据 $\hat{\mathcal{I}}(N_i, C_i, H_o, W_i \times H_f)$ 较小的访问代价优先选择循环（算法 2 中第 6 行至第 10 行），以实现更快的内存访问。在卷积实现的过程中，输入张量、卷积核张量和输出张量是三个主要的张量数据，按照由大到小排序，它们的访问代价通常为 $\mathcal{I} > \mathcal{O} > \mathcal{F}$。

2.2.2 基于 im2win 变换的张量卷积

通过上述的 im2win 变换算法可以将原始的输入张量变换得到 im2win 张量，im2win 张量与传统的 im2col 变换得到的 im2col 矩阵对比能够有效降低内存空间的消耗，并且相比于原始输入张量能够保证每个点乘窗口的元素内存访问连续。此外，通过 im2win 变换得到的 im2win 张量仍然是一个四维张量，相比于传统的 im2col 变换得到的 im2col 二维矩阵能够保留更多的高维隐含信息。基于 im2win 变换的张量卷积算法首先通过 im2win 变换将输入张量 $\mathcal{I}$ 变换为 im2win 张量 $\hat{\mathcal{I}}$ 后，在通过 im2win 张量 $\hat{\mathcal{I}}$ 和卷积核张量 $\mathcal{F}$ 计算出输出张量 $\mathcal{O}$，基于 im2win 变换的张量卷积的定义如下式所示：

$$\mathcal{O}_{(i,j,m,n)}=\sum_{j=0}^{C_i-1}\sum_{m=0}^{H_f-1}\sum_{n=0}^{W_f-1}\left(\hat{\mathcal{I}}_{(i,j,m,(n\times s+v)\times u)}\times\mathcal{F}_{(r,j,u,v)}\right) \tag{2-3}$$

其中各变量的取值范围如下：

$$m=0,1,..,H_o-1,n=0,1,..,W_o=1,u=0,1,..,H_f-1,v=0,1,..,W_f-1,$$
$$i=0,1,..,N_i-1,r=0,1,..,C_i-1,k=0,1,..,W_i-1.$$

算法 3：基于 im2win 变换的张量卷积算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

```
1:  Î[N_i][C_i][H_o][W_i × H_f] = im2win(I, F, s)
2:  for i = 0 to N_o − 1 do
3:      for j = 0 to C_o − 1 do
4:          for m = 0 to H_o − 1 do
5:              for n = 0 to W_o − 1 do
6:                  for r = 0 to C_f − 1 do
7:                      for u = 0 to H_f − 1 do
8:                          for v = 0 to W_f − 1 do
9:                              O[i][j][m][n] += Î[i][r][m][n × s × W_f + v × H_f + u]
                                × F[j][r][u][v]
```

根据式 2-3 实现的基于 im2win 变换的张量算法如算法 3 所示，该算法嵌套多个循环的结构，通过不同层的循环迭代值来获取 im2win 张量 $\hat{\mathcal{I}}$、卷积核张量 $\mathcal{F}$ 和输出张量 $\mathcal{O}$ 的索引值，之后执行乘加运算并将结果存储至相应位置。在式 2-3 中，在 C_i、H_f 和 W_f 三个维度上进行累加 im2win 张量 $\hat{\mathcal{I}}$ 和卷积核张量 $\mathcal{F}$ 的

积表示卷积操作中单个点乘窗口运算。但 im2win 张量中数据排列是按照连续的点乘窗口排列，因此在基于 im2win 变换的张量卷积实现时可以一次运算多个窗口而不是单个窗口来提升卷积操作的运算效率。由于基于 im2win 变换的张量卷积在实现时需要考虑不同平台的软硬件体系结构，所以本文将在后续第 3 章中结合 CPU 平台的特性详细介绍基于 im2win 变换的张量卷积实现及优化，在第 4 章中结合 GPU 平台的特性详细介绍基于 im2win 变换的张量卷积实现及优化。

2.3 im2win 变换算法的对比分析

根据上述 im2col 变换和 im2win 变换的过程，本节对变换得到的 im2win 张量和 im2col 矩阵的内存占用进行分析对比。假设输入张量 $\mathcal{I}$ 的大小为 $N_i \times C_i \times H_i \times W_i$，卷积核张量 $\mathcal{F}$ 的大小为 $N_f \times C_f \times H_f \times W_f$，步长为 s（$H_f$ 和 W_f 都不小于 s）。根据上述算法 2，im2win 张量 $\hat{\mathcal{I}}$ 的大小如下式所示：

$$\hat{\mathcal{I}} = \begin{cases} N_i \times C_i \times H_f \times W_i \times (\dfrac{H_i - H_f}{s} + 1), & H_f > s \\ N_i \times C_i \times H_i \times W_i, & H_f = s \end{cases} \tag{2-4}$$

当 $H_f > s$ 时，im2win 张量 $\hat{\mathcal{I}}$ 的大小为 $N_i \times C_i \times H_f \times W_i \times ((H_i - H_f)/s + 1)$；而当 $H_f = s$ 时，im2win 张量 $\hat{\mathcal{I}}$ 的大小等于输入张量 $\mathcal{I}$ 的大小。此外，im2win 张量 $\hat{\mathcal{I}}$ 的大小与 W_f 的值无关。根据上文 2.1.3 节中详细介绍的 im2col 变换的过程，im2col 矩阵 M 的大小如下式所示：

$$M = \begin{cases} N_i \times C_i \times H_f \times W_f \times (\dfrac{H_i - H_f}{s} + 1) \times (\dfrac{W_i - W_f}{s} + 1), & H_f > s \,||\, W_f > s \\ N_i \times C_i \times H_i \times W_i, & H_f = W_f = s \end{cases} \tag{2-5}$$

当 $H_f > s$ 时或 $W_f > s$ 时，通过 im2col 变换得到的 im2col 矩阵 M 的大小为 $N_i \times C_i \times H_f \times W_f \times ((H_i - H_f)/s + 1) \times ((W_i - W_f)/s + 1)$；当 $H_f = W_f = s$ 时，im2col 矩阵 M 的大小等于输入张量 $\mathcal{I}$ 的大小。

根据上述分析，假设 $H_f = W_f$，则 im2col 矩阵 M 与 im2win 张量 $\hat{\mathcal{I}}$ 的差异 δ 如下式所示：

$$\begin{aligned} \delta &= N_i \times C_i \times H_f (W_f \times (\frac{H_i - H_f}{s} + 1) \times (\frac{W_i - W_f}{s} + 1) - W_i \times (\frac{H_i - H_f}{s} + 1)) \\ &= N_i \times C_i \times H_f \times (\frac{H_i - H_f}{s} + 1) \times (W_i - W_f) \times (\frac{W_f}{s} - 1) \end{aligned} \tag{2-6}$$

im2col 矩阵 M 与 im2win 张量 $\hat{\mathcal{I}}$ 的比值 ρ 如下式所示：

$$\begin{aligned}\rho &= \frac{N_i \times C_i \times H_f (W_f \times (\frac{H_i - H_f}{s} + 1) \times (\frac{W_i - W_f}{s} + 1)}{N_i \times C_i \times H_f \times W_i \times (\frac{H_i - H_f}{s} + 1))} \\ &= \frac{W_f}{W_i} \times (\frac{W_i - W_f}{s} + 1)\end{aligned} \tag{2-7}$$

其中 $H_o, H_f, C_i, W_i, W_f, s$ 均大于 0，且 $W_i \geq W_f$。如式 2-6 所示，当 $H_f = W_f = s$ 时，得到 $\delta = 0$，即 im2win 张量和 im2col 矩阵的大小相同；当 $H_f = W_f > s$ 时，得到 $\delta > 0$，即 im2col 矩阵的大小大于 im2win 张量的大小。因此，无论何种情况下，im2win 张量的大小都要小于等于 im2col 矩阵的大小。两者的比值 ρ 如式 2-7 所示，当 s 越小时，im2col 矩阵 M 和 im2win 张量 $\hat{\mathcal{I}}$ 的比值 ρ 就越大，意味着 im2col 矩阵 M 要比 im2win 张量 $\hat{\mathcal{I}}$ 花费更多的内存。图 2-5 说明了 im2col 变换和 im2win 变换的过程对比，其中输入张量 $\mathcal{I}$ 的大小为1×3×3×3，卷积核张量 $\mathcal{F}$ 的大小为1×3×2×2，步长 $s = 1$。从图中可以看出，im2col 矩阵中有 48 个元素，其中有大量的冗余重复元素，而 im2win 张量中只有 36 个元素，减少了冗余的程度。在这个简单的例子中，im2win 张量相比于 im2col 矩阵除了有更好的数据局部性外，还降低了1/3的内存存储。

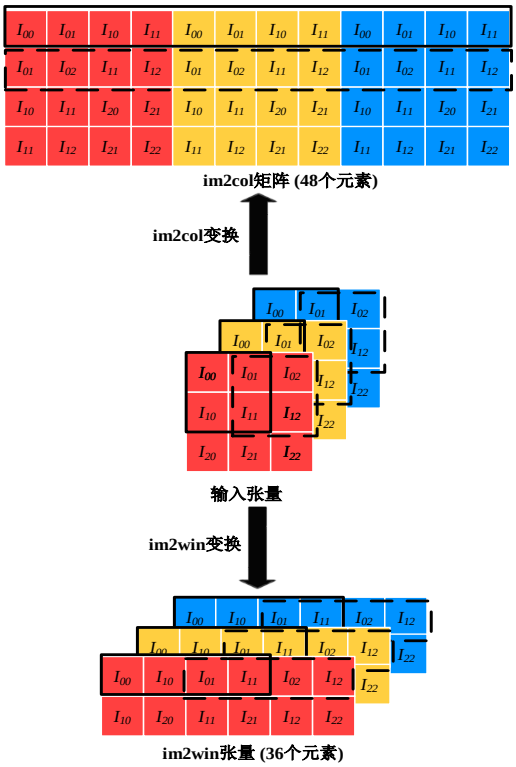

图 2-5 im2col 变换与 im2win 变换对比

2.4 本章小结

本章详细介绍了目前主流的卷积算法的实现过程并分析了它们的优缺点，包括直接卷积算法，基于 im2col 变换的卷积算法。针对现有卷积算法的缺点，提出了一种新的基于窗口序的张量卷积算法，该算法包含了新设计的 im2win 变换和张量乘法。本章还对 im2win 变换得到的 im2win 张量和传统的 im2col 变换得到的 im2col 矩阵的大小进行分析对比，展示了本文提出的 im2win 变换相较于传统的 im2col 变换能够提供更好的数据局部性并有效降低内存消耗。

第 3 章 基于 im2win 变换的张量卷积在 CPU 平台上的优化实现

上一章介绍了卷积操作和算法的基本概念和原理，分析了传统算法的问题和不足，并提出了一种基于张量代数的卷积算法，该卷积算法利用一种新的 im2win 变换和张量乘法来实现高效灵活的卷积操作。本章的目的是利用上文提出的基于 im2win 变换的张量卷积算法，在 CPU 平台上实现高效的张量卷积算法，并对其进行优化和测试。首先介绍了 CPU 硬件体系结构、SIMD 向量指令集和 CPU 并行技术的基本概念；然后详细描述了基于 im2win 变换的张量卷积算法及其在 CPU 上的优化策略，针对 CPU 架构特点，本文建立了一个优化分析模型，并采用了 SIMD 向量指令集、多线程并行、循环展开等技术来加速矩阵乘法和充分利用 CPU 资源。最后通过一个完备的测试基准，对比了该算法与现有方法在不同网络结构和输入规模下的性能差异，并分析了该算法的通用性和有效性。实验结果表明，该算法在内存消耗和计算性能方面均优于 CPU 平台上常见的深度学习库中的卷积算法实现。

3.1 CPU 平台的体系结构

随着计算机科学和技术的不断发展，CPU 平台的体系结构也在不断演进。CPU 的体系结构对计算机系统的性能和功耗都具有至关重要的影响。在现代计算机中，CPU 通常是整个系统的核心部件，承担着执行各种指令以完成计算和控制任务的重要职责。从硬件结构角度来看，CPU 通常由运算器、控制器、寄存器和缓存等组成，实现高性能卷积算法的关键在于如何让这些组件之间高效协作。SIMD 向量指令集作为一种并行计算技术，可以同时对多个数据进行相同的操作，从而提高卷积算法的计算效率。因此，在许多科学计算和图形应用中，SIMD 向量指令集受到了广泛应用。此外，随着计算机体系结构的不断演变，CPU 并行技术也越来越重要。利用多核 CPU 的并行计算能力，可以使卷积算法实现更快的计算速度和更高的能效比。

3.1.1 CPU 的硬件结构

中央处理器（CPU）是计算机系统最为核心的组成部分之一，主要承担计算机系统中命令执行、数据流控制和算术逻辑处理等任务。其硬件结构通常由控制单元、算术逻辑单元、高速缓存和寄存器等主要组件构成。在实现 CPU 平

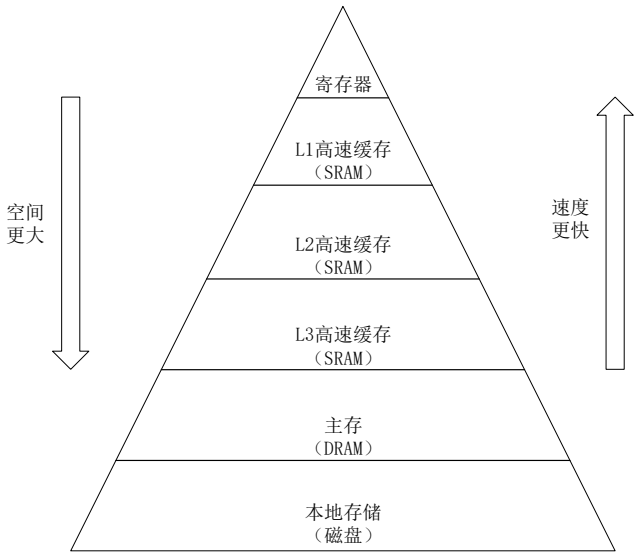

图 3-1 内存的层次结构

台上的张量卷积算法时，运算器与存储器之间的高效协作是取得算法高性能的关键，即 CPU 中算术逻辑单元（ALU）、高速缓存和寄存器间的相互配合协作。ALU 作为 CPU 的“大脑”，主要负责各种算术和逻辑运算，如加、减、乘、除、与、或、非等运算，其输入来自寄存器，输出结果也保存在寄存器中。在卷积操作中，浮点数的乘法和加法运算是主要的运算，而 ALU 则能够高效地完成这些运算。在现代 CPU 中，ALU 的运算能力已经相当强大，因此在像卷积操作这样的计算密集型操作中，CPU 的浮点计算能力通常并不是性能瓶颈。相反，数据从主存搬运到寄存器中的速度往往无法跟上浮点计算的速度，因此对于卷积算法的性能优化而言，合理分配存储器并优化数据加载方式更为重要。在现代计算机系统中，内存的层次结构包括本地储存、主存、高速缓存和寄存器等多个层次（如图 3-1 所示），这些存储器的空间大小和访问延迟依次减小。内存层次结构的特性对于卷积算法设计有着显著的影响，这些内存储存层次之间以及与 CPU 之间的数据传输速度不同，因此在选择合适的内存层次进行数据存储时需要考虑到访问延迟和带宽等方面的限制。在卷积算法中，由于局部性原理的存在，使用高速缓存可以大大降低访问延迟，并提高算法的性能。同时，将数据尽可能地保存在寄存器中也是提高算法性能的一种有效手段。然而，高速缓存容量和成本的限制也会影响算法的设计，因为当数据量超出高速缓存容量时，可能需要频繁地从主存中获取数据，进而带来额外的访问延迟。因此，在后续卷积算法设计和优化过程中，本文根据内存层次结构的特点和限制来优化算法访问内存的方式，以达到最优的算法性能。

3.1.2 SIMD 向量指令集

单指令多数据流（SIMD）向量指令集是一种计算机处理器指令集，其设计思想是将多个数据元素打包成一个向量，以单条指令的方式同时对这些向量进行操作，从而实现高效的并行计算（如图 3-2 所示）。相比于传统的标量指令，SIMD 指令集能够更好地利用处理器中的硬件资源和执行单元，提高计算效率和吞吐量，特别是在并行执行卷积操作这种部分数据类似或相同的运算时。

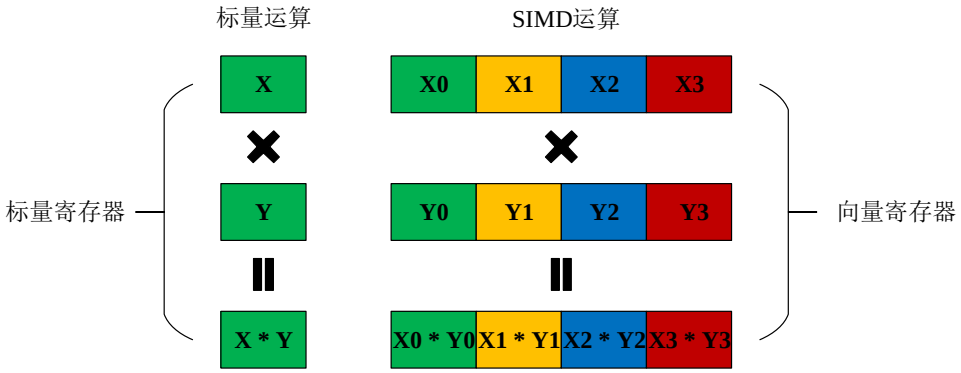

图 3-2 标量运算与 SIMD 运算对比

流式 SIMD 扩展（SSE）和高级向量扩展（AVX）是 Intel x86 架构上广泛使用的两种 SIMD 指令集（如表 3-1 所示）。它们都支持向量化运算，使得数值密集型计算任务可以显著提升性能。（1）SSE 指令集是第一个适用于 x86 平台的 SIMD 指令集，该指令集支持向量长度为 128 位，并提供了多种针对浮点数、整数、字符等数据类型的算术逻辑运算、数学函数、移位和数据加载/存储等指令。SSE2、SSE3、SSSE3、SSE4 以及 SSE4.2 版本在此基础上增加了更多的功能和优化，如扩展了整数和浮点数运算范围、增加了更高效的数据访问模式、引入了新的算术逻辑运算方式等，并且还改进了先前指令集存在的缺陷和限制；（2）AVX 指令集是 SSE 指令集的扩展，在保留了 SSE 所有功能和兼容性的同时，支持更长的向量长度（256 位），新增了更多丰富而强大的功能和优化。AVX 最大的亮点是 Fused Multiply-Add（FMA）指令，可在一条指令内完成乘加法操作，并且减少了中间结果舍入误差，从而提高浮点运算效率和精度。此外，AVX 还新增了更多针对浮点数、整数等数据类型的算术逻辑运算指令、向量处理指令、数据加载和存储指令等，并且引入了更高级灵活方便地寄存器命名方式，如 YMM 寄存器。

SSE 和 AVX 都采用相同地操作方式——将多个数据元素打包成一个向量，并通过单条或少条 SIMD 指令同时对这些向量进行操作。在程序中使用这些 SIMD 指示进行编程时有三种方法：（1）编译器自动向量化技术；（2）Intrinsics 函数；（3）Assembly 汇编语句。为了获得最佳性能和效率，在使用 SIMD 指令进行编程时需要根据具体应用场景选择合适地向量长度和 SIMD 指令，并对原

始代码进行适当地重构与优化。

表 3-1 部分 SIMD 向量指令集的特性

指令集	向量寄存器长度	特性
SSE	128bit	只支持单精度浮点数运算
SSE2	128bit	增加整型数据支持和双精度浮点数运算
SSE3	128bit	增加浮点数到整数转换，对超线程技术的支持
AVX	256bit	支持 256 位浮点运算，FMA 指令
AVX2	256bit	增加 256 位整型数据支持，三运算指令
AVX512	512bit	支持 512 位运算

3.1.3 CPU 并行加速技术

随着科学计算、机器学习和数据分析等领域中计算任务规模和复杂度的不断增加，需要大量的计算资源来满足需求。然而，传统的串行 CPU 已经无法满足这一需求，因为它们只能按照指令流水线的顺序逐个执行指令，无法同时处理多个指令或数据块。为了提高计算效率，多种 CPU 并行加速技术被提出，如指令级并行加速、线程级并行加速和数据级并行加速等。这些技术旨在通过同时执行多个计算任务以最大限度地利用 CPU 资源，从而提高计算效率，满足超大规模复杂计算任务的需求。

（1）指令级并行加速

指令级并行加速（Instruction-Level Parallelism，ILP）[89][90]旨在通过同时执行多条指令来提高计算效率。尽管现代计算机中 CPU 的时钟频率已经达到了较高水平，但是由于单个指令需要访问多个数据存储器，因此每个指令在执行期间都会有等待时间，从而降低了 CPU 的利用率。为了充分利用 CPU 资源，提高计算效率，指令级并行加速技术应运而生。在指令级并行加速中，CPU 中的不同功能单元同时执行指令，从而最大限度地提高计算效率。如果 CPU 中不同功能单元执行的指令操作之间没有数据依赖，则这些操作指令可以同时并行执行。通过乱序执行、超标量执行等技术，CPU 可以在一个时钟周期内执行多个指令，从而提高计算效率。由于张量卷积操作是计算密集型操作，因此在设计张量卷积操作的微内核时，需要结合 CPU 指令级并行的特点排列微内核中指令操作的布局，以使微内核在执行时可以同时并行处理取出的多条指令操作，从而最大限度地提高 CPU 资源的利用率。

（2）线程级并行加速

线程级并行加速（Thread-Level Parallelism，TLP）[91][92]的主要目的是通过

同时执行多个线程来提高计算机系统的效率。随着现代计算机中多核处理器的普及，每个核心可以同时执行一个或多个线程，因此线程级并行加速技术成为了提高计算效率的重要手段。相对于指令级并行加速，线程级并行加速更注重并行处理多个任务或者应用程序的能力。在一个多核处理器中，每个核心都有自己的线程调度器，负责管理和调度该核心上正在运行的线程。本文中所使用的多线程编程环境是 OpenMP[93]将张量卷积算法转换为一个适合并行执行的模型，并使用 OpenMP 的 Fork/Join 模式实施并行化（如图 3-3 所示）。其中，Fork 阶段将任务进行分解，并创建相应的子线程，各个子线程并行执行对应的计算任务；Join 阶段则将各个子线程的计算结果合并起来。需要注意的是，在对张量卷积算法使用线程级并行加速技术时，需要解决线程之间的数据共享和同步问题。因此，在实施并行化前，需要仔细考虑数据共享的情况，并选择合适的同步策略以确保并行化方案的正确性。通过这种方式，可以充分利用多核处理器的能力，并在保证正确性的前提下，提高卷积操作计算的效率和系统的响应速度。

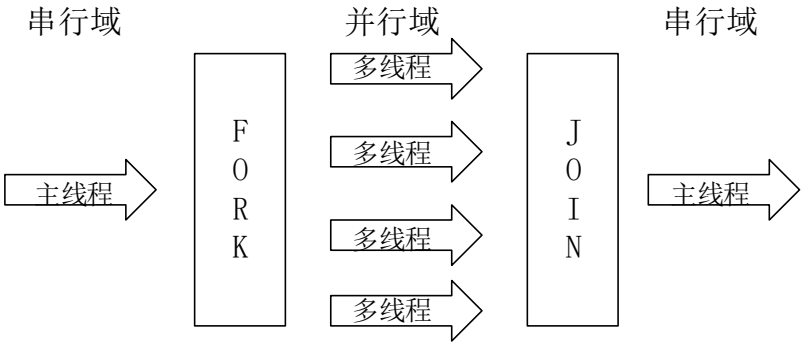

图 3-3 Fork/Join 并行执行模式

（3）数据级并行加速

数据级并行加速（Data-Level Parallelism，DLP）[94][95]是一种利用多个处理单元同时执行同一个操作的并行计算技术。在数据级并行加速中，大量的数据被切分成若干个小块，每个小块都可以被独立地操作。这些小块可以分配给不同的处理单元进行并行计算，从而有效地利用了多核处理器的能力。相对于指令级并行加速和线程级并行加速，数据级并行加速更注重并行化处理数据的能力。数据级并行加速常用于需要处理大规模、密集型的数据集的应用程序中，如图像处理、信号处理和机器学习等领域。在这些领域中，大量的数据需要进行复杂的运算和变换，并且这些运算和变换通常都可以被分割成许多小的、独立的操作。通过将这些小操作分配给不同的处理单元并行执行，可以显著提高程序的效率，缩短运行时间。在本文设计实现的张量卷积算法中，在张量数据的批次维度对数据进行了初步划分，之后将不同列的连续点乘窗口作为基本单

位划分数据块，在确保不同线程的操作之间没有数据依赖的前提下尽可能得提高单个线程的连续操作的数据重用。

3.2 CPU 上张量卷积算法的实现及优化

在之前的章节 2.2 中，详细介绍了基于 im2win 的张量卷积算法，包括内存更高效的 im2win 数据变换的过程和算法，以及基于 im2win 的张量卷积算法的数学定义。本节中将主要介绍在 CPU 平台上对 im2win 变换得到的 im2win 张量进行卷积操作运算的具体实现及优化。

3.2.1 基于 im2win 变换的张量卷积实现

根据算法 3 所定义的基于 im2win 变换的张量卷积，该算法可被 CPU 上的串行结构有效映射，因而使得实现此算法的 CPU 串行逻辑与算法 3 的基本结构高度一致，如算法 4 所示。该算法采用嵌套多个 for 循环的结构，通过不同层循环的迭代值来计算 im2win 张量$\hat{\mathcal{I}}$、卷积核张量$\mathcal{F}$和输出张量$\mathcal{O}$的索引值，随后执行乘加运算，并将结果存储至相应位置。在算法 4 中，首先通过上文提出的 im2win 变换算法（如算法 2 所示）将输入张量$\mathcal{I}$变换为按连续点乘窗口顺序排列的 im2win 张量$\hat{\mathcal{I}}$。接着，通过外部的四层 for 循环的迭代值获取输出张量$\mathcal{O}$的四个维度对应的索引值（算法 4 中第 1 行至第 4 行）。之后，通过内部的三层 for 循环的迭代值获取卷积核张量$\mathcal{F}$的索引值（算法 4 中第 5 行至第 7 行）。最后，在最内层的循环中，执行乘加操作并存储计算结果至输出张量$\mathcal{O}$中对应的位置（算法 4 中第 8 行）。算法 3 中实现的基于 im2win 变换的张量卷积是在 CPU 上基础的版本，在保证计算结果正确的前提下采取了简单直观的设计。本文将在此基础上进行一系列改进与优化，并与基础版本进行了实验比对分析。

3.2.2 基于 im2win 变换的张量卷积算法优化设计

上节提出的基础的基于 im2win 变换的张量卷积算法存在许多局限性和瓶颈，计算效率和访存效率都不高。因此，在本节中将详细介绍如何对基础版本的基于 im2win 变换的张量卷积算法进行优化设计，从分析模型、循环重排列、循环展开、向量化、FMA 指令、存储器分块和并行策略等多个方面来提出有效的优化设计方案，最终实现高性能的基于 im2win 变换的张量卷积算法，并使其适应于不同的卷积场景。

（1）张量卷积算法的分析模型

本文建立了一个基于硬件特性的张量卷积的分析模型，以指导高性能的张

量卷积算法的设计。张量卷积是一种计算密集型操作，它与矩阵乘法有类似之处，因此可以参照矩阵乘法的分析模型建立张量卷积的分析模型。本文建立的张量卷积的分析模型架构类似于高性能矩阵乘法的分析模型架构[96]，该分析模型从向量寄存器、向量操作指令、内存层次结构和带宽方面进行设计分析。该分析模型架构具有以下特点：

——向量寄存器。本文假设分析模型使用单指令多数据流（SIMD）向量指令集，这意味着每个指令同时在多个标量数据上执行操作。假设一共有 N_{REG} 个向量寄存器，每个向量寄存器都可以存储 N_{VEC} 个浮点元素（在不支持 SIMD 向量指令集的设备上，N_{VEC} 等于 1，这意味着只能进行标量运算），因此寄存器可以保留的元素数量为 $N_{REG}N_{VEC}$。

——FMA 指令。假设分析模型存在 N_{FMA} 个支持融合乘加指令（FMA）的计算单元，每个 FMA 指令都会计算一个乘法和一个加法操作，即处理器浮点计算单元在每个指令周期可以进行 $2N_{VEC}N_{FMA}$ 次浮点运算（单个 FMA 指令为 2 次浮点运算）。每个 FMA 计算单元都可以在每个计算周期计算一个 FMA 指令，但每个 FMA 指令具有 L_{FMA} 个周期的延迟，这是两个相互依赖的连续 FMA 指令之间的最小周期数。这意味着自发出 FMA 指令之后，必须经过 L_{FMA} 个周期才能发出后续依赖的 FMA 指令。

——加载/存储架构。假设分析模型的架构是一种加载/存储架构，在对数据进行操作之前，必须将数据先加载到寄存器中。

——带宽。假设分析模型中高速缓存和寄存器之间的带宽足够大，不会成为张量卷积计算中的性能瓶颈。

算法 4： CPU 上基础的基于 im2win 变换的张量卷积算法

输入： 输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出： 输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量： $\hat{\mathcal{I}} = im2win(\mathcal{I}, \mathcal{F}, s)$

for $i = 0$ **to** $N_o - 1$ **do**
  for $j = 0$ **to** $C_o - 1$ **do**
    for $m = 0$ **to** $H_o - 1$ **do**
      for $n = 0$ **to** $W_o - 1$ **do**
        for $r = 0$ **to** $C_f - 1$ **do**
          for $u = 0$ **to** $H_f - 1$ **do**
            for $v = 0$ **to** $W_f - 1$ **do**
              $\mathcal{O}[i][j][m][n] += \hat{\mathcal{I}}[i][r][m][n \times s \times W_f + v \times H_f + u] \times \mathcal{F}[j][r][u][v]$

（2）基于循环重排列和循环展开优化的张量卷积

基础的基于 im2win 变换的张量卷积由 7 个嵌套的 for 循环构成（如算法 4 所示），其中每一个 for 循环都是相互独立的，因此可以任意改变循环的顺序而不会影响计算结果。对于整个算法中的三个不同张量，即 im2win 张量$\hat{\mathcal{I}}$、卷积核张量$\mathcal{F}$和输出张量$\mathcal{O}$而言，访问代价由大到小排列通常是$\hat{\mathcal{I}}>\mathcal{O}>\mathcal{F}$。对于同一个张量的不同维度而言，不同维度的访问代价也存在差异（在 2.2.1 中详细分析了不同维度的访问代价）。为了提高张量卷积操作中加载和存储数据的性能，应该优先考虑计算访问代价较低的维度，并且保持连续的内存访问顺序。因此，基础的基于 im2win 的张量卷积中的循环顺序被重新设计（如算法 5 中所示）。

算法 5：向量化与循环重排列优化的基于 im2win 变换的张量卷积算法

输入：输入张量$\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核$\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量$\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量： $\hat{\mathcal{I}}=im2win(\mathcal{I},\mathcal{F},s)$

```
1: for j = 0 to C_o − 1 do
2:     for i = 0 to N_o − 1 do
3:         for m = 0 to H_o − 1 do
4:             for r = 0 to C_f − 1 do
5:                 for n = 0 to W_o − 1 do
6:                     for v = 0 to W_f − 1 do
7:                         DOT_PRODUCT(j,i,r,m,n,v,s)
8: 函数定义 DOT_PRODUCT(j,i,r,m,n,v,s):
9:     for u = 0 to H_f / N_vec − 1 do
10:        iu = n × s × W_f + v × H_f + u
11:        SIMD_LOAD(Î[i][r][m][iu : iu + N_vec])
12:        for u = 0 to H_f / N_vec − 1 do
13:            SIMD_LOAD(F[j][r][v][u : u + N_vec])
14:            FMA(Î,F,O[i][j][m][n])
15:    SIMD_STORE(O[i][j][m][n])
```

在基础的基于 im2win 变换的张量卷积中（如算法 4 所示），im2win 张量$\hat{\mathcal{I}}$的访问代价是三个主要张量中最昂贵的一个，所以应该尽量减少对它的访问次数。由于 im2win 张量$\hat{\mathcal{I}}$与输出张量$\mathcal{O}$的C_o维度和卷积核张量$\mathcal{F}$的所有维度相互独立，因此，可以将加载 im2win 张量$\hat{\mathcal{I}}$数据的操作上提到更外层的循环中，保持 im2win 张量$\hat{\mathcal{I}}$数据在寄存器中直到不再使用（算法 5 中第 9 行至第 11 行）。

这样一来可以减少对 im2win 张量$\hat{\mathcal{I}}$访存次数并提升计算效率，这种优化技术叫做 hoist[97]。在最内层的循环中，加载了卷积核张量$\mathcal{F}$中的数据后（算法 5 中第 13 行），执行 FMA 指令完成乘加操作（算法 5 中第 14 行）。

在确定张量卷积的整体循环顺序后，可以考虑对单个循环进行循环展开优化。循环展开[98]可以减少缓存/内存的读取次数，同时会减少处理器在每次迭代中产生的错误分支预测的数量。在基础的基于 im2win 变换的张量卷积中的最内层循环具有循环独立性（算法 4 中第 8 行），这意味着循环中每一个迭代都不会依赖于它的前一个迭代，因此对张量卷积中的最内层循环进行循环展开（算法 5 中第 13 行至第 14 行）。为了算法的简洁起见，算法 5 中没有明确显示具体的循环展开策略。

（3）基于向量化和 FMA 指令优化的张量卷积

本文的分析模型中有N_{REG}个大小为N_{VEC}的向量寄存器，有N_{FMA}个单元用于计算 FMA 指令，每个 FMA 指令的延迟为L_{FMA}个时钟周期。当模型达到最大性能时，每个 FMA 计算单元必须至少发出L_{FMA}个独立的 FMA 指令，才不会在张量卷积操作的浮点运算指令流水线中引入停顿。由于每个 FMA 指令可以计算N_{VEC}个输出元素，这意味着模型达到最大性能时如下式所示：

$$\theta \geq N_{VEC} N_{FMA} L_{FMA} \tag{3-1}$$

其中θ是每个时钟周期L_{FMA}中必须计算的独立输出元素的最小值。

在本文的实现中，使用了 AVX2 指令集。该指令集具有 16 个逻辑寄存器，每个寄存器可以存储 256 位的数据。如果操作中使用的基本数据是 32 位的单精度浮点数，则每个寄存器可以容纳 8 个元素。这意味着每个 SIMD 指令可以同时操作 8 个单精度数。在基础的基于 im2win 变换的张量卷积中，最内层每次执行一个标量操作（算法 4 中第 8 行），因此可以使用 SIMD 寄存器使得操作向量化。由于向量寄存器是根据连续内存中的数据进行加载操作，所以在H_f维度上进行向量化[99]（算法 5 中第 9 行和第 12 行）。为了获得更好的向量化性能，可以选择将循环按照W_f和H_f维度一起排列，以便于一次加载一个点乘窗口的数据。由于通过 im2win 变换算法将输入张量$\mathcal{I}$变换为根据连续点乘窗口排序的 im2win 张量$\hat{\mathcal{I}}$，所以大多数窗口元素在向量寄存器中加载数据时都可以被重复利用，这极大地提高了$\hat{\mathcal{I}}$中元素的可重用性和缓存命中率。

（4）基于存储器分块优化的张量卷积

寄存器分块：如式 3-1 所示，维持模型 SIMD 架构系统峰值性能所需的最小输出元素数目为θ。θ受到所有向量寄存器[100]中可保留的元素数量的限制，如下式所示：

$$\theta \le N_{REG} N_{VEC} \tag{3-2}$$

为了实现算法的最大性能，需要尽可能多得将张量元素保存在寄存器中，并最小化 SIMD 向量加载次数。因此可以采取寄存器分块将多个点乘窗口得元素分别加载到寄存器中。由于 im2win 张量$\hat{\mathcal{I}}$得点乘窗口按$N_i \times C_i \times H_o \times W_o \times W_f \times H_f$排序，因此在寄存器级别上考虑三个连续循环$W_o \times W_f \times W_h$。在算法 6 中，对$W_o$和$W_f$维度上应用寄存器分块，随后在卷积核的$H_f$维度上执行 SIMD 向量化加载（算法 6 中 11 行和 13 行）。

算法 6：CPU 上高性能的基于 im2win 变换的张量卷积算法

输入：输入张量$\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核$\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量$\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量： $\hat{\mathcal{I}} = im2win(\mathcal{I}, \mathcal{F}, s)$

1: **for** $jj = 0$ **to** $C_o/C_{o,b}$-1 **in parallel do**
2: **for** $i = 0$ **to** $N_o - 1$ **do**
3: **for** $m = 0$ **to** $H_o - 1$ **do**
4: **for** $r = 0$ **to** $C_f - 1$ **do**
5: **for** $nn = 0$ **to** $W_o/W_{o,b}$-1 **do**
6: **for** $vv = 0$ **to** $W_f/W_{f,b}$-1 **do**
7: $DOT\_PRODUCT(jj, i, r, m, nn, vv, s)$
8: **函数定义** $DOT\_PRODUCT(jj, i, r, m, nn, vv, s)$:
9: **for** $n' = 0$ **to** $W_{o,b} - 1$ **do**
10: **for** $v' = 0$ **to** $W_{f,b} - 1$ **do**
11: **for** $u = 0$ **to** $H_f / N_{vec} - 1$ **do**
12: $iu = (nn \times W_{o,b} + n') \times s \times W_f + (vv \times W_{f,b} + v') \times H_f + u$
13: $SIMD\_LOAD(\hat{\mathcal{I}}[i][r][m][iu : iu + N_{vec}])$
14: **for** $j' = 0$ **to** $C_{o,b} - 1$ **do**
15: $SIMD\_LOAD(\mathcal{O}[i][jj \times C_{o,b} + j'][m][nn])$
16: **for** $v' = 0$ **to** $W_{f,b} - 1$ **do**
17: **for** $u = 0$ **to** $H_f / N_{vec} - 1$ **do**
18: $SIMD\_LOAD(\mathcal{F}[jj \times C_{o,b} + j'][r][vv \times W_{f,b} + v'][u : u + N_{vec}])$
19: $FMA(\hat{\mathcal{I}}, \mathcal{F}, \mathcal{O}[i][jj \times C_{o,b} + j'][m][nn])$
20: $SIMD\_STORE(\mathcal{O}[i][jj \times C_{o,b} + j'][m][nn])$

高速缓存分块：在具有更多级别的内存层次结构中，例如具有高速缓存的

架构中，因此可以进一步将 im2win 张量$\hat{\mathcal{I}}$划分为更小的分区，以使其适合高速缓存的分块级别。由于张量卷积操作过程是在输出张量$\mathcal{O}$的C_o维度上不断迭代新的卷积核$\mathcal{F}$中的窗口，并与$\hat{\mathcal{I}}$进行点乘运算，因此提高了$\hat{\mathcal{I}}$中元素的数据重用率。在使用存储器分块优化的算法中选择将迭代输出通道维度C_o划分为较小的分区以适应下一级层次的内存（算法 6 中第 14 行）。

（5）基于并行策略优化的张量卷积

在并行计算中，“易并行计算（an embarrassingly parallel）”[101]被定义为可以轻松将整个任务分成许多并行任务的工作量。原则上，可以独立处理的所有循环都可以并行化处理。为了设计并行算法，必须确定可以独立处理的循环，观察到在算法 5 中，所有输出张量$\mathcal{O}$中元素彼此独立且可以进行易并行计算处理。由于输出是一个四维张量$\mathcal{O}(N_i, C_o, H_o, W_o)$，这意味着可以选择任何一个四个维度中的一个进行并行处理。为了在并行化后平衡各个线程之间的负载，应选择尺寸较大的维度划分并行策略。在并行的基于 im2win 变换的张量卷积实现中是从输出通道C_o维度上并行化的（算法 6 中第 1 行），因为这个维度是卷积层中可以预先设计的参数，在增加卷积层深度时有助于多个线程之间的负载平衡。每个线程被分配一个输出张量$\mathcal{O}$的数据块，大小为$H_o \times W_o \times C_o / t$，其中$t$是使用的线程数目。

本节对基于 im2win 变换的张量卷积实现优化提供了一系列策略，首先进行了分析建模，之后将循环结构映射到对应的平台架构上，对数据处理和数值运算向量化、存储器分块和并行策略等优化，最终实现了一个高性能的基于 im2win 变换的张量卷积（如算法 6 所示）。

3.3 实验与结果分析

上一节详细介绍和分析了基于 im2win 变换的张量卷积算法在 CPU 上的实现及其后续的一系列高性能优化。本节将对实现的算法进行计算性能和内存占用的验证实验，并采用一个完备的测试基准，以便对其进行全面分析并得到更可靠的结果。

3.3.1 测试实验的整体方案

表 3-2 CPU 平台软件环境的版本型号

软件环境	版本型号
PyTorch	1.10.0a0
OpenMP	4.5.0
GCC	7.5.0

表 3-3 由 12 个卷积层组成的卷积神经网络测试基准

卷积层	输入 $C_i \times H_i \times W_i$	卷积核，步长 $C_o \times H_f \times W_f, s_h(s_w)$	输出 $C_o \times H_o \times W_o$
Conv1	$3\times227\times227$	$96\times11\times11,4$	$96\times55\times55$
Conv2	$3\times231\times231$	$96\times11\times11,4$	$96\times56\times56$
Conv3	$3\times227\times227$	$64\times7\times7,2$	$64\times111\times111$
Conv4	$64\times224\times224$	$64\times7\times7,2$	$64\times109\times109$
Conv5	$96\times24\times24$	$256\times5\times5,1$	$256\times20\times20$
Conv6	$256\times12\times12$	$512\times3\times3,1$	$512\times10\times10$
Conv7	$3\times224\times224$	$64\times3\times3,1$	$64\times222\times222$
Conv8	$64\times112\times112$	$128\times3\times3,1$	$128\times110\times110$
Conv9	$64\times56\times56$	$64\times3\times3,1$	$64\times54\times54$
Conv10	$128\times28\times28$	$128\times3\times3,1$	$128\times26\times26$
Conv11	$256\times14\times14$	$256\times3\times3,1$	$256\times12\times12$
Conv12	$512\times7\times7$	$512\times3\times3,1$	$512\times5\times5$

（1）硬件环境

本文 CPU 上的测试实验是在 2 个主频为 2.20GHz 的 Intel Xeon Silver 4214 处理器上进行，进行实验时一共有 24 个核心（每个处理器有 12 个核心）。每个处理器中有三级高速缓存，L1 缓存的大小为 32KB、L2 缓存的大小为 1MB 以及 L3 缓存的大小为 16.5MB。

（2）软件环境

进行测试对比的算法包括本文提出的基于 im2win 变换的张量卷积算法，PyTorch 中的基于 im2col 变换的卷积算法以及简单实现的直接卷积算法。在测试实验中，基于 im2win 变换的张量卷积算法使用 C++代码实现并使用 OpenMP 环境进行并行，数据结构使用 PyTorch 中定义的张量数据类型，每个张量中的元素都是单精度浮点型。所有的算法在测试时使用 GCC 编译，并且添加“-Ofast –march=native”编译选项。软件环境的具体版本型号如表 3-2 所示。

（3）测试基准

本文期望对常用的卷积神经网络中大多数的卷积层进行测试实验，然而如果选择某一个神经网络模型作为测试基准无法覆盖大多数的卷积层，例如 VGG-16[27]神经网络模型中所有的卷积层的卷积核大小都为 3×3，而 ResNet-50[29]神经网络模型中只包含三种不同尺寸的卷积核。因此，在本文的测试实验中，使用了一个十分完备的卷积神经网络测试基准。这个测试基准中包含 12 个参数不同的独特卷积层，具体的参数如表 3-3 所示。

（4）评价指标

在实验测试中主要对比分析不同卷积算法之间运算性能和内存占用这两方面的测试结果。运算性能除了运行时间这个指标外，本文选取每秒浮点计算次数（floating-point operations per second, FLOPS）作为测试实验的速度性能评价指标。其计算公式如下式所示：

$$FLOPS = \frac{float\ ops}{t} \tag{3-3}$$

其中，t 是卷积操作的运行时间，$float\ ops$ 是卷积操作的总的浮点运算次数（乘法和加法操作次数的总和），可以通过卷积层的参数计算得出，其计算公式如下式所示：

$$float\ ops = C_o \times H_o \times W_o \times (2 \times C_i \times H_f \times W_f) \tag{3-4}$$

（5）实验方法

在确定了实验的软硬件环境、测试基准和评价指标后，本文选择 4 种卷积算法的实现进行测试，包括基于 im2col 变换的卷积算法、直接卷积算法和基于 im2win 变换的张量卷积算法，其中基于 im2win 变换的张量卷积算法包括基础版本和进行一系列优化后的高性能版本。在实验中，使用 C++标准库的 wall-clock 函数来记录不同算法完成卷积的运行时间，为了避免处理器运行时出现的偶然误差，每个算法在测试基准的每个层上运行 5 次，并选取 5 次运行中最佳的运行时间记录。在测试基准中 12 个不同的卷积层上运行卷积算法时，为了使处理器中每个线程在运行时都有一定的负载工作量，选取输入张量 $\mathcal{I}$ 的批次维度 N_i 大小为 128，在 PyTorch 框架中生成每个卷积层对应参数的张量数据。

3.3.2 运算性能结果分析

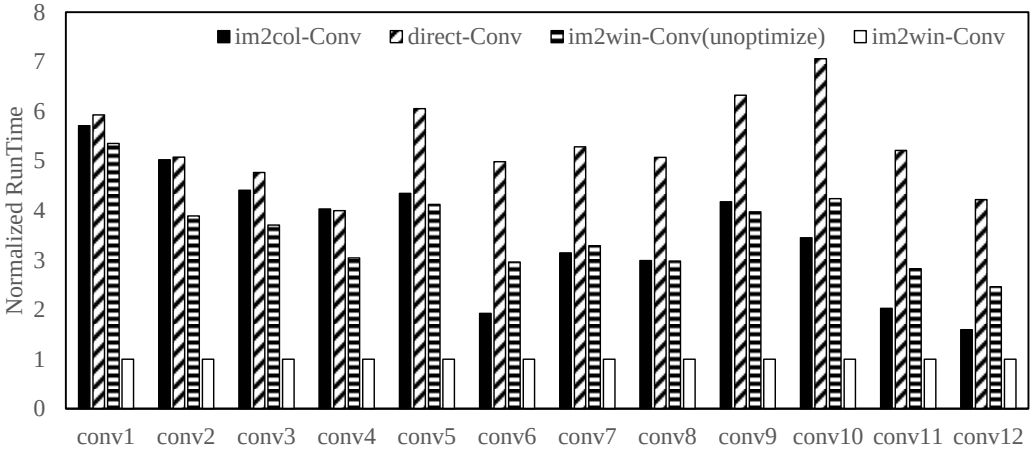

图 3-4 不同卷积算法的运行时间归一化对比

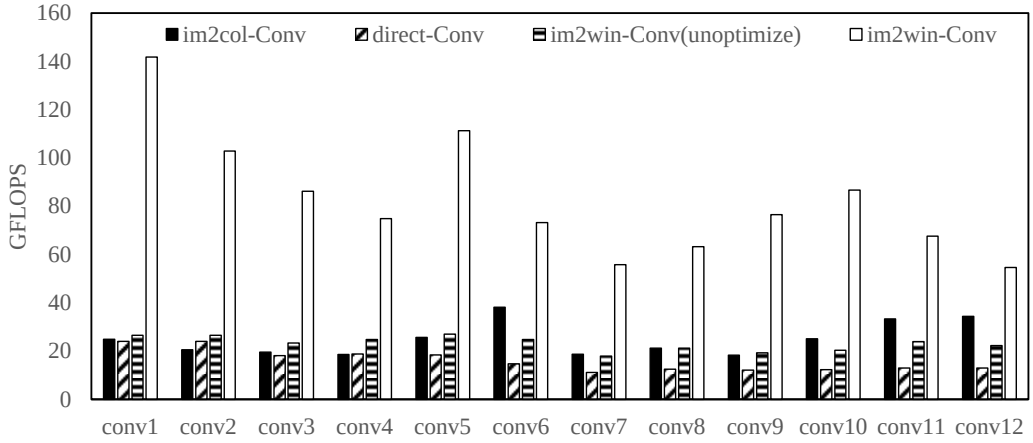

图 3-5 不同卷积算法的 GFLOPS

本节是对基于 im2col 变换的卷积算法、直接卷积算法和本文提出的张量卷积算法的基础版本与高性能版本在测试基准中的运算性能结果分析，测试基准中一共有 12 个独特的卷积层，其详细参数如表 3-3 所示。对于运算性能测试实验记录了两个评价指标，不同卷积算法的运行时间和 GFLOPS。

图 3-4 展示了不同算法在测试基准中的运行时间结果，包括基于 im2col 变换的卷积算法、直接卷积算法和本文提出的张量卷积算法的基础版本与高性能版本四种方法。为了方便比对，实验结果以本文提出的张量卷积算法的高性能版本为基准进行归一化处理。在测试基准的 12 个层中，本文提出的张量卷积算法都具有最快的运行时间。具体来看，基于 im2col 变换的卷积算法的运行时间最慢约为本文提出的张量卷积算法的 5.7 倍，平均约为本文提出的张量卷积算法的 3.6 倍。而直接卷积算法的运行时间最慢约为本文提出的张量卷积算法的 7 倍，平均约为本文提出的张量卷积算法的 5.3 倍。综合上述分析可以看出，本文提出的张量卷积算法在测试基准中有最快的运行时间，当卷积层的输入特性图大小、卷积核尺寸和步长均发生改变时，相比于基于 im2col 变换的卷积算法和直接卷积算法仍然能取得最好的运行时间表现。

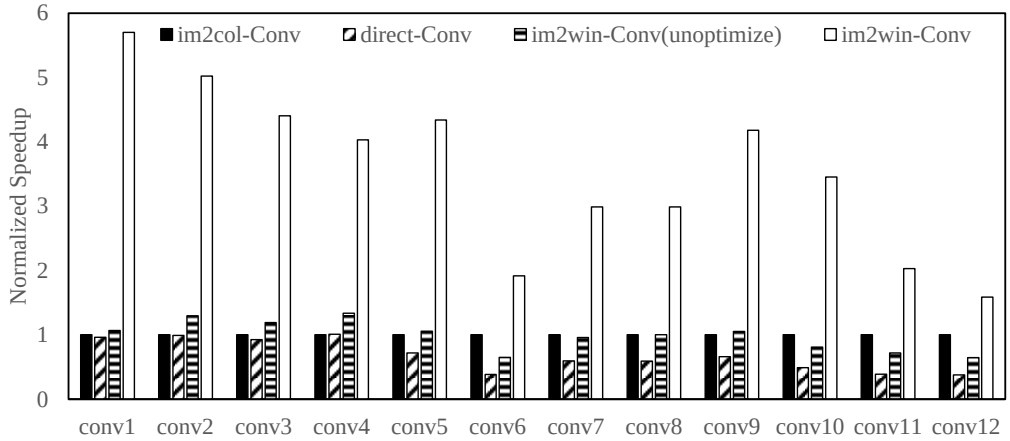

图 3-6 不同卷积算法的 GFLOPS 归一化对比

图 3-5 展示了不同算法在测试基准中的 GFLOPS 性能结果，并在图 3-6 中对实验结果以基于 im2col 变换的卷积算法为基准进行归一化处理。从图 3-5 可以看出，本文提出的张量卷积算法在测试基准的 12 个卷积层中都具有最高的 GFLOPS 性能，并且在大多数卷积层上要远远高出基于 im2col 变换的卷积算法和直接卷积算法。此外，从图 3-6 中可以观察到，本文提出的张量卷积算法相对于基于 im2col 变换的卷积算法，其 GFLOPS 性能的提升最高可达 5.8 倍，最低可达 2.2 倍；而相对于直接卷积算法，其 GFLOPS 性能的提升最高可达 7.1 倍，最低可达 3.8 倍。值得注意的是，在本文提出的张量卷积算法在测试基准的 12 个卷积层中的 GFLOPS 性能中，存在两处明显的性能下降趋势，分别出现在第 1 层至第 4 层以及第 10 层至第 12 层。通过结合测试基准中的详细参数（如表 3-3 所示）可以发现，第 1 个卷积层至第 4 个卷积层间卷积核尺寸和步长大小递减，而第 10 个卷积层至第 12 个卷积层间输入特征图尺寸逐渐减小。这意味着本文提出的张量卷积算法在输入特征图、卷积核和步长大小较大的卷积场景中表现最佳。

综合上述分析，本文提出的张量卷积算法相比于基于 im2col 的卷积算法和直接卷积算法能够有效地提升运算性能表现。

3.3.3 内存占用结果分析

本节是对基于 im2col 变换的卷积算法、直接卷积算法和本文提出的张量卷积算法的基础版本与高性能版本在测试基准中的内存占用结果分析，测试基准中一共有 12 个独特的卷积层，其详细参数如表 3-3 所示。对于内存占用测试实验，分别记录了不同卷积算法的数据内存占用情况和算法运行时内存占用的峰值。

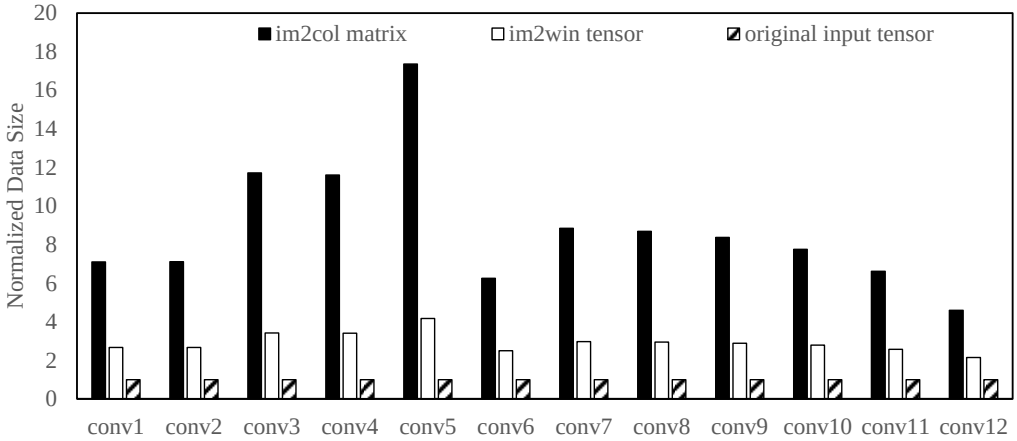

图 3-7 不同卷积算法的数据使用内存大小

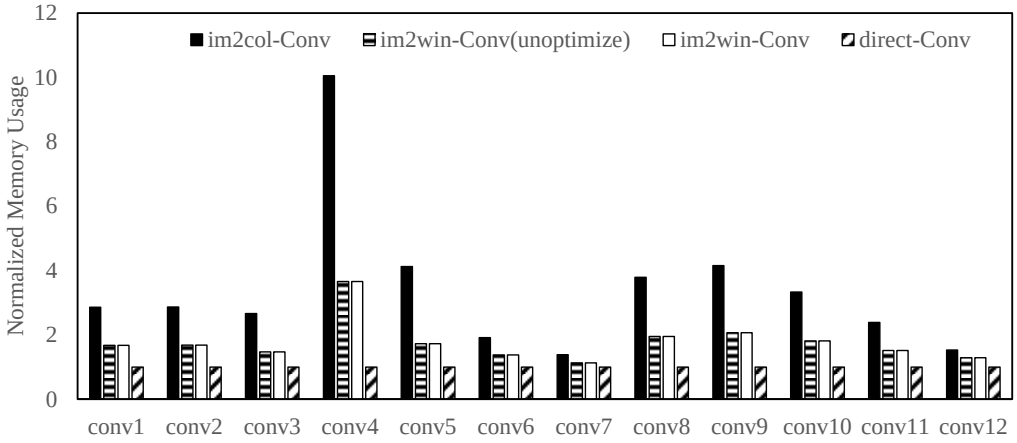

图 3-8 不同卷积算法运行时内存占用的峰值

图 3-7 展示了不同算法在测试基准中运行时数据占用内存的大小，即原始输入张量、im2col 矩阵和 im2win 张量内存占用，使用三者中最小的原始输入张量为基准对实验结果进行归一化处理。可以从图中观察到，im2col 矩阵在测试基准的 12 个卷积层上都具有最大的内存占用且远远大于原始输入张量的内存大小。这个结果与本文在 2.1.3 节中讨论得到的结论一致，即基于 im2col 变换的卷积算法中 im2col 变换算法会导致巨额的内存消耗。具体来看，im2col 矩阵的内存占用在测试基准最多达到原始输入张量的 17.3 倍，这无疑极大地降低了基于 im2col 变换的卷积的性能并增大了算法部署在内存有限的设备上的难度；本文提出的张量卷积算法中的 im2win 张量在测试基准的 12 个卷积层上内存占用都比 im2col 矩阵更小，内存占用平均约为 im2col 矩阵的 30%。

图 3-8 展示了不同卷积算法在测试基准上运行时内存占用的峰值，即基于 im2col 变换的卷积算法、直接卷积算法和本文提出的张量卷积算法的基础版本与高性能版本四种方法，所记录的峰值内存占用结果考虑了不同算法整个卷积过程中输入张量、卷积核张量、输出张量、数据转换和其他操作的所有内存占用。考虑到直接卷积算法没有额外的数据变换内存消耗，为了方便比对，以直接卷积算法的峰值内存占用为基准对实验结果进行归一化处理。在测试基准的 12 个卷积层上本文提出的张量卷积算法的内存占用都要低于基于 im2col 变换的卷积算法，平均降低了 41.6%的内存占用。

综合上述分析，本文提出的张量卷积算法比基于 im2col 的卷积算法能够有效地减少内存占用。

3.4 本章小结

本章主要介绍了本文提出的张量卷积算法在 CPU 平台上的实现及优化，首先介绍分析了 CPU 平台体系结构的特性，包括 CPU 平台的软硬件结构和并行

加速技术等，之后详细阐述了本文提出的张量卷积算法在 CPU 平台的实现方法及结合 CPU 平台的一系列优化技术实现的高性能卷积算法，最后通过一个完备了测试基准验证了本文提出的张量卷积算法相比于基于 im2col 的卷积算法和直接卷积算法有更好的性能表现，相比于基于 im2col 的卷积算法能够有效降低内存占用。

第 4 章 基于 im2win 变换的张量卷积在 GPU 平台上的优化实现

上一章介绍了本文提出的基于 im2win 变换的张量卷积算法在 CPU 平台上实现，并采用了多种优化技术来适应 CPU 架构特点和提高计算性能，并通过测试基准展示了该算法与其他方法的计算性能和内存占用对比以及通用性分析。本章主要介绍基于 im2win 变换的张量卷积算法在 GPU 平台上的高性能实现及优化技术，并与 GPU 平台上常用的深度学习框架中的卷积算法进行实验分析对比。首先介绍了 GPU 的硬件体系结构、软件编程模型以及计算统一设备架构 CUDA，针对 GPU 的 CUDA 架构模型，对张量卷积算法的索引进行重构，使得 GPU 可以很好地并行张量卷积算法；然后详细介绍了利用 Tiling 策略来减少全局内存访问，并利用共享内存与寄存器来缓存局部数据；同时还采用微内核设计、向量化加载/存储、双缓冲与数据预取等技术来提高计算吞吐率；最后，在一个完备的测试基准上对比评估了本文实现的基于 im2win 变换的张量卷积算法与现有方法在运行时间、每秒浮点运算次数和内存占用等指标上的性能，并且进行了消融实验，以分析本文提出的算法在不同场景下的通用性和有效性。实验表明，本章在 GPU 平台上实现的张量卷积算法，与常见的深度学习库中的卷积算法相比，在减少内存消耗的同时，计算性能更优或相当。

4.1 GPU 平台的体系结构

图形处理器（GPU）是一种高并发、高吞吐量和高效能的处理器，已经成为人工智能和深度学习领域的主要计算平台。本文所研究的张量卷积算法是一种高效的卷积算法，它可以在 GPU 平台上获得很好的性能。为了实现高性能的张量卷积算法，需要结合 GPU 平台的体系结构特点进行优化设计。本节将介绍 GPU 平台的硬件结构、软件编程模型和优化技术，并分析它们对张量卷积算法性能的影响。

4.1.1 GPU 的硬件结构

由于高度并行的架构设计，现代 GPU 在图像和图形处理方面展现了极高的效率。相比之下，通用处理器 CPU 更适合处理逻辑复杂的任务。以下将从硬件结构和指令级别两个角度，对 GPU 和 CPU 在大数据块并行处理算法方面的性能差异进行分析和比较。

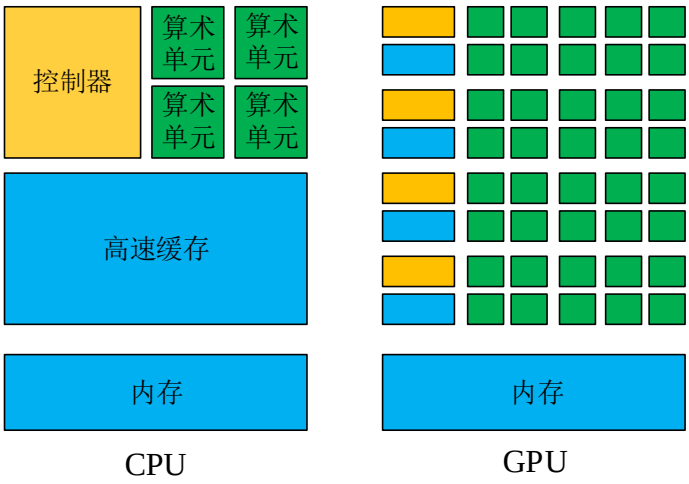

图 4-1 CPU 架构与 GPU 架构

首先，从硬件结构角度来看，CPU 和 GPU 存在很大差异。如图 4-1 所示，CPU 拥有众多功能模块，可以适应复杂的运算环境。而 GPU 构成相对简单，其中流处理器和显存控制器占据了绝大部分晶体管。这种不同的资源分配方式决定了 CPU 和 GPU 的计算能力和特点。

其次，从指令级别来看，CPU 和 GPU 也存在巨大差异。CPU 由专门为顺序串行处理而优化的少量核心组成。而 GPU 则由数以千计的更小、更高效的核心组成，并且这些核心专门为同时处理多任务而设计。通过图 4-2 可以更好地理解串行运算和并行运算之间的区别。传统串行编写软件具备以下特点：运行在具有单一 CPU 的计算机上；一个问题被分解为一系列离散指令；指令必须依次执行；任何时刻只能执行一条指令。而并行计算则改进了许多重要细节：使用多个处理器运行；一个问题可以被分解为可同时解决的离散指令；每个部分进一步细分为一系列指示；每个部分问题可以同时在不同处理器上执行。

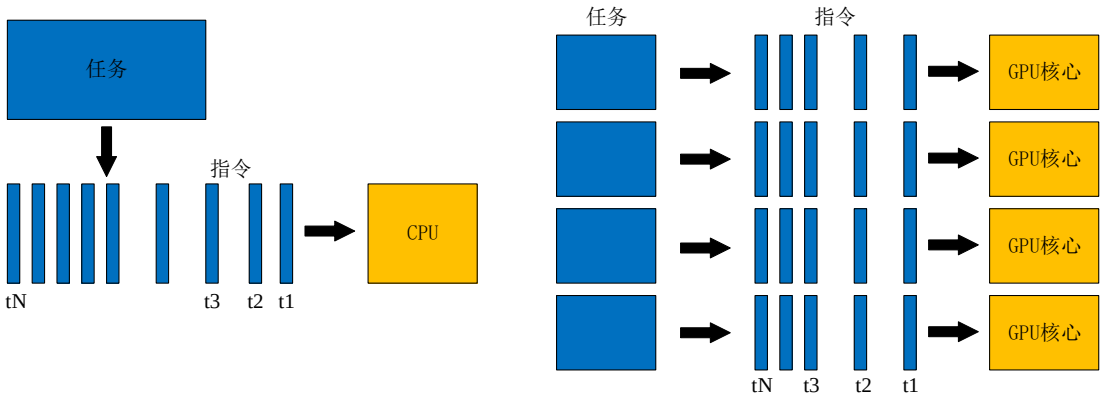

图 4-2 CPU 串行计算和 GPU 并行计算

基于以上两个角度，可以得出结论：GPU 之所以适用于处理数据并行计算任务，是因为它拥有大量简化了复杂逻辑控制单元和缓存体系的计算核心。这

使得 GPU 可以同时执行数千个线程，并在大规模数据并行问题上表现显著优于 CPU。以 NVIDIA GeForce RTX 3090 GPU 为例，该 GPU 包含 10496 个 CUDA 核心，而 Intel Core i9-11900K CPU 则只有 8 个核心。这意味着，在同样条件下，GPU 可以同时执行比 CPU 多出上千倍的线程数量。

4.1.2 GPU 的软件编程模型

GPU 的软件编程模型主要有 OpenCL[102]、CUDA[47]等。本文采用 NVIDIA 的 GPU 解决方案，并基于其 CUDA 架构实现张量卷积及优化。下面介绍 CUDA 架构的基本概念和特点。

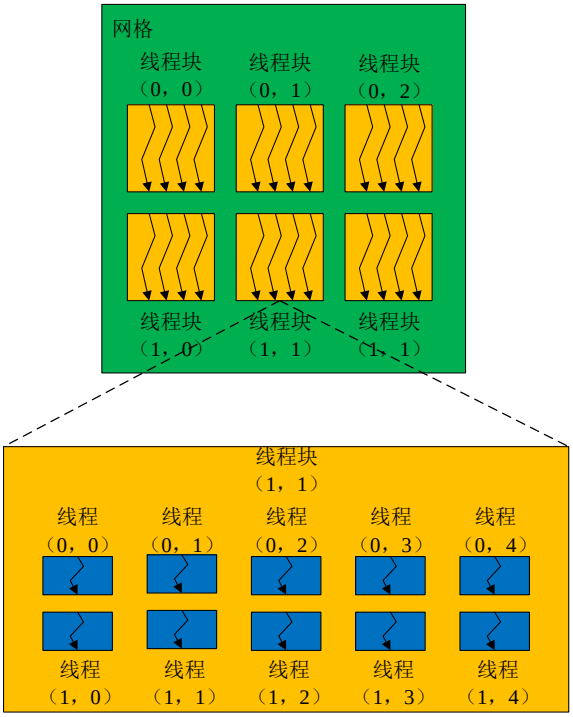

图 4-3 CUDA 线程层次结构

（1）CUDA 架构

统一计算设备架构（Compute Unified Device Architecture，CUDA）是 NVIDIA 推出的一种利用 GPU 进行通用并行计算的平台和编程模型。它包含了专为 GPU 计算设计的指令集架构和内部并行计算引擎。该架构具有以下三个关键抽象特性：1、线程组层次：将大量线程组织成网格，每个网格由多个线程块组成，每个线程块又包含多个线程。这种划分使开发人员可以根据数据并行度和硬件资源来安排线程执行。2、共享内存：位于每个多处理器上、可被同一个线程块中所有线程共享的高速缓存，由开发人员显式管理。共享内存可以提高数据访问速度，并减少对设备内存的传输。3、屏障同步机制：通过__syncthreads()函数实现，在该函数处，所有线程都会等待直到同一个线程块中

的所有其他活动线程都到达该点。在 CUDA 架构下，一个程序由主机端和设备端两部分组成。主机端在 CPU 上执行，设备端在 GPU 上执行。开发人员需要精确控制数据传输和计算任务之间的交互，以达到最佳的性能表现。

（2）CUDA 线程层次

在 CUDA 架构中，为了充分利用 GPU 的并行性能，线程层次被设计为网格、线程块和线程三个不同的级别。具体而言，网格（grid）表示一个核函数的启动，线程块（block）表示在一个流多处理器（Streamimg Multiprocessor, SM）中执行的一组线程，而线程（thread）则表示最小的计算单元。每个线程都执行相同的代码，并且可以访问同一块共享内存。线程块中的所有线程可以通过屏障同步机制进行协调。在硬件层面上，每个 SM 以每 32 个线程为一组创建、管理、调度和执行这些线程，并称之为 warp。当一个 SM 被分配有一个或多个线程块时，则会被分成多个 warps，并且每个 warp 都由调度器进行调度。

（3）CUDA 内存模型

CUDA 内存模型是一种显式的内存模型，可以通过它控制 GPU 中不同层次的内存空间。CUDA 内存模型包括以下几种类型的内存：

——全局内存：GPU 中最大的一块存储器，用于存储所有线程都可以访问的数据。全局内存适合处理大规模计算问题，但是访问速度较慢，应尽量减少对它的访问。

——共享内存：每个 SM 都有一定数量由线程块分配的共享内存，用于线程块内部的线程协作。共享内存可以提高程序性能，避免访问全局内存，但是需要同步机制来防止数据冲突。

——L1 和纹理缓存：用于缓存最近访问过的数据，加快 GPU 程序执行速度。L1 缓存在每个 SM 上，纹理缓存在每个流多处理器上。

——L2 缓存：用于处理较大规模计算问题时使用。L2 缓存在所有 SM 之间共享，比 L1 缓存在更大但是相对较慢。

——寄存器：GPU 中最快的一种类型的内存，每个线程都有自己独立的寄存器文件。寄存器通常用于保存频繁访问和计算结果等变量。

4.1.3 GPU 的优化技术

（1）共享内存和寄存器

共享内存是 GPU 中每个线程块内部的高速缓存，可被该线程块中的所有线程共享访问。它可以降低对全局内存的依赖，提高数据局部性和并行效率。寄存器是 GPU 上最快速的内存类型，为每个线程提供临时变量空间。由于寄存器数量有限，需要合理分配和使用寄存器，避免发生寄存器溢出。

（2）数据并行性

GPU 是一种基于数据并行模型设计的处理器，能够同时执行多个相同或不同的任务。为了充分利用 GPU 的并行能力，需要保证程序中有足够多的独立可并行执行的数据。这就要求在编写程序时避免不必要的串行操作和同步操作，尽量使用向量化指令和循环展开等技术来增加指令级并行度。

（3）提高占用率

占用率是指一个线程块中活跃线程数占最大线程数（通常为 32 或 64）的比例。占用率越高，表示 GPU 上空闲资源越少，吞吐量越大。提高占用率可以通过合理设置线程块大小、寄存器数量、共享内存大小等参数来实现。

（4）合并访存和内存对齐

合并访存和内存对齐合并访存是指多个线程同时访问连续或相邻地址时，将这些访问合并成一次或几次较大粒度的访问。这样可以减少总体访问次数和延迟，并提高带宽利用率。为了实现合并访存，需要保证数据在全局内存中是连续或相邻排列，并且与缓冲区边界对齐。

（5）快速指令和指令级并行

快速指令是指执行时间较短或没有延迟开销的指令。例如，在 CUDA 中有一些特殊函数叫做 intrinsics，它们可以直接映射到硬件操作而不需要调用库函数。使用快速指令可以减少运算时间和节省资源。另外，在 GPU 上执行时，每个线程块会被划分为若干个 warp（通常为 32 个线程），每个 warp 会以单指令多数据（SIMD）方式执行相同或不同的指令流。为了提高 warp 级别的并行度，需要避免分支散度和控制流散度等情况发生。

4.2 GPU 上张量卷积算法实现

在之前的第 3 章中，详细介绍了基于 im2win 变换的张量卷积算法结合 CPU 平台的一系列优化技术实现的高性能卷积算法，并通过测试基准验证了基于 im2win 变换的张量卷积算法在运算性能和内存占用方面的有效提升。然而，GPU 平台的设计架构比 CPU 平台更适合于计算密集型操作，如卷积运算。此外，GPU 平台的可用内存远远小于 CPU 平台的可用内存，因此，在 GPU 平台上有效降低卷积算法的内存占用更加关键。在本节中，将重点介绍在 GPU 平台上对 im2win 变换得到的 im2win 张量进行卷积操作运算的具体实现及优化。

4.2.1 张量卷积算法的索引映射

根据算法 3 所定义的基于 im2win 变换的张量卷积，该算法采取了 7 层嵌套

循环结构。GPU 平台的 CUDA 编程模型架构能够有效地并行化循环结构，但其维度数量有限，每个维度的取值范围较大。在算法 3 中，共存在 7 个 for 循环的维度，其中各个维度的取值范围通常较小（例如卷积核尺寸维度 H_f 和 W_f），这意味着算法 3 中的卷积结构难以直接映射到 CUDA 架构上（因为 CUDA 架构的维度数量较少，但每个维度的取值范围通常较大，与算法 3 中情况相反）。因此，本文提出了一种类似于隐式矩阵乘法卷积[103]的结构设计，能够通过重新映射卷积过程中某些维度的索引值来优化计算。这种设计使得基于 im2win 变换的张量卷积算法可以更好地适配 CUDA 架构，如算法 7 所示。

算法 7：索引映射的基于 im2win 变换的张量卷积算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量： $\hat{\mathcal{I}} = imwin(\mathcal{I}, \mathcal{F}, s)$

维度： $M = C_o, N = N_o \times H_o \times W_o, K = C_f \times H_f \times W_f$

1: for $m = 0$ to $M - 1$ do
2: $o_c = m$
3: for $n = 0$ to $N - 1$ do
4: $o_n = i_n = n / (H_o \times W_o)$
5: $o_h = (n\%(H_o \times W_o)) / W_o$
6: $o_w = (n\%(H_o \times W_o))\%W_o$
7: for $k = 0$ to $K - 1$ do
8: $f_c = i_c = k / (H_f \times W_f)$
9: $k_{res} = k\%(H_f \times W_f)$
10: $f_h = k_{res} / W_f$
11: $f_w = k_{res}\%W_f$
12: $i_h = o_h \times s + f_h$
13: $i_w = o_w \times s + f_w$
14: $\mathcal{O}(o_n, o_c, o_h, o_w) += \hat{\mathcal{I}}(i_n, i_c, i_h, i_w) \times \mathcal{F}(f_n, f_c, f_h, f_w)$

算法 7 展示的是索引映射的基于 im2win 变换的张量卷积算法的串行结构，原本采取的 7 层嵌套循环结构被重新映射到了 3 个维度上，即 $M = C_o, N = N_o \times H_o \times W_o, K = C_f \times H_f \times W_f$。在该算法中，im2win 张量 $\hat{\mathcal{I}}$、卷积核张量 $\mathcal{F}$ 和输出张量 $\mathcal{O}$ 的索引值不能直接得到，而是通过 M, N, K 三个维度上的迭代值进行映射得到，输出张量 $\mathcal{O}$ 各个维度的索引值通过 M 和 N 维度得到（算法 7 中第 2 行和第 4 行至第 6 行），im2win 张量 $\hat{\mathcal{I}}$ 和卷积核张量 $\mathcal{F}$ 各维度

的索引值在最内层 K 维度得到（算法 7 中第 8 行至第 13 行）。在最内层循环中，计算得到各个张量的索引后进行乘加赋值操作（算法 7 中第 14 行）。

4.2.2 CUDA 架构上的张量卷积算法

算法 7 中的索引映射基于 im2win 变换的张量卷积算法，可以通过 CUDA 编程模型并行化，但需要考虑线程之间的独立性。在整个张量卷积算法中，不同输出张量 $\mathcal{O}$ 之间是相互独立的，因此可以将输出张量的维度映射到 CUDA 模型的线程上。由于输出张量 $\mathcal{O}$ 的维度只与 M 和 N 两个循环有关（算法 7 中第 1 行至第 6 行），因此可以将这两个循环映射到 CUDA 模型中的不同线程上。由于现代 GPU 中线程块的大小一般最大为 1024，而两个循环的迭代值的积可能会大于 1024，因此不能将这两个循环映射到线程块（block）层次，只能映射到网格（grid）层次。算法 8 展示了在 GPU 上实现基于 im2win 变换的张量卷积算法，线程层次划分如下，一共有 $M/32 \times N/32$ 个线程块，每个线程块有 32×32 个线程，即总共有 $M \times N$ 个线程。这样一来，算法 7 中的三层循环结构中的两层可以映射到 CUDA 模型的线程中。通过网格维度和线程块维度的编号，可以计算出输出张量的索引值（算法 8 中第 1 行至第 6 行）。通过剩下的维度循环计算 im2win 张量和卷积核张量各维度的索引值并进行乘加赋值操作（算法 8 中第 7 行至第 14 行）。

4.3 GPU 上张量卷积算法优化设计

在上节中，详细介绍了如何通过 CUDA 模型的多线程特性将张量卷积算法进行并行化实现（如算法 8 所示）。然而，在算法 8 中，由于所有的数据都存储在全局内存中，导致在全局内存上的读取以及操作具有很高的延迟，从而导致该算法性能表现不佳。为了提高算法性能，本节将介绍针对张量卷积算法的优化设计。这些优化技术包括 tiling、共享内存和寄存器、微内核设计、向量化加载/存储、双缓冲和预取等技术。通过采用这些优化技术，可以最大化工作负载和数据并行性，并减少数据访问延迟，从而实现高性能的张量卷积算法。

4.3.1 基于 tiling 策略优化的张量卷积

Tiling 技术[104]是一种基于数据并行性的优化技术，旨在通过将数据划分为更小的块（称为 tile）来减少内存访问冲突和提高内存带宽利用率。在 GPU 上，tile 块通常被划分为线程块级别和线程级别，作为基本的计算单元来执行计算任务。在算法 8 中，由于将索引重新映射到了 M, N, K 三个维度上，每个维度都

算法 8：GPU 上基础的基于 im2win 变换的张量卷积算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量： $\hat{\mathcal{I}} = imwin(\mathcal{I}, \mathcal{F}, s)$

维度： $M = C_o, N = N_o \times H_o \times W_o, K = C_f \times H_f \times W_f$

 # blocks 数目： $M / 32 \times N / 32$

 # 每个 block 中 threads 数目： 32×32

$m = bx \times 32 + tx$

$n = by \times 32 + ty$

$o_c = m$

$o_n = i_n = n / (H_o \times W_o)$

$o_h = (n\%(H_o \times W_o)) / W_o$

$o_w = (n\%(H_o \times W_o))\%W_o$

for $k = 0$ **to** $K - 1$ **do**

　$f_c = i_c = k / (H_f \times W_f)$

　$k_{res} = k\%(H_f \times W_f)$

　$f_h = k_{res} / W_f$

　$f_w = k_{res}\%W_f$

　$i_h = o_h \times s + f_h$

　$i_w = o_w \times s + f_w$

　$\mathcal{O}(o_n, o_c, o_h, o_w) += \hat{\mathcal{I}}(i_n, i_c, i_h, i_w) \times \mathcal{F}(f_n, f_c, f_h, f_w)$

有足够大的取值范围，因此可以将 im2win 张量和卷积核张量在线程块级别划分为大小为 $M_B \times N_B \times K_B$ 的 tile 块，在线程级别划分为大小为 $M_T \times N_T$ 的 tile 块（如算法 9 所示）。这样一来可以将 tile 块作为卷积计算过程中的基本计算单元，进而通过减少数据访问次数和提高数据局部性来改善计算性能。tile 的大小可以适应于共享内存或寄存器的大小，这样可以大幅降低延迟，从而提高数据重用率和缓存命中率。通过采用 tiling 技术，可以将计算任务划分为多个小块，以便在 GPU 上并行执行，从而提高算法的性能表现。此外，Tiling 技术还可以通过动态调整 tile 的大小来进一步优化性能，以适应不同的硬件环境和计算任务。

4.3.2 基于共享内存和寄存器优化的张量卷积

GPU 设备上的内存由四个层次结构组成：全局内存、共享内存、L1/L2 缓存（在 CUDA 中不可编程控制）和寄存器。从全局内存到共享内存，再到寄存

器，访问延迟逐渐降低，大小也逐渐减小。因此，在 GPU 计算中，为了提高性能，需要充分利用这些不同层次结构的特点。算法 9 中，将 im2win 张量和卷积核张量划分为 tile 块之后，可以为每个线程分配大小为$M_T \times N_T$的寄存器，而为每个线程块分配大小为$M_B \times N_B$和$K_B \times N_B$的共享内存（算法 9 中第 1 行至第 2 行），并将位于全局内存中的 im2win 张量和卷积核张量加载到共享内存和寄存器中（算法 9 中第 3 行至第 7 行）。由于共享内存由线程块内的所有线程共享，因此将数据在共享内存中重用远快于在全局内存中重用，这可以显著提高性能。因此，通过结合 GPU 设备内存的层次结构合理分配共享内存和寄存器，可以将数据从全局内存中移动到更快的内存层次结构中，以减少访问延迟和提高数据重用率，进一步提高算法的性能表现。

4.3.3 基于微内核设计优化的张量卷积

微内核技术[105]是一种基于微架构设计的优化技术，通过将 GPU 的硬件架构分解为多个小型、独立的内核来实现，从而提高 GPU 的并行性和性能。在算法 8 中，最内层 for 循环中的一个乘加操作只计算输出张量的一个元素（算法 8 中第 14 行）。微内核设计通常实现为向量的外积乘法，通过在 GPU 上每个线程中执行微内核，每个线程可以负责计算输出张量的多个元素。微内核的大小被设计为$M_T \times N_T$（算法 9 中第 12 行和第 19 行），在线程级别上进行划分。这种设计使得 GPU 上的每个线程可以在本地计算微内核，并减少对全局内存的访问次数，从而降低了内存瓶颈的影响。此外，微内核设计还可以提高数据重用效率，因为相邻的线程计算相邻的输出张量元素，这样就可以复用之前加载到共享内存中的数据，从而进一步减少了内存访问次数。

4.3.4 基于向量化加载/存储优化的张量卷积

向量化加载/存储是一种在 GPU 上提高内存访问效率的技术，其主要目的是通过将多个数据元素一次性地加载或存储到寄存器中，从而改善数据 I/O 效率和内存带宽利用率。在 GPU 上进行卷积计算时，数据 I/O 和内存带宽通常是性能瓶颈之一。针对这一问题，上文提出了 im2win 变换，即将输入张量转换为连续的窗口排序的 im2win 张量，以便于使用向量化加载将输入张量中的数据一次性地加载到寄存器中（算法 9 中第 6 行至第 7 行），并在寄存器中进行后续计算。这种方法可以有效地减少内存访问次数，提高内存带宽利用率，从而进一步提高卷积计算的性能。

4.3.5 基于双缓冲和数据预取优化的张量卷积

算法 9：GPU 上高性能的基于 im2win 变换的张量卷积算法

输入：输入张量 $\mathcal{I}[N_i][C_i][H_i][W_i]$，卷积核 $\mathcal{F}[N_f][C_f][H_f][W_f]$，步长 s

输出：输出张量 $\mathcal{O}[N_o][C_o][H_o][W_o]$

im2win 张量：$\hat{\mathcal{I}} = imwin(\mathcal{I}, \mathcal{F}, s)$

维度：$M = C_o, N = N_o \times H_o \times W_o, K = C_f \times H_f \times W_f$

 # blocks 数目：$M / M_B \times N / N_B$

 # 每个 block 中 threads 数目：$M_B / M_T \times N_B / N_T$

1: 分配寄存器：$R_{\hat{\mathcal{I}}}[2][N_T], R_{\mathcal{F}}[2][M_T], R_{\mathcal{O}}[M_T \times N_T]$

2: 分配共享内存：$S_{\hat{\mathcal{I}}}[2][K_B \times N_B], S_{\mathcal{F}}[2][M_B \times K_B]$

3: $S_{\hat{\mathcal{I}}}[0][k_B \times n_B] \leftarrow k_B \times n_B$ 从 $\hat{\mathcal{I}}(0, by)$ 中

4: $S_{\mathcal{F}}[0][m_B \times k_B] \leftarrow m_B \times k_B$ 从 $\mathcal{F}(bx, 0)$ 中

5: $\_\_syncthreads()$

6: $R_{\hat{\mathcal{I}}}[0][n_T] \leftarrow n_T$ 从 $S_{\hat{\mathcal{I}}}[0][0 \times n_B]$ 中 //向量化加载

7: $R_{\mathcal{F}}[0][m_T] \leftarrow m_T$ 从 $S_{\mathcal{F}}[0][m_B \times 0]$ 中 //向量化加载

8: **for** $kk = 0$ **to** $C_f \times H_f \times W_f / K_{f,b} - 1$ **do**

9: **for** $k' = 1$ **to** $K_{f,b} - 1$ **do**

10: $R_{\hat{\mathcal{I}}}[load][n_T] \leftarrow n_T$ 从 $S_{\hat{\mathcal{I}}}[store][k' \times n_B]$ 中 //预取下一次的数据

11: $R_{\mathcal{F}}[load][m_T] \leftarrow m_T$ 从 $S_{\mathcal{F}}[store][m_B \times k']$ 中 //预取下一次的数据

12: $R_{\mathcal{O}}[m_T \times n_T] += R_{\mathcal{F}}[store][m_T] \times R_{\hat{\mathcal{I}}}[store][n_T]$ //微内核

13: **if** $kk \neq C_f \times H_f \times W_f / K_{f,b} - 1$ **then**

14: $S_{\hat{\mathcal{I}}}[load][k_B \times n_B] \leftarrow k_B \times n_B$ 从 $\hat{\mathcal{I}}(kk + 1, by)$ 中 //预取下一次的数据

15: $S_{\mathcal{F}}[load][m_B \times k_B] \leftarrow m_B \times k_B$ 从 $\mathcal{F}(bx, kk + 1)$ 中 //预取下一次的数据

16: $\_\_syncthreads()$

17: $R_{\hat{\mathcal{I}}}[0][n_T] \leftarrow n_T$ 从 $S_{\hat{\mathcal{I}}}[store][0 \times n_B]$ 中 //向量化加载

18: $R_{\mathcal{F}}[0][m_T] \leftarrow m_T$ 从 $S_{\mathcal{F}}[store][m_B \times 0]$ 中 //向量化加载

19: $R_{\mathcal{O}}[m_T \times n_T] += R_{\hat{\mathcal{I}}}[1][n_T] \times R_f[1][m_T]$ //微内核

20: $\mathcal{O}(bx, by) \leftarrow R_{\mathcal{O}}[m_T \times n_T]$

双缓冲技术[106]是指使用两个缓冲区来存储输入张量和滤波器张量以进行流水线并发计算。具体来说，在算法 9 中，为共享内存和寄存器时分配了两倍的空间（算法 9 中第 1 行至第 2 行），其中一个缓冲区用于计算，另一个用于加载下一次计算中使用的新数据。当计算完成后，两个缓冲区的角色会交换，即原始缓冲区变为新的加载缓冲区，原始加载缓冲区变为新的计算缓冲区。这种技术可以有效地隐藏数据 I/O 延迟，提高计算效率，从而进一步优化卷积计算的

性能。与此同时，数据预取技术[107]是通过将下一次卷积计算使用的张量数据提前加载到寄存器（或共享内存）中，以隐藏从全局内存读取数据的延迟和开销（算法 9 中第 10 行至第 11 行和第 14 行至第 15 行），从而减少等待时间并提高计算效率。预取技术通常是基于数据局部性原理实现的，即当前访问的数据很可能在未来也被访问，因此将其预先加载到寄存器或共享内存中可以有效地提前完成数据 I/O 操作，从而避免了等待时间并提高了计算效率。

4.4 实验与结果分析

上一节详述了基于 im2win 变换的张量卷积算法在 GPU 平台上的实现，并针对 GPU 平台的特性进行了一系列优化措施，以提高算法的计算性能。为了验证本文提出的算法和优化技术的有效性和可靠性，本节将进行计算性能和内存占用方面的评估实验，以及消融实验。实验采用了一个全面且完备的测试基准，涵盖了各种卷积场景。

4.4.1 测试实验的整体方案

（1）硬件环境

本文 GPU 上的测试实验是在英伟达公司的 GeForce RTX 3090 型号的 GPU 设备上进行的，该设备的详细参数如下表所示：

表 4-1 3090 硬件参数

GPU 型号	GeForce RTX 3090
流处理器	10496 个
核心频率	基础频率：1395MHz
显存容量	24GB GDDR6X
显存位宽	384bit

（2）软件环境

表 4-2 GPU 平台软件环境的版本型号

软件环境	版本型号
PyTorch	1.10.0a0
CUDA	11.1
cuBLAS	11.2
cuDNN	8.0.1

本文将基于 im2win 变换的张量卷积算法与 PyTorch 中基于 im2col 变换并调用 cuBLAS[72]的卷积算法以及调用 cuDNN[73]的卷积算法进行性能和内存使用方面的对比。由于 cuBLAS 和 cuDNN 的 API 是预定义的，不能直接支持基于 im2win 变换的张量卷积，因此本文使用 CUDA 11.1 实现了该算法。为了简便起见，以下分别将这三种卷积方法称为 im2col+cuBLAS 卷积、cuDNN 卷积和 im2win 张量卷积。在实验测试中，数据结构使用 PyTorch 中定义的张量数据类型，每个张量中的元素都是单精度浮点型。软件环境的具体版本型号如表 4-2 所示。需要注意的是，cuDNN 提供了多种不同的卷积算法，在通过 PyTorch 调用时会根据卷积层参数自动选择最优的算法。表 4-3 列出了 cuDNN 支持的所有卷积算法及其简要描述。在后续实验结果中，将结合测试基准具体分析 cuDNN 选择的卷积算法。

表 4-3 cuDNN 中的卷积算法

cuDNN 中卷积算法	描述
IMPLICIT_GEMM	基于矩阵乘法的隐式卷积算法
IMPLICIT_PRECOMP_GEMM	基于预先计算的矩阵的隐式卷积算法
GEMM	基于矩阵乘法的显式卷积算法
FFT	基于 FFT 的卷积算法
FFT_TILING	基于 FFT 分块的卷积算法
WINOGRAD	基于 Winograd 的卷积算法
WINOGRAD_NONFUSED	基于 Winograd 的非融合卷积算法

（3）测试基准

本文期望对常用的卷积神经网络中大多数的卷积层进行测试实验，因此实验采用了一个涵盖了各种卷积场景的测试基准。该测试基准包含了 12 个参数不同的卷积层，在上文中介绍了这个测试基准中各个卷积层的具体的参数，如表 3-3 所示。在通过 PyTorch 调用 cuDNN 时会根据卷积层参数自动选择最优的算法，cuDNN 在测试基准中 12 个卷积层自适应选择的最优卷积算法如下表所示：

（4）评价指标

在实验测试中主要对比分析不同卷积算法之间运算性能和内存占用这两方面的测试结果。本文选取运行时间和 GFLOPS 作为测试实验的速度性能评价指标。上文中详细介绍了 GFLOPS 的计算公式，如式 3-3 所示。

（5）实验方法

在确定了实验的软硬件环境、测试基准和评价指标后，本文选择 3 种卷积进行测试，包括 im2col+cuBLAS、cuDNN 和 im2win，其中 cuDNN 中实现了多

种不同的卷积算法。在测试实验中，使用 C++标准库的 wall-clock 函数并添加同步函数来记录不同算法在 GPU 上完成卷积操作的运行时间，为了使 GPU 在运行时充分预热，每个算法在测试基准的每个层上运行 50 次，并选取 50 次运行中最佳的运行时间记录。在测试基准中 12 个不同的卷积层上运行卷积算法时，为了使 GPU 中足够的任务量进行并行操作，每个卷积层的输入张量的批次维度 N_i 大小为 128，在 PyTorch 框架中生成每个卷积层对应参数的张量数据。

表 4-4 cuDNN 在测试基准中选择的最优算法

卷积层	cuDNN 自适应选择的最优卷积算法
Conv1	IMPLICIT_PRECOMP_GEMM
Conv2	IMPLICIT_PRECOMP_GEMM
Conv3	IMPLICIT_PRECOMP_GEMM
Conv4	IMPLICIT_PRECOMP_GEMM
Conv5	WINOGRAD_NONFUSED
Conv6	IMPLICIT_PRECOMP_GEMM
Conv7	IMPLICIT_PRECOMP_GEMM
Conv8	FFT
Conv9	WINOGRAD_NONFUSED
Conv10	WINOGRAD_NONFUSED
Conv11	WINOGRAD_NONFUSED
Conv12	IMPLICIT_PRECOMP_GEMM

4.4.2 运算性能结果分析.

本节是对 im2col+cuBLAS、cuDNN 和 im2win 三种算法在测试基准中的运算性能结果分析，测试基准中一共有 12 个独特的卷积层，其详细参数如表 3-3 所示。对于运算性能测试实验记录了两个评价指标，不同卷积算法的运行时间和 GFLOPS。

图 4-4 展示了不同算法在测试基准中的运行时间，并在图 4-5 中对实验结果以 im2win 为基准进行归一化处理。从图 4-4 可以看出，本文提出的 im2win 张量卷积算法在测试基准的 12 个卷积层中都比 im2col+cuBLAS 卷积算法更快，在大多数层上比 cuDNN 更快。此外，从图 4-5 中可以观察到，im2col+cuBLAS 卷积算法的运行时间最慢约为 im2win 张量卷积算法的 8.4 倍，平均约为 im2win 张量卷积算法的 4.3 倍。而 cuDNN 卷积的运行时间最慢约为 im2win 张量卷积算法的 1.8 倍，平均约为 im2win 张量卷积算法的 1.1 倍。

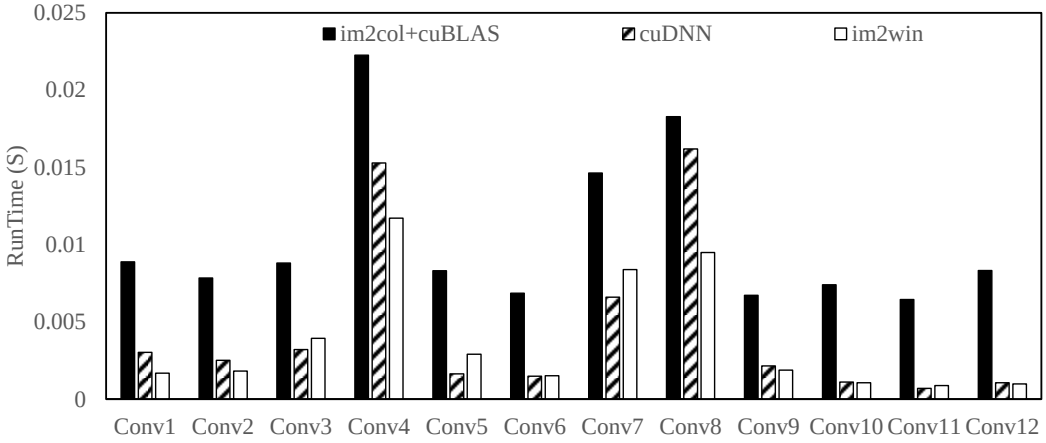

图 4-4 不同卷积算法在测试基准上的运行时间

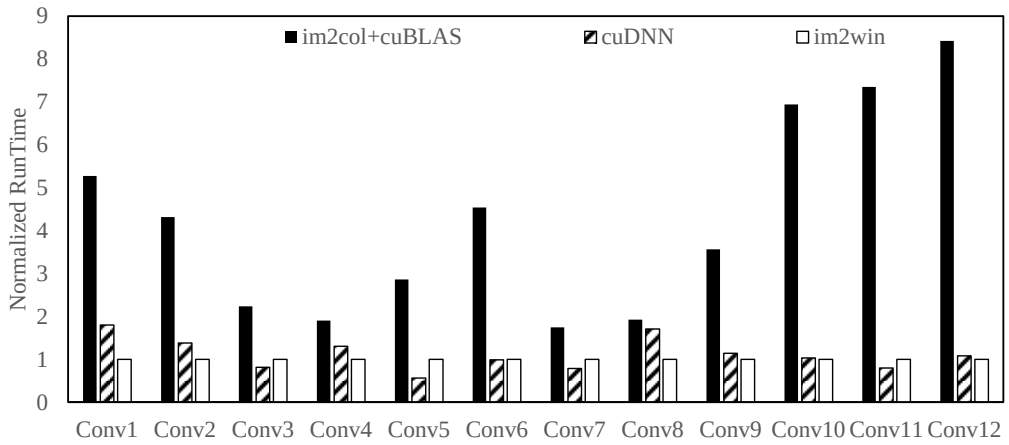

图 4-5 不同卷积算法在测试基准上归一化运行时间

图 4-6 展示了不同算法在测试基准中的 GFLOPS 性能结果，并在图 4-7 中对实验结果以 im2col+cuBLAS 卷积算法为基准进行归一化处理。从图 4-6 可以看出，本文提出的 im2win 张量卷积算法的 GFLOPS 性能在测试基准的所有卷积层上要远远高出 im2col+cuBLAS 卷积算法，在测试基准的大多数卷积层比 cuDNN 卷积更高。此外，在第 8 层到第 12 层 im2col+cuBLAS 卷积算法有明显的下降趋势，而 im2win 张量卷积算法没有明显趋势，这说明当输入张量的尺寸变小时，im2col+cuBLAS 卷积算法的性能会下降，而 im2win 张量卷积算法一定程度上改善了这种问题。从图 4-7 中可以观察到，im2win 张量卷积算法相对于基于 im2col 变换的卷积算法平均约为 3.5 倍的性能；而相对于 cuDNN 卷积表现出相当的性能，最高约为 1.8 倍的性能。

综合上述分析，本文提出的 im2win 张量卷积算法相比于 im2col+cuBLAS 卷积算法能够大幅度提升运算性能表现，相比 cuDNN 表现出相当的性能。但是需要注意，cuDNN 在测试基准中的每一层都选取了最优的卷积算法，这意味着 im2win 与 cuDNN 在每层的最优算法有相当的性能，而 cuDNN 选取的最优算法可能会导致使用更多的内存，本文将在下节内存占用分析章节中讨论。

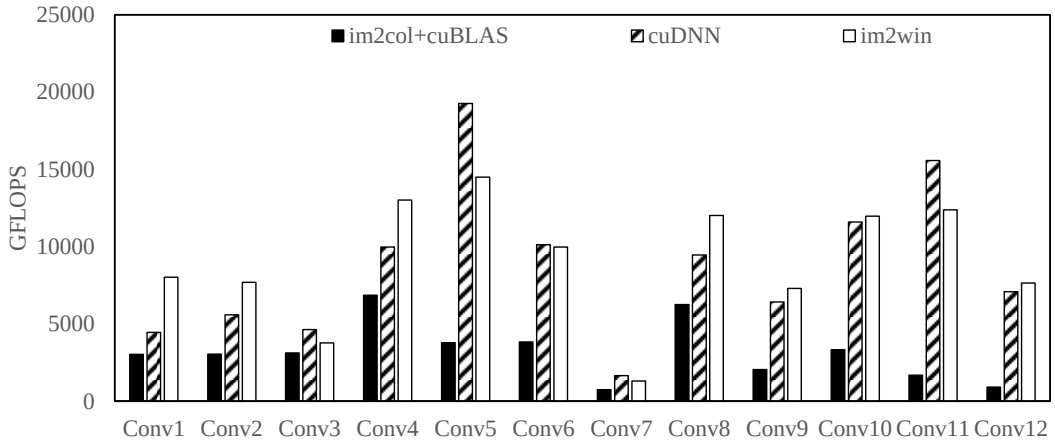

图 4-6 不同卷积算法在测试基准上 GFLOPS 性能

4.4.3 内存占用结果分析

本节是对 im2col+cuBLAS、cuDNN 和 im2win 三种算法在测试基准中的内存占用结果分析，测试基准中一共有 12 个独特的卷积层，其详细参数如表 3-3 所示。对于内存占用测试实验，分别记录了不同卷积算法的运行时内存占用的峰值。

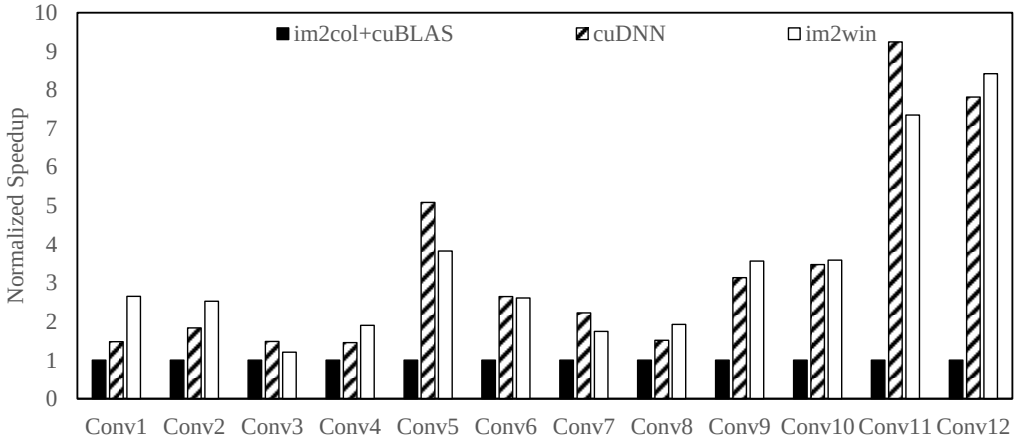

图 4-7 不同卷积算法在测试基准上归一化 GFLOPS 性能

图 4-8 中展示了不同算法在测试基准上的内存占用情况，与 im2col+cuBLAS 的卷积和 cuDNN 在测试基准上选择的最快的卷积相比，本文提出的 im2win 张量卷积算法在所有 12 个层上都明显使用了更少的内存空间。平均而言，im2win 张量卷积算法比 cuBLAS 节省了 23.1% 的内存，比 cuDNN 节省了 32.8% 的内存。cuDNN 在测试基准的部分卷积层上会选择基于 FFT 和基于 Winograd 的卷积，这两种卷积方法在一些场景下有很高的性能表现，但是内存占用会远远高于基于 GEMM 的卷积算法。考虑到单个 GPU 的内存通常不大（即使是 Nvidia A100 也最多只有 80GB 的内存），本文提出的 im2win 张量卷积

支持在单个 GPU 上处理更大规模的张量数据，比 cuBLAS 和 cuDNN 更可取。

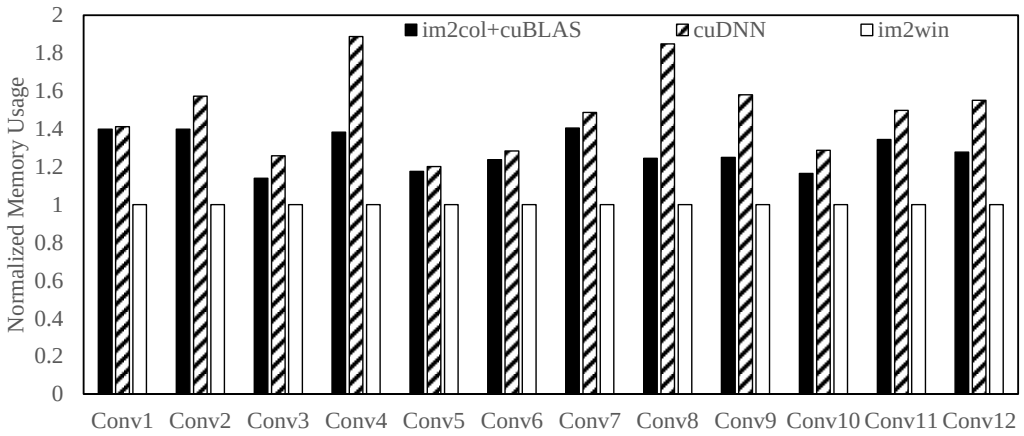

图 4-8 不同卷积算法在测试基准上归一化内存占用

4.4.4 消融实验结果分析

为了探索在 im2win 张量卷积算法中应用的不同优化技术对性能的影响，本节对预取、向量化加载和微内核等优化技术在高性能的 im2win 张量卷积算法中进行了消融实验。消融实验以高性能的 im2win 张量卷积算法作为基准，它包含了所有的优化技术。对于其他三种变体，每次移除一种技术来研究其有效性。图 4-9 显示了不同优化技术对 im2win 张量卷积的 GFLOPS 性能的影响。分别移除预取、向量化加载和微内核优化技术进行消融实验。在十二个基准测试中，微内核优化技术给出了最大的性能提升，其次是向量化加载，而预取对 im2win 张量卷积提升最差。微内核通过给每个线程分配更多的计算任务来减少数据处理的延迟，合理划分微内核可以最有效地提高性能。向量化加载可以一次加载多个标量来减少读取指令的数量，这也有一定的性能好处。双缓冲和预取是为了隐藏全局内存访问延迟而设计的，在 im2win 张量卷积中并不显著地改善性能，可能是因为共享内存已经充分地缓解了访问延迟。

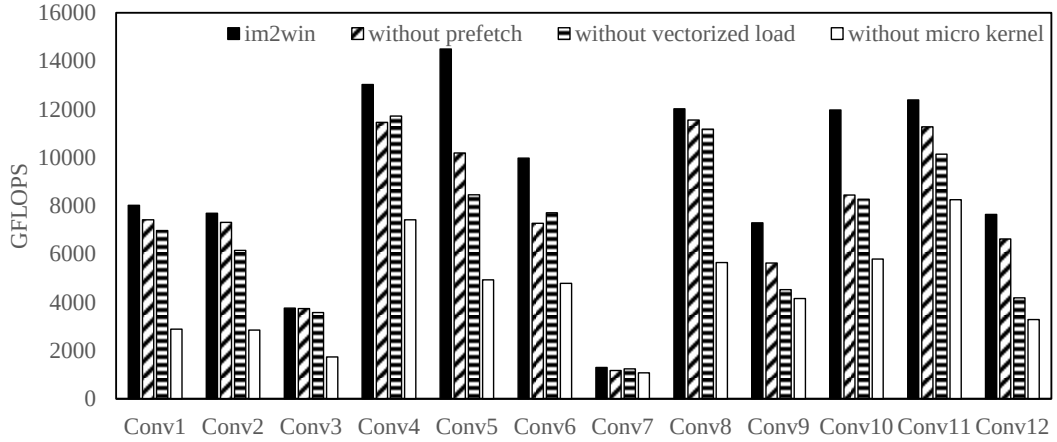

图 4-9 im2win 张量卷积算法在测试基准上的消融实验

4.5 本章小结

本章主要介绍了本文提出的张量卷积算法在 GPU 平台上的实现及优化，首先介绍分析了 GPU 平台体系结构的特性，包括 GPU 平台的软硬件结构和并行加速技术等，之后详细阐述了本文提出的张量卷积算法在 GPU 平台的实现方法及结合 GPU 平台的一系列优化技术实现的高性能卷积算法，最后通过一个完备了测试基准验证了本文提出的张量卷积算法相比于基于 im2col 的卷积算法和直接卷积算法有更好的性能表现，相比于基于 im2col 的卷积算法能够有效降低内存占用。

本文提出的张量卷积算法在 CPU 平台和 GPU 平台上有不同的实现方式和实验结果评价。这主要取决于 CPU 和 GPU 的架构、内存、并行能力等方面的差异。具体来说，本文算法在两种平台上的区别如下：在 CPU 平台上，利用多线程或向量化操作、缓存等技术，提高卷积运算效率和降低内存访问开销，根据卷积核大小和步长选择合适的算法；在 GPU 平台上，利用 CUDA 或其他框架、共享内存等技术，加速卷积运算和减少全局内存访问开销，根据卷积核大小和步长选择合适的线程块大小和网格大小。 实验结果评价方面，考虑以下指标：（1）内存占用：GPU 内存较小，本文算法在 GPU 上减少内存消耗更有意义。（2）加速比：GPU 更适合数值密集计算，本文算法在 GPU 上相对基础线性代数库加速比更明显。（3）准确性：本文算法在两种平台上都保证了计算结果与理论值之间误差在一定范围内。

第 5 章 总结与展望

上述几章描述了卷积神经网络中的卷积算法的计算过程和主要实现方式，针对传统的卷积算法内存消耗大、计算性能低下和实现方式不灵活的问题，提出了一种基于张量代数的卷积算法，在不同硬件平台上实现了高性能且低内存占用的张量卷积算法，并进行了相关实验验证。本章将总结本文所做的工作，并分析卷积神经网络性能优化的发展趋势。

5.1 本文工作总结

随着深度学习技术的快速发展和广泛应用，神经网络在各个领域取得了令人瞩目的准确性，但同时也导致了神经网络的深度和复杂度不断增加，从而带来了模型计算和访存的巨大开销。卷积神经网络作为深度学习中最重要的网络模型之一，在图像处理、语音识别、自然语言处理等领域具有广泛应用价值。然而，在常见的卷积神经网络中，卷积操作占据了模型总执行时间的 90%以上，因此卷积操作是卷积神经网络性能优化的重点。传统的卷积算法在处理高维数据、不同尺寸或稀疏数据、不同形式或结构化数据时，往往存在冗余计算、内存浪费和灵活性不足等问题。这些问题不仅限制了卷积神经网络的性能和效率，也影响了其在更多场景下的应用潜力。为此，本文以降低卷积算法的内存消耗和提高卷积算法的性能为目标，在不同硬件平台上实现了高效且低内存消耗的张量卷积算法，并进行了相关实验验证。本文主要研究内容包括以下三个方面：

（1）为了解决传统卷积算法在内存占用和内存访问方面的问题，本文提出了一种基于张量代数的卷积算法，即基于 im2win 变换的张量卷积。该算法分为两个步骤：第一步是 im2win 变换，它是一种将输入图像按照点乘窗口运算访问顺序重新排列数据的方法；第二步是张量乘法，它是一种将输入图像和卷积核都视为四维张量，并进行点乘运算来实现卷积操作的方法。相比于传统的 im2col 变换，im2win 变换能够显著减少内存开销，并保证点乘窗口内存访问连续性和数据重用性；相比于传统的矩阵乘法，张量乘法能够保留更多高维隐含信息，并避免重复计算和数据移动。这样就可以降低计算复杂度和内存消耗，并使得卷积实现更加灵活适应不同硬件平台和卷积场景。

（2）为了提高 CPU 平台上卷积算法的性能和效率，本文在 CPU 平台上实现了基于 im2win 变换的张量卷积算法，并采用了多种优化技术来加速运行过程。首先建立了一个分析优化空间与策略选择的数学分析模型，并结合 CPU 架构特

性和张量卷积计算过程设计完善该模型；然后利用 SIMD 向量指令集来加速向量乘法，并通过多线程并行来充分利用 CPU 资源；最后采用循环展开等技术来减少分支判断和函数调用开销。实验表明，本文提出的张量卷积算法与 CPU 平台上常见深度学习库进行对比，在内存消耗和计算性能方面都有较大提升。

（3）为了提高 GPU 平台上卷积算法的性能和稳定性，本文在 GPU 平台上实现了基于 im2win 变换的张量卷积算法，并采用了多种优化技术来进一步提高运行效率。首先根据 GPU 的 CUDA 架构模型，对张量卷积算法的索引进行重构，使得该算法可以更好地映射在 GPU 的架构上；然后利用 Tiling 策略来减少全局内存访问，并利用共享内存与寄存器来缓存局部数据；最后采用微内核设计、向量化加载/存储、双缓冲与数据预取等技术来提高计算吞吐率。实验表明，本文提出的张量卷积算法与 GPU 平台上常见的深度学习库对比，在减少内存消耗的同时，计算性能更优或相当。

5.2 展望

本文以降低卷积算法的内存占用和提升卷积算法的性能为目标，基于张量代数理论，在不同硬件平台上实现了高性能且低内存占用的张量卷积算法，最终在很多场景下都取得了很好的表现。但在实验的过程中发现，仍有一些不足之处，需要在未来的工作中继续改进，后续研究可以从以下几个方面开展：

（1）多精度数据支持。本文提出的方法仅在单精度浮点数（FP32）上实现，而深度学习模型前向推理对于数据精度的要求不高，可以使用半精度浮点数（FP16），甚至是 8bit 整数（Int8）来保存数据。这样可以减少深度学习模型所需的存储空间，并且可以提高模型推理的性能。因此，在后续工作中需要增加对 FP16 和 Int8 的支持，这也有利于提高本文方法在移动设备或边缘计算场景下的适应性和可移植性。

（2）多算子融合优化。本文主要关注卷积算子本身，而在神经网络中还有其他算子与之配合使用，例如激活函数、批规范化等。如果能将多个算子融合在一个内核函数里实现，则可以充分利用硬件资源和并行能力，从而进一步提高深度神经网络计算速度。

（3）不同平台的可移植性。本文提出的方法基于张量代数理论，可以在不同硬件平台上实现高性能且低内存占用的张量卷积算法。为了实现 CPU 与 GPU 不同平台的可移植性，需要考虑以下几个方面：（a）编程语言和框架的选择。不同的硬件平台可能支持不同的编程语言和框架，例如 C/C++、Python、CUDA、OpenCL 等。为了保证代码的可移植性，需要选择一种通用且高效的编程语言和框架，或者使用跨平台的工具和库，例如 TensorFlow、PyTorch 等。

（b）算法和数据结构的设计。不同的硬件平台可能有不同的计算能力和内存特性，例如 CPU 通常有较少但更强大的核心，而 GPU 通常有较多但更专用的核心。为了保证算法和数据结构的可移植性，需要考虑不同平台的特点，例如并行度、缓存大小、内存带宽等，并进行相应的优化和调整。（c）任务划分和性能评估。不同的硬件平台可能有不同的性能表现，例如 CPU 通常适合处理串行或低并行度的任务，而 GPU 通常适合处理高并行度或密集型的任务。为了保证性能的可移植性，需要在不同平台上进行更加充分的测试和评估，并比较不同平台之间的性能差异和优劣。

（4）针对特殊场景下的卷积优化。本文设计并实现了通用且高效地张量卷积算法，在大多数场景下都有较好地表现。但是，在某些特殊场景下可能还有更适合或更优化地方法存在。例如，在稀疏或低秩张量上进行卷积时是否可以利用其结构特点进行压缩或加速？在处理超大规模或超高维度地张量时是否可以采用分布式或并行计算方式？这些问题值得进一步探索。

参考文献

[1] Turing A M. Computing machinery and intelligence (1950)[J]. The Essential Turing: the Ideas That Gave Birth to the Computer Age, 2012: 433-464.

[2] LeCun Y, Bengio Y, Hinton G. Deep learning[J]. nature, 2015, 521(7553): 436-444.

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

[4] Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors[J]. nature, 1986, 323(6088): 533-536.

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

[6] 张仕良. 基于深度神经网络的语音识别模型研究[D]. 合肥: 中国科学技术大学, 2017.

[7] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Communications of the ACM, 2017, 60(6): 84-90.

[8] Law S, Seresinhe C I, Shen Y, et al. Street-Frontage-Net: urban image classification using deep convolutional neural networks[J]. International Journal of Geographical Information Science, 2020, 34(4): 681-707.

[9] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9.

[10] Melis G, Dyer C, Blunsom P. On the state of the art of evaluation in neural language models[J]. arXiv preprint arXiv:1707.05589, 2017.

[11] Ardila D, Kiraly A P, Bharadwaj S, et al. End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography[J]. Nature medicine, 2019, 25(6): 954-961.

[12] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.

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

[14] 唐贤伦, 杜一铭, 刘雨微, 等. 基于条件深度卷积生成对抗网络的图像识别方法[J]. 自动化学报, 2018, 44(5): 855-864.

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

[16] Ren S, He K, Girshick R, et al. Faster r-cnn: Towards real-time object detection with region proposal networks[J]. Advances in neural information processing systems, 2015, 28.

[17] 张顺, 龚怡宏, 王进军. 深度卷积神经网络的发展及其在计算机视觉领域的应用[J]. 计算机学报, 2019, 42(3): 453-482.

[18] Ahonen T, Hadid A, Pietikäinen M. Face recognition with local binary patterns[C]//Computer Vision-ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic,

May 11-14, 2004. Proceedings, Part I 8. Springer Berlin Heidelberg, 2004: 469-481.

[19] Parkhi O M, Vedaldi A, Zisserman A. Deep face recognition[J]. 2015.

[20] Dong C, Loy C C, He K, et al. Image super-resolution using deep convolutional networks[J]. IEEE transactions on pattern analysis and machine intelligence, 2015, 38(2): 295-307.

[21] Zhang Y, Tian Y, Kong Y, et al. Residual dense network for image super-resolution[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 2472-2481.

[22] Chen Y. Convolutional neural network for sentence classification[D]. University of Waterloo, 2015.

[23] Hassan A, Mahmood A. Deep learning for sentence classification[C]//2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT). IEEE, 2017: 1-5.

[24] Ma N, Zhang X, Zheng H T, et al. Shufflenet v2: Practical guidelines for efficient cnn architecture design[C]//Proceedings of the European conference on computer vision (ECCV). 2018: 116-131.

[25] Sze V, Chen Y H, Yang T J, et al. Efficient processing of deep neural networks: A tutorial and survey[J]. Proceedings of the IEEE, 2017, 105(12): 2295-2329.

[26] Crowley E J, Gray G, Storkey A J. Moonshine: Distilling with cheap convolutions[J]. Advances in Neural Information Processing Systems, 2018, 31.

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

[28] Zhong Z, Jin L, Xie Z. High performance offline handwritten chinese character recognition using googlenet and directional feature maps[C]//2015 13th international conference on document analysis and recognition (ICDAR). IEEE, 2015: 846-850.

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

[30] 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. 2016: 2818-2826.

[31] Zhang J, Franchetti F, Low T M. High performance zero-memory overhead direct convolutions[C]//International Conference on Machine Learning. PMLR, 2018: 5776-5785.

[32] Heinecke A, Henry G, Hutchinson M, et al. LIBXSMM: accelerating small matrix multiplications by runtime code generation[C]//SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016: 981-991.

[33] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing[C]//Tenth international workshop on frontiers in handwriting recognition. Suvisoft, 2006.

[34] Mathieu M, Henaff M, LeCun Y. Fast training of convolutional networks through ffts[J]. arXiv preprint arXiv:1312.5851, 2013.

[35] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library[J]. Advances in neural information processing systems, 2019, 32.

[36] Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning[C]//Osdi. 2016, 16(2016): 265-283.

[37] 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. 2014: 675-678.

[38] Panagakis Y, Kossaifi J, Chrysos G G, et al. Tensor methods in computer vision and deep learning[J]. Proceedings of the IEEE, 2021, 109(5): 863-890.

[39] Matthews D A. High-performance tensor contraction without transposition[J]. SIAM Journal on Scientific Computing, 2018, 40(1): C1-C24.

[40] 杨礼吉,王家祺,景丽萍,于剑.基于张量计算的卷积神经网络语义表示学习[J].计算机学报,2023,46(03):568-578.

[41] 宋冰冰,张浩,吴子锋,刘俊晖,梁宇,周维.自动化张量分解加速卷积神经网络[J].软件学报,2021,32(11):3468-3481.DOI:10.13328/j.cnki.jos.006057.

[42] Oseledets I V, Savostyanov D V, Tyrtyshnikov E E. Linear algebra for tensor problems[J]. Computing, 2009, 85: 169-188.

[43] Kolda T G, Bader B W. Tensor decompositions and applications[J]. SIAM review, 2009, 51(3): 455-500.

[44] Nie C, Wang H. Tensor neural networks via circulant convolution[J]. Neurocomputing, 2022, 483: 22-31.

[45] Novikov A, Podoprikhin D, Osokin A, et al. Tensorizing neural networks[J]. Advances in neural information processing systems, 2015, 28.

[46] Sze V, Chen Y H, Yang T J, et al. Efficient processing of deep neural networks: A tutorial and survey[J]. Proceedings of the IEEE, 2017, 105(12): 2295-2329.

[47] NVIDIA CUDA Toolkit Documentation . https://docs.nvidia.com/cuda/index.html ,2021-03-21.

[48] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit[C]//Proceedings of the 44th annual international symposium on computer architecture. 2017: 1-12.

[49] HUAWEI Ascend AI Processor [EB/OL]. https://e.huawei.com/en/products/cloud-computing-dc/atlas/ascend-series ,2021-03-21.

[50] Kolda T G. Orthogonal tensor decompositions[J]. SIAM Journal on Matrix Analysis and Applications, 2001, 23(1): 243-255.

[51] Kolda T G, Sun J. Scalable tensor decompositions for multi-aspect data mining[C]//2008 Eighth IEEE international conference on data mining. IEEE, 2008: 363-372.

[52] Phan A H, Sobolev K, Sozykin K, et al. Stable low-rank tensor decomposition for compression of convolutional neural network[C]//Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIX 16. Springer International Publishing, 2020: 522-539.

[53] Oymak S, Soltanolkotabi M. Learning a deep convolutional neural network via tensor decomposition[J]. Information and Inference: A Journal of the IMA, 2021, 10(3): 1031-1071.

[54] Denton E L, Zaremba W, Bruna J, et al. Exploiting linear structure within convolutional

networks for efficient evaluation[J]. Advances in neural information processing systems, 2014, 27.

[55] Rigamonti R, Sironi A, Lepetit V, et al. Learning separable filters[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2013: 2754-2761.

[56] Han S, Pool J, Tran J, et al. Learning both weights and connections for efficient neural network[J]. Advances in neural information processing systems, 2015, 28.

[57] Chen W, Wilson J, Tyree S, et al. Compressing neural networks with the hashing trick[C]//International conference on machine learning. PMLR, 2015: 2285-2294.

[58] 柯圣财, 赵永威, 李弼程, 等. 基于卷积神经网络和监督核哈希的图像检索方法[J]. 电子学报, 2017, 45(1): 157.

[59] Qin C, Liu E, Feng G, et al. Perceptual image hashing for content authentication based on convolutional neural network with multiple constraints[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2020, 31(11): 4523-4537.

[60] Chen H, Hu C, Lee F, et al. A supervised video hashing method based on a deep 3d convolutional neural network for large-scale video retrieval[J]. Sensors, 2021, 21(9): 3094.

[61] Molchanov P, Tyree S, Karras T, et al. Pruning convolutional neural networks for resource efficient inference[J]. arXiv preprint arXiv:1611.06440, 2016.

[62] Liu Z, Li J, Shen Z, et al. Learning efficient convolutional networks through network slimming[C]//Proceedings of the IEEE international conference on computer vision. 2017: 2736-2744.

[63] He Y, Zhang X, Sun J. Channel pruning for accelerating very deep neural networks[C]//Proceedings of the IEEE international conference on computer vision. 2017: 1389-1397.

[64] 孙彦丽, 叶炯耀. 基于剪枝与量化的卷积神经网络压缩方法[J]. 计算机科学, 47(8): 261-266.

[65] Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network[J]. arXiv preprint arXiv:1503.02531, 2015.

[66] 高钦泉, 赵岩, 李根, 等. 基于知识蒸馏的超分辨率卷积神经网络压缩方法[J]. 计算机应用, 2019, 39(10): 2802-2808.

[67] Zhao B, Cui Q, Song R, et al. Decoupled knowledge distillation[C]//Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition. 2022: 11953-11962.

[68] Zhu Z, Hong J, Zhou J. Data-free knowledge distillation for heterogeneous federated learning[C]//International Conference on Machine Learning. PMLR, 2021: 12878-12889.

[69] Georganas E, Avancha S, Banerjee K, et al. Anatomy of high-performance deep learning convolutions on simd architectures[C]//SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018: 830-841.

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

[71] San Juan J , Kwon Y , Kim H . Optimizing Convolutional Neural Networks on Intel® Xeon® Scalable Processors[J]. Intel Corporation White Paper, 2020.

[72] NVIDIA C U B L A S Library User Guide[C]. NVIDIA Corporation White Paper. 2008.

[73] NVIDIA C U D A N N Library User Guide[C]. NVIDIA Corporation White Paper. 2017.

[74] Lipshitz B, Ballard G, Demmel J, et al. Communication-avoiding parallel strassen: Implementation and performance[C]//SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012: 1-11.

[75] Smith T M, Van De Geijn R, Smelyanskiy M, et al. Anatomy of high-performance many-threaded matrix multiplication[C]//2014 IEEE 28th International Parallel and Distributed Processing Symposium. IEEE, 2014: 1049-1059.

[76] Anderson A, Vasudevan A, Keane C, et al. Low-memory gemm-based convolution algorithms for deep neural networks[J]. arXiv preprint arXiv:1709.03395, 2017.

[77] Trusov A V, Limonova E E, Nikolaev D P, et al. p-im2col: Simple yet efficient convolution algorithm with flexibly controlled memory overhead[J]. IEEE Access, 2021, 9: 168162-168184.

[78] Cho M, Brand D. Mec: memory-efficient convolution for deep neural network[C]//International Conference on Machine Learning. PMLR, 2017: 815-824.

[79] Mathieu M, Henaff M, LeCun Y. Fast training of convolutional networks through ffts[J]. arXiv preprint arXiv:1312.5851, 2013.

[80] Lavin A, Gray S. Fast algorithms for convolutional neural networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 4013-4021.

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

[82] Huang X, Wang Q, Lu S, et al. Evaluating FFT-based Algorithms for Strided Convolutions on ARMv8 Architectures?[J]. ACM SIGMETRICS Performance Evaluation Review, 2022, 49(3): 28-29.

[83] Huang X, Wang Q, Lu S, et al. NUMA-aware FFT-based convolution on ARMv8 many-core CPUs[C]//2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE, 2021: 1019-1026.

[84] Dongarra J J, Du Croz J, Hammarling S, et al. A set of level 3 basic linear algebra subprograms[J]. ACM Transactions on Mathematical Software (TOMS), 1990, 16(1): 1-17.

[85] Wang E, Zhang Q, Shen B, et al. Intel math kernel library[J]. High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures, 2014: 167-188.

[86] Xianyi Z, Qian W, Yunquan Z. Model-driven level 3 BLAS performance optimization on Loongson 3A processor[C]//2012 IEEE 18th international conference on parallel and distributed systems. IEEE, 2012: 684-691.

[87] Intel, “Intel oneAPI deep neural network library (oneDNN).” Accessed: Mar. 09, 2022. [Online]. Available: https://github.com/ oneapi-src/oneDNN

[88] Gunnels J A, Henry G M, Van De Geijn R A. A family of high-performance matrix multiplication algorithms[C]//Computational Science—ICCS 2001: International Conference San Francisco, CA, USA, May 28–30, 2001 Proceedings, Part I 1. Springer Berlin Heidelberg, 2001: 51-60.

[89] Wall D W. Limits of instruction-level parallelism[C]//Proceedings of the fourth international

conference on Architectural support for programming languages and operating systems. 1991: 176-188.

[90] Rau B R, Fisher J A. Instruction-level parallelism[M]//Encyclopedia of Computer Science. 2003: 883-887.

[91] Oplinger J T, Heine D L, Lam M S. In search of speculative thread-level parallelism[C]//1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425). IEEE, 1999: 303-313.

[92] Blake G, Dreslinski R G, Mudge T, et al. Evolution of thread-level parallelism in desktop applications[C]//Proceedings of the 37th annual international symposium on Computer architecture. 2010: 302-313.

[93] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming[J]. IEEE computational science and engineering, 1998, 5(1): 46-55.

[94] Espasa R, Valero M. Exploiting instruction-and data-level parallelism[J]. IEEE micro, 1997, 17(5): 20-27.

[95] Minakova S, Tang E, Stefanov T. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs[C]//Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20. Springer International Publishing, 2020: 18-35.

[96] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, “Analytical modeling is enough for high-performance blis,” ACM Transactions on Mathematical Software (TOMS), vol. 43, no. 2, pp. 1–18, 2016.

[97] Lei L, Wang T J. Determining optimal cyclic hoist schedules in a single-hoist electroplating line[J]. IIE transactions, 1994, 26(2): 25-33.

[98] J. J. Dongarra and A. Hinds, “Unrolling loops in fortran,” Software: Practice and Experience, vol. 9, no. 3, pp. 219–226, 1979.

[99] Weinhardt M, Luk W. Pipeline vectorization[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2001, 20(2): 234-248.

[100] Eichenberger A E, Wu P, O'brien K. Vectorization for SIMD architectures with alignment constraints[J]. Acm sigplan notices, 2004, 39(6): 82-93.

[101] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, The art of multiprocessor programming. Newnes, 2020.

[102] Stone J E, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems[J]. Computing in science & engineering, 2010, 12(3): 66.

[103] Zhou Y, Yang M, Guo C, et al. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators[C]//2021 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2021: 214-225.

[104] M. Wolfe, “Iteration space tiling for memory hierarchies,” in Proceed- ings of the Third SIAM Conference on Parallel Processing for Scientific Computing. USA: Society for Industrial and Applied Mathematics, 1989, p. 357–361.

[105] Liedtke J. On micro-kernel construction[J]. ACM SIGOPS Operating Systems Review, 1995,

29(5): 237-250.

[106] Huang J, Yu C D, van de Geijn R A. Implementing Strassen's algorithm with CUTLASS on NVIDIA Volta GPUs[J]. arXiv preprint arXiv:1808.07984, 2018.

[107] Vanderwiel S P, Lilja D J. Data prefetch mechanisms[J]. ACM Computing Surveys (CSUR), 2000, 32(2): 174-199.

硕士期间发表的学术论文和参与的科研项目

一、发表的学术论文

[1] Shuai Lu, Jun Chu, and Xu T. Liu. "Im2win: Memory Efficient Convolution On SIMD Architectures." *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022.（oral，已发表，EI 检索，高性能计算和嵌入式计算交叉领域的顶级会议之一）

[2] Shuai Lu, Jun Chu, and Xu T. Liu. "Im2win: An Efficient Convolution Paradigm On GPU." *2023 International Conference on Parallel and Distributed Computer (Euro-Par)*. Springer, 2023.（oral，已录用，CCF-B 类，欧洲并行处理和分布式计算领域的顶级会议之一）

二、参与的科研项目

a) 国家自然科学基金项目：面向安防监控的复杂环境下长时目标跟踪方法研究，No. 62162045， 2022.01-2025.12.

b) 江西省科技支撑计划项目-工业领域一般项目：面向视频监控的跨摄像机目标跟踪关键技术研究，20192BBE50073， 2019.01.01-2021.12.

致谢

三年的硕士生涯即将画上句号，这是一场梦幻般的旅程，让我经历了无数的风雨和彩虹。在这段充满收获和成长的时光里，我要感谢所有给予我支持和帮助的人，在此向他们致以最诚挚的谢意。

首先，我要感谢我的导师储珺教授。储老师不仅在学术上给予了我悉心的指导和培养，还在生活上给予了我温暖的关怀和鼓励。储老师严谨的治学精神、博学的学术造诣和敬业的科研态度，都深深地影响了我，并让我受益终身。能成为储老师的学生，是我的荣幸和幸运。在此，我衷心地祝福储老师身体健康，工作顺利，幸福快乐！

其次，我要感谢我的表哥郭卢安政博士以及华盛顿大学的刘旭博士。他们是我进入高性能计算领域的引路人，也是我完成本论文的重要合作者。他们在技术上给予了我很多宝贵的建议和指导，让我在科研上有了突破和进步。他们在生活上给予了我很多关心和鼓励，让我在困难面前有了勇气和信心。他们是我的良师益友，也是我的榜样。

此外，我还要感谢实验室所有的老师和同学。感谢老师们为我们提供了优良的科研环境和条件，感谢同学们为我们营造了和谐的学习氛围。在实验室里，我们相互交流、相互学习、相互进步。在这里，我收获了知识、友情和成长。

最后，感谢我的家人。感谢他们一直给予我的爱、鼓励、关怀和包容，感谢他们为我付出的一切！正是他们的支持才让我不断成长与坚强，不断进取与奋斗。他们永远是我精神的支柱，力量的源泉！

再次感谢所有述及和未述及的人，感谢有你们的陪伴，让我不断超越自己，迎接挑战。在此，我向你们致以最诚挚的敬意和最美好的祝福！