量化快速卷积方法在 ARM 设备上的优化研究

（申请清华大学工程硕士专业学位论文）

培 养 单 位 ： 软件学院

工 程 领 域 ： 软件工程

申 请 人 ： 温 发 琥

指 导 教 师 ： 丁 贵 广 副教授

二〇二〇年五月

Research on Optimization of Quantitative Fast Convolution Method on ARM Devices

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the professional degree of

Master of Engineering

by

Wen Fahu

(Software Engineering)

Thesis Supervisor: Associate Professor Ding Guiguang

May, 2020

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。

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

来自于边缘设备的数据越来越丰富，边缘设备的计算能力也有所提升，在此机遇下，边缘设备对于智能应用的需求也在不断升级。而结合物联网（IOT）与人工智能（AI）的应用 AIoT，也会在即将到来的时代，对很多领域实现重新定义。目前移动端最为广泛使用的芯片架构为 ARM Cortex-A，相比于计算资源充足的桌面端和服务端，该平台的计算能力相当有限。而边缘设备有限的计算能力和当前机器学习模型，特别是深度神经网络，庞大的计算需求之间的矛盾成为在边缘设备上部署智能应用的一大难点。对于视觉任务而言，卷积神经网络是当前最为主要的模型，而其中的计算瓶颈便是卷积操作，而 3x3 卷积是目前卷积网络中占比最高的卷积操作。加速卷积的实现，特别是 3x3 卷积，是加速卷积网络模型并且实现其在边缘设备上部署最为有效也最为根本的手段。近年来被应用于卷积网络计算的 Winograd 卷积是目前实现中运算复杂度最低的卷积算法，并且特别适用于 3x3 卷积这种小尺度的卷积。同时，另外一方面，模型量化方法近年来日臻完善。通过量化方法，以低精度数据表示的网络模型仍然可以在视觉任务上有着较高的准确率，同时实现模型的压缩，运行时计算吞吐量的提升。

结合上述两点，实现量化卷积网络的 Winograd 卷积，将有效提升卷积网络在移动端的部署效率。而当前在移动端最为主流的卷积实现方法为 Im2Col 算法，Winograd 算法相比之下有着明显的算法复杂度优势。本文针对 Winograd 整数卷积的优化问题，提出了局部区域多通道 Winograd 卷积算法，在计算资源有限的 ARM 设备上，充分利用了 Winograd 卷积的空间局部性以及通道可并行化，有效解决了整数卷积操作在嵌入式设备的加速问题。另外，为了进一步提升整数卷积的速度，本文利用 ARM NEON 指令的高效并行计算能力，针对于轻量化网络的参数特征，提出了缓存友好、计算吞吐高的中小规模整数矩阵乘法，进一步实现了计算效率的优化。最后，本文将优化的卷积算法实现集成入计算内核库 QNNPACK，同 TensorFlow Lite 量化卷积计算对比，同等参数配置的卷积计算操作，在 ARM Cortex A72 设备上实现了 2 倍加速效果。

关键词：Winograd；ARM NEON；卷积；量化；GEMM

Abstract

The demand for intelligent computing on edge devices is increasing with the improvement of computation capacity of edge devices and the demand for more and more complex applications due to the richness of data. The application of AIoT, which combines the Internet of Things (IOT) and artificial intelligence (AI), is also about to redefine many fields. At present, the most widely used chip architecture of the mobile devices is ARM Cortex-A. Compared with the desktop and server with sufficient computing resources, the computing capability of the platform is quite limited. The contradiction between the limited computing capability of edge devices and huge computation demand of current machine learning models, especially deep neural networks, has become a major difficulty in deploying intelligent applications on edge devices.

For visual tasks, the convolutional neural network is currently the dominated model, and its computational bottleneck is the convolution operation. Moreover, the 3x3 convolution is the mostly used convolutional operation in the current convolution network design. The implementation of efficient convolution, especially 3x3 convolution, is the most effective and fundamental means to accelerate the convolutional network model and implement its deployment on edge devices. In recent years, Winograd convolution, which has been widely used in digital signal processing, is applied to the computation of convolutional networks and proves to be the convolution algorithm with the lowest computational complexity in the current implementation. Also Winograd convolution is particularly suitable for small-scale convolution, such as the widely used 3x3 convolution.

On the other hand, model quantification methods have become more sophisticated in recent years. Through the quantization approach, the network model represented by low-precision data can still have a high accuracy in visual tasks. Compressed model size and increased computation throughput make quantized models ideal to be deployed on mobile devices.

To conclude, the implementation of quantized Winograd convolution will effectively improve the deployment efficiency of the convolutional network on the mobile devices. At present, the most mainstream implementation of convolution on the mobile devices is the Im2Col algorithm, and the Winograd algorithm has obvious advantages in algorithm complexity in comparison. In this paper, a spatially local multi-channel Wino-

grad convolution algorithm is proposed for the optimization problem of Winograd integer convolution. On ARM devices with limited computing resources, the spatial locality of Winograd convolution and the parallelization of channel dimension are fully utilized to effectively accelerated quantized convolution on embeded devices; Secondly, in order to further improve the speed of integer convolution, this paper utilizes the efficient parallel computing capability of ARM NEON instructions and addressing the parameter characteristics of lightweight networks, proposes cache-friendly and high computational throughput small-to-medium-scale integer matrix multiplication further optimizes the calculation efficiency. Finally, this paper integrates the optimized convolution algorithm into the computation kernel library QNNPACK. Compared with the TensorFlow Lite quantization convolution computation, the convolution operation with the same parameter configuration, achieved a 2x acceleration on the ARM Cortex A72 device.

Key Words: Winograd; ARM NEON; Convolution; Quantization; GEMM

目 录

主要符号对照表

HPC	高性能计算 (High Performance Computing)
SIMD	单指令多数据（Single Instruction Multiple Data）
ILP	指令级并行化（Instruction Level Parallelization）
TLP	线程级并行化（Thread Level Parallelization）
GEMM	通用矩阵乘法（GEneral Matrix Multiplication）
packing	打包
Register Blocking	寄存器分块
Cache Blocking	缓存分块
micro kernel	微内核
block panel	子板块
vector register	矢量寄存器
OLA	重叠加 (Overlap Add)
FFT	快速傅立叶变换 (Fast Fourier Transform)
DFT	离散傅立叶变换 (Discrete Fourier transform)

第 1 章　引言

1.1　研究背景

神经网络赋能嵌入式设备智能是当前智能应用的一大热点。作为神经网络与边缘计算的交叉领域，这一研究需要调和深度神经网络模型计算资源消耗巨大与嵌入式设备端计算资源有限，功耗低的矛盾，权衡模型的精度和在嵌入式硬件环境中的运行效率，实现智能应用的高效部署。一方面，在网络模型设计中，很多适用于低计算复杂度的轻量级模型相继提出，如 MobileNet 系列，MnasNet 等，实现了在低参数量和低计算复杂度的场景在对应任务场景下的高精度表现。另外一方面，关于神经网络的压缩方法的研究如各类剪枝、量化方法也取得了较为成熟的应用，使得深度网络模型的计算得以简化。

智能手机作为使用最为广泛和频繁的嵌入式设备也便成为边缘智能最为热门的应用场景。同时，智能化的服务也会为智能手机的使用带来颠覆性的体验。目前主流的智能手机的硬件环境均使用 ARMv8 指令架构的芯片，而这一系列的芯片具有 SIMD 扩展 NEON，这一功能使得在 ARM 平台上实现高效的矩阵计算成为可能。而矩阵计算正是神经网络模型在计算中的核心。“GEMM is at the heart of deep learning”（Pete Warden，technical lead of the mobile and embedded TensorFlow group, in Google’s Brain team）。

1.1.1　边缘智能计算的必要性

尽管云计算的高速发展和越来越快的网络技术（5G 网络）正在深刻的改变着数据获取以及被处理的手段，很多计算和业务也有着向云端靠拢的趋势，但是在可见的未来，在边缘设备上智能计算的需求明显是远远高于在云端的。尽管云端的智能计算比如 Microsoft Azure Conginitive Serivce，AWS SageMaker 等服务具有丰富的计算资源和应用生态，但仍然有很多场景是这些智能服务所无法覆盖的。比如很多物联网设备的智能计算，自动驾驶汽车，可穿戴设备等等，在这些具有相当实时性要求的边缘智能设备上，远在云端的请求延时是无法容忍的。除此之外，很多边缘设备也无法接入到云服务中，在大多情形下，这些设备需要独立运行，自我决策。而即便是可以计入云计算的设备，智能物联网设备生成的庞大的数据量，也让云方案的可规模化大打折扣，同时大规模数据的传输也让网络陷入极大的压力。因此，在边缘设备上仍然需要执行相当一部分的计算负载，以实现安全，自动，实时的智能决策。而这又同边缘嵌入式设备有限的计算资源和极低的功耗相

矛盾，特别是智能应用往往是计算密度极高的，这为实现边缘智能（edge AI）带来了相当的挑战。

1.1.2　ARM 架构计算设备的应用前景

基于 ARM 架构的芯片的计算应用不仅仅是在嵌入式计算场景下的主导，同时近年来，基于 Arm 的处理器已经出现在 HPC 领域，为主要由 x86 处理器主导现状提供了一种替代方案。对于机器学习应用而言，各类服务器级别的处理器，比如 x86 架构的处理器（Intel，AMD）和 GPGPU（Nvidia），目前均已经有了相对成熟的生态和广泛的应用。ARM 架构的处理器为这一现状提供了一种额外的可能和新的选择。Marvell（ThunderX2），Ampere（eMAG），华为（Kunpeng 920），富士通（A64FX）和亚马逊（Graviton）等处理器制造商均推出基于 Arm 的服务器级处理器，将 HPC 应用迁移向成本更低，计算性价比更高的 ARM 平台。

ThunderX2 处理器是 Marvell 的第一个大规模商用的服务器级处理器。ThunderX2 处理器使用 Armv8 指令集，并且专门针对服务器工作负载而设计。Arm Neon 技术是 Arm Cortex-A 和 Cortex-R 系列处理器的高级单指令多数据（SIMD）架构扩展。Neon 寄存器被视为相同数据类型元素的向量，Neon 指令同时对多个元素进行操作。NEON 支持多种数据类型，包括浮点和整数运算。NEON 旨在通过加速音频和视频的编码和解码，用户界面，2D / 3D 图形和游戏来改善多媒体体验。Neon 还可以加速信号处理算法和功能，以加快诸如音频和视频处理，语音和面部识别，计算机视觉和深度学习之类的应用程序。

1.1.3　卷积网络在 ARM 设备上的应用现状与对应计算优化

卷积神经网络的使用使得嵌入式场景的视觉应用获得了普遍的效果提升。而移动端的 ARM 设备同桌面端及服务端具备大量计算资源的设备不同，ARM 移动设备往往具有低功耗，计算能力相对较低，内存有限的劣势，而将计算密集的卷积网络计算部署在 ARM 设备也需要针对性的实现多种优化。

模型量化方法是一种针对于模型运算加速和功耗的降低均有着显著效益的网络模型轻量化方法，这一领域的研究证明，深度神经网络在低精度的表示下仍然具有着相当的准确率和在对应任务上的表现力。具有量化参数的低位数学运算与对神经网络的中间计算进行量化相结合，可带来较大的计算增益和更高的性能。除了性能优势之外，由于内存访问成本降低同时计算效率提升，量化神经网络还提高了功耗效率。使用低位量化数据仅需要较少的片内和片外数据移动，从而减少了存储器带宽并节省了大量能量。较低精度的数学运算（例如 8 位整数乘法）消耗

更少的能量并提高了计算效率，从而降低了功耗。此外，减少用于表示神经网络参数的位数可以减少存储空间。而在产业界的应用中，结合嵌入式设备硬件条件，实现对于量化网络（8 位整数矩阵）计算的原生支持以及真正意义上的端到端 low precision 计算，是实现网络模型性能加速，功耗降低的重要途径。

对于视觉任务而言，卷积神经网络的应用占据着主导地位。在这一网络模型中，卷积操作占据了绝大多数的计算，从而性能加速和功耗优化的瓶颈为卷积操作的实现。而在卷积的计算实现中，目前主要有 GEMM-based Convolution (im2col) 和快速卷积算法（FFT-based Convolution，Winograd Convolution）。基于 GEMM 的卷积算法的实现相对简单，在对数据实现简单的重组之后，便可以直接调用成熟的矩阵乘法实现卷积操作。而快速卷积方法，尽管具有比较高的实现难度，但可以有效实现卷积计算复杂度的降低。结合应用于移动端的轻量化网络的特点，网络中主要的卷积操作，其卷积核均为 3x3。在理论研究和实践中，winograd 快速卷积方法对于 3x3 卷积具有明显的计算复杂度优势。而另外一方面，结合量化网络参数的特点，由于 winograd 卷积自身的实现复杂性以及量化计算中数值表示精度的频繁变换，使得适用于桌面端以及浮点运算的 Winograd 算法不能高效实现。目前在移动端最为通用的卷积算法为 im2col 及其变种。而 Winograd 卷积的量化计算实现，目前在研究与应用中都不够成熟。

此外，卷积网络中的计算瓶颈在于卷积计算，而卷积计算的实现往往同矩阵乘法密切相关。主流的卷积实现方法都会直接或间接的将卷积运算转换为矩阵乘法计算。因此，卷积神经网络中的有效推理问题中的一大重点和难点在于矩阵乘法（在线性代数库中也称为 GEMM）的有效实现。尽管 HPC 领域已经对于 GEMM 有了相当成熟的研究成果，但 HPC 领域对于 GEMM 的研究则是针对服务端具有大量的计算资源和充分的计算能力的 x86 架构的设备，同在边缘计算领域资源和能力均受限，并且以 ARM 架构设备为主导的场景存在着较大的偏差，而在这种场景下的量化计算则更是寥寥无几。另外，高性能计算中的矩阵乘法所关注的问题往往是规模较大的矩阵，而部署在移动端的神经网络中所涉及的矩阵运算往往没有达到这种规模，因此很多针对大矩阵运算的优化策略，在这里是多余的，同时还会带来额外的开销。

结合现有的针对于移动端网络模型以及模型轻量化方法研究，充分利用硬件平台的计算能力，结合快速卷积方法，对于移动端应用的算子库（kernel library）实现优化设计，对于实现高效，低功耗的深度网络模型的部署，边缘智能，或者说智能物联网（AIoT）具备着不言而喻的重要性和必要性。

1.2　本文主要研究内容与贡献

1.2.1　主要工作及贡献

本文针对于实现 ARM 设备上的快速卷积算法进行研究，实现在实际硬件上对于卷积操作以及卷积网络通过量化计算和快速卷积算法的效率加速。主要内容和贡献如下：

- 针对 Winograd 卷积方法的整数卷积优化问题，提出了局部区域多通道 Winograd 卷积算法，针对于嵌入式设备有限的计算资源，充分利用了卷积操作的空间局部性和 Winograd 的算法的通道可并行化，有效解决了整数卷积操作在嵌入式设备的加速问题。
- 为了进一步优化整数卷积的速度，利用 ARM 的 NEON 指令的高效并行计算能力，针对于移动端轻量级卷积网络的参数特征，提出了中小规模整数矩阵乘法缓存友好的高吞吐实现策略，从而更好地利用有限的存储和计算并行能力，避免传统高性能计算中的矩阵乘法的额外开销，进一步加速了整数卷积在 ARM Cortex-A 架构上的速度。
- 将优化的卷积算法集成到 QNNPACK，并与 TensorFlow Lite 的量化卷积速度进行了对比，验证了所提方法的有效性。

1.2.2　论文组织结构

本篇论文的组织结构如下：

第一章介绍了当前边缘设备实现深度网络模型的运行的应用和现状，分析了量化神经网络在移动端应用的可行性，以及快速卷积算法量化计算在移动端的应用的空缺和前景。并简要阐述了本文的主要工作和结构。

第二章系统总结了当前在卷积操作计算加速方法，神经网络量化研究以及网络量化计算在当前实现工作中的支持。对于卷积操作的直接实现，基于 GEMM 的实现和快速卷积方法的实现展开说明，并分析各个方法的优劣；通过总结网络模型量化研究，论证神经网络模型具有在低精度数值表示下仍然在对应任务上保持高准确率的能力，并且存在可行的方法获得模型的量化表示；最后，结合业界对于量化卷积的实现，说明量化卷积计算特别是量化快速卷积在 ARM 设备实现的应用前景，并分析当前实现所存在的缺陷。

第三章主要阐释了 Winograd 卷积在 ARM 设备实现中的一种对于当前主流 ARM 硬件较为友好的方法。同时对于 HPC 领域提出的高效 GEMM 算法针对性的在 ARM 边缘计算场景和现代卷积网络量化计算的特征和限制做了针对性的改进和调优，为 Winograd 卷积在嵌入式量化计算的实现奠定基础。

第四章主要总结并分析了 8 位无符号整数的 Winograd 卷积计算在 ARMv8-A 架构下的实现。总结了实现过程中需要考虑的硬件因素，快速卷积方法在实现中的额外优化，分析了这一卷积实现的有效性，以及集成于 QNNPACK 实现卷积网络加速的效果。

第五章总结了本篇论文的主要研究内容和贡献点，并对当前方法的缺陷进行进一步分析，提出了未来研究工作的相关思路和展望。

第 2 章　相关研究综述

卷积是在深度学习和图像处理中最为常用的一种操作，直观而言，卷积神经网络中的卷积操作可以理解为将卷积核在图像上滑窗，并这个位置计算卷积核与对应的图像输入位置像素的点积并输出到对应的位置。

卷积核大小为 $k_1 \times k_2$ 的卷积操作作用于 $M \times N$ 的矩阵的直接卷积算法具有 $O(MNk_1k_2)$ 的计算复杂度，而对于存储的复杂度则为 $O(MN)$。因此，从理论上讲，像 GEMM（矩阵乘法，也称为 Dense Layer，线性层 (Linear layer) 或仿射层 (Affine Layer)）一样，卷积是受计算限制 (compute bounded) 的操作不是为内存限制 (memory bouned) 的操作。同时，对于多通道的图像输入，内存复杂度变为 $O(MNC)$，计算复杂度变为 $O(MNCk_1k_2)$。

2.1　卷积实现方法概述

2.1.1　直接卷积方法

由于通用矩阵乘法（GEMM）和卷积都是受到计算资源限制的操作（computation bounded operation）, 用于实现高效矩阵乘法的很多方法可以借鉴用于提高卷积操作的效率。Cache blocking 和 Register blocking 是常用的用于实现高效矩阵乘法的两种策略，值得注意的是，这两种策略并不改变矩阵乘法的算法复杂度 $O(n^3)$，而是从硬件角度考虑，充分利用到硬件的计算资源，对矩阵乘法实现硬件友好（hardware friendly）的重新调度。工作[1] 和[2] 中的直接卷积方法实现均借鉴了这一对于矩阵乘法的优化策略，不改变卷积计算本身的复杂度，而是对于直接卷积计算考虑更加硬件友好的实现:

对于卷积操作输入的四个维度，这里记 minibatch 的大小为 N，输入的 feature map 的数目为 C，空间尺度为 HxW, 而与之对应的输出的 feature map 数目，或者说输出的通道数为 K，输出特征的空间尺度为 PxQ，以及卷积操作由四维张量所表示的权重，各个维度的尺度分别为，输入和输出的特征通道数 C, K 卷积核的空间维度 R 和 S。

于是直接卷积算法的处理过程中，如 Algorithm 1所示，具有 6 重循环，其中输入的特征为 I，卷积的权重为 W，而输出的特征为 O

而考虑到现代主流硬件普遍支持向量化处理（vectorization）的并行计算，在对应的架构下，处理器有矢量寄存器（vector register）实现数据的并行化处理。而

Algorithm 1 直接卷积算法

for $k \leftarrow 1 \rightarrow K$ **do**
  for $c \leftarrow 1 \rightarrow C$ **do**
    for $p \leftarrow 1 \rightarrow P$ **do**
      for $q \leftarrow 1 \rightarrow Q$ **do**
        for $r \leftarrow 1 \rightarrow R$ **do**
          for $s \leftarrow 1 \rightarrow S$ **do**
            $O_{n,k,p,q} += I_{n,c,p+r,q+s} \times W_{k,c,r,s}$
          end for
        end for
      end for
    end for
  end for
end for

这一并行处理的过程同处理器架构和数据类型相关，比如对于 x86 架构的 AVX512 矢量计算，如果输入为 32 位浮点数那么计算过程中可以支持 16 个 32 位浮点数的并行计算。而为了充分利用矢量寄存器的并行处理能力，需要将计算中最内层，迭代最为频繁的这一维度的数据，分块打包成大小同矢量寄存器所能容纳数据的最大值相等的子块。比如上述的 AVX 512 矢量计算中，需要将计算最内层的数据分为多个 16 个 32 位浮点数组成的小块。除此之外，在寄存器层面，可以使用 register blocking 技术实现寄存器中的数据复用，并减小 L1 缓存中的数据传输压力，以期减小操作延时。工作[1] 中将 register blocking 应用在输出特征的空间维度层面上，因此在空间维度的迭代过程中，其中的每个点的值均可以独立计算。在综合了寄存器角度的优化策略之后，即矢量化（vectorization）和 register blocking，[1] 对于直接卷积算法提出了如 Algorithm2 所示的优化，这里 $C_{o,b}$ 表示经过 Blocking 方法实现的输出通道量的分块，算法中针对 W 的表示类似。

直接卷积方法无需考虑额外的变换，也没有额外的储存需求，因而在存储角度考虑实际上是相当高效的，而且直接卷积算法也不对卷积本身的设定有任何的预设，可以直接应用于各种类型的卷积操作。而另外一方面，直接卷积算法的缺陷也同样是显而易见的，这一实现实际上对于缓存相当不友好。对于相当常用的 3x3 卷积而言，如果输入的数据类型使用 32 位表示，在计算第一行中的输入的 3 个值的过程中会加载与其位于同一缓存行中的 16 个值，而在下一步的计算中，CPU 需要处理下一行中的 3 个数值，同时又会将这些值位于同一缓存行中的值加载到缓

存中，而直到第三行的数据操作完毕之后，由于缓存的容量一般很小，第一行的数据已经从缓存中抹去了，而这时的计算却正好需要这些数据。这些存储访问相关的问题都会限制计算密度的最大化和计算资源的最有效利用。

Algorithm 2 优化直接卷积算法

for $\tilde{j} \leftarrow 1 \rightarrow C_o/C_{o,b}$ **do**
  for $\tilde{i} \leftarrow 1 \rightarrow C_i/C_{i,b}$ **do**
    for $l \leftarrow 1 \rightarrow H_o$ **do**
      for $\tilde{k} \leftarrow 1 \rightarrow W_o/W_{o,b}$ **do**
        for $n \leftarrow 1 \rightarrow H_f$ **do**
          for $m \leftarrow 1 \rightarrow W_f$ **do**
            for $ii \leftarrow 1 \rightarrow C_{i,b}$ **do**
              for $kk \leftarrow 1 \rightarrow W_{o,b}$ **do**
                for $jj \leftarrow 1 \rightarrow C_{o,b}$ **do**

$$O_{\tilde{j}C_{o,b}+jj,\tilde{k}W_{o,b}+kk,l} \mathrel{+}= I_{\tilde{i}C_{i,b}+ii,s\tilde{k}W_{o,b}+kk+m,ls+n} \times W_{\tilde{i}C_{i,b}+ii,\tilde{j}\times C_{a,b}+jj,m,n}$$

                end for
              end for
            end for
          end for
        end for
      end for
    end for
  end for
end for

2.1.2　基于 GEMM 的卷积实现

2.1.2.1　Im2col （Image to Columns）

Im2col(Image Block to Column) 是当前最流行的 CPU 卷积方案。这一卷积实现方法自 Caffe[3] 起，便在卷积神经网络计算的实现中有着相当广泛的支持，而也正是 caffe 使得基于 GEMM 的卷积实现方法得以流行。这一方法最早见于[4] 它依赖于经过数十年优化的矩阵乘法（GEMM），以充分利用 CPU 缓存。这样的优点是计

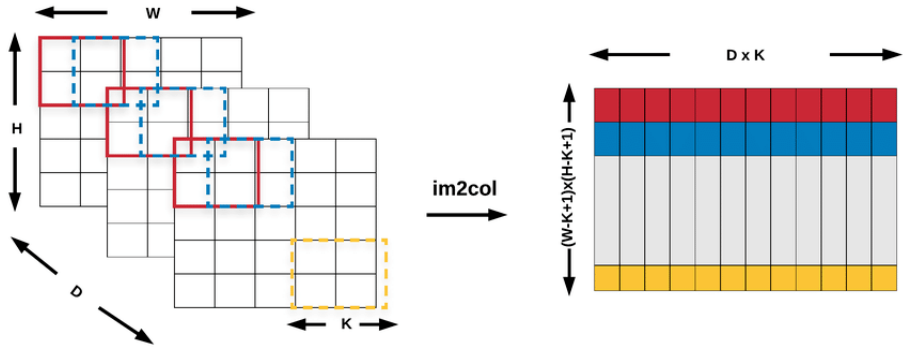

图 2.1　Im2Col 方法

算速度更快，但会占用更多的内存。使用 im2col 运算将输入图像转换为矩阵，然后将该矩阵与变换后的卷积核相乘。最后，再使用 col2im 操作将这个相乘后的矩阵重新转换为图像。

从微观角度来看，卷积操作基本上就是卷积核参数同移动窗口所选择的同卷积核尺度相同的局部区域之间的点积（dot product）。Im2col 的基本想法就在于将所有可能的窗口在内存中展开，然后使用矩阵乘法实现点积，毕竟点积和矩阵乘法的基本流程都是对应位置乘积的和（sum of element-wise product）。从而可以用更大的内存消耗为代价而换取实现的简化和计算加速。

简而言之，im2col 技术，每个窗口截取，将其展平，然后将它们堆叠为矩阵中的列。再将内核展平为行向量并在两者之间进行矩阵乘法，则在输出 reshape 后可以获得完全相同的结果。

在 im2col 方法中内存复制的复杂度为 $O(MNC)$，而卷积的复杂度为 $O(MNCk_1k_2)$，因此，在大多情形下，内存复制所需要的时间开销同卷积本身相比仍然较低的，并且在卷积核尺度较小的场景下 Im2col 方法的内存复杂度相对而言也是可接受的，从而总体上 im2col 可以实现对于卷积操作的加速。

2.1.2.2　Memory Efficient Convolution

MEC 方法[5] 作为对于 im2col 的一种改进和扩展，旨在改善 im2col 方法中对于内存的额外需求。Im2col 实际是一种矩阵的降维手段（lowering scheme），将输入的 3D 张量通过 im2col 可以转换为 2D 矩阵并且可以直接使用 BLAS 实现来完成矩阵乘法。Im2col 的方法变换之后的矩阵中存在着相当多的数据冗余，这也表示这种方法可以通过去除这其中的数据冗余实现对于 im2col 方法中过多的 memory overhead。Im2col 方法会将 $W_i \times H_i \times C_i$ 的三维张量转换为 $(H_f \times W_f \times C_i) \times (H_o \times W_o)$ 的二维矩阵，可见这一过程中，额外需要的内存会随着问题规模存在着平方级别的增长趋势。

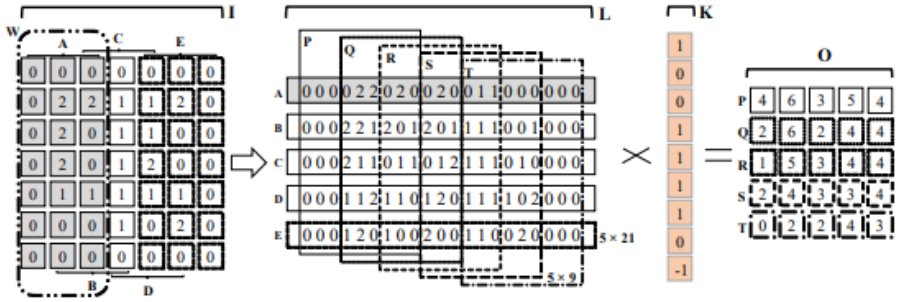

图 2.2　MEC 卷积方法

Im2col 方法中将输入的矩阵中的一个子块做降维，同 Im2col 方法不同的是，MEC 方法在实现的过程中会一次将多列值降维，并且通过额外的矩阵乘法调用减少冗余的内存占用。

如图2.2 中所示，对于一个 7x7 的输入，和 3x3 的卷积核，这里需要实现降维的对象是输入矩阵中的 3 列数据，记输入为 I，则这里需要将 $I[0:7, 0:3]$ 这一区域中的数据平摊为一维，即图中的第二步所示，依次按照滑窗实现输入的按照列的降维，在 stride 为 1 的情形下，这里在第二步得到了降维后的 5 个一维向量，构成一个 5x21 的矩阵，而在实现矩阵乘法的过程中，则又需要对于这个 5x21 的矩阵做划分为 5x9 的子矩阵，在划分的过程中每向右移动三个单位（横向 stride 乘卷积核宽度）取一个子矩阵。每一个子矩阵和平摊（flatten）为一维向量的卷积核之间的乘积构成输出的一行。尽管这一过程中需要更多的矩阵乘法调用，但实际的乘法/加法操作则同 im2col 方法一致。在计算复杂度上没有变化。

MEC 方法可以有效的减少 im2col 方法中在垂直方向上的数据冗余，在上图所示的计算过程中已经达到了相对于 im2col 方法 54% 的存储开销减少。这种方法甚至可以达到相对于 im2col 方法 3.2 倍的存储资源节省。

2.1.3　快速卷积方法实现

前面所提到的卷积算法，其优化的出发点都在于将卷积实现本身以高性能计算的角度解决问题 (直接卷积方法)，或者说将卷积转换为研究相对充分的高性能计算问题（基于 GEMM 的方法），而对于卷积操作本身的复杂度和运算量均没有实质性的降低。而卷积这一源自于信号处理的概念，则实际上在信号处理方法中已经有着相当长远的研究，并且存在着很多可以实现卷积计算复杂性简化的方法。这些方法的主要目的在于减少卷积实现中的乘法操作。因此这一类方法被称为快速卷积方法。

2.1.3.1　FFT 卷积

从信号处理中的角度而言，频域中的乘法对应于时域中的卷积。使用 DFT 将输入信号转换到频域，再乘以滤波器的频率响应，然后使用 Inverse DFT 将输入信号转换回时域。从傅立叶时代开始这种基本技术就广为人知。但是，因为计算 DFT 所需的时间比直接计算卷积所需的时间长，所以使用傅里叶变换实现卷积的方法并没有受到足够的重视。随着 1965 年快速傅立叶变换（FFT）的发展，这种情况发生了变化。通过使用 FFT 算法计算 DFT，与直接对时域信号进行卷积相比，通过频域进行卷积可以更快。而卷积的运算量则获得了有效的缩减。一般而言，2 维场景下，直接卷积的计算复杂度为 $O(n^4)$, 而 FFT 卷积则可以将计算复杂度降低到 $O(n^2 \log_2 n)$。而具体的在 FFT 卷积中的每一个步骤所需的计算复杂度同直接卷积方法的对比可以参考工作[6]。

基于 FFT 的卷积方法[7][6] 可以表示为

$$f * g = F^{-1}(F(f) \cdot F(g)) \tag{2-1}$$

其中的 F 与 F^{-1} 表示傅里叶变换和逆傅里叶变换，在离散场景下，f 和 g 需要具有相同数目的元素，这一点可以通过向两者中较短的那一方实现 zero padding 实现。DFT 卷积的结果是一个循环卷积（circular convolution/ cyclic convolution），而从中获得有效的卷积结果仍然需要从循环卷积的结果中抽取其中的最后 $|f|-|g|+1$ 个元素。

在 FFT 卷积方法中，卷积的输入和权重在经过离散傅里叶变换之后被变换为复数表示，而此后则需要执行二者之间的复数矩阵乘法。而在这一过程中，又具有着以下两种特性能够进一步降低乘法操作的数目。

- 实数值傅里叶变换的 Hermitian 对称性；
- 快速复数乘法计算；

实数域中的信号的 FFT 变换是具有 Hermitian 对称性（Hermitian symmetry）的，即实数矩阵在傅里叶变换之后的矩阵同其共轭转置矩阵（conjugate transpose）是相等的，这使得在 FFT 卷积方法中的矩阵乘法部分的实际有效乘法量可以进一步减半。一个尺寸为 $m \times m$ 的实数值矩阵的离散傅里叶变换由于 Hermitian 对称性可以用 $m \times (floor(\frac{1}{2}+1))$ 个复数来表示。同时又由于 $U^H V^H = (UV)^H$，矩阵乘法结果中的一半元素的值（上/下三角）可以通过取已计算值的共轭获得。

复数乘法计算　复数计算在信号处理中有着广泛的应用，然而不幸的是，很多现代硬件在底层的指令上则缺少复数计算的支持，这使得复数计算的实现不得不面临着额外的挑战和付出。而针对于 ARM 平台，尽管在 ARMv8.2 指令集（ISA）之后添加了 FCADD (Floating-point Complex Add), FCMLA (Floating-point Complex Multiply Accumulate) 操作，然而实现这一指令集的芯片微架构（microarchitecture）并不占据主流，主要集中在高端智能手机芯片上，比如苹果应用于 iPhone X 上的 A11 芯片及其之后的应用于 iPhone XS 系列的 A12 芯片以及 iPhone 11 的 A13 芯片，在很多的嵌入式设备场景下的 low profile 的计算资源场景下，指令级别的复数操作支持仍然是缺失的。

一般的复数乘法中，一对复数相乘需要四次复数乘法，即

$$(u_r + iu_i)(v_r + iv_i) = u_r v_r - u_i v_i + i(u_i v_r + u_r v_i) \tag{2-2}$$

而对于2-2 稍加变换即可以得到复数乘法的快速算法，该方法只需要三个实数乘法。

$$(u_r + iu_i)(v_r + iv_i) = [u_r v_r - u_i v_i, i(u_r v_i + u_i v_r)] \tag{2-3}$$

$$= [u_a - u_c, i(u_a + u_b)] \tag{2-4}$$

where

$$u_a = v_r(u_r + u_i) \tag{2-5}$$

$$u_b = u_r(v_i - v_r) \tag{2-6}$$

$$u_c = u_i(v_r + v_i) \tag{2-7}$$

这一方法，通过使用更多的加法实现在复数乘法计算中乘法的减少。

快速复数乘法在傅里叶卷积中的应用　在执行图像块的变换后，产生一个复数张量 $U = U_r + iU_i$ ，可以计算实值张量 $U_r + U_i$ 并将其与 Ur 和 Ui 一起存储。类似地，卷积核权重在离散傅里叶变换之后得到一个复数张量 $V = V_r + iV_i$, 张量 $V_i - V_r$ 和 $V_r + V_i$ 可以在内核变换期间计算，并与 V_r 一起存储（不必存储 V_i ）。这里每个变换前的实数域的输入对应于变换后的三个实数值，而不是直观理解上的每个变换后的值对应两个用来表示复数实部和虚部的两个实数。然后，可以用三个独立的实值张量的元素间乘积替换复数张量的元素间乘积. 而由此计算得到的三个实数

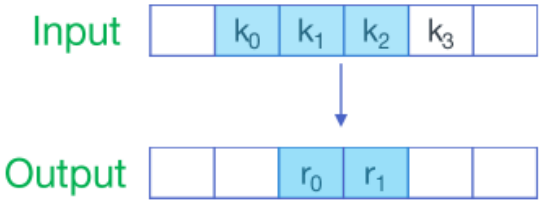

图 2.3　1D 卷积

张量则可以在计算逆变换的过程中被隐式的转换为单个实数张量。总体来讲，快速复数乘法的实现，可以使得原本的复数域中的一个矩阵间乘法转换为 3 个实数域中的矩阵乘法，同直接计算复数乘法相比，可以将乘法的操作数减少 25%。

FFT 卷积在执行计算的过程中需要将权重填充到和输入同等大小，这会导致额外并且不必要的内存负担。而在卷积核本身毕竟小的状态下，这种额外开销的负面影响则更加明显。信号处理中对于这一问题也有着比较成熟的解决方案，即 overlap add 和 overlap save 方法。切换到深度学习的场景下，可以实现对于输入的图像的切分，在每个小块上执行 FFT 卷积，再将得到的结果通过类似 overlap add / overlap save 的方法拼凑为整个图像输入的结果。但是，这样的方法还需要将内核权重填充到合适的大小（通常为 2 的幂）以达到最优性能。即使将内核权重填充到架构寄存器大小的较小倍数（例如 8 或 16），也会导致内存需求增加 7 到 28 倍。此外可以通过动态执行 FFT, 将 FFT 作为卷积层计算的一部分来最小化这种额外填充卷积核权重并将其转换到频域的 overhead。但是，这会产生大量的性能开销，使得资源受限的嵌入式设备雪上加霜。

2.1.3.2　Winograd 卷积

Winograd minimal filtering 算法是在信号处理中常用的一类快速算法，并且在信号处理领域已经有了相当时间的应用和研究。而近年来[8] 才将这一方法引入卷积网络的计算中。下面将从卷积网络计算的角度，考虑这一算法在卷积计算中的具体应用场景，简要解释介绍 Winograd 算法对于卷积计算过程的优化。

从 1D 的卷积过程来考虑，对于卷积核大小为 3 的情形，对于连续两次的卷积过程，如图2.3 所示，输出 r_0 由输入 k_0, k_1, k_2 同卷积核参数做元素间乘法相加得到，同样的，输出中的 r_1 则由输入 k_1, k_2, k_3 计算得出，这一过程中输出两个值需要有 4 个输入，将这一过程用矩阵乘法来做表示

$$\begin{pmatrix} k_0 & k_1 & k_2 \\ k_1 & k_2 & k_3 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \tag{2-8}$$

直接做矩阵乘法，这一过程需要 6 次乘法和 4 次加法：

$$r_0 = k_0 w_0 + k_1 w_1 + k_2 w_2 \tag{2-9}$$

$$r_1 = k_1 w_0 + k_2 w_1 + k_3 w_2 \tag{2-10}$$

而如果对上述的乘法计算过程做如下的变换:

$$\begin{pmatrix} k_0 & k_1 & k_2 \\ k_1 & k_2 & k_3 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} m_0 + m_1 + m_2 \\ m_1 - m_2 - m_3 \end{pmatrix} \tag{2-11}$$

其中，

$$m_0 = (k_0 - k_2) w_0 \tag{2-12}$$

$$m_1 = (k_1 + k_2) \frac{w_0 + w_1 + w_2}{2} \tag{2-13}$$

$$m_0 = (k_0 - k_2) w_0 \tag{2-14}$$

$$m_2 = (k_2 - k_1) \frac{w_0 - w_1 + w_2}{2} \tag{2-15}$$

由于在卷积的过程中，卷积核权重的值都是确定的，所以其中 m_1, m_2 中同 w 相关的这一部分均可以在计算矩阵乘法（卷积操作）之前预先计算。从而这一过程可以将乘法的数目减少到 4 次，而上述的这一过程即为 $F(2,3)$ 卷积在 1D 场景下的实现，而二维的场景，只需要将 1D 的场景做进一步的嵌入即可实现。

具体而言，对于如图所示的 $F(2x2, 3x3)$ 的场景

用矩阵乘法的形式表示这一卷积过程

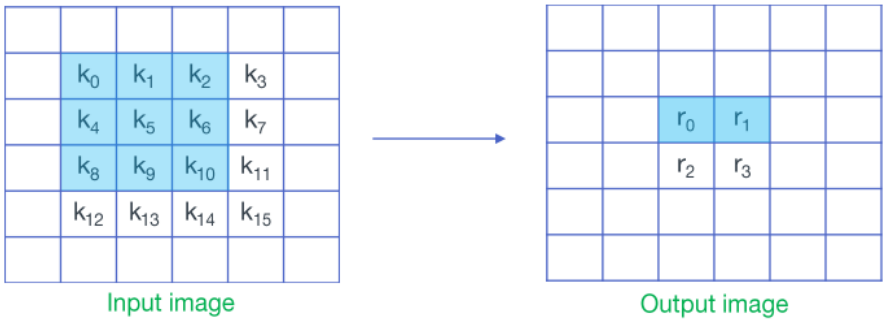

图 2.4　F(2x2, 3x3) 卷积

$$\begin{pmatrix} k_0 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 & k_9 & k_10 \\ k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 & k_9 & k_10 & k_11 \\ k_4 & k_5 & k_6 & k_7 & k_8 & k_9 & k_10 & k_11 & k_12 & k_13 & k_14 \\ k_5 & k_6 & k_7 & k_8 & k_9 & k_10 & k_11 & k_12 & k_13 & k_14 & k_15 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

类比于 1D 场景下的处理方法，这里可以得到相似的处理形式，不同点在于，1D 场景下简化的乘法2-11中表示的值均为 Scalar，而 2D 场景下对应的值为一个子矩阵，可见在 2D 场景下，乘法操作的数目可以由直接矩阵乘法计算的 36 次，减少到 16 次，从而以乘法计算的复杂度来考虑，可以达到 2.25 倍的复杂度降低。

2.1.3.3　FFT 卷积实现同 Winograd 卷积的对比

离散傅里叶变换方法的有效卷积可以视作 Winograd 卷积 3-1 的一种特殊情形，其中的矩阵 A^T, B^T 和 G 均处于复数域，并且由多项式点（polynomial point）为单位根的范德蒙矩阵（Vandermonde matrices）推导出。

Winograd 与 FFT 卷积中主要能够降低计算成本的一点在于，权重的变换可以预先计算，而卷积输入的变换则可以多次复用，而计算本身中比重最大的部分仍然是矩阵乘法，而矩阵乘法的规模正好通过变换过程得以降低。

2.2　神经网络量化研究

减小计算资源的需求并提高功耗利用是在边缘设备上实现智能应用的一大挑战。量化方法便是解决这一挑战的一条可行的途径。量化过程做一个简单的比喻，可以理解为是图片的编码，现实世界中的图像视觉信号是连续的模拟信号，量化过程便是使用有限的编码位数，用离散的整数值逼近重建实际的连续值。

在机器学习应用中，神经网络由激活节点，节点之间的连接以及与每个连接关联的权重参数组成。这些权重参数和激活节点计算可以量化。在硬件上运行神经网络往往存在着数百万的乘法和加法运算。具有量化参数的低位数学运算与对神经网络的中间计算进行量化相结合，可带来较大的计算增益和更高的性能。

2.2.1　网络模型量化方法

量化网络模型在很多应用场景下的存在精度下降的问题，为实现在嵌入式场景，移动场景等边缘设备的高效模型部署和智能应用处理，近些年也有很多研究着力于实现使用低位数的模型 (low-bit model) 低精度的数据表示 (low-precision representation) 在对应的机器学习任务上获得较高的精度指标。相当多的工作在着力实现量化对整个网络准确性的影响最小。量化背后的核心思想是神经网络对噪声的弹性或者说容忍性。尤其是深度神经网络，经过训练可以识别关键模式并忽略噪声。不逊于全精度（full precision）网络的准确度加上显着减少的内存占用，功耗和计算速度的提高，使量化成为将神经网络部署到嵌入式硬件的有效方法。

而在网络训练结束之后单独对于网络权重做量化（post-training quantization）处理，这一网络量化处理最为直观和朴素的方法对于较大的网络模型一般实现的实际精度效果仍然是可接受的，而对于参数比较小的模型，这种方法则会造成相当的精度损失。参数较大的模型具有更丰富的表达能力，同时抗噪性也更强，而小模型本身的模型表达能力便受到容量的限制，参数量化过程则会对模型的表现能力造成巨大的波动。网络模型中参数的分布可能会存在比较极端的情形，这可能存在于同一个卷积操作的不同输出 channel 中，也可能存在于网络中不同的层次中。这使得我们在实现有限位数的量化的过程中，权重差距比较小的部分会存在较大的量化误差。另外一方面，这种直接的量化策略下，离群点对于整体量化的效果会存在比较明显的影响。

工作[9] 中在模型的训练部分，提出了可以在模型训练的过程中模拟量化效果，从而填补模型训练和模型量化之间所存在的 gap。这一文中描述了目前通用的执行网络中端到端整数计算的策略，并且提出在网络的训练过程中可以在计算节点间插入 Fake Quantization 操作，使得网络在前向推理的过程中，效果上执行的是端到

端的整数计算过程，而网络本身的权重参数仍然为浮点数，从而网络反向梯度更新仍然可以像普通的训练过程一样执行。

工作[10] 为了训练可以有效离散化而又不损失性能的网络，引入了可微分的量化过程。通过将权重上的连续分布和网络的激活转换为量化网格上的分类分布，可以实现可区分性。随后将它们放宽 (relaxed) 为连续值替代，从而可以进行基于梯度的有效优化。

2.2.2　模型量化计算实现支持

诸如 TensorRT，TensorFlow，PyTorch，MXNet 等框架均在不同程度上具有了处理量化网络模型的能力。一些模型的计算过程中会包含量化和反量化操作，即量化计算的过程并不是端到端的，即网络中某个操作（通常是如全连接或者卷积这种计算密集型操作）的输入仍然是 32 位浮点数，在执行该操作之前，可以通过量化操作将对应的输入转换为 8 位整数表示执行计算，最后再将计算的结果反量化回 32 位浮点表示，这种量化计算方法在多个连续的网络计算操作的进行过程中，可以通过去除计算流程（computation flow）中的冗余的量化/反量化操作达到将浮点运算转换为定点计算（即整数型计算），从而使得输入的 32 位浮点表示的特征在经过一次变换之后，通过多个支持整数计算的操作，最终在输出前再反量化为浮点表示。而这一流程的优势也仅仅存在于，计算流程中的每一步都需要支持整数型计算，否则将会引入额外的量化/反量化过程，使得计算整体的效率得不偿失。

这些深度学习网络框架对于机器学习从业者而言，很多都只是定义了计算的接口，而真正的计算则是由支持 PyTorch, TensorFlow 等框架的底层的 kernel library 来实现的，这些计算库才是深度模型计算负载的真正执行者。而计算库的实现也是同执行运算的硬件平台密切相关的。就 CPU 上的量化计算而言，在 Intel x86 芯片架构的桌面和服务器端，Intel MKL(Math Kernel Library) 提供了各种针对硬件高度优化的数学计算支持，同时也包括整数型的矩阵乘法和卷积实现。而且同时针对于近些年来普遍应用于深度学习视觉任务中的小尺度卷积而言，在 Intel 硬件平台则由 LIBXSMM[11] 实现针对性的计算优化。顾名思义，这里的 XSMM 即指在 x86 平台的小尺度矩阵乘法计算（Small Matrix Multiplication）而 LIBXSMM 的小尺度卷积实现中，并没有使用快速卷积算法，而是在小尺度的情形下使用高度优化的直接卷积算法。同时 LIBXSMM 支持十六位整数和 8 位整数的计算。

而 Facebook 则专门针对于深度网络的量化计算推出 QNNPACK，实现在 ARM 移动平台和 x86 桌面端的量化网络计算需求，而其中的卷积计算方法则是采用了 Indirect Convolution[12](一种 im2col 方法的变种，从计算实现角度改善 memory

overhead)。另外，在服务器端的 CPU 低精度模型推理优化计算。FBGEMM 的实现中采用了和 QNNPACK 中类似的 im2col 改进策略。

GEMM 方法实现的卷积很大程度上依赖于高性能计算研究中对于矩阵乘法的优化，然而 HPC 毕竟同深度学习模型推理之间是存在着不匹配的。很多 HPC 库并不提供针对于量化场景的相关计算，HPC 针对的场景往往是大规模的矩阵计算，甚至是规模大到需要在分布式集群执行的计算，而没有针对于深度学习中的很多矩阵的计算，特别是近些年来的小尺度的矩阵的计算做优化，因而不能充分利用到深度网络权重矩阵的特性。

第 3 章　整数快速卷积实现

3.1　Winograd 卷积实现简介

Winograd 卷积的一般过程包括输入变换（input transform），权重变换（weight transform），矩阵乘法（GEMM）以及输出变换（output transform）。其中权重的变换只用操作一次，便可以在不同的输入复用。一般考虑到运行时效率，权重变换可以在卷积操作实际执行前操作，从而 Winograd 卷积的实际执行过程可以概况为以下步骤：

1. 将输入的图像块转换到 Winograd 域
2. 执行在 Winograd 域中执行变换后的输入和权重的矩阵乘法
3. 将矩阵乘法的结果转换回空间域

若使用 $F(m,r)$ ① 有 m 个输出的 r-tap FIR 滤波器，F(m, r) 接收的输入的大小为 (m + r - 1)，则 Winograd 卷积算法可以表示为

$$Y = A^T[(Gg) \circ (B^T d)] \tag{3-1}$$

在式 3-1 中，$\circ$ 表示 Hadamard Product.

Winograd 卷积算法中的矩阵乘法量同输入的尺寸一致，即 $F(m,r)$ 的 Winograd 卷积，其中需要执行 m + r -1 个乘法。而更高维度的 Winograd 算法 $F(m\times n, r\times s)$，则可以通过在对应的维度嵌套一维的卷积算法3-1 $F(m,r)$ 和 $F(n,s)$ 实现。特别是对于卷积网络中，最为广泛使用的方形滤波器，即卷积核的宽高相等，$F(m\times m, r\times r)$ 表示卷积核尺寸为 $r\times r$，对应的输出尺寸为 $m\times m$, 二维的 Winograd 算法可以表示为：

$$Y = A^T[(GgG^T) \circ (B^T d B)]A \tag{3-2}$$

而与之对应的直接卷积方法中对应的矩阵乘法则为 m^2r^2 （输出为 m^2 个，每个输出对应着 r^2 个输入，每个输入都要同 filter 中的对应值做一次乘法），而这里二维场景的 Winograd 卷积实现，其中的乘法操作的规模为 $(m+r-1)^2$。而在卷积

① 下面对于卷积实现中的一些符号做如下约定。卷积操作作用于输入特征的局部，并且局部特征的卷积结果输出也仅由这一局部的输入所决定。对于输入局部（input tile）尺寸位 mxm，卷积核（kernel）尺寸为 $k\times k$，输出块（output tile）特征大小为 $r\times r$ 的卷积操作，可以记为 $F(r\times r, k\times k, m\times m)$.

算法的实现中，矩阵乘法（GEMM）无疑是最为占据主导的操作，自然也是效率优化的瓶颈。于是，以乘法的规模作为复杂度的度量，Winograd 卷积相对于直接卷积方法，实现的复杂度简化为式3-3

$$\frac{m^2 r^2}{(m+r-1)^2} \tag{3-3}$$

因此，对于较为常用的两类 Winograd 卷积实现 F(2, 3), F(4, 3)，即卷积核大小均为 3，输出的尺寸分别为 2 和 4 的卷积，理论上分别可以达到 2.25 和 4 倍的加速。而在实际的实现中，还需要考虑到对于输入和输出的变换所带来的额外开销，在卷积的规模，特别是矩阵乘法的规模没有达到一定的限度时，Winograd 卷积方法不一定能够实现理论的加速值。尽管使用更大尺度（输出的尺度和卷积尺度）的 Winograd 卷积可以实现更大程度的复杂度简化，但同时也受限于变换过程的数值稳定性，更大尺度的 Winograd 卷积一般也意味着不准确，可能会存在精度的损失。

另外，关于 Winograd 卷积中使用的变换矩阵 $\boldsymbol{G}$, $\boldsymbol{B}^T$, $\boldsymbol{A}^T$ 的具体表示的推导可以通过 Cook Toom 算法来实现。具体过程可以概述为，使用拉格朗日插值多项式，将相当于卷积的多项式乘法，转换为多项式在固定数目的插值点处取值的逐元素乘法。Cook-Toom 算法的缺点是，随着变换大小的增加，变换很快变得不稳定。但是，它们非常适合卷积神经网络中使用的 3x3 小卷积。

Winograd 快速卷积算法的更一般形式是使用中国剩余定理。Cook Toom 算法只是其中之一。而这一过程的具体不在本文的范畴，有兴趣的可以参考工作[8] 中的补充材料。

快速卷积算法相对于直接卷积和基于 GEMM 的卷积（比如 im2col，im2row 等）的实现的主要一大优化目标在于减小运算中乘法的规模，尽管快速卷积方法往往为实现减少乘法的数目而不惜引入一些额外的加法操作以及对于输入的变换，但在运算中乘法操作达到一定的规模后，乘法数目减少所带来的计算收益是可以超过这些额外的开销的。值得注意的是，快速卷积方法中为减少乘法操作而不惜引入加法的这一优化手段，并不是因为加法操作会比乘法操作快，这个概念具体到现代的硬件设备上的乘法和加法实现实际上是不合理的。而且实际上在现代硬件设备中乘法和加法操作往往是可以聚合（fused）在一起的，也就是乘累加（multiply and acculate）操作。从而即使额外引入加法，在仔细设计算法执行后，是不会带来额外的计算开销的。另外，通过充分利用 FMA 实现计算加速，在算法设计中添加加法来减少乘法，需要考虑到计算中的乘法和加法的平衡，加法操作需要同乘法操作匹配，否则，额外的加法会带来计算负担，甚至会超过乘法减少和 FMA 换取

的计算量的减少值，反而使得计算变得低效。

而另外一方面，作为另外一种更为常用的快速卷积方法实现，快速傅里叶变换（FFT）方法在整体的形式上同 Winograd 方法3-1类似，其中的变换过程则会对应为 FFT 和 inverse 类似，其中的变换过程则会对应为 FFT 和 inverse FFT，即变换矩阵 G 和 B^T 表示 FFT，而 A^T 则表示 inverse FFT。同时，式3-1 中的也将对应的修改为复数乘法，同实数乘法不同的是，直接的复数乘法需要 4 次实数乘法来实现。然而，卷积网络中的输入往往都是实数，而实数的傅里叶变换存在 Hermitian 对称性，从而可以将矩阵复数乘法的复杂度减半，只需计算矩阵中的一半的值，另外一半只需要取对应的已计算值的共轭复数即可。但即便如此，FFT 卷积实现中仍然需要达到 64x64 的输入尺度才能在乘法规模的优化程度上达到和 Winograd 卷积 F(4, 3) 在输入为 6x6 时的同等水准，使用 FFT 实现卷积操作的加速需要相比于 Winograd 卷积更大的内存需求，

3.2　Winograd 算法数值稳定性

尽管 Winograd 算法是现已知的在计算复杂度而言最优的卷积算法，然而 Winograd 卷积算法具有着数值计算不稳定的特质。在卷积的尺度相对比较小（比如 $F(2,3), F(4,3)$），并且参与运算的数值为单精度或者双精度浮点数时，Winograd 卷积的结果仍然是相当精确的，但是当卷积的尺度变大，或者参数计算的数值精度降低时，这一情况却不容乐观了。这也限制了 Winograd 卷积的适用范围，在卷积核或者卷积操作输入的子块的尺度较大的情形下，Winograd 卷积的精确性便会有比较明显的损失。Winograd 卷积计算的精确性可以通过找出更加合适的 Winograd 算法的导出式有关，即通过 Toom-Cook 算法或者 CRP 理论选择合适的插值点导出对应的输入变化，权重变换和输出变换矩阵。这里需要说明的一点是，Winograd 卷积的数值不稳定性，带来的只是一种边际数值误差（marginal error），因此，Winograd 卷积在很多情形下只是应用于模型的推理阶段，即网络在训练的过程中使用常规的卷积算法，而在模型前向推理的过程中，是完全可以将对应的满足条件的卷积操作可以替换为 Winograd 卷积算法。

表 3.1 中对比了不同尺度的 Winograd 卷积算法在不同精度的数据表示下的准确度及数值稳定性，这里以直接卷积算法的准确率作为对照，可见对于输出尺度大于 2 的 Winograd 卷积在 low precision 的场景下在对应视觉任务上的准确率会大幅下降，而这里 Winograd 卷积指的是在工作[8] 以及附带的 Toom Cook 方法代码中所导出 Winograd 算法形式。究其原因在于，在 Winograd 卷积的输出尺度较大时, 如3-4 可见，在变换矩阵 A^T, G, B^T 中对应的数值的范围会变大，并且注意，在

表 3.1　ResNet 18 在不同尺度和不同精度卷积算法下 CIFAR-10 分类准确率

	32 bit	16 bit	8 bit
Direct Conv	93.16	93.60	93.22
Winograd F(2, 3)	93.16	93.48	93.21
Winograd F(4, 3)	93.14	19.25	17.36
Winograd F(6, 3)	93.11	11.41	10.95

卷积神经网络中常用的 2D 卷积中，将其转换为整数型计算时，权重变换矩阵 G 需要扩大 24 倍，则卷积计算过程中，至少要保证在 $\pm 24^2$ 范围内数据的表示和计算数值精度。

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{pmatrix} G = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{pmatrix} \tag{3-4}$$

$$B^T = \begin{pmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{pmatrix} \tag{3-5}$$

由于在数值精度较低的情形下 Winograd 算法表现出明显的数值不稳定性，导致计算精度下降，进而使得在对应视觉任务上的准确性下降，因此在后续的 Winograd 量化卷积实现中，本文主要针对于 F(2，3) 卷积的实现。

3.3　局部区域化多通道的 Winograd 的卷积算法

按照上述的 Winograd 算法的流程的描述，针对于输入的特征的每个通道做对应的变换，再将变换后的值分别同对应位置的值做乘法似乎是一种直观的实现方法，这种实现不仅在读取输入的过程中缓存友好，而且只需要有限的矢量寄存器就可以完成变换操作，但是在量化计算的场景下，存在这以下新的特性：

- 数值表示的精度降低使得计算过程的吞吐得以提升

• 使用定点数实现计算存在频繁的数值表示的变换

首先，相对于单精度浮点数表示而言，量化的数值表示可以在同样长的寄存器中存储更多的数值，这不仅使得计算的吞吐变高，同时也需要重新考虑算法的设计与硬件特性的支持。ARM NEON 提供了最大 128 位的 SIMD 矢量寄存器，尽管相对服务端与桌面端芯片的矢量寄存器所支持的位长相当有限，但应用于 8 位量化整数的并行计算仍然是有着充分的计算能力。而在 Winograd 卷积算法中，各个 channel 之间的数据的计算也实际上是相互独立的，结合硬件计算支持而言，针对于输入的局部区域在通道维度实现并行化计算实则是更为合理的，这样也避免了在量化计算过程中，数值位长的变换带来的算法设计上的频繁改动。而单通道算法在实现 Winograd 卷积的量化计算中，还需要针对于变换的过程中实现矢量寄存器中数据的转置操作，以及执行矩阵乘法前对于数据的重新排布，这些向量和矩阵的转置操作都是需要相当大的额外开销的，在资源本就有限的 ARM 设备上这样的操作应该在算法的设计过程中予以避免。而针对于局部的输入区域的多通道特征，恰好可以避免这些转置操作，完成变换的输入和权重可以直接对应到参与矩阵乘法的 16 个矩阵中。而在寄存器数量允许的条件下，多通道方法尽可能多的使用了可用的寄存器资源，最大化了计算指令与访存指令的比例，更有效的利用了 ARM 处理器流水线，可以有效实现计算的高吞吐。

以下针对多通道维度并行化和空间局部性，从 Winograd 卷积计算中的数据排布，Winograd 变换，卷积计算的空间局部化阐述本文在移动设备的量化计算中对于 Winograd 卷积的实现设计。

3.3.1　适用于量化计算的灵活数据排布

卷积网络中常用的数据排布方式（data layout）存在 Channel First 即 NCHW 和 Channel Last 即 NHWC 。而对应的在数据加载的过程中，由于在 ARMv8-A 架构下，存在 32 个 128 位 SIMD 寄存器，因此每个寄存器可以用来存储 4 个 32 位整数（int32）或者 16 个 8 位整数（uint8）或者 8 个 16 位整数（int16）。以 32 位整数举例，在 NCHW 数据排布下，在完成 128 位的数据加载后，一个 SIMD 寄存器可以用来储存矩阵中一行的四个值，而在 NHWC 数据排布下，同样的单个 SIMD 寄存器可以用来加载矩阵中位于同一个位置的连续四个通道（channel ）的值。而在 Winograd 快速卷积的实现中，NHWC 的数据表示是具备着实现灵活性上的相当的优势的，特别是在量化计算的场景下，定点（fixed point）表示的数据在计算过程中会频繁涉及数据位长（data width）或者说数据精度（precision）的变换，NHWC 的表示可以灵活的实现并行算法。同时在考虑不同规模的算法下，NHWC 的 data

layout 也可以实现寄存器的高效利用。下面在具体的 Winograd 卷积的实现中会做对比和具体的说明。

3.3.2　多 channel Winograd 变换及逆变换

对于卷积操作 $F(2x2, 3x3, 4x4)$，Winograd 卷积输入变换最为常用的特征方程是

$$X^T x X = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} x \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \tag{3-6}$$

在这里输入变换的输入特征 x 无论是 NHWC 表示还是 NCHW 表示，抛开 Channel 维度而言，矩阵在平面表示上都是 row major 的，即同一行中的元素位于矩阵表示中的更内层。而矩阵 x 左乘一个矩阵可以表示为 x 中的行元素的线性组合（这一点会在矩阵乘法一节详细展开说明）。上述矩阵的变换过程可以分为两步:

- 第一步计算 $X^T x$。这里可以借助矩阵 X^T 的简单形式和矩阵左乘的特性快速实现。
- 第二步计算 $X^T x X = (X^T (X^T x)^T)$，可见这一计算过程只需将第一步的计算结果转置，再重复第一步的计算过程，最终再将结果转置。

在这里简单讨论 NHWC 表示和 NCHW 表示在这里对于这一计算过程的影响。首先使用 NCHW 表示，则一个 SIMD 寄存器中将包含处于矩阵中同一行的元素。我们将应用矩阵左乘的线性组合性，实现输入特征的变换。同时由于这种数据表示，上述算法中的转置操作也不可避免，而转置操作在硬件实现上则是代价相对较高的一种操作；更重要的一点是，在量化计算中，NCHW 的表示会为算法设计带来更多的困难，比如这里的输入变换中，如果输入的数据类型是 32 位整数（int32），则在 ARMv8-A 中的 SIMD 计算中，可以用 4 个 128 位的寄存器来存储一个 4x4 矩阵的值，计算过程和上述的数学表示相差无几。而如果输入的特征是 8 位整数（uint8），则上面的算法就需要重新设计，或者说，NCHW 的数据表示下，不得不针对每种类型的数据重新设计算法。最后，如果考虑到后续的计算，NCHW 表示的计算结果中位于同一寄存器中的值必须分散（scatter）到不同的位置来实现矩阵乘法，而在 NHWC 的布局下，每个寄存器都是同一位置不同 channel 的值，对于输入变换过程，可以使用 16 个 SIMD 寄存器来表示算法中的 4x4 矩阵，这一表示允许更加灵活的使用 SIMD 实现对不同 channel 的数据的同步计算，而不必针对于

输入数据的位长而重新设计算法，数据位长或者精度只会改变 SIMD 计算中可以同时操作的 channel 数目，数据位长越低，则能同步处理更多 channel 的数据，同一个 128 位的寄存器，可以同时处理 4 个 32 位的整数或者 8 个 16 位的整数。

同时，也是由于 Channel last 的数据排布方式，使得在实现上述的变换算法过程中，只需要对应的改变多维数组中的索引值，而不需要执行编写难度和运行开销较大的向量转置指令。

此外，NCHW 的 data layout 的影响还会受限于变换算法过程中的矩阵大小，$F(3,2)$ 的情形下矩阵是 4x4 的，而在 $F(3,4)$ 的场景下，输出的矩阵则是 6x6 的。如果使用 NCHW 的表示，这样将会使得寄存器中存储一个 6x6 的矩阵变得十分棘手。在 32 位整数的情形下，需要一个半寄存器来表示 6x6 矩阵中的一行，这样便只能在算法的过度设计和寄存器的浪费中二选一了。

在量化计算的场景下，这里输入的特征为被量化后的 8 位无符号整数，执行 16 次数据加载之后可以获得 4x4 的输入矩阵，ARMv8-A NEON 指令中支持一次加载一个 64 位寄存器或者一个 128 位寄存器，实现中采取一次加载 8 个 8 位无符号整数（一个 64 位寄存器），即这里实际加载到的输入是 4x4x8 的。同时在这里考虑到定点计算过程中的溢出的影响需要将 8 位整数转换为 16 位整数，而转换为 16 位整数之后，同一位置的多个 channel 的数据刚好占满一个 128 位的寄存器。最终变换输出的结果也是由 16 位带符号整数表示。

而对于权重变换而言，在通用的 Winograd 卷积变换中，同上述输入变换所对应的权重变换矩阵 G 为

$$G = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{pmatrix} \tag{3-7}$$

为实现这一计算过程的整数化，只需要对这一矩阵扩大两倍即可，即用于整数计算的权重变换矩阵为

$$\begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 2 \end{pmatrix} \tag{3-8}$$

同时，对应的，由于在权重变换的过程中 G 和 G^T 均乘 2，所以在最终的输出

变换过程中需要对输出值除 4。

输出变换矩阵 A^T 则仍然使用

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix} \tag{3-9}$$

3.3.3　卷积输入局部区域化方法

在信号处理中，在很多场景下，需要处理的信号都非常长，计算过程中往往没有足够的内存可以容纳需要处理的整段输入。而另外一方面，由于卷积本身是线性的，因此可以通过简单的将输入的每个输出块相加来计算长序列信号的输出。因此可以通过 overlap-add 方法实现长序列的分割，从而将长序列分解成为更易于处理的短序列。而对于卷积网络中所使用的对于 filter 尺寸为 r 且输出为 m 的卷积，通过 OLA（overlap-add）方法，卷积的输入被分割为 $m+r-1$ 的小块，而块之间存在着 $r-1$ 的重叠（overlap）。而对于所有输入图像中相同位置的图块，计算输出图像大小为 m 的图块。

3.3.4　区域多通道 Winograd 卷积实现流程概述

输入的图像或者 3D 的特征首先会在空间维度分区，按照上面所述的图片分块方法，分割为 $(m+r-1)$ 尺度的小块，记这里输入的通道数为 C，则每个小块的尺度为 $(m+r-1)x(m+r-1)xC$。记这里被分割出的小块的数目为 R，这 R 个小块可以并行的实现到 Winograd 域的输入变换，在这一变换过程中 channel 维度的变换也是各 channel 之间相互独立的，因此这里的输入变换也是通过 SIMD 同步在对应的 channel 实现的。而在变换结束之后，每个输入的小块被转换为 Winograd 域中的一个 $4x4xC$ 的矩阵，而这样的小块则一共存在 R 个。而对应于卷积中后续的步骤，这里单个 channel 中的 4 x 4 的值需要执行同来自权重变换的一个 channel 中的 4x4 的权重值的乘法，而变换过程中的指令并行化是在 channel 维度实现的，也就是说，在执行一次变换之后，有 4x4 的矢量寄存器，并且每个矢量寄存器中包含有对应位置的连续 channel 中的值，这 4x4 个数值需要分发到 16 个 GEMM 操作，而这里变换之后的输出的值也按照在 channel 维度连续的方式做储存。即这里做了输入变换之后的整个特征在内存中的数据排布方式为 16 x R x C.

在权重的变换方面，要保证输出的 channel 的维度在数据的排布方式中位于最内层，即输出通道维度对应的 stride 为 1。这里同一般的卷积网络中的数据表示有着相当的不一致，而好在权重的值在做 inference 的过程中是固定不变的，所以可以预先对权重的数值做矩阵转置，以保证输出通道在 data layout 的最内层 (实际上

权重变换的整个过程都可以在网络运行 workload 之前完成, 模型运行中只要复用这个变换值即可)。假如一个 3x3 卷积的输入 channel 数为 C，而输出的 channel 数为 M，每个 3x3 的权重变换之后为 4x4 的矩阵，在 channel 维度使用矢量计算指令同步计算出权重变换的结果，并将其按照 $16xCxM$ 的 layout 写入存储。而在上述的两种变换过程中，由于寄存器数目有限，并不能一次性处理全部的通道或者特征分块，每次在一部分子块的变换之后将变换结果按照上述的 layout 约束写出的时候，前面所提到的矩阵 stride 的概念便可以使得这一过程实现的非常直观。举个例子来说，输入变换中处理的第 r 个子块 4x4xC 个值可以按照算法3写出:

Algorithm 3 输入变换的 Channel 维度写出

```
for i ← 0 → 4 do
    for j ← 0 → 4 do
        store(output + n* stride, V[i][j] )
        n+=1
    end for
end for
```

上述的两者的变换为有效实现矩阵乘法提供了必要的条件。变换后的平面尺度为 4x4，所以这里需要 16 个 GEMM 计算来实现 Winograd 域的乘法操作，而每个 GEMM 实现中对应的矩阵乘法为一个 R x C 的矩阵与 C x M 的矩阵的乘积，而前面在变换过程中对于输出的 layout 的限制，也是为了在这里可以更方便的实现这些 GEMM 计算。当然在实际的卷积计算中 R，C，M 都可能是比较大的值，这里便需要精心的设计矩阵乘法，充分利用 cache 和 vector register，实现高效计算。毕竟，对于乘法操作的减少是 Winograd 卷积的最大优势，高效的 GEMM 实现才能将 Winograd 卷积的加速效果真正表现出来，同时，从另外一个角度来看，如果对于 Winograd 卷积中各个阶段（输入输出变换，GEMM）的时间占比做一个拆分，在保证 GEMM 高效实现的前提下，GEMM 所占的比重越大，证明 Winograd 卷积省去的乘法操作越多，实现的加速效果越明显。

GEMM 执行结束之后，将计算结果按照 16xRxM 的 layout 写出。

类比如输入变换，输出变换的过程可以看作是输入的逆过程，这里将参与输出变换的 4x4 多 channel 的中间结果从 GEMM 计算结果的输出中的 16 个位置读入，然后通过矢量计算同步处理每个 4x4 矩阵多个 channel 的结果，同时按照对应子块的索引和卷积输出的尺度信息将输出变换的结果填充到卷积输出的对应位置，从而得到 H x W x M 的卷积输出，完成整个卷积操作。这一完整过程如图3.1所示。

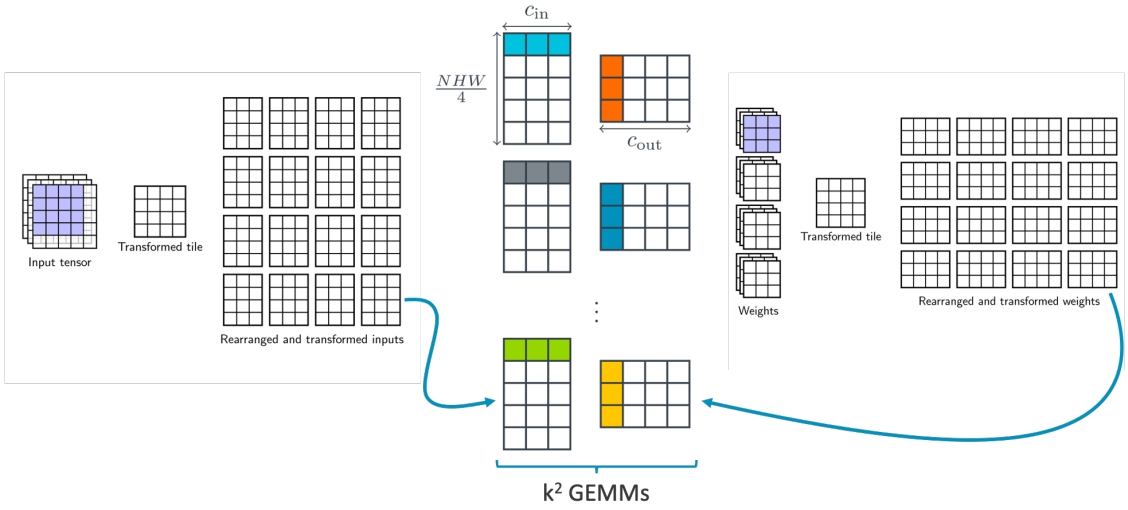

图 3.1　ARM 设备 Winograd 卷积实现

3.4　高效量化矩阵乘法实现

在输入和权重均经过 Winograd 变换之后，变换后的矩阵会执行元素间的乘法，然后将对应位置的乘积结果累加（element-wise addition of Hadamard products）。表示输出通道为 m，输入通道为 c 的权重值可以在 c 通道中的所有输入区域被复用，而 c 通道中位置为 (i, j) 的输入则在所有的 (i, j) 位置输出的区域被复用。这一观察表明，实际上这一操作本身可以转化为研究较为充分的通用矩阵乘法（GEMM）。对于输入的 channel 数为 C，输出的 channel 数为 M，且具有 R 个分块的卷积，在 $F(2,3)$ 的 Winograd 卷积中输入的元素为 16 个，那么对应的需要执行 16 次 $[RxC]x[CxM]$ 的矩阵乘法。

ARM 设备上没有成熟通用的矩阵乘法实现，特别是针对于量化场景的矩阵乘法计算。另外一方面，高性能计算（HPC）领域实现的矩阵乘法，并不同移动设备上的卷积网络的运算完全相匹配。HPC 领域所涉及的矩阵乘法面向的是上千维度的浮点数矩阵乘法，而本文所面对的，是矩阵维度从数十到数百近千不等的 16 位整数乘法。HPC 领域的典型的优化矩阵乘法仍然有着相当的参考意义，然而在优化矩阵乘法过程中的每一步操作都有着额外的开销，因此这里需要选择性的针对移动端卷积网络参数的特性设计优化的矩阵乘法实现。

轻量级网络中的 Winograd 卷积中的矩阵乘法可以属于小规模的矩阵乘法（small matrix multiplication）[11]。对于矩阵乘法 A x B，其中 A 矩阵的大小为 $M \times K$，而 B 矩阵的规模为 $K \times N$，对于矩阵乘法的规模参数 M, K, N，K 一般不会超过 1024，K 和 N 的范围一般在 32 到 256 之间，这一场景下的矩阵往往可以直接容纳于处理器的缓存中，并且在矩阵乘法的微内核（micro kernel）实现中，微内核中读入的数据不超过 16KB。在这里我们将满足这一特性的矩阵乘法称为中小规模

的矩阵乘法。

高效的矩阵乘法实现需要考虑以下几点：

- 参与计算的数值表示。不同类型的参数存储需求不同，实现同样的操作也需要使用针对各个数据类型的不同指令。另外，矢量寄存器是定长的，比如 128 位长的矢量寄存器只能同时处理 4 个 32 位值或者是 8 个 16 位值，因此数值表示也决定了并行算法设计。
- 矩阵的规模。在矩阵的规模较小时，最为朴素的矩阵乘法实现并没有表现出性能问题，而当矩阵规模变大时，尽管问题的复杂度没有变，但是朴素的矩阵乘法并不能充分利用处理器的并行能力和存储的层次化结构，从而计算性能大幅下降。而各类优化矩阵乘法的方法，其适用性均是在一定的运算规模下的，盲目使用优化方法可能导致性能提升有限，反而引入了额外的开销。
- 并行加载/存储支持。数据的计算和加载在处理器中是由不同的单元实现的，而数据也只有在被加载到积存器中之后才能被用于计算。寄存器之间的操作远远快于寄存器同内存之间的操作，为提高计算性能，需要最大化计算/访存比，单次访存操作实现多个参数的加载和存储的能力，对此有着重要的意义，而在算法的调度上也必须满足，单次加载的多个数据被充分利用之后，再被并行存储。
- 矢量寄存器的位数及数量。这一点相当程度上代表了处理器的并行能力。更长的矢量寄存器意味着单个 SIMD 指令可以同时实现更多数据的计算，而矢量寄存器的数量又决定了可以有多少这样的并行计算可以同时执行。
- 乘累加指令操作延时（latency）。单个乘累加指令需要多个时钟周期完成，指令延时的存在需要结合 pipeline 考虑对于算法效率的影响。单个乘累加指令会存在较高的延时，而乘累加操作却是矩阵乘法中最为频繁使用的指令，而 pipeline 可以减小指令的平均延时，提高计算效率。算法的调度中要减少数据依赖，从而避免 pipeline stall。
- 缓存信息，包括缓存行（cache line）大小，缓存放置策略（cache placement policy），缓存置换策略（cache replacement policy）。缓存的置换策略一般考虑 LRU。缓存行的大小影响到线程并行化中数据分配的设计，要使得多个线程的数据不要处于同一缓存行，造成 Cache False Sharing，大幅拉低效率。同时缓存大小需要结合数据规模和计算执行的模式考虑，在一般的 ARM Cortex A 芯片的多级缓存中，会具备两级缓存，L1 缓存速度远远高于 L2 缓存，但 L1 缓存的容量远小于 L2 缓存，同时 L1 缓存分为数据缓存和指令缓存。因此算法设计中，需要将频繁使用的数据放在 L1 缓存中，而且这一部分数据

要足够小，从而可以为 L1 缓存所容纳，而复用程度相对较低的数据可以放在 L2 缓存中。同时，算法中最为密集的计算，模型不能够不复杂，否则会造成指令缓存的浪费，引起性能下降。

以下将针对于 ARM Cortex A 架构的特征和轻量级量化网络中的参数特征，考虑上述高效矩阵乘法计算的要点展开阐述。在正式介绍矩阵乘法的算法设计之前，由于 Winograd 算法和矩阵乘法中都存在着频繁的对于多维矩阵的索引（indexing）及交换（permutation）操作，在3.4.1 章节中先对于本文所采取的内存结构中多维矩阵的表示做简要介绍，这一矩阵表示方式可以实现简单且高效的索引机制，同时尽可能规避开销较大的交换操作。3.4.2 章节中则对于 ARM Cortex A 架构硬件的特性做概述，作为3.4.3 章节中的矩阵乘法设计的背景。

3.4.1　矩阵在内存中的灵活表示

矩阵在存储空间中是以一整块连续的存储块空间所表示的。在矩阵的存储（storage）中有两个同内存（memory）相关的的关键问题：维度（dimension）和跨度（stride）。

跨度是指在遍历矩阵的过程中，在通过不同维度的过程中需要跨过的字节(bytes) 数。跨度的存在可以使得很多矩阵的处理变得更加快速有效，同时，对于跨度的理解也有助于更加容易的理解矩阵操作的实现。矩阵存储在被称为 data buffer 的连续（contiguous）的同质（homogeneous）内存块 (block of memory) 中。Strides 可以作为一个矩阵的 meta data（元数据）实现多维矩阵之中的不同维度的 index（索引）与它在连续 memory block 中的位置的映射（mapping）。简而言之，strides 表示对于每个维度的连续元素之间的字节间距（byte-separation）。同时矩阵存储空间的同质性也保证了矩阵在存储空间中的连续元素间的单位距离的一致性，比如对于 int32 类型的矩阵，这里计量矩阵元素的单位距离就是 4 个 bytes，每两个矩阵元素在存储中的距离为 4。于是对于多维矩阵而言，比如对于 int 类型的二维矩阵 A，A 在矩阵表示的最内侧的元素在 memory 中连续存储，即二维矩阵中的 column（列）元素在实际的存储中相邻，在二维场景下可以成为 row-major。从 $A[0,0]$ 所表示的元素移动到 $A[0,1]$ 所表示的元素，即在同一行（第 0 行）中的元素间，从第 0 列移动到第 1 列，这里需要在 data buffer 中移动 4 个 byte，也就是在列所对应的维度上的一个 stride。然而，如果从 $A[0,0]$ 所表示的元素的位置移动到 $A[1,0]$, 即在同一列（第 0 列），移动到下一行（从第 0 行移动到第 1 行）所在元素的位置，直观而言，需要先遍历这一行中剩余的元素，才能够到达下一行，然后再下一行中遍历元素到达目标元素，比如矩阵 A 是一个 $3x3$ 大小的矩阵，那么从 $A[0,0]$ 到

$A[0,1]$ 需要遍历 $3x4$ 个 bytes。也就是说矩阵 A 在行的维度上相邻元素间的间距（row stride）是 12 个 bytes。Stride 的存在可以使得矩阵的处理变得更加灵活，也使得很多矩阵操作轻量化，矩阵的转置以及 reshape 等操作可以不用实际操作矩阵，而仅仅是改变矩阵各个 dimension 所对应的 stride 这一 meta data，比如矩阵的转置实际上可以只是交换对应的 stride 并且改变矩阵的 shape 信息。比如，对于前面例子中的 3x3 矩阵 A，假如这里使用 $(12,4)$ 表示矩阵 A 在第 0 维度（row）的 stride 为 12 个 bytes，而在第 1 维度（column）的 stride 为 4 个 bytes。它的转置 A^T 并不需要对于 A 所对应的内存块中的元素做实际的重新排布（reorder），而只需要将 A 原本所对应的 stride 信息修改为 $(4,12)$ 即可。同样的，在考虑到矩阵的 stride 这一元信息（meta info）的场景下，对于矩阵的处理也不能直观的认为矩阵的最内层（innermost）元素之间的距离一定是单位该元素数据类型的尺度。而是在沿着矩阵的某个维度实现遍历处理的过程中，这一维度上对应元素间的距离一定是这一维度对应的 stride。

除此之外，基于原有的矩阵的创建一个新的矩阵对象的操作也可以因此而轻量化。创建一个新的数组元数据，该元数据使用相同的 data buffer 来创建该数据缓冲区的新视图，该视图对缓冲区的解释 (interpolation) 不同（例如，形状，偏移量，字节顺序等），但共享相同的数据字节。例如 slice 操作, 只需要创建一个新的矩阵元数据信息，对应的更改其起始位置的 offset 和各个维度的 stride，而不需要从 memory 中拷贝 data buffer 中的相关数据元素。

3.4.2　ARMv8-A 架构硬件特征

ARMv8 指令集架构的设计中包含 32 个 64 位（doubleword）NEON 寄存器，或者称为 D 寄存器，同时，这 32 个 D 寄存器也可以视为 16 个 128 位（quadword）寄存器，称为 Q 寄存器。

NEON 作为一个 SIMD 并行计算的协处理单元，具有以下流水线执行特征

- NEON 指令运行在其独立的 10-stage 的流水线中
- ARM 在每个执行周期（cycle）可以调度（dispatch）两个 NEON 指令
- 可以容纳 16 条指令的指令队列（16-entry instruction queue）可以用于在指令实际进入 pipeline 之前作为缓冲
- 可以容纳 12 条数据记录的队列用于存储 ARM 寄存器的值，在 ARM 调度指令之后保存当时 ARM 寄存器的值

由于以上的特征，在实现 ARMv8 指令集的大多架构下，由 ARM 到 NEON（由 CPU 处理器到 SIMD 协处理器）的数据传输是相对高效迅速的，而由 NEON

到 ARM 的数据传输则相对有较高的延时（latency）; 同时在 ARM 处理器一方不会因为 NEON 协处理器的执行而停滞，除非 NEON 协处理器的队列已满。ARM 处理器可以向 NEON 协处理器调度分布一系列的指令后继续处理器本身的任务，直到 NEON 指令的结果返回；另外，NEON 指令的实际执行同在编写代码中直观观察到的执行并不是一致的，由于队列的存在，NEON 指令的实际执行是有所延后的。这就导致，如果程序修改了另外一个程序所需要的缓存行（cache line），会导致 ARM 处理器侧的停滞，直到 NEON 的执行同步。

3.4.3　硬件相关的矩阵乘法性能优化

根据前文对于矩阵乘法实现要点的分析，以下将针对于各个要点展开。具体而言，体现为如下具体步骤：

- 首先确定问题建模。确定矩阵乘法计算中的数据类型以及矩阵乘法的规模；
- 硬件层级自底向上分析各个层级的最高效实现，包括：
  - 指令层级：并行访存指令和并行计算指令下的基本算法单元;
  - 寄存器层级：最小化加载/存储操作，提升计算/访存比;
  - 缓存层级：充分利用缓存性能，尽可能减少访存开销;

3.4.3.1　量化计算中的数据表示

讨论数字格式 (numerical format) 时，有两个主要属性。第一个是动态范围，它是可表示数字的范围。第二个是动态范围内可以表示多少个值，这又决定了格式的精度/分辨率（两个数字之间的距离）。

对于所有整数格式，动态范围为 [-2n-1..2n-1-1]，其中 n 是位数。因此，对于 INT8，范围是 [−128..127]，对于 INT4，范围是 [−8..7]。整数型数字可表示值的数量为 2^n。而 32 位浮点数的动态范围为 $\pm 3.4x10^38$，可以表示大约 $4.2x10^9$ 的值。这里可以立即发现 FP32 更具通用性，因为它能够准确地表示各种分布。这的确是满足深度学习模型需求的一个属性，在深度学习模型中，权重和激活的分布通常非常不同（至少在动态范围内）。此外，模型中各层之间的动态范围可能会有所不同。为了能够用整数格式表示这些不同的分布，比例因子 (scale factor) 可以被用于将张量的动态范围映射到整数格式范围。但是，相比于浮点数，整数可表示值的数量要少得多，即分辨率要低得多。在很多情况下，比例因子可以是浮点数。因此，即使使用整数，也会保留一些浮点计算。另外一方面，乘法运算也可以使用位移（shift）实现，这样可以不必乘以 scalar factor，从而消除了浮点运算。在 GEMMLWOP 中，原本未 32 位浮点数的比例因子便是使用整数乘法加上移位运算来近似得出的。在

许多情况下，这种近似对精度的影响可以忽略不计。

卷积和完全连接的层涉及将中间结果存储在累加器 (accumulators) 中。由于整数格式的动态范围有限，如果我们将相同的位宽用于权重和激活以及累加器，则可能会很快溢出。因此，累加器通常以更高的位宽实现。

两个 n 位整数相乘的结果最多为 2n 位数字。在卷积层中，此类乘法被累加 $c \cdot k^2$ 次，其中 c 是输入通道的数量，k 是内核宽度（假定为正方形内核）。因此，为避免溢出，累加器应为 $2n + M$ 位宽，其中 M 至少为 $\log_2(c \cdot k^2)$。在许多情况下，使用 32 位累加器，但是对于四位整数及更低版本，可能会使用少于 32 位的累加器，具体取决于预期的使用情况和层宽度。

因此，可以确定在 Winograd 卷积中使用的累加器为 32 位整数，即这里矩阵乘法的结果将保存为 32 位整数。而另外一方面，量化场景下卷积网络中的参数和特征的表示均为 8 位无符号整数，而在实现 Winograd 变换中为了尽可能保持数值精度，变换过程中将输入转换为 16 位有符号整数计算，并将变换后的结果以 16 位整数存储。到此为止，可以确认，参与矩阵乘法的输入的数据类型为 16 位整数，输出为 32 位整数。

3.4.3.2　轻量化网络中的矩阵乘法规模

不同的网络中参数的规模会存在比较大的差异，在网络计算中实现的矩阵乘法的规模也随之会处于比较大的变化范围。表**??** 中对比了 VGG19 和 ResNet18 中的卷积操作的参数规模，以及各自对应到 Winograd 卷积中的矩阵乘法的参数规模。由于本文实现的 Winograd 卷积只针对于 stride 为 1 的 3x3 卷积操作，因此表**??**中也只统计计算了网络中满足这一条件的卷积，但其中所统计的卷积已经占据了网络结构中的绝大多数卷积操作，对于这些卷积操作的统计结果在很大程度上也反映了整个网络运行时的性能。

显然，轻量级的卷积网络在矩阵乘法中所处理的问题规模同传统的卷积网络相比是截然不同的。轻量化的网络中需要解决的矩阵乘法问题的规模远小于传统的卷积网络。此外，在本文所针对的量化计算场景下，数据在存储设备（内存，缓存，寄存器）中的开销将变得更小。

至此，已经完善了对于应用于移动端的量化 Winograd 卷积中，矩阵乘法问题的建模。总结而言，这里需要解决的问题是，中小规模的 16 位整数矩阵的乘法，乘积则为 32 位整数表示。为方便后文表述，这里定义两个矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$, 对应的尺度分别位 $m \times k, k \times n$，记为 $\boldsymbol{A}(m,k)$ 和 $\boldsymbol{B}(k,n)$，则两者的矩阵乘积可以表示为**??**

表 3.2　卷积操作参数以及 Winograd 卷积中对应的矩阵乘法

网络	输入（HxWxC）	卷积核 (CxHxWxK)	矩阵乘法 (AxB)
VGG19	224x224x64	3x3x3x64	(12544x3)x(3x64)
	224x224x64	64x3x3x64	(12544x64)x(64x64)
	112x112x64	64x3x3x128	(3136x64)x(64x128)
	112x112x128	128x3x3x128	(3136x128)x(128x128)
	56x56x128	128x3x3x256	(784x128)x(128x256)
	56x56x256	256x3x3x256 (x3)	(784x256)x(256x256)
	28x28x256	256x3x3x512	(196x256)x(256x512)
	28x28x512	512x3x3x512 (x3)	(196x512)x(512x512)
	14x14x512	512x3x3x512 (x4)	(49x512)x(512x512)
ResNet18	56x56x64	64x3x3x64 (x4)	(784x64)x(64x64)
	28x28x128	128x3x3x128 (x3)	(196x128)x(128x128)
	14x14x256	256x3x3x256 (x3)	(49x256)x(256x256)
	7x7x512	512x3x3x512 (x3)	(16x512)x(512x512)

卷积核参数后的括号内表示该卷积操作在网络中的重复次数

$$C(m,n) = A(m,k)xB(k,n) \tag{3-10}$$

直观而言，矩阵乘法可以视作是矩阵 A 中的行同矩阵 B 中的对应的列的点积的集合，是对应矩阵元素的乘积的和（sum of element-wise multiplication）。对应的实现如算法4：

Algorithm 4 矩阵乘法

```
for i = 0 → m do
    for j = 0 → m do
        for p = 0 → m do
            C[i, j] += A[i, p] × B[p, j]
        end for
    end for
end for
```

对这一简单直接的矩阵乘法的实现的可视化可如下图3.2所示

而一个针对于高性能计算场景下的矩阵乘法计算则需要针对性的实现多层级的优化。以 BLIS[13] 为例，其中的矩阵乘法的实现策略如图3.3 所示。

总体来看，对于矩阵乘法而言，图3.3 中所描述的算法循环从外而内，意味着

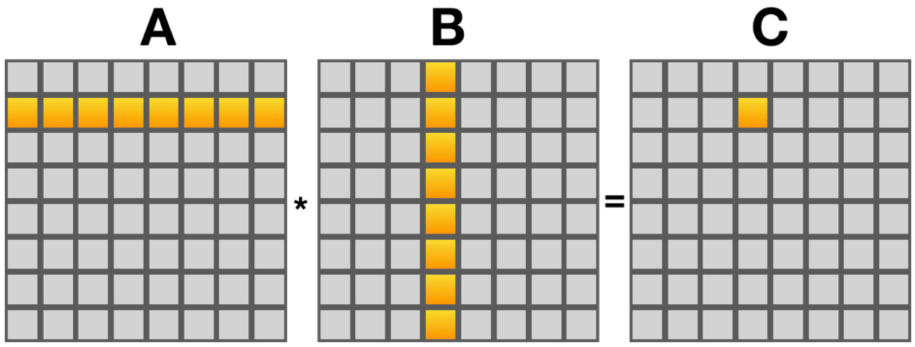

图 3.2　naive 矩阵乘法

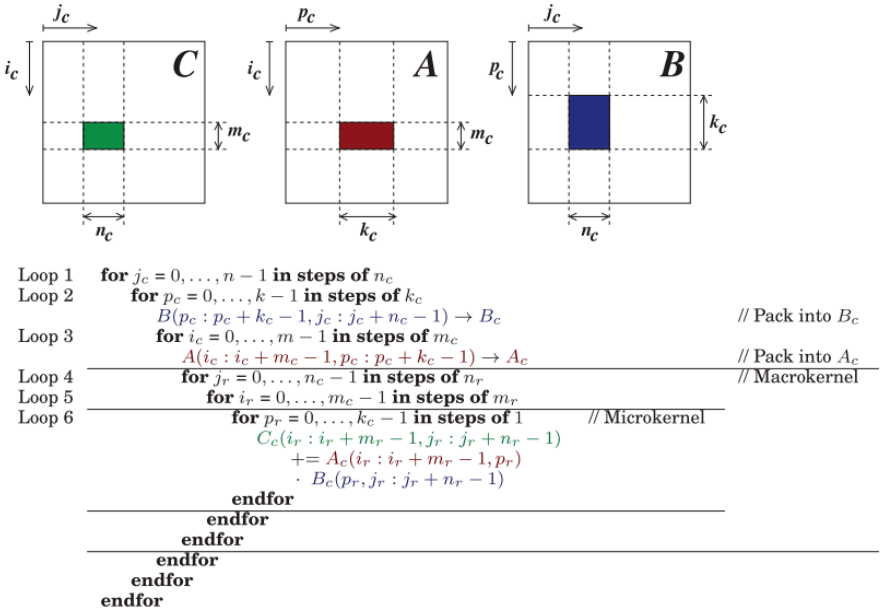

图 3.3　矩阵乘法的高效实现。图片来自 BLIS[14]

问题的分析由宏观入微观。在对于矩阵乘法问题分解的角度而言，这里由外而内针对的问题对象分别是矩阵，缓存块（cache block），寄存器块（register block），寄存器内的矢量（vector）。从计算实现的角度而言，由外及里则分别是矩阵乘法（GEMM），Macro Kernel 和 Micro Kernel。从问题处理的规模而言，则由外而内依次减小。换言之，工作[13] 中的优化策略将矩阵乘法自上而下分解为一系列规模递减的问题，在不同规模的层次上有着针对性的优化方法。正因如此，结合前文中对于本文所涉及的矩阵乘法问题的分析，以下便可以结合本文问题的规模，针对性的实现中小规模量化矩阵的高效乘法实现。

在此之前，仍然有必要简要介绍[13] 中的矩阵乘法实现。对于矩阵乘法问题自上而下的分解，实际上是同具体的硬件特征相关的。图3.4 大致上描述了这一算法中各级子问题同硬件设备的对应关系，同时也是算法实现中各个层级之间数据在不同硬件设备上转移的流程。首先，关于自上而下的子问题的拆分，在如图3.3所

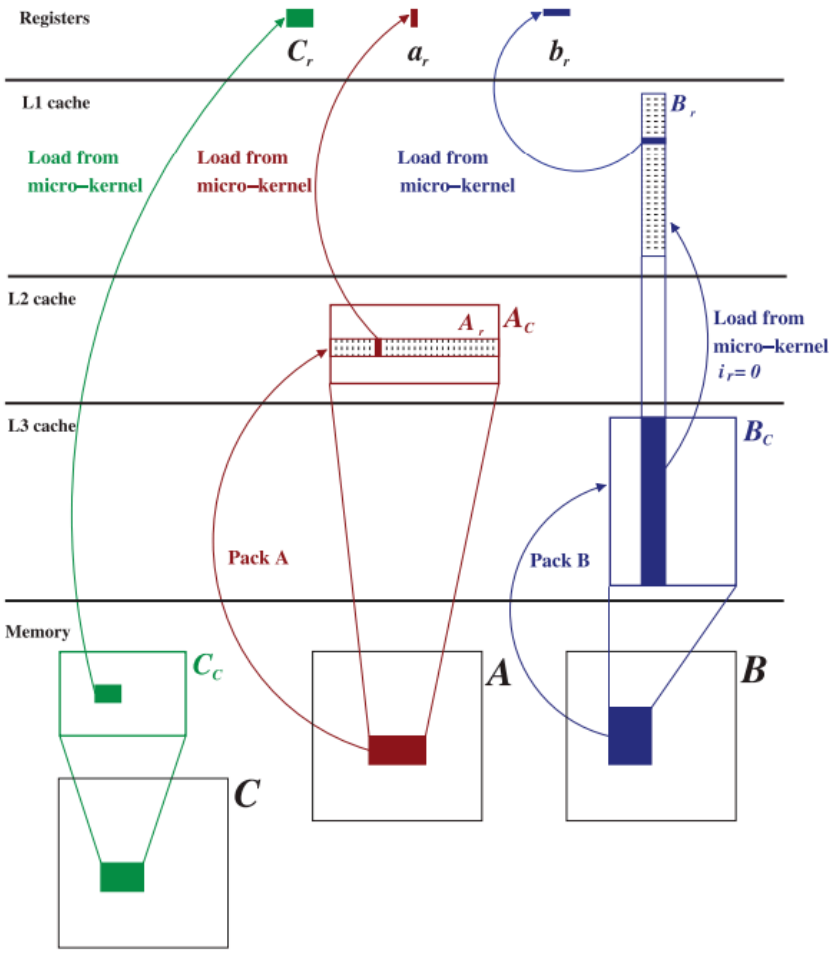

图 3.4　矩阵乘法子问题与硬件关联。图片来自[14]

示的矩阵乘法算法中。在最高层次上，将大小为 m，n 和 k 的矩阵乘法在 k 所表示的维度以 k_c 大小的高速缓存块进行分割，从而创建 rank 为 k 的子问题。在这一阶段，为了实现在计算的最基础层次上上，参与计算的元素在存储中是完全连续的，即矩阵表示中最低维度对应的 stride 为 unit stride，需要将 B 以一种特殊的格式分块打包到连续存储中（标记为 B ）。然后，将每个 rank 为 k 的子问题沿 m 维度，按照缓存块大小 m_c 分块，从而创建 block-panel 子问题。然后将当前的 $m_c \times k_c$ 块，记为 A_i，按照计算中的最基础级别，参与计算的元素在存储中是连续的的原则，打包到 A。然后将剩下的该 block-panel 子问题（$C_i = C_i + A_i B$）实现为高度优化的汇编计算内核。而该内核将继续沿 n，m_c 划分矩阵，最后在 k_c 维划分矩阵。另外，从数据的转移角度来说，即考虑存储设备的访问开销，这种对于问题的划分也是基于频繁复用的数据应该尽可能存储在高速存储设备中的原则。

这一矩阵乘法中的多个循环的使用在于根据缓存块大小和寄存器的容量，精心设计矩阵的分块策略，从而使得子矩阵的分布同存储的层级结构相匹配，以达

到数据复用的目的。同时 A 和 B 的子矩阵被复制打包到临时工作区（workspace），从而使得矩阵乘法操作的 micro-kernel 能够在内存中连续访问矩阵元素，从而提高了缓存和 TLB 性能。而一般打包过程的成本可以通过计算本身来摊销，在计算的规模足够大时，这一成本是可以忽略不计的。

而这其中决定最内层的循环的调用的寄存器相关的参数 m_r, n_r, 以及较外层的循环调用的高速缓存相关参数 m_c, k_c, n_c 一般由具体硬件的特性所决定，比如矢量寄存器的大小，缓存大小，缓存的相关性等。工作[14] 为这些参数的选择提供了一种分析性模型。

至此，对于本文解决的 Winograd 卷积中的矩阵乘法的分析，以及一般意义上的高效矩阵乘法的实现阐述均已完备。以下将自下而上，从问题实际和硬件特性的角度针对性的设计矩阵乘法。

3.4.3.3　SIMD 矢量化矩阵乘法计算基本单元

首先针对于矩阵乘法计算中的最内层，在矢量级别处理矩阵乘法的计算。考虑硬件提供的并行数据加载机制，在 ARM Cortex A 设备上，可以单次加载多个顺序存储的数据到 NEON 矢量寄存器中。这意味着，在加载矩阵中的元素时，针对于 16 位整数，一个加载指令可以最多加载 8 个同一行中的矩阵元素。考虑矩阵乘法 $A \times B = C$ ，C 中第 i 行，第 j 列的值，由 A 中第 i 行的所有元素，同第 j 列的所有元素的点积确定。而这里的并行加载机制对于访问矩阵中的列元素的支持是不友好的。比较直接的一种方案是对于 B 中的元素做转置，转置之后，B 矩阵中的元素便也可以发挥并行加载指令的优势，减少存储访问。但转置本身是一个成本比较高的操作，因此这里考虑另外一种优化策略，从矩阵乘法的计算方式角度，提出解决方案。

矩阵乘法有一个属性在于：矩阵 A 乘矩阵 B 实际上是将矩阵 A 中的列以矩阵 B 中的列元素作为参数的线性组合，或者说是，矩阵 B 中的行和矩阵 A 中的行元素作为参数的线性组合。

首先，矩阵的右乘，可以视作列的组合；对于简单的矩阵右乘一个列向量的情形

$$\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix} \times \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} ax_1 + by_1 + cz_1 \\ ax_2 + by_2 + cz_2 \\ ax_3 + by_3 + cz_3 \end{pmatrix} \tag{3-11}$$

而对于这一过程做一个直观的可视化可以表示为图3.5

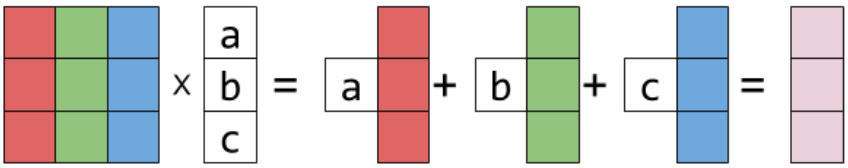

图 3.5　列向量右乘矩阵

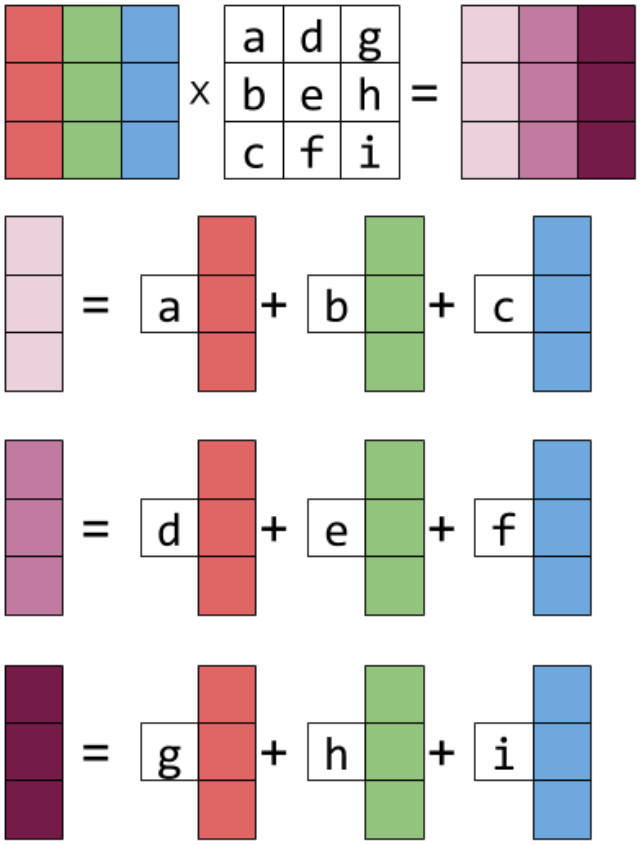

图 3.6　矩阵右乘

矩阵中的每一列同列向量中的每个元素相乘，矩阵中的一列同列向量中的单个元素（Scalar）的乘积仍是一个列向量，而最终的结果则是这些列向量的和。这一情形推广到右乘一个矩阵则是相似的情形，乘积的结果的矩阵中的每一列仍是矩阵列的线性组合。

$$\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix} \times \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} = \begin{pmatrix} ax_1+by_1+cz_1 & dx_1+ey_1+fz_1 & gx_1+hy_1+iz_1 \\ ax_2+by_2+cz_2 & dx_2+ey_2+fz_2 & gx_2+hy_2+iz_2 \\ ax_3+by_3+cz_3 & dx_3+ey_3+fz_3 & gx_3+hy_3+iz_3 \end{pmatrix} \tag{3-12}$$

这一过程的可视化表示如图所示为图3.6

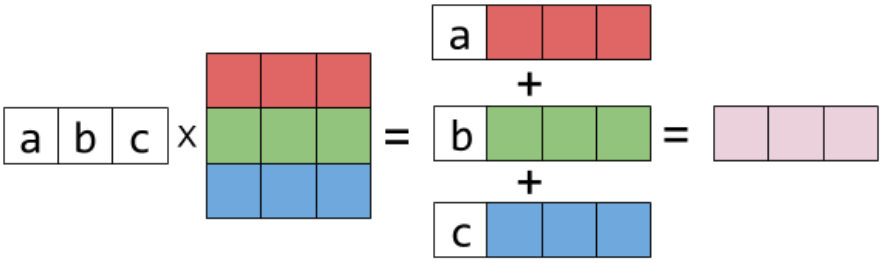

图 3.7　行向量左乘矩阵

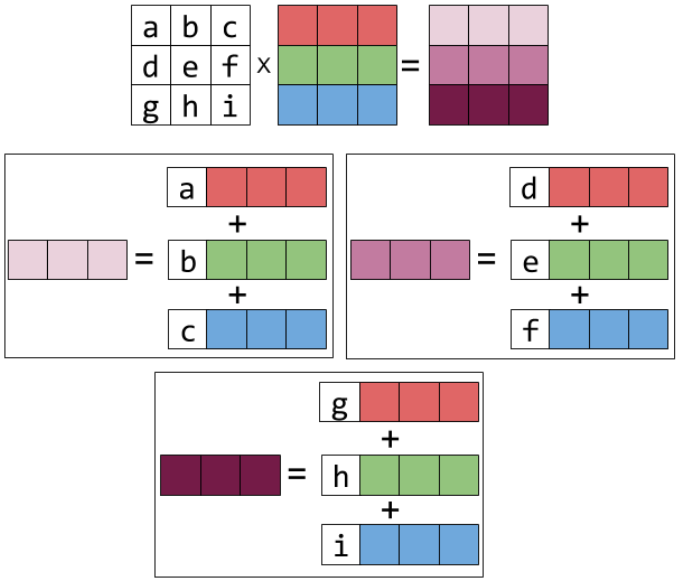

图 3.8　左乘矩阵

而对于左乘一个矩阵则可以视为是矩阵行的线性组合。首先对于一个矩阵左乘一个行向量的情形，可以表示为3-13

$$\begin{pmatrix} a & b & c \end{pmatrix} \times \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix} = \begin{pmatrix} ax_1 + bx_2 + cx_3 & ay_1 + by_2 + cy_3 & az_1 + bz_2 + cz_3 \end{pmatrix} \tag{3-13}$$

可视化表示为3.7。

而在左乘一个矩阵的过程中，则是左乘行向量的情形重复作用于矩阵中的每一行。相对应的可视化表示可以为图3.8。

而另外一方面，ARM Cortex A 架构的 NEON 指令中是支持按照矢量寄存器的

data lane 进行计算的，因此可以加载矩阵 A 同一行中的多个元素到矢量寄存器 R，再将 R 中各个 data lane 的元素分别同 B 中同一行的多个顺序元素分别相乘。对于 B 中的多行元素重复上述步骤，便得到矢量同矩阵的乘积结果。再对于 A 中的多行元素重复矢量-矩阵乘法，并对结果实现前述的线性组合，便得到矩阵乘法结果。

至此，本文针对于硬件的数据加载结构和计算指令的 data lane 特征，利用矩阵计算的线性组合性，对于矩阵乘法的基本实现单元实现了修改。下文中将在较高的层次，矢量寄存器级别，实现对于矩阵乘法的优化。

3.4.3.4　矢量寄存器分块化处理– 优化最内层循环

众所周知，寄存器对于数据的操作，总体上存在三个阶段：加载，计算和存储写出。对于计算的执行而言，现代 CPU 一般存在多个用于调度分配 SIMD 计算的执行端口，和多个用于调度数据存取的端口。这意味着，在一个时钟周期里可以实现调度执行多条读取数据的指令，或者调度执行多条乘法计算指令。而在存储访问方面（数据加载和写出），从读写的速度考虑，处理器中的寄存器间的操作的速度是远远高于缓存的读写性能，而缓存的读写性能又远远高于内存，内存的读写性能则又远远高于硬盘等外部存储。计算机存储结构的设计是层次化的，算法设计中对于存储设备的访问也需要遵守这一特性。高效计算的过程中对于存储性能的优化的目标在于，尽可能的减小访问存储的操作同计算操作的比例，提高高速设备中数据的利用率，在算法的设计上保证加载到高速设备上的数据得到充分利用，并可复用。而在实现矩阵乘法的背景下，这就意味着，从矩阵 A 中加载的数据需要被用来执行多次计算操作，这一点可以由3.4.3.3章节所述方法中，A 中的元素同来自 B 中的同一行中不同列的元素做计算而保证；同时，B 中的元素也需要被用于执行不止一次计算，而要实现这一点，则可以通过加载数据时，从矩阵 A 中每次加载多行的元素，从而使得 B 中的每一列的元素可以得到复用。而从计算结果来看，乘积矩阵 C 中的每一行的元素都是一个矢量累加值，并且对应于矩阵 A 中的行，每次处理多行 A 中的元素，需要在计算中具备与之数目相同的累加器用来暂时存储矩阵 C 的结果。这一过程中使用了多个寄存器实现了矩阵中的一个小块（block）的计算，这一过程被称为寄存器 blocking （register blocking) 。

图3.9 对于 register blocking 实现做了比较直观的可视化。一般 register blocking 方法存在这两个参数，垂直方向的 blocking size m_r 和水平方向的 blocking size n_r。这两个值的上限受到矢量寄存器的数目所限制，而当 m_r 和 n_r 均为 1 时，算法退化为朴素矩阵乘法。为了尽可能避免数据依赖和指令操作延时导致的 pipeline stall，m_r 和 n_r 的取值应该足够大，但并不是越大越好。因为还需要分配一部分寄存器作

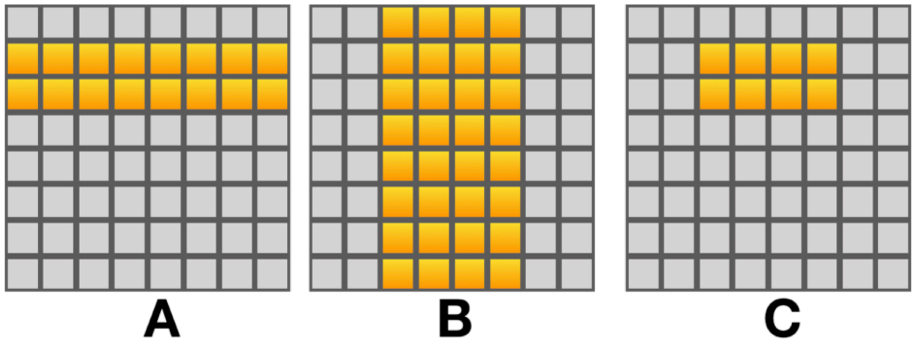

图 3.9　矩阵乘法实现 + Register Blocking

为累加器暂存结果矩阵 C 中的值。在单个的 register blocking 计算过程中，计算输出的结果的维度为 $m_r \times n_r$，因此用于暂存结果累加器的数目由 m_r, n_r 共同决定。寄存器总数有限，这里要求用于暂存 C 中的输出的寄存器尽可能少，从而有足够多的寄存器用于计算矩阵 A，B 中的输入，因此这里的 blocking factor m_r, n_r 的取值需要做一定的权衡。

工作[14] 中对于寄存器级别的 blocking factor 的选择提出了一种分析模型。从硬件参数通过式3-14可以推算合适的 m_r 和 n_r 值。其中 N_{VEC} 表示单个矢量寄存器所能容纳的数据的数目，N_{VFMA} 表示可执行矢量乘累加计算（Vector FMA）的寄存器的数目，而 L_{VFMA} 则表示乘累加计算的操作延时（latency）。

$$m_r = \left\lceil \frac{\sqrt{N_{VEC} L_{VFMA} N_{VFMA}}}{N_{VEC}} \right\rceil N_{VEC} \tag{3-14}$$

$$n_r = \left\lceil \frac{N_{VEC} L_{VFMA} N_{VFMA}}{m_r} \right\rceil \tag{3-15}$$

由这一过程推算出的寄存器分块因子（register blocking factor）m_r 和 n_r 在大多情形下同成熟的矩阵计算库中通过经验型调优方法获得的值吻合。本文经过对于主流的 ARM Cortex A 设备的相关硬件参数值做统计，并结合本文实现的矩阵乘法背景设置，分别取 m_r，n_r 的值为 8，8。

此外，图3.3 中的最内层循环，即 Loop 6 中在 k_c 维度的循环中的步长为 1 ，而本文的矩阵乘法在3.4.3.3 中所述的矩阵乘法实现在最基本单元的单元，就已经确定了矩阵 A 的多行元素和矩阵 B 中的多列元素的并行化计算，这一设计将同时影响到 packing 策略，将在后文展开阐述。本文的实现中，考虑矢量寄存器长度和数量的限制，同时兼顾编译器的循环优化中的 loop unrolling 和数据预取（data prefetching），本文中取在 k_c 维度的循环步长为 4 。

至此，寄存器级别的计算优化基本完善。以下将从计算机存储设备中的层次化结构中，数据存取速度次于寄存器的高速缓存的硬件特征，分析在矩阵乘法中

计算规模更高一级的问题的优化。

3.4.3.5　缓存优化策略讨论 Packing 与 Blocking

处理器每次从主内存中（main memory ）中加载数据的过程中，会同时将该数据所处的一个缓存行（cache line ）加载到 L1 高速缓存中，而此后参与计算的数据如果同该数据处于同一缓存行，那么 CPU 只需从 L1 缓存中加载数据，而不必访问需要较大储存访问开销的内存。缓存友好的数据组织方式要求参与计算的数据依次顺序排列在储存中。为实现数据访问模式的缓存友好性，可以针对于章节3.4.3.4中所描述的计算模式，对于参与计算的数据重新排布。这一过程一般成为 packing。Packing 可以使得参与计算的数据均位于连续的存储中，最大化缓存命中率，同时还可以使数据在缓存行（cache line）边界和内存页（page）边界对齐，从而可以使用访问对齐数据的指令，加速数据访问；此外，Packing 还更有利于数据在对应的各级缓存中的预加载。

在3.3 中所描述的寄存器级优化方法下，矩阵 A 中的数据需要重新排列为 column-major 的格式，即储存中连续的数据处于同一列中，而这一过程需要将矩阵 A 转置。转置操作的计算开销不容忽略，而另外一方面，关联到神经网络中的应用，A 对应于特征输入，而 B 则代表模型的权重。在网络模型的执行过程中，A 的值是不定的，网络每次的输入值在实际应用中是不同的，对于 A 的优化不能得到复用。综合这两点考虑，packing 对于输入的矩阵 A 的处理，所带来的计算效率提升将相当有限。

而在3.4.3.4 中所描述的矩阵乘法实现下，对于矩阵 A 而言，计算过程中会有多列元素参与，在没有 packing 的实现下会具有比3.3 中方法更高的缓存效率。另外一方面，参考**??** 中的轻量级网络的参数规模，在量化场景下，L2 缓存的容量便可以满足 A 中的参数量 (这一点将在后面展开)，这在一定程度上保证了 A 中数据的缓存效率底线。考虑矩阵乘法中的另一输入，矩阵 B 的值是固定的，从而对于 B 所执行的各种优化处理，则可以在每次的执行中被复用。并且这些处理可以在预处理阶段完成，而不给网络模型的运行时带来开销。矩阵 B 的 packing 需要按3.4.3.4中的算法，将多行中的同列的长度 n_r 的向量首位相接，如图3.10所示，按照 Z 字形遍历矩阵，完成矩阵的重新排布。

而在 Intel 的 CPU 高性能计算库 MKL 的实现中，有发现[15] 表明尽管 Packing 方法在较大规模的矩阵乘法中有着比较明显的效率提升，但是在机器学习模型中常用的规模比较小的矩阵乘法中的性能提升效果有限，并且 packing 引入额外的开销无法在小规模的计算中被充分摊销，从而甚至会引起性能的负向优化。本文将

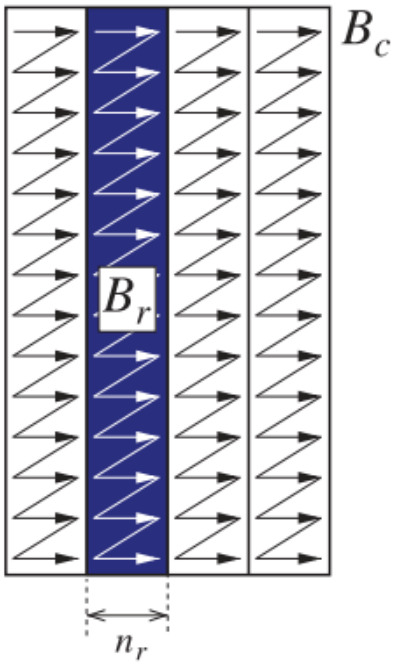

图 3.10　矩阵 B 的 packing 策略

在后文中通过实验分析 Packing 策略对于 ARM Cortex A 设备上的中小规模矩阵乘法效率的具体影响。

而在 packing 之外，另一可以有效提升缓存效率的方法为 Cache Blocking。Blocking 方法的目的在于减少对于内存带宽的压力，避开内存带宽瓶颈。Cache blocking 通过组织数据，将数据切分成适合缓存大小的块，将规模较大的矩阵中的一小部分加载到高速缓存中，并且在矩阵乘法实现中尽可能复用缓存中的矩阵子块，而不必访问存储开销更大的内存，提高缓存效率。

图3.3 中的最外围的三层循环 Loop1 ，Loop 2，Loop 3 即是对于 cache block 的划分过程。而其内层的两重循环 Loop 4 和 Loop 5 则代表了在 Cache Block 内部计算各个 Register block 的过程。以下对于 Cache Blocking 的实现做简要描述。

假设 A, B, C 矩阵都是 nxn 的矩阵，这里对于矩阵的分块每一个小块的尺度则是 bxb，记 $N = n/b$ ，这一算法在低速存储设备上的存储访问操作（memory operation）M 为3-19

$$M = N \times n^2 \quad \text{read each block of B} N^3 \times (N^3 \times \frac{n}{N} \times \frac{n}{N}) \tag{3-16}$$

$$+ N \times n^2 \quad \text{read each block of A} N^3 \tag{3-17}$$

$$+ 2 \times n^2 \quad \text{read and write of block C} \tag{3-18}$$

$$= (2 \times N + 2)n^2 \tag{3-19}$$

Algorithm 5 Cache Blocking 矩阵乘法

```
for i ← 1 → N do
    for j ← 1 → N do
        read block C_{i,j} to fast memory
        for k ← 1 → N do
            read block A_{i,k} to fast memory
            read block B_{k,j} to fast memory
            C_{i,j} = A_{i,k} × B_{k,j} // matrix multiplication on tile
            write C_{i,j} to slow memory
        end for
    end for
end for
```

于是在这一算法实现下的矩阵乘法中，计算操作同访存操作的比值为 $2 \times N^3/((2 \times N + 2)n^2) \approx n/N = b$，在 n 足够大时，这一近似成立。因而可以通过矩阵分块计算中的 block size 块大小实现这一算法的效率提升，同时 block size 也并不是可以任意大，这一优化的出发点在于优化利用快速缓存，而这一值同时也受到缓存大小的限制，上述矩阵乘法中的三个块 $\boldsymbol{A}_{i,k}, \boldsymbol{B}_{k,j}, \boldsymbol{C}_{i,j}$ 均必须不能溢出快速缓存的容量限制，

传统的 GEMM 实现中往往对矩阵重新打包以更好地利用缓存层次结构，以期在大量计算上分摊打包开销，由于缓存关联性有限（limited cache associativity)，在每次 micro kernel(图3.3 中的最内层循环 Loop 6) 中不得不读取尽可能多的可能用于计算的输入进入 cache，在此过程中，矩阵子块中的不同行的值可能落入同一 cache set，造成缓存冲突，从而导致性能下降。而这一情形，在量化参数的表示下将不再成为问题。

对于如表**??**中所示的本文所针对的经量化网络中的矩阵乘法参数的规模，以在智能手机上普遍应用的 ARM Cortex A77 芯片为例，该结构的芯片对应的 L1 指令缓存和数据缓存均为 64KB，而 L2 缓存可以为 128KB，256KB 或者 512KB（由具体的芯片制造商确定），同时 Cortex A77 还具有可以高达 4M 的 L3 缓存。这使得量化的 A 矩阵可以完全容纳于 L2 缓存中。同时，矩阵乘法最内层循环中的 B 矩阵中的子块，即图3.4 中的 $\boldsymbol{B}_r$ 可以完全存放在 L1 缓存中，$\boldsymbol{B}_r$ 在3.3 中的循环 Loop 5 中得以复用。总而言之，对于轻量化的量化网络模型参数而言，在实现了本文前面所述的 Register Blocking 策略，k 维度并行化计算和对于 B 矩阵的 Packing 之后，ARM Cortex A 设备的缓存可以完全满足计算需求，参与计算的各个矩阵子块，可

以完全容纳于各自对应的快速储存设备的层次结构中，因而这里缓存块优化策略（Cache Blocking）将不会具有明显的优化效果，而 cache blocking 方法又会引入对于每个划分出的 cache block 中的 data movemnet 开销，从而在低精度小规模的参数场景下，可以去除这些不必要的参数的内存转换以及输入的重新打包。因此本文优化中在尝试了 cache blocking 的优化可能性之后，去除了 cache blocking。

综合上述的分析，本文中最终确定的矩阵乘法设计如6所示

Algorithm 6 适用于量化轻量级网络中的 Winograd 卷积的矩阵乘法

$m_r \leftarrow 8$

$n_r \leftarrow 8$ ▷ Blocking factor

$B_r \leftarrow PackMatrix(B)$ ▷ Pack B matrix

for $j_r \leftarrow 0, \cdots, n$ **in steps of** n_r **do**

  for $i_r \leftarrow 0, \cdots, m$ **in steps of** m_r **do**

    for $p_r \leftarrow 0, \cdots, k$ **in steps of** 4 **do**

$$C(i_r : i_r + m_r, j_r : j_r + n_r) = A(i_r : i_r + m_r, p_r : p_r + 4) \cdot B(p_r : pr + 4, j_r : j_r + n_r)$$

    end for

  end for

end for

3.4.4　矩阵乘法性能优化验证

矩阵乘法优化的主要目标在于提高计算操作同访存操作之间的比例，缓存性能是这一目标的一个重要指标。以下我们通过对于缓存性能的分析，通过实验证明本文所提出的优化矩阵乘法的有效性。

首先考虑实际的应用场景，这里的矩阵乘法的参数设置参照表**??** 中的 ResNet 18 网络中的卷积操作对于的矩阵乘法，矩阵的输入参数的数值精度为 16 位整数，输出为 32 位整数。实验所使用的硬件设备为 64 位 Arm 架构的 AWS Graviton 处理器。AWS Graviton 由 Amazon Web Services 使用 64 位 Armv8 Cortex A72 微架构定制而成，属于 ARM 的第一代 16nm Neoverse 平台。为在 Amazon EC2 中运行的云工作负载提供更高的性价比。Graviton 支持 NEON SIMD，具有 32KiB 的 L1 指令缓存，48KiB 的 L1 数据缓存和 2MB 的 L2 缓存，单核主频 2.3GHz。本实验中使用单个四核 Graviton 处理器。其中 L1 数据缓存 cache line 大小 64 bytes，每个 Cache Set 中包含两个 Cache Line（2-way set associativity），L2 数据缓存 cache line 大小为

表 3.3　矩阵乘法性能实验参数配置

矩阵乘法	参数大小
m1	(784x64)x(64x64)
m2	(196x128)x(128x128)
m3	(49x256)x(256x256)
m4	(16x512)x(512x512)

表 3.4　4x8 Register Blocking 矩阵乘法缓存效率

矩阵乘法	L1 数据缓存 miss rate/%
m1	2.0
m2	2.8
m3	28.4

64 bytes，每个 Cache Set 中包含 64 个 Cache Line 。

实验中使用 Valgrind 组件 Cachegrind 来测量矩阵乘法实现的缓存效率。Cachegrind 是一个缓存分析器。它可以对 CPU 中的 I1，D1 和 L2 缓存进行详细的模拟，因此可以准确地指出代码中的缓存未命中的位置。它以功能，模块和整个程序的摘要来标识针对每一行源代码执行的高速缓存未命中次数，内存引用和指令。Cachegrind 最大的短板在于其执行速度，Cachegrind 运行程序的速度比正常运行慢 20 到 100 倍。

为表述方便，这里对于表**??** 中 ResNet18 中的卷积操作对应的矩阵乘法的表示做如表3.3 所示约束：

首先是使用 4 x 8 的 register blocking 策略（OpenBLAS 实现中采用的 blocking factor）。对于这一实现实验得出的 L1 数据缓存的 miss rate 如表3.4 （缓存未命中的统计量，越小证明缓存使用越高效）所示。

可见在仅使用 4x8 的 reigster blocking 优化下，在矩阵乘法的规模相对较小的情形下，缓存效率是可接受的。但在达到实验配置中的 m3 ，即 A 矩阵为 49x256，B 矩阵为 256x256 的场景下，就已经出现了非常明显的缓存资源浪费了。在此基础上，这里针对于 m3 所示的矩阵乘法，在实现中加入 cache blocking，并对其中的分块因子（blocking factor）展开分析，尝试不同的组合，期望获得最优的缓存效率。以下针对于 m3 中的矩阵乘法参数配置，分析 cache blocking 方法实际带来的缓存利用率提升。这里针对于 m3 中的较大两个维度即 k 和 n 维度实现了不同的 blocking factor，具体的结果如表3.5。

从表 3.5 中的实验结果可见，cache blocking 方法在针对参数做细致的调节之

表 3.5　Cache Blocking 矩阵乘法缓存效率对比

k_c	n_c	L1 数据缓存 miss rate/%
16	32	18.7
16	64	15.7
16	128	13.3
16	256	12.3
32	32	24.8
32	64	19.9
64	32	46.1
64	64	46.6

表 3.6　4x8 Register Blocking 矩阵乘法缓存效率

矩阵乘法	L1 数据缓存 miss rate/%
m1	0.1
m2	0.2
m3	2.6
m4	2.0

后，依然可以达到相当程度的缓存效率提升，但是缓存效率依然不够理想。其中最好的 cache miss rate 依然高达 12.3%。Cache Blocking 在本文所处理的矩阵乘法中，优化效果十分有限，证实了前文中对于 cache blocking 方法在本文问题中的适用性的论述和预测。

以下我们对于前文分析所确定的矩阵乘法算法6以及前文分析的 register blocking factor 和对于 B 矩阵的 packing 策略，完善实现之后，测量这一实现的缓存性能，结果如表3.6 所示。

表 3.6 中的结果证明了前文分析所得出的矩阵乘法算法的有效性，同时也得出了同 Intel MKL 库在 x86/64 硬件设备下实现矩阵乘法[15] 不同的结论，证实了对于网络权重的 packing，在我们的实现背景是有着明显的优化效果和必要性的。算法6 所实现的矩阵乘法高效利用了处理器的并行计算能力和数据并行加载机制，同时充分适应了计算机访问速度由高变低，容量由小变大的层次化存储结构，解决了 Winograd 卷积中的计算瓶颈，为实现高效的 Winograd 卷积算法奠定了坚实的基础。

表 3.7　实验中的卷积操作配置

标记	输入	卷积核
conv1	56x56x64	3x3x64
conv2	28x28x128	3x3x128
conv3	14x14x256	3x3x256
conv3	7x7x512	3x3x512

3.5　实验设计与结果分析

为了验证本文中实现的整数快速卷积实现的有效性。本节基于此方法针对于移动设备实现了该卷积算法，并在移动硬件设备上进行了运行效率的实验。下面将分别介绍实验中的所涉及的实验硬件环境，评价标准，实验设计和实验结果。

3.5.1　实验环境与评价标准

实验中所使用的处理器同上述矩阵乘法实验。实验中的卷积操作使用 ARM NEON Intrinsics 实现，通过 GCC ARM 9.2 交叉编译，多线程支持使用 OpenMP 实现。

实验评价标准为单个卷积操作在上述硬件处理器设备上的卷积运行时间，每个卷积操作执行 100 次取平均执行时间。

3.5.2　实验设计

为验证本文整数卷积方法在移动端实现的优化有效性，这里同在移动设备应用最为广泛且同时支持量化卷积操作的 TensorFlow Lite 的量化卷积操作做对比。Tensorflow Lite 中的卷积实现采用 im2col 算法，使用 GEMMLowp 高度优化的量化矩阵乘法实现。

实验中的卷积操作的参数配置和输入尺度来自于 ResNet[16]。我们对于实验中所使用到的卷积操作的输入和参数配置做如表3.7 所示的表示。

3.5.3　实验结果及分析

TensorFlow Lite 同本文实现的 8 位整数量化卷积计算的执行效率对比如表3.8 所示，这里 TensorFlow Lite 的卷积操作时间测量使用 TFLite Model Benchmark Tool 实现①可见在移动设备中，本文实现的卷积算法达到了接近理论预期的 2 倍以上加速效果。

① TFLite Model Benchmark Tool 是 TensorFlow 团队官方提供的用于对 TFLite 模型及各个运算操作实现基准测试的工具，详情可见 https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark

表 3.8　不同卷积核及输入尺度下卷积操作时间（ms）对比

卷积	tf lite	本文	加速比
conv1	27.797	15.04	1.848
conv2	24.252	9.00	2.695
conv3	22.816	8.02	2.845
conv3	23.096	10.02	2.305

表 3.9　不同输入及卷积尺度下 Winograd 卷积中各个阶段的时间（ms）分布

卷积	input transform	GEMM	output transform
conv1	0.73	10.7	1.68
conv2	0.37	10.18	0.60
conv3	0.34	10.53	0.23

下面进一步讨论不同的卷积操作中 Winograd 算法的各个阶段的时间占比分布，表3.9 中统计了 Winograd 卷积在运行时的三个阶段，输入变换（input transform），矩阵乘法（GEMM）和输出变换（output transform）中分阶段的平均计时（权重变换可以 offline 完成并在运行时复用）。图3.11 则更直观的显示了三阶段在时间占比的分布。可见在 Winograd 卷积操作中，对于输入和输出的变换仅仅占到整个卷积运行时间的一小部分，并且矩阵乘法是在卷积操作中的时间占比会随着卷积 channel 的增大而增大，在卷积 channel 较大的场景下，矩阵乘法占据了整个卷积操作中 90% 以上的时间。

一方面，上述结论证明 Winograd 卷积中对于输入和输出的变换所带来的额

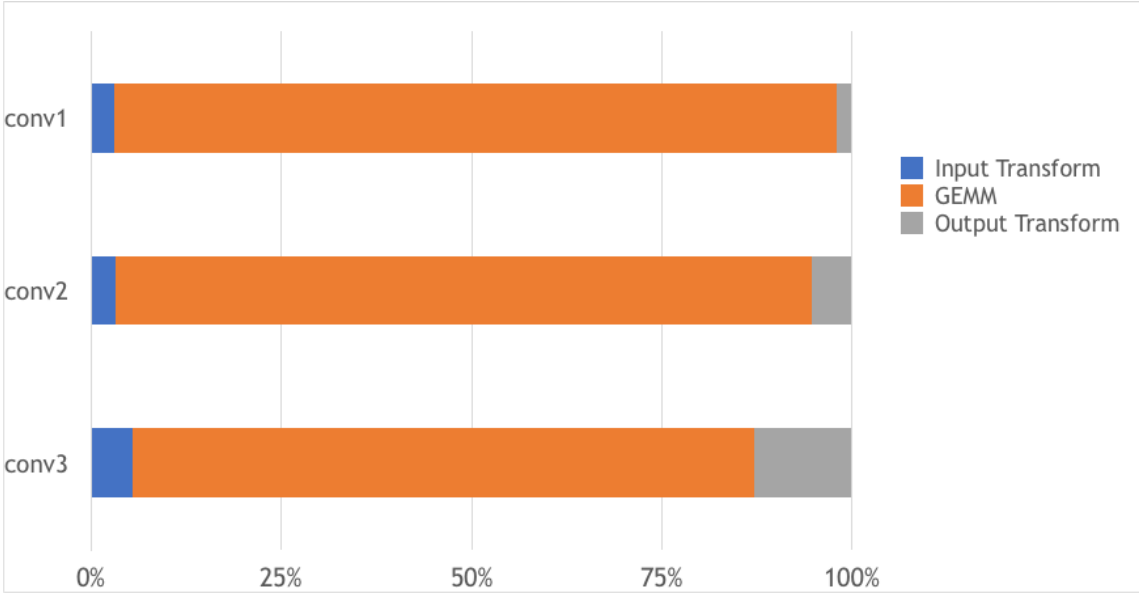

图 3.11　不同输入及卷积尺度下 Winograd 卷积中各个阶段的时间（ms）分布

外计算开销相对而言并不大，特别是在 channel 维度较大的场景下更是微乎其微。输入变换所带来的开销可以在变换后的输入的多次复用中得以充分补偿，而输出变换所带来的开销其实也可以通过矩阵乘法计算量的减少而摊销。另外一方面，Winograd 卷积中矩阵乘法占据整个卷积操作中的绝大多数时间，也正好最大化了 Winograd 卷积的加速增益。Winograd 卷积相对于直接卷积方法或者基于 GEMM 的卷积方法最大的优势便在于其减小了乘法的规模，矩阵乘法的占比在卷积运行时的占比越高，则 Winograd 卷积相对于基于 GEMM 的卷积方法的加速越明显，其实际加速效果越接近于其理论加速值。

第 4 章　量化卷积操作在嵌入式设备的实现优化

前面的章节已经对于本文中针对于加速卷积计算所提出的主要的方法，包括多通道的局部区域化 Winograd 卷积以及中小规模量化矩阵的乘法，以下将主要针对于前文所提出的量化卷积操作作为卷积神经网络中的组成部分，分别从指令并行优化，线程并行优化，支持端到端整数计算的量化/反量化（quantize/dequantize）策略以及网络部署中的卷积相关的操作融合，展开描述。并将这一卷积实现集成到具体的卷积网络中，验证其对于卷积网络的加速效果。

4.1　NEON 计算指令并行优化

到目前实现为止，在 ARM 处理器上最为广泛得以应用的并行计算资源便是 NEON，这一技术自 ARMv7-A ISA 起成为 ARM 芯片的处理计算密集型应用的最为通用的主力解决方案，而 ARM 也在 ARM v8.2A 指令架构中提出了对于 NEON 的矢量计算的扩展 SVE(Scalable Vector Extension)，然而作为一项可选的扩展，目前的通用 ARM 计算设备中几乎没有支持这一特性 (富士通的下一代超算使用 ARM 架构，支持 SVE)。另外例如高通这样的集成电路设计者会在 SoC 方案中添加一些专门的计算单元，比如高通的 Hexagon DSP 针对性的优化了对于量化计算的支持，HTA（Hexagon Tensor Accelerator），HVX(Hexagon Vector eXtension) 的支持，使得在这种设备上的 on device 模型推理计算有了额外的优化途径，并减轻了 CPU 的压力。同时 NVIDIA 也为其面向移动端的 ARM 架构芯片提供了 GPU 和 Cuda 支持。但是这些额外的专门硬件的支持网络模型推理计算的方案，局限性也是显而易见的，即受到硬件本身的局限，不具备通用性。在实现可移植，强适应性，高效率的计算密集应用下，NEON 还是目前最优的选择。

在细节上实现高效的 NEON 指令的调用以及调度对于在 ARM 架构设备上的高效智能计算具有着至关重要的意义。在实现量化的快速卷积过程中，本作遵循了以下优化原则：

- 连续的计算指令之间不得存在数据依赖。某些计算指令，比如乘累加，的操作延时（latency）比较高，为了减少指令等待时间，需要避免将当前指令的目标寄存器用作下一条指令的源寄存器。在实现中，这里将 vld(数据加载)，vmlal（乘累加），st（数据写出）等操作在调度的过程中实现第一阶段通过多个 vld 指令加载各项所需的数据，第二阶段通过乘累加操作计算相互独立的矢量计算，这里每个乘累加操作需要有各自独立的累加器，否则累加过程会

存在 data dependence 造成指令 stall。第三阶段再将乘累加的结果写出。

- NEON/FPU 是一个独立的处理单元，Load/Store Unit 也是一个独立处理单元，两个单元可以并行执行。Load/Store Unit 可以在加载到第一个数据的时候立即把数据 forward 给 NEON/FPU，这样在只有一条 VLDM 指令时，只需要引入 1 个 CPU cycles 的延迟了。这就提示我们通过合理的打散 VLDM 指令并和 NEON/FPU 指令交织（interleave），可以提升指令执行的并行度，并继续提升软件的性能。
- NEON 指令执行计算的过程中不得出现条件分支。NEON 指令集中没有分支跳转指令。当需要跳转时，使用 ARM 的跳转指令。在 ARM 处理器中，分支预测技术被广泛使用。但是一旦分支预测失败，那么惩罚就会很高。因此，最好避免使用跳转说明。

4.2　线程并行化

前面描述的方法对于计算的并行化处理大多处于指令级别，通过 SIMD 矢量处理器的数据并行化处理，实现指令级别的并行化（Instruction Level Parallelism, ILP）。除此之外，多线程执行也是实现并行计算的重要手段，称为线程级并行化（Thread Level Parallelism, TLP）。线程级的并行化处理可以对于处理器内核（core）的计算资源的更为有效的利用，多线程的处理可以使得 superscalar① 架构的处理器的执行单元被全部调用，同时通过流水线（pipeline）调度来隐藏访存所需要的延时（memory latency）。而使用线程并行化的同时也存在着一些额外的需求或者开销：首先需要额外的存储满足线程上下文（thread context）的需求，而线程并行化也要求被优化的程序具有在最基础的指令并行之外的并行度；另外一方面，多线程也会对于内存带宽带来额外的压力，更多的线程，意味着每个单独的线程所获得的缓存空间会更小。

当前的绝大多数的基于 ARM 架构的 SoC 都会集成多个 CPU core, 另外 ARM 也在计算密集的场景下针对性的对于处理器添加了 SMT(Simultaneous multithreading) 支持，比如针对于自动驾驶应用优化的 Cortex A76AE 微架构。

在实现计算的多线程并行化的过程中，不仅仅要保证被并行的任务中的子任务之间没有依赖，保证子任务执行的独立性，同时需要避免在多线程执行中可能存在的 False Sharing 和 Cache Line Ping-pong.

① 超标量体系结构是许多处理器中使用的并行计算方法。在超标量计算机中，中央处理单元（CPU）管理多个指令流水线，以在一个时钟周期内同时执行多个指令。这是通过将不同的流水线通过处理器中的多个执行单元来实现的。为了成功实现超标量架构，CPU 的指令获取机制必须智能地检索和委托指令。否则，可能会发生 pipeline stall，从而导致执行单元经常处于空闲状态。

- 当不同的线程具有程序中未共享的数据，但此数据被映射到共享的缓存行 (cache line) 时，就会发生 False Sharing。例如，假设一个程序具有一个整数数组，其中一个线程执行读取和写入具有偶数索引的所有数组条目，而另一个线程执行读取和写入具有奇数索引的条目。在这种情况下，线程实际上将不会共享数据，但是它们将共享高速缓存行，因为每条高速缓存行都将包含奇数和偶数索引值。
- 高速缓存行 ping-pong 是在多个 CPU（或内核）之间快速连续传输高速缓存行的效果。本质上，如果多个 CPU 试图在同一高速缓存行中读取和写入数据，则该高速缓存行可能必须快速连续地在两个线程之间传输，并且这可能会导致性能显着下降（甚至与单线程相比，性能可能更差）。

针对于此，实现多线程处理中需要特别注意以下几点：

- 尽可能向同一个 CPU 执行写入操作；
- 为每个 CPU 指定对应的内存；将线程锁定在对应的 CPU；
- 避免将频繁读写但本身并没有关联的数据放在同一个 Cache Line 中;

因此在实现过程中，这里结合本文中的 Winograd 卷积实现设计的空间局部区域的独立性，在特征的输入变换，GEMM 实现，以及结果的输出变换中，对于整体的图像或者中间特征做了区域性的划分，并在这一层级上实现多线程的并行，保证每个单独的 region 的处理都是独立的，并且向各个处理阶段对应的 buffer 输出处理结果的过程中，各个 region 所对应的地址空间避免处于同一 cache line 的情形。从而保证这一 regional wise 的 Winograd 算法实现在多线程并行下的高效性。

4.3　端到端整数计算支持

在量化计算的量化策略中，本文选择了一种相对简单有效且在实际平台可以具有广泛应用和实现的方案。

矩阵中的每个值均以放射变换的方法（线性变换）通过比例因子和零点值被量化映射到低精度的表示，因此量化域 (这里的实现为 8 为无符号整数) 中的计算可以直接映射到实数域。比例因子和零点值被矩阵中的多个值所共享，即矩阵中的所有行使用相同的比例因子和零点。量化矩阵中的值 x_r 可以通过如下简单的变换方式实现到实数矩阵中的值 x_q 的变换。

$$x_r = a_{scale}(x_q - a_{zero\_point}) \tag{4-1}$$

这里的比例因子（scale factor）a_scale 一般而言是一个浮点值，而参考在[9] 对

于比例因子的处理，按照上面的实数域和量化域的表示，实数域的值 x_r, y_r, z_r 所对应的量化域中的值分别为 x_q, y_q, z_q，三者各自对应的比例因子为 $x_{scale}, y_{scale}, z_{scale}$，零点为 $x_{zero}, y_{zero}, z_{zero}$ 那么对应的实数域的乘法的计算可以表示为

$$z_r = x_r \cdot y_r \tag{4-2}$$

$$z_{scale} \cdot (z_q - z_{zero}) = (x_{scale} \cdot (x_q - x_{zero})) \cdot (y_{scale} \cdot (y_q - y_{zero})) \tag{4-3}$$

$$z_q - z_{zero} = \frac{x_{scale} \cdot y_{scale}}{z_{scale}} (x_q - x_{zero})(y_q - y_{zero}) \tag{4-4}$$

$$z_q = \frac{x_{scale} \cdot y_{scale}}{z_{scale}} (x_q - x_{zero})(y_q - y_{zero}) + z_{zero} \tag{4-5}$$

这里4-2 可见实数乘法所对应的量化值乘法，也具备着同实数值和量化值的映射所类似的线性变换形式，而这里的比例因子 $M = \frac{x_{scale} y_{scale}}{z_{scale}}$，按照[9] 中的经验性统计可以表示为

$$M = 2^{-n} M_0 \tag{4-6}$$

而其中，$M_0 \in [0.5, 1)$ 且 n 为非负整数。M_0 现在很适合表示为定点乘法器（例如，int16 或 int32 取决于硬件）。例如，如果使用 int32，则表示 M_0 的整数是最接近 $2^{31} M_0$ 的 int32 值。由于 $M_0 > 0.5$，因此该值始终至少为 2^{30}，因此始终至少具有 30 位相对精度。因此，与 M_0 的乘法可以实现为定点乘法。同时乘以 2^{-n} 可以有效移位来实现

另外，这里的实现中计算操作的输入和网络的权重均为 8 位无符号整数表示，8 位整数的乘法一般会使用 32 位的累加器，而在 Winograd 卷积实现的将输入由空间域向 winograd 域变换过程中，F（2，3）的 Winograd 算法中基本只涉及加减操作，在这一过程中将输入的无符号 8 位整数转换为 16 位带符号整数做处理，并且在后续的矩阵乘法操作中实现中使用 32 位的累加器，每次乘累加操作中，将 16 位整数乘法的结果 widen 到 32 位再加到 32 位的累加器中。而在一般的基于 GEMM 的量化卷积实现中，累加结束后，会再次通过量化，将 32 位的整数表示转换为 8 位整数的表示，这一过程可以称为再量化（re-quantization ）或者后累加量化（post-accumulation quantization）而 Winograd 卷积的实现中，乘累加操作结束之后，还需要执行输出变换操作，将 Winograd 域中的矩阵乘法结果转换回空间域，此后再执行量化，并将量化之后的 8 位整数输出，从而降低对于内存带宽的压力和缓存占用。

4.4　卷积相关操作的融合

此外，现在通用的卷积网络中，卷积操作之后通常还带有 bias，而在卷积操作之后还通常会紧随着 Batch Normalization 操作和 Relu 激活函数，这一流程构成了卷积网络中常用的一大范式即，Conv（+ Bias）→ BatchNorm → Relu。相比于卷积操作而言添加 Bias，并对输出的特征作 Batch Normalization 和 relu 激活本身从计算复杂度角度而言，都是不值一提的，但是在 ARM 设备的这种嵌入式场景下，将这几个流程作为单独的操作在模型的运行时顺序执行，则是需要额外的存储消耗的（memory footprint）。每次执行单个操作的过程中，都需要从 memory 系统中读入上一操作输出的特征，并且把简单计算操作之后的结果再写入 memory，这同在嵌入式场景计算和存储资源受限的条件下需要最大化计算/访存操作的比例的优化目标是相悖的。而注意到这些后续的操作实际上都是可以转化为区域性（regional wise）inplace 的操作的，特别是 Batch Normalization 在模型训练结束之后，单纯作 inference 的过程中是可以和卷积操作实现融合的。以下简述 Batch Normalization 同卷积的融合的过程。

对于 Batch Norm 的输入特征 x, Batch Norm 本身的参数 γ 和 β, 以及训练过程中对于输入特征的统计量（exponential moving average）μ σ. 操作对应的输出则为 $\tilde{x}$，Batch Norm 操作的形式如下：

$$\tilde{x_i} = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{4-7}$$

而这一操作可以通过 1x1 卷积实现。给定一个输入特征 F，其 data layout 为 $C \times H \times W$, 其对应的 Batch Norm 的结果 $\hat{F}$ 可通过计算每个空间位置 i, j 的如下矩阵-矢量乘法获得：

$$\begin{pmatrix} \hat{F_{1,i,j}} \\ \hat{F_{2,i,j}} \\ \vdots \\ \hat{F_{C,i,j}} \end{pmatrix} = \begin{pmatrix} \frac{\gamma_1}{\sqrt{\sigma_1^2+\epsilon}} & 0 & \cdots & 0 \\ 0 & \frac{\gamma_2}{\sqrt{\sigma_2^2+\epsilon}} & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & \frac{\gamma_C}{\sqrt{\sigma_C^2+\epsilon}} \end{pmatrix} + \begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ \vdots \\ F_{C,i,j} \end{pmatrix} + \begin{pmatrix} \beta_1 - \gamma_1 \frac{\mu_1}{\sqrt{\sigma_1^2+\epsilon}} \\ \beta_2 - \gamma_2 \frac{\mu_2}{\sqrt{\sigma_2^2+\epsilon}} \\ \vdots \\ \beta_C - \gamma_C \frac{\mu_C}{\sqrt{\sigma_C^2+\epsilon}} \end{pmatrix} \tag{4-8}$$

而 Batch Norm 操作往往又紧随卷积操作，于是可以将 Batch Norm 操作融合入卷积操作，将上述的 batch norm 的矩阵表示 表示为 $\hat{F} = W_{bn} \cdot F + b_{bn}$, 其中 $W_{bn} \in R^{CxC}, b_{bn} \in R^C$, 而嵌入 Batch Norm 操作前继的卷积操作之后，这一计算过

表 4.1　ResNet18 网络总体结构

结构名称	输出尺寸	结构
conv1	112x112	7x7, 64, stride 2
block1	56x56	$\begin{bmatrix} 3x3, & 64 \\ 3x3, & 64 \end{bmatrix} x2$
block2	28x28	$\begin{bmatrix} 3x3, & 128 \\ 3x3, & 128 \end{bmatrix} x2$
block3	14x14	$\begin{bmatrix} 3x3, & 256 \\ 3x3, & 256 \end{bmatrix} x2$
block4	7x7	$\begin{bmatrix} 3x3, & 512 \\ 3x3, & 512 \end{bmatrix} x2$

程则为:

$$\hat{f_{i,j}} = W_{bn} \cdot (W_{conv} \cdot f_{i,j} + b_{conv}) + b_{bn} \tag{4-9}$$

于是卷积和 Batch Norm 两个操作可以通过一个权重为 $W = W_{bn} \cdot W_{conv}$, Bias 为 $b = W_{bn} \cdot b_{conv} + b_{bn}$ 的卷积操作来替代。

除此之外，对卷积的添加 Bias 和 relu 激活，则可以在 Winograd 卷积的每个子区域的输出变换过程中，在重量化之前，in-place 的实现添加 Bias，而在重量化之后，通过输出值截断（output clamping），实现诸如 Relu 或者 Relu6 这样的简单的激活函数，从而减少模型运行时对于内存的开销。

4.5　卷积网络加速实验验证

为将本文实现的卷积算法同卷积神经网络推理框架整合集成，并且验证其对于卷积神经网络推理在 ARM Cortex A 设备上加速的有效性，本节针对于在移动端高效实现的卷积网络，在 ARM Cortex A 设备上运行集成了本文局部区域性多通道 Winograd 卷积算法的卷积网络，以下从实验环境，实验设计和结果分析展开介绍。

4.5.1　实验环境与实验设计

这里的硬件环境配置同第三章3 一致。本节实验中，使用 ResNet18 作为验证卷积网络加速效果的模型。这里的对比实现同样为 TensorFlow Lite 中量化卷积的 Im2col 卷积实现。本文将这里实现的快速卷积方法集成入 PyTorch，实验结果由在

表 4.2　ResNet18 网络及子结构加速效果

结构	Im2Col(ms)	本文 (ms)	加速效果
conv1	51.047	31.529	1.619
block1	114.559	65.675	1.744x
block2	95.723	76.505	1.251
block3	86.587	70.223	1.233
block4	95.916	81.382	1.179
整体网络	446.786	326.678	1.368

PyTorch 上运行模型的时间为准。本文实现所作用的有效对象为网络结构中的 3x3，stride 为 1 的卷积。

ResNet18 结构中包含多个 Residual Block，可参见4.1本节将针对于各个卷积 Residual Block 的加速效果和网络整体的加速效果展开实验与分析。

4.5.2　实验结果与分析

表4.2 中显示了在 ResNet18 中的各个卷积子结构的加速效果以及网络整体的加速效果。可见本文提出的局部区域多通道卷积算法对于在实际网络中的卷积计算加速有着显著的提升，同时对于移动端所使用的卷积神经网络在边缘设备上的运行效率提升也具有着明显的意义。

第 5 章　总结与展望

5.1　论文工作总结

本文主要研究整数型 Winograd 卷积在 ARM CPU 上的高效实现，从而加速卷积神经网络在 ARM 上的运行效率。对于智能应用在移动和边缘设备的部署具备着积极的意义。

移动设备上运行卷积网络当前最大的痛点在于卷积网络庞大的计算量与移动设备有限的计算能力之间的矛盾。而同时，尽管模型量化方法已经存在了众多研究，但在支持量化网络模型在移动设备上的高效执行方面，业界依然存在着相当的空白。本文针对上述的难点和痛点，将目前已经的计算复杂度最低的卷积算法针对 ARM 设备硬件特性，实现改造调整，并且结合线性量化策略，实现卷积网络中端到端的整数计算流程。对比于现有的移动端框架对于卷积网络的实现，取得了相对的效率提升。以下是本文完成的主要工作内容：

- Winograd 算法可以在广泛存在移动端和边缘计算设备上的 ARM CPU 上实现计算复杂度的降低和计算功耗的节省。Winograd 卷积运行的过程中有着输入变换和输出变换带来的额外开销，Winograd 卷积主要是加速效果来自于减少了乘法操作，因而这二者带来的开销需要通过 GEMM 计算来做摊销。所以在 Winograd 卷积的运行中可见，卷积的输出通道数越大，卷积的加速效果越明显越接近于 Winograd 卷积加速的理论值。文中针对于 Arm Cortex A 设备，通过空间局部区域多通道处理方法，高效的使用 ARMv8-A NEON SIMD 指令，实现了移动/嵌入式场景下内存有限，计算能力受限的状态下，Winograd 卷积的高效实现。对每个卷积输出的子区域，独立并行计算这一多通道区域的输入变换，矩阵乘法和输出变换。输入变换的 16 个空间位置的结果被 scatter 到 16 组 Winograd 域中的 GEMM 计算中，之后并行执行这 16 组 GEMM，即输入变换和权重变换的结果的乘积，最后从对应的 16 组 GEMM 中 gather 到每个区域对应的结果，执行输出变换，将最后的结果变换回空间域的值并作输出。并且在 data layout，权重预处理，中间结果存储（store）等等方面为此算法的实现和 ARM Cortex A 架构的特征做了设计。
- 结合上面的整数卷积方法实现，本文应用静态线性量化策略，使用整数模拟逼近浮点数计算，实现了在网络中的端到端完全的整数计算。通过引入位移和和加法操作，使得量化模型中的整数值计算同全精度模型中的实数计算相对应。而且引入的额外操作的复杂度相对于网络本身的计算而言可以忽略不

记。最后考虑到卷积网络在移动端的实现，并在卷积的操作中实现了 batch norm，relu，bias 等操作的融合，并入，尽可能减少在存储和计算上不必要的开销，实现卷积网络在移动端 CPU 的执行效率。

5.2　未来工作展望

本文实现了针对于移动端 ARM 设备的整数型 Winograd 快速卷积。一定程度上提高了移动端卷积网络的运行效率。但是，目前的工作仍然有着相当的提升空间。

- 本文中优化的 Winograd 卷积算法为 Winograd 卷积算法中尺度较小的 F（2，3）卷积，该算法理论上的加速上限为 2.25 倍。而 Winograd 算法会随着卷积尺度的增大而加速效果更加明显，同时，其数值精度也会下降，而在模型参数的数值精度比较低的场景下，即量化网络的场景下，在卷积规模变大时，计算的数值稳定性和精度会大幅下降。这里一项重要的原因在于没有找到在卷积尺度变大的场景下适合与低精度计算的 Winograd 算法的形式，即对应的三个变换矩阵。目前主流使用的 Winograd 变换中的数值的范围在卷积尺度变大时明显变大，而量化网络中数值的范围包括中间特征的范围则是非常有限的。所以完全发挥出 Winograd 卷积在量化场景下的加速效果，还需要找到数值范围相对较小的 Winograd 变换的形式，克服这一场景下 Winograd 卷积的数值不稳定性。
- 目前的实现中计算的主要负载由 NEON 实现，但在众多的 ARM 设备中除了 CPU 中的 SIMD 协处理器可以用于实现并行计算，很多设备同时还具有 GPU 支持，而对于移动设备中种类繁多的 GPU 存在着统一可移植的通用接口 - Vulkan。在 ARM 设备上使用 Vulkan API 在 GPU 上执行神经网络模型的推理计算将更有效的实现，同时更充分的利用设备硬件资源。

参考文献

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

[2] Zhang J, Franchetti F, Low T M. High performance zero-memory overhead direct convolutions // ICML. 2018.

[3] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding // MM '14. 2014.

[4] Chellapilla K, Puri S, Simard P Y. High performance convolutional neural networks for document processing // 2006.

[5] Cho M, Brand D. Mec: Memory-efficient convolution for deep neural network // ICML. 2017.

[6] Mathieu M, Henaff M, LeCun Y. Fast training of convolutional networks through ffts. CoRR, 2013, abs/1312.5851.

[7] Zlateski A, Jia Z, Li K, et al. Fft convolutions are faster than winograd on modern cpus, here is why. ArXiv, 2018, abs/1809.07851.

[8] Lavin A, Gray S C. Fast algorithms for convolutional neural networks. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015:4013-4021.

[9] Jacob B, Kligys S, Chen B, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2017:2704-2713.

[10] Louizos C, Reisser M, Blankevoort T, et al. Relaxed quantization for discretized neural networks. ArXiv, 2019, abs/1810.01875.

[11] Heinecke A, Henry G, Hutchinson M, et al. Libxsmm: Accelerating small matrix multiplications by runtime code generation. SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016:981-991.

[12] Dukhan M. The indirect convolution algorithm. ArXiv, 2019, abs/1907.02129.

[13] Van Zee F G, van de Geijn R A. BLIS: A framework for rapidly instantiating BLAS functionality [J/OL]. ACM Transactions on Mathematical Software, 2015, 41(3):14:1-14:33. http://doi.acm.org/10.1145/2764454.

[14] Low T M, Igual F D, Smith T M, et al. Analytical modeling is enough for high-performance blis. ACM Trans. Math. Softw., 2016, 43:12:1-12:18.

[15] Goto K, Guney M E, Knepper S. Reducing packing overhead in matrix-matrix multiplication [EB/OL]. https://techdecoded.intel.io/resources/reducing-packing-overhead-in-matrix-matrix-multiplication/.

[16] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016:770-778.

致　谢

感谢丁贵广老师多年来对于我的研究学习生活的指导，关照和支持。老师治学严谨，工作勤勉，待人谦逊和蔼，有师长之风，言传身教，使我受益良多。

感谢 417 实验室的伙伴们，有彼此相互扶持，共同学习和生活的时光，使我的研究生生涯充实且欢乐。

感谢中国科学院自动化所的赵天理博士对于本工作给予的指点和支持，感谢 ARM 的工程师 Pablo Marquez，Gian Marco Iodice 以及 Andrew Mundy 对于本文工作中问题的解答。

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

1994 年 9 月 15 日出生于甘肃省天祝藏族自治县。

2012 年 9 月考入清华大学工程物理系工程物理专业，2014 年转入清华大学软件学院软件工程专业，2017 年 7 月本科毕业并获得工学学士学位。

2017 年 9 月免试进入清华大学软件学院攻读工程硕士学位至今。