面向现代GPU的Winograd卷积加速研究

童 敢[1],黄立波[1],吕雅帅[2]
(1. 国防科技大学计算机学院,湖南长沙 410073;2. 华为技术有限公司,北京 100031)

摘 要: 卷积运算是现代卷积神经网络中必不可少的组成部分,同时也是最耗时的. 为了解决卷积算子的性能问题,包括快速傅里叶变换(Fast Fourier Transform,FFT)和Winograd在内的快速卷积算法被提出. Winograd卷积可被用于提高小卷积核的推理性能,是目前卷积神经网络中的主流实现方法. 然而,Winograd卷积在许多高度优化的深度神经网络库和深度学习编译器中的实现比较低效. 由于Winograd卷积的四个阶段的复杂数据依赖关系,面向GPU对其进行优化非常具有挑战性. 本文针对现代GPU体系结构优化了Winograd卷积算子的性能. 本文提出了Winograd计算阶段的等价变化及其利用Tensor Core进行计算的无同步实现,并进一步提出了利用不同GPU内存层级的部分计算核融合方法PKF(Partial Kernel Fusion). 基于张量虚拟机(Tensor Virtual Machine,TVM)和代码重构器PKF-Reconstructor(Partial Kernel Fusion Reconstructor),实现了高性能的Winograd卷积. 对真实应用中卷积神经网络的卷积算子的评估表明,与cuDNN相比,本文所提算法实现了7.58~13.69倍的性能提升.

关键词: Winograd卷积;低精度;部分计算核融合;卷积加速;GPU内存层级;Tensor Core
基金项目: 国家自然科学基金(No.61872374)
中图分类号: TP183 **文献标识码:** A **文章编号:** 0372-2112(2024)01-0244-14
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20211641

Research on Winograd Convolution Acceleration for Modern GPU

TONG Gan[1], HUANG Li-bo[1], LYU Ya-shuai[2]
(1. *College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan* 410073, *China*;
2. *Huawei Technologies Corporation Limited, Beijing* 100031, *China*)

Abstract: Convolution operation is an indispensable part of modern convolutional neural networks, and it is also the most time-consuming. In order to solve the performance problem of convolution operators, fast convolution algorithms including FFT (Fast Fourier Transform) and Winograd have been proposed. Winograd convolution is used to improve the inference performance of small convolution kernels and is currently the mainstream implementation method in convolutional neural networks. However, the implementation of Winograd convolution in many highly optimized deep neural network libraries and deep learning compilers is relatively inefficient. Due to the complex data dependence of the four stages of Winograd convolution, it is very challenging to optimize it for GPU. In this paper, the performance of the Winograd convolution operator is optimized for modern GPU architecture. This paper proposes the equivalent transformation of the Winograd calculation stage and its non-synchronization implementation using Tensor Core, and further proposes a partial kernel fusion method utilizing different GPU memory hierarchies, i.e. PKF (Partial Kernel Fusion). Based on TVM (Tensor Virtual Machine) and a code reconstructor named PKF-Reconstructor (Partial Kernel Fusion Reconstructor), a high-performance Winograd convolution is implemented. The evaluation of the convolution operators from real-world convolutional neural networks shows that the proposed algorithm achieves a performance improvement of 7.58~13.69 times compared with cuDNN.

Key words: Winograd convolution; low precision; partial kernel fusion; accelerating convolution; GPU memory hierarchy; Tensor Core
Foundation Item(s): National Natural Science Foundation of China (No.61872374)

收稿日期:2021-12-09;修回日期:2022-11-03;责任编辑:王天慧

1 引言

深度学习是近年来机器学习领域新兴的研究方向,在诸多领域取得了令人惊艳的效果. 其中基于卷积运算的卷积神经网络(Convolutional Neural Network, CNN)是深度学习中的典型网络之一,通过多层卷积算子构造深层网络模型,利用学习训练建立模型的参数,可以得到逼近现实的输入输出关系. 而更深的神经网络也意味着更多的参数和更大的计算量,带来了更大的时延和更高的成本.

卷积运算可以转换为一般矩阵乘法(GEneral Matrices Multiplication, GEMM),并在各类硬件平台上调用更高效的线性运算库来加速运行,但此类方法存在冗余的乘法计算. 为了达到实时的人工智能(Artificial Intelligence, AI)应用体验,降低训练、推理成本,研究者提出了包括快速卷积算法在内的诸多优化方案. 快速卷积算法是卷积运算的一类等价算法,通过对卷积运算进行数学等价变化,将输入线性变换到另一个空间,以减少计算中的乘法次数. 2015年提出的基于Winograd最小滤波算法的Winograd卷积就是典型的快速卷积算法,它以加法次数的增加为代价换取乘法数量的减少[1].

在引入Winograd卷积之后,大量研究工作将Winograd卷积拓展到了更一般的卷积算子上,并在CPU, GPU和FPGA等平台完成了实现. 但截至目前,供应商提供的神经网络计算库中,Winograd卷积的实现均很低效. 也有部分工作对特定平台上Winograd卷积的性能进行了优化,但面向NVIDIA GPU的Winograd卷积优化工作均未利用Tensor Core的高性能,且相关优化方法存在两方面的问题:一是基于对底层汇编器的修改,难以整合到流行的深度学习计算库中因此不便使用;二是利用了部分规范中未明确的GPU特性,不具备对新GPU架构的兼容性. 本文旨在提出一种高级编程语言层面的、使用Tensor Core的高效Winograd卷积,以加速神经网络中卷积算子在GPU上的推理性能. 这也是首个利用Tensor Core实现低精度Winograd卷积推理的工作.

然而由于Winograd卷积的内在复杂性,在GPU这类非定制硬件平台上进行优化存在困难. Winograd卷积由四个分离的阶段组成. 一方面,这四个阶段的访存模式不尽相同,因此难以利用GPU的多级内存层级;另一方面,计算阶段的运算为小规模矩阵的对应位相乘,无法充分利用GPU的并行处理能力和Tensor Core这类高效运算部件. 同时,现有相关工作尝试了对Winograd卷积进行阶段的融合,但也均存在实现困难或可移植性差的情况. 总之,面向现代GPU优化Winograd卷积存在一定的困难,但对GPU上的卷积神经网络推理加速有重大意义.

针对上述问题,本文首先引入低精度,在降低模型大小的同时利用Tensor Core的强大计算能力,并提出了一种避免了线程同步的算法;其次,创新地提出新的融合策略,利用GPU的多层内存层级来融合Winograd卷积的部分阶段;最后对设计空间进行了探索,分析和对比了算法复杂性,并基于一个流行的端到端深度学习框架TVM和代码重构技术实现了优化的Winograd卷积. 通过实验验证,本文提出的方法与NVIDIA cuDNN中的Winograd卷积实现相比实现了7.58~13.69倍的显著性能提升.

2 Winograd卷积优化相关工作

自从Winograd卷积引入以来,国内外研究者做了很多优化和实现工作. 牺牲精度换取更少的内存占用和更高的运算效率是CNN优化中的常见方法. 将CNN模型的参数从32位浮点数转变为16位浮点数,或量化为8-bit定点数甚至更低精度,可以在基本不损失模型精度的情况下压缩模型并提升运算效率. 最早研究者在引入Winograd卷积时就同时测试了单精度和半精度浮点数[1],但实验表明使用半精度浮点数会导致较大的绝对误差. Meng等人[2]提出由均匀的仿射量化生成量化的卷积核,并以8位无符号整数和动态范围表示. Gong等人[3]提出动态在CNN上分层应用不同的卷积实现和量化以降低计算复杂度,其中也包括Winograd卷积的量化. Fernandez等人[4]提出应用Winograd卷积到8-bit网络上,并用学习解决精度损失问题. Liu等人[5]在引入RNS(Residual Number System,余数系统)变换的同时也使量化可以在低精度上操作. Zhang等人[6]提出对精度损失建模,为特征映射和卷积核使用不同的量化级别. Barabasz等人[7]用勒让德基多项式取代Winograd变换中的规范基多项式,提出基于基变技术的量化. Li等人[8]将线性量化直接嵌入Winograd域以实现低精度量化. Han等人[9]进一步探索了用Winograd算法优化4~6位精度的卷积核. Sabir等人[10]在特征映射切片上应用量化,应用粒子群优化技术找到量化的阈值[10]. 此外,还有一些研究者也在Winograd卷积上应用了8-bit量化技术[11~14]. 这些研究证明了低精度与量化技术在Winograd卷积上的可行性,但结合Tensor Core的低精度Winograd卷积推理还没有相关研究. NVIDIA cuDNN神经计算库中的Winograd卷积实现也没有支持Tensor Core,因此本文提出的利用Tensor Core的低精度Winograd卷积推理尚属首次.

Huang等人[15]和Jiang等人[16]分别将大尺寸卷积核和非单位步长的卷积分解为小卷积核以解决数值精度问题. 这些研究证明了Winograd卷积在更大尺寸卷积核上的适应性,Vincent等人[17]还证明了在不修改

Winograd卷积算法的前提下，仅改变变换矩阵的元素值就可以提高数值稳定性，进一步支撑了低精度与量化的可行性. Hong等人[18]在大规模GPU集群上利用Winograd卷积的数据并行性和切片内并行性实现了多维并行训练，他们的思路一定程度上启发了本文的研究. Jia等人[19]利用MegaKernel技术将Winograd卷积的四个阶段融合，利用精心设计的任务映射算法在GPU上达成了显著的性能提升. 这四个阶段的融合并非一般意义上的算子融合，而是将每个阶段视为任务流的一个节点，本质上是更高层次的计算图调度算法. Yan等人[20]使用SASS汇编器优化Winograd卷积，合并了全局访存并使共享访存无冲突，利用缓存设计流水线、提高计算强度，还利用常规寄存器填补了谓词寄存器不足的缺陷. 他们的工作一方面基于对底层汇编器的修改，难以整合到流行的深度学习计算库中，因此不便使用；另一方面利用了部分规范中未明确的GPU特性，不具备对新GPU架构的兼容性. 而本文提出的融合技术和低精度与量化技术具备更好的通用性，通过对设计空间的分析也确保了算法对未来GPU的兼容，还具备向其他类型硬件设备推广的理论可行性.

3 相关基础

3.1 Winograd卷积

Winograd[21]在1980年提出了有限脉冲响应(Finite Impulse Response，FIR)的最小滤波算法，该算法给出了卷积运算最少需要的乘法数量. 对于由r拍的FIR滤波器生成的m个输出，即F(m,r)，需要的最少乘法数量为

$$\mu\left(\mathrm{F}\left(m,r\right)\right)=m+r-1$$

对于F(2,3)的情况，所需的最小乘法数为4，与直接卷积中的6次乘法相比减少了2次. 可以用矩阵形式表示Winograd最小滤波算法：

$$\boldsymbol{Y}=\boldsymbol{A}^{\mathrm{T}}\left[\left(\boldsymbol{Gg}\right)\odot\left(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}\right)\right]$$

其中，$\boldsymbol{g}$为滤波器向量，$\boldsymbol{d}$为输入数据向量，$\boldsymbol{Y}$为输出数据向量，$\boldsymbol{G}$表示滤波器变换矩阵，$\boldsymbol{B}^{\mathrm{T}}$表示数据变换矩阵，$\odot$表示矩阵的对应位乘法(Hadamard积)运算，$\boldsymbol{A}^{\mathrm{T}}$表示输出变换矩阵. 对于特定的$m$和$r$，可以通过中国剩余定理计算出相应的变换矩阵$\boldsymbol{G}$和$\boldsymbol{B}$. Lavin等人[1]首先将Winograd最小滤波算法应用在CNN中，利用减少的乘法次数提升卷积算子性能. 他们通过嵌套一维最小滤波算法F(m,r)，得到了二维的最小滤波算法F($m\times m$,$r\times r$)：

$$\boldsymbol{Y}=\boldsymbol{A}^{\mathrm{T}}\left[\left(\boldsymbol{GgG}^{\mathrm{T}}\right)\odot\left(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{dB}\right)\right]\boldsymbol{A}$$

此时，滤波器$\boldsymbol{g}$大小为$r\times r$，输出$\boldsymbol{Y}$大小为$m\times m$，则输入$\boldsymbol{d}$大小为$(m+r-1)\times(m+r-1)$. 二维最小滤波算法所需乘法数为$(m+r-1)^2$，而原始卷积算法需要乘法数为$m\times m\times r\times r$. 对于F($2\times2$,$3\times3$)而言，乘法次数从36降低到了16，减少了55.56%，即使将额外带来的加法运算考虑在内也有可观性能收益.

可以自然地将Winograd卷积分为如下四个分离的阶段(如图1所示)：

(1) 输入变换(Input Transformation，ITrans)：$\boldsymbol{g}'=\boldsymbol{GgG}^{\mathrm{T}}$，将输入张量变换到Winograd域，$\boldsymbol{g}'$的尺寸为$(m+r-1)\times(m+r-1)$.

(2) 卷积核变换(Kernel Transformation，KTrans)：$\boldsymbol{d}'=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{dB}$，将卷积核变换到Winograd域，$\boldsymbol{d}'$的尺寸与$\boldsymbol{g}'$的尺寸相同.

(3) 对应位相乘(Element-wise Matrix Multiplication，EWMM)：$\boldsymbol{Y}'=\boldsymbol{g}'\odot\boldsymbol{d}'$，是Winograd卷积的计算阶段，$\boldsymbol{Y}'$的尺寸与$\boldsymbol{g}'$的尺寸相同.

(4) 输出变换(Output Transformation，OTrans)：$\boldsymbol{Y}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{Y}'\boldsymbol{A}$，将结果从Winograd域逆变换到特征向量域，$\boldsymbol{Y}$的尺寸为$m\times m$.

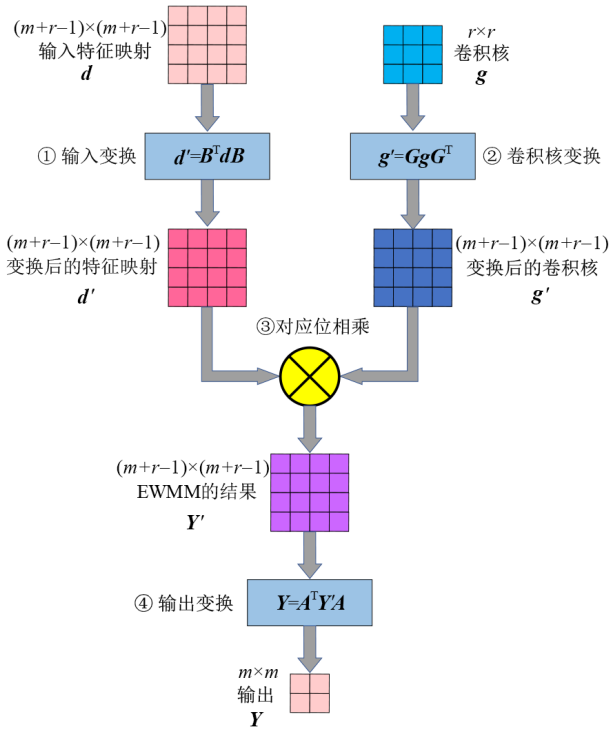
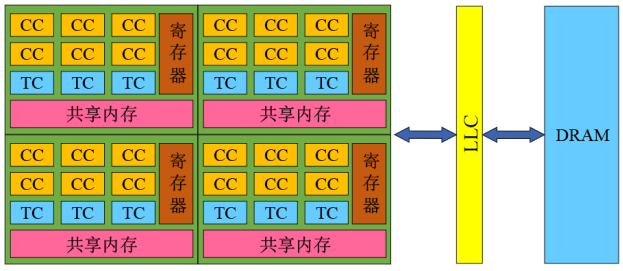


图1 Winograd卷积的四个阶段

3.2 现代GPU体系结构

现代GPU体系结构如图2所示. 一块GPU包含若干个流处理器(Streaming Multiprocessor，SM)，而每个SM由包括CUDA Core和Tensor Core在内的大量计算核心、大容量的寄存器组和一个共享内存(Shared Memory，也称L1 Cache)组成. NVIDIA GPU从Volta架构开始引入Tensor Core，从一般意义上来说，一个Tensor Core单元可以在一个时钟周期内完成一个小规模矩

阵乘法的计算,而此前的CUDA Core计算单元每个时钟周期仅能完成一个融合浮点乘加运算(Fused Multiply-Add,FMA),这是革命性的提升.然而使用Tensor Core需要操作数满足一定的条件,更需要合适的工作负载到线程的映射,这是本文需要解决的第一个问题.L2 Cache(LLC)和高带宽的动态随机访问内存(Dynamic Random-Access Memory,DRAM)由所有的SM共享.距离计算核心更近的内存层级具备更小的内存但同时也具备更高的带宽,反之离DRAM越近的内存层级具备更大的容量但带宽是有限的.利用好数据局部性可以高效利用这种多级内存层级,进而显著改善性能、如何利用好内存层级实现阶段的融合是本文需要解决的另一个问题.



(TC表示Tensor Core,CC表示CUDA Core)

图2　简化的典型GPU体系结构

3.3　循环融合与算子融合技术

在深度学习领域,算子融合[22,23]指的是将多个算子组合起来成为一个融合的算子.与传统编译中的循环融合[24]不同,算子融合中的数据依赖关系由其计算图和算子语义隐式给出.但不管是算子融合还是循环融合,两者都是尝试通过将计算合并到一起,以减少从内存中读写中间计算结果的开销.由于多级内存层级中,不同层级的内存带宽存在很大的差异,因此这种开销是显著的.Winograd卷积天然地由四个阶段组成,在GPU实现上也一般为4个分离的kernel,可以考虑将这些kernel融合起来.但在数学上,没有已知方法可以将Winograd卷积中的运算变换为单个运算,因此无法利用传统的循环融合和算子融合技术.但Winograd卷积的四个阶段存在数据相关性,故存在利用数据局部性的可能性.CPU上已经有研究利用L3 Cache重用卷积核来加速Winograd卷积,而在GPU上还没有相关的探索[25].因此,可以考虑利用GPU中的缓存暂存每个阶段的中间运算结果,在进入下一阶段时直接从高速缓存中获取数据以减少数据流开销.

4　面向现代GPU加速Winograd卷积

鉴于对相关工作的介绍和对现代GPU体系结构以及融合技术的分析,本文提出一种面向现代GPU的Winograd卷积加速方法,该方法由4部分组成.

首先是面向Tensor Core的低精度Winograd.现有一些研究探索了Winograd卷积上的低精度技术,并在FPGA等平台上实现.在GPU上,低精度Winograd卷积虽存在相关研究,但其实现中没有利用高效的Tensor Core,因此可以进一步优化.本文提出了EWMM阶段的一种等价形式,以提高算术强度和该阶段的可并行性,并提出应用低精度技术,使该等价形式可以利用Tensor Core进行加速,在此基础上,本文提出的调度还避免了运算过程中读取数据和计算之间的线程同步指令.

然后是提出了部分计算核融合技术(Partial Kernel Fusion,PKF).算子融合通常应用在不同的算子上,比如CNN中典型的"卷积-批归一化-激活函数"结构就可以通过算子融合技术融合为单个算子.但Winograd卷积有其特殊性,由于Winograd变换和逆变换的存在,Winograd卷积天然地包含四个分离的部分.这四个部分存在数据依赖关系,在GPU实现上表现为四个独立的计算核(kernel),可以看作四个算子.但Winograd变换与逆变换互为不可融合的类型,因此无法融合全部kernel为单个kernel,仅可以融合包括EWMM阶段在内的两个kernel.而EWMM阶段存在算术强度过低的情况,且无法利用Tensor Core进行计算.本文针对这些难点,提出利用GPU的多级内存层级,用两种新的融合策略及其实现来融合不可融合的算子,并针对GPU上的Winograd卷积提出了利用新融合策略的PKF技术.

接着是优化策略的设计空间探索.应用低精度技术和PKF技术,可以拓展Winograd实现的设计空间,同时也对设计空间产生了约束.低精度技术的应用使每个时钟周期可以读取更多的操作数和执行更多的运算,这也对工作负载到线程的分配映射提出了要求.PKF技术旨在更好地利用多级内存层级,因此受限于两级缓存的大小.通过对应用这两种技术的Winograd卷积进行了全面分析,本文探索了应用低精度技术和PKF技术后的Winograd卷积的设计空间并对比了算法复杂性.

最后是优化的Winograd卷积算子实现.基于流行的深度学习编译软件栈(Tensor Virtual Machine,TVM),可以将算子的计算与调度分开独立实现.通过TVM中的调度原语,可以实现设计空间中的方案,并利用Intrinsic映射调用Tensor Core.但由于TVM的对可融合算子的类型限制,PKF技术无法在TVM中直接实现.本文研究了以最小代价突破这种限制的方法,将PKF技术的应用与算子基础代码生成分离,实现了可在TVM生成的代码上应用PKF技术的代码重构器,最终实现了应用了本文优化的Winograd卷积算子.

4.1　面向Tensor Core的低精度Winograd算法

Winograd卷积的计算集中在EWMM阶段,执行的

运算是对应位的矩阵乘法，即 Hadamard 积. EWMM 阶段的输入是变换后的输入和变换后的卷积核，在不考虑通道数的情况下，二者均为 $(m+r-1)\times(m+r-1)$ 的矩阵. 因此，这一阶段需要读取 $2(m+r-1)^2$ 个元素并执行 $(m+r-1)^2$ 次乘法操作，最后将 $(m+r-1)^2$ 个元素的运算结果写回内存.

可以用算术强度（Arithmetic Intensity）这一概念来评估算法对内存带宽的需求[26]，其定义为算法的操作数与访存字节数的比例，即平均每读取一个字节的数据可以支持的运算次数. 运算强度越大，则表示算法对内存带宽的需求越小；反之内存带宽更容易成为算法性能的瓶颈. 假设 Winograd 卷积的数据类型为单精度浮点数，那么根据算术强度的定义，EWMM 阶段的算术强度 $\mathrm{AI}_{\mathrm{EWMM}}$ 为

$$\mathrm{AI}_{\mathrm{EWMM}}=\frac{(m+r-1)^2\mathrm{FLOPS}}{[2(m+r-1)^2+(m+r-1)^2]\times 4\mathrm{B}}=\frac{1}{12}\mathrm{FLOPS/B}$$

此值远低于 GPU 的运算带宽比（ops：byte ratio，以 NVIDIA Volta V100 为例，其运算带宽比为 40~139 FLOPS/B），因此是典型的内存限制型（Memory Limited）运算. 这类运算一方面受限于 GPU 的内存带宽，另一方面无法以充足的计算量在流水线中隐藏访存延迟. 若要提高 Winograd 卷积 EWMM 阶段的性能，首先需要提高计算强度. 提高算术强度的方法之一是对算法本身作调整，另一个方法是使用低精度类型以节约带宽. 另外，考虑利用 Tensor Core 的算力，则也需要使算法匹配 Tensor Core 支持的操作.

4.1.1 EWMM 阶段的变化

完整的二维 Winograd 卷积中，计算阶段并非简单的二维矩阵上的 EWMM，还需要在 C 维度上规约累加，如算法 1 所示（算法中张量均为 Winograd 域张量）.

算法 1 原始 EWMM 阶段处理单批次 Winograd 域输入张量的算法

输入：输入张量 $\boldsymbol{I}(\mathrm{ntiles}, C, H_{\mathrm{tile}}, W_{\mathrm{tile}})$ 和卷积核 $\boldsymbol{F}(K, C, H_{\mathrm{tile}}, W_{\mathrm{tile}})$

输出：输出张量 $\boldsymbol{O}(\mathrm{ntiles}, K, H_{\mathrm{tile}}, W_{\mathrm{tile}})$

```
FOR n=0→ntiles
  FOR k=0→K
    // C 维度上的规约累加
    FOR c=0→C
      // HW 维度上的 EWMM
      FOR h=0→H_tile
        FOR w=0→W_tile
          O[n,k,h,w] += I[n,c,h,w]*F[k,c,h,w]
RETURN O
```

这种乘累加的运算模式在形式上与 GEMM（GEneral Matrices Multiplication）是一致的：矩阵 $\boldsymbol{A}_{m\times k}$ 和矩阵 $\boldsymbol{B}_{k\times n}$ 的 GEMM 就是将 k 维度上的所有元素按位相乘后累加. 因此，仅仅通过变换数据布局和交换循环顺序，就可以将原始的 EWMM 后规约累加转变为算术强度更高的 BGEMM（Batched GEMM，批量矩阵乘），如算法 2 所示.

算法 2 与原始 EWMM 阶段等价的 BGEMM 算法

输入：输入张量 $\boldsymbol{I}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{ntiles}, C)$ 和卷积核 $\boldsymbol{F}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, K, C)$

输出：输出张量 $\boldsymbol{O}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{ntiles}, K)$

```
// 内层的矩阵乘法需要执行 H_tile*W_tile 个批次
FOR h,w=0→H_tile,W_tile
  // 等价于 A_{ntiles×C} 和 B_{C×K} 的矩阵乘法
  FOR n=0→ntiles
    FOR k=0→K
      FOR c=0→C
        O[h,w,n,k] += I[h,w,n,c]*F[h,w,k,c]
RETURN O
```

经过算法调整后，原 EWMM 阶段的 EWMM 后规约累加运算变成了 BGEMM 运算，记为 BGEMM 阶段. 根据算法 2 中的描述可知，BGEMM 阶段矩阵乘法的批次为 $H_{\mathrm{tile}}W_{\mathrm{tile}}$，对应的矩阵乘法规模“$M$”“$N$”“$K$”分别为 ntiles，$K$ 和 C. 现对 BGEMM 阶段的算术强度重新分析：考虑单批次的矩阵乘法，输入矩阵大小分别为 (ntiles, C) 和 (K, C)，完成单批次矩阵乘需要读取 $\mathrm{ntiles}\times C+K\times C$ 个元素，需要 $\mathrm{ntiles}\times C\times K$ 次浮点乘法运算和 $\mathrm{ntiles}\times C\times K$ 次浮点加法运算，再将 (ntiles, K) 大小的计算结果写回内存需要操作 $\mathrm{ntiles}\times K$ 个元素，因此 BGEMM 阶段的算术强度 $\mathrm{AI}_{\mathrm{BGEMM}}$ 为

$$\mathrm{AI}_{\mathrm{BGEMM}}=\frac{(\mathrm{ntiles}\times C\times K+\mathrm{ntiles}\times C\times K)\mathrm{FLOPS}}{(\mathrm{ntiles}\times C+K\times C+\mathrm{ntiles}\times K)\times 4\mathrm{B}}$$

$$=\frac{\mathrm{ntiles}\times C\times K}{2(\mathrm{ntiles}\times C+K\times C+\mathrm{ntiles}\times K)}\mathrm{FLOPS/B}$$

不妨带入典型卷积算子实例，令卷积算子 $C=K=64$，$R=S=3$，输入为 $H=W=224$ 的特征映射，Winograd 卷积中的 m 取常用的 4. 此时 $H_{\mathrm{tile}}=W_{\mathrm{tile}}=6$，ntiles=3 136，带入可得到新的算术强度 $\mathrm{AI}_{\mathrm{BGEMM}}$ 约为 15.84，较之前的 $\mathrm{AI}_{\mathrm{EWMM}}=1/12$ 有近两百倍的提升. 同时，转换后得到的批量矩阵乘法也为使用 Tensor Core 创造了可能性. 但调整后的算法仍然有改进的空间. 一方面算术强度还不够大，另一方面 32 位精度在 Ampere 架构之前的 Tensor Core 上也不受支持. 这两方面的问题都可以通过使用低精度数来解决.

4.1.2 到 Tensor Core 的映射

Volta 以来的三种架构中的 Tensor Core 对数据类型有不同的要求，如表 1 所示.

可以看到，三种架构的 GPU 都支持 FP16 数据类型，而从 Turing 架构开始又引入了对 INT8 精度的支持. 为保持对早期版本的兼容性，本文选择 FP16 类型作为首选数据类型. 另外，INT8 精度 Winograd 卷积的可行

表 1　不同架构Tensor Core上支持的数据类型

架构	Tensor Core支持的数据类型
NVIDIA Volta	FP16
NVIDIA Turing	FP16, INT8, INT4, INT1
NVIDIA Ampere	FP64, TF32, bfloat16, FP16, INT8, INT4, INT1

性已得到验证,但未在GPU上利用Tensor Core付诸实施,因此本文将INT8数据类型作为研究目标尚属首次. 使用Tensor Core需要解决以下两个问题:一是使用Tensor Core对Winograd卷积参数有何要求? 二是如何分配任务以在使用Tensor Core的同时最大化并行性?

通过算法2提出的变换,得到了规模"M""N""K"分别为ntiles,K和C的矩阵乘法. 对于FP16和INT8,通过WMMA API调用Tensor Core允许三种$M\times N\times K$的矩阵规模:16×16×16, 32×8×16和8×16×32. 可以看出,WMMA的"K"维度必须为16,对应BGEMM阶段的C. 因此,要完美使用Tensor Core加速Winograd卷积,卷积的输入通道数必须为16的倍数. 同理,ntiles和K也必须为8的倍数,且二者之积需要能被256整除. 对流行的CNN网络架构进行统计发现,卷积参数大多满足以上要求.

在任务分配方面,相应的调度和映射算法如图3所示,记作BGEMM-TC算法.

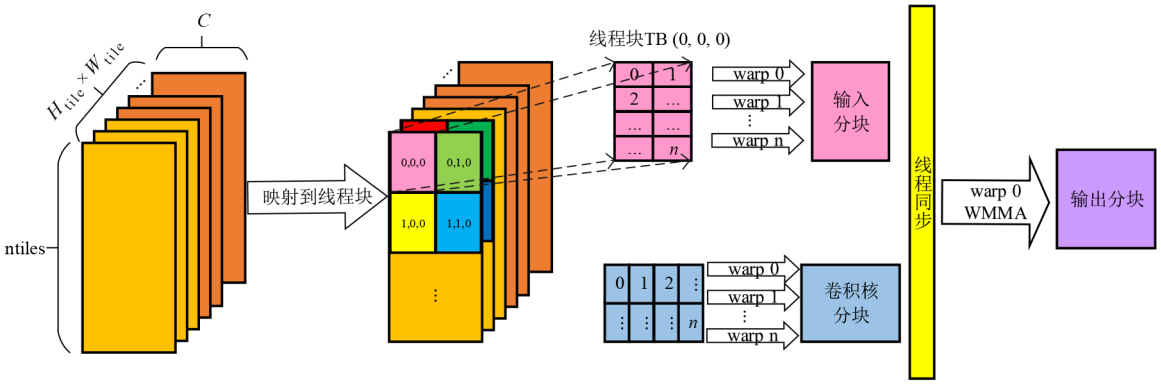


图3　面向Tensor Core的低精度Winograd卷积任务映射(BGEMM-TC)算法

首先将BGEMM任务映射到线程块上,对输出张量进行划分以根据数据依赖关系溯源到相关的输入张量. 对于输出张量,可以视为$H_{tile}\times W_{tile}$个批次(ntiles, K)的矩阵. 将HW维度映射到线程块的一个维度上的索引,并在ntiles维度和C维度上进行划分. 对应到GEMM的输入,线程块会分别在输入张量的ntiles维度和C维度和卷积核张量的K维度和C维度上迭代,即每个线程块在一次迭代中负责$TB_M\times TB_N\times TB_K$规模的矩阵乘法. 接着,将线程块一次迭代中的任务映射到线程块内的warp上. 使线程块内所有的warp协同完成矩阵乘法输入数据的访问,在完成输入张量切片和卷积核张量切片的访问后,执行WMMA操作. 所有的Tensor Core相关指令都是warp级别的指令,这意味着一个warp内的所有线程应该同时执行相同的指令. 在完成数据访存后,每个warp内的线程将协同执行WMMA指令来计算其中的一个子矩阵.

4.1.3　同步指令的消除

将一个线程块负责的(TB_M, TB_N)大小的输出矩阵划分为($WMMA_M$, $WMMA_N$)大小的分块,由每个warp负责计算其中一个分块. 那么每个warp使用的数据为输入分块中高度为$WMMA_M$的一行和卷积核分块中宽度为$WMMA_N$的一列,所以warp之间使用的数据存在重叠. 也就是说,由一个warp读取的数据会作为其他warp的操作数,这必然导致warp读取的数据和计算时使用的数据不一致的情况. 因此,为了避免重复的数据访问,需要在访存阶段将线程块内warp需要的所有数据加载完毕,warp才可以开始计算,这也就引入了上述算法中的线程同步指令. 为了消除这种同步开销,本节提出了一种无需线程同步的改进算法,图4和算法3详细描述了该算法,记为Sync-free BGEMM-TC算法.

首先,BGEMM的批次维度不再直接映射到线程块索引. 仍然是沿着ntiles维度和K维度对工作负载进行切分,并将分块映射到不同的线程块. 但与BGEMM-TC算法不同的是,H_{tile}维度将直接映射到循环迭代,而W_{tile}维度则映射到一个线程块内的warp索引. 此时,每个线程块负责处理$H_{tile}W_{tile}$批次的子矩阵的计算,而每个线程块由W_{tile}个warp组成,即这些warp相互独立地处理不同批次的子矩阵,与线程块内的其他warp相互独立. 这样一来,每个warp在读取据后就可以立即执行WMMA指令,而无需等待其他warp到达同步点.

4.2　面向GPU内存层级的部分kernel融合

4.2.1　Winograd阶段融合分析

分析Winograd卷积中的kernel能否融合,首先需要

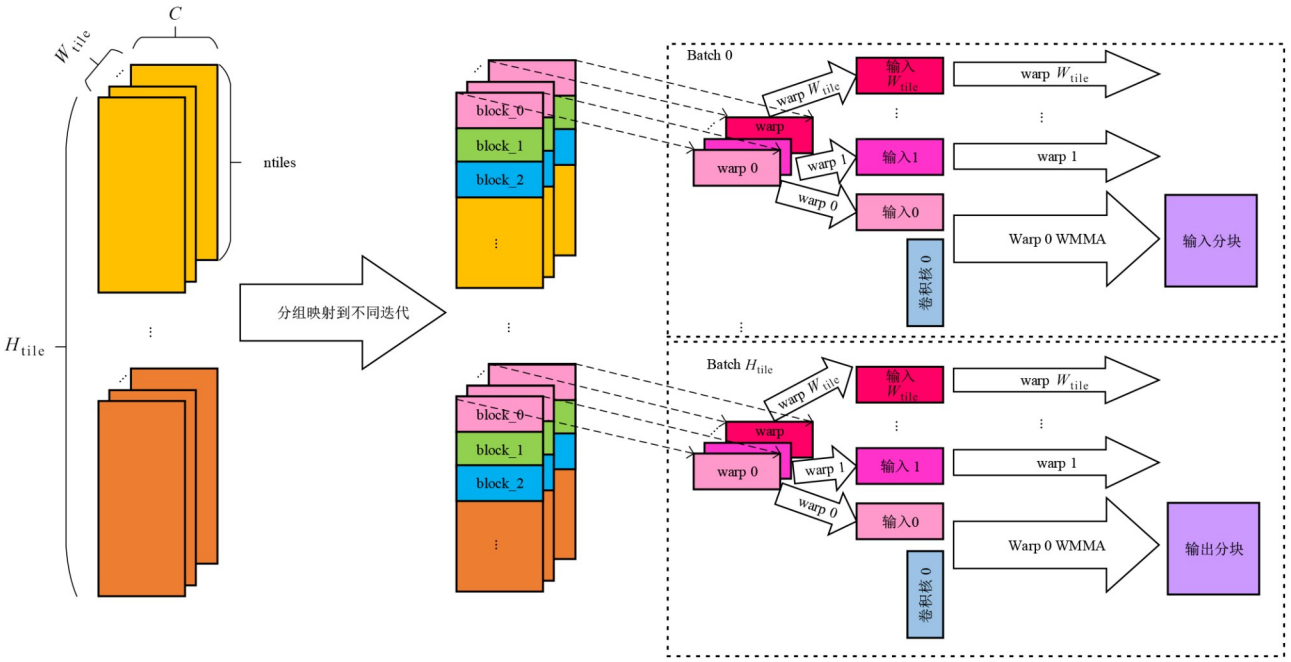


图4 无需额外线程同步指令的Sync-free BGEMM-TC算法

算法3 无需额外线程同步指令的Sync-free BGEMM-TC算法

输入: 输入张量 $\boldsymbol{I}(H_{\text{tile}}, W_{\text{tile}}, \text{ntiles}, C)$ 和卷积核 $\boldsymbol{F}(H_{\text{tile}}, W_{\text{tile}}, K, C)$

输出: 输出张量 $\boldsymbol{O}(H_{\text{tile}}, W_{\text{tile}}, \text{ntiles}, K)$

// 划分一个批次的GEMM为 $\text{TB}_M \times \text{TB}_N \times \text{TB}_K$ 规模的矩阵乘法

// 映射到线程块的 xy 索引维度

FOR $n_{\text{outer}}, k_{\text{outer}} = 0 \rightarrow \text{ntiles}/\text{TB}_M, K/\text{TB}_N$

  // 作为线程块的迭代维度

  FOR $c_{\text{outer}} = 0 \rightarrow C/\text{TB}_K$

    // 作为warp的迭代维度

    FOR $h = 0 \rightarrow H_{\text{tile}}$

      // 映射到warp的索引维度

      FOR warp_id $= 0 \rightarrow W_{\text{tile}}$

        // 读取此warp需要的所有数据

        WARP_LOAD(warp_id)

        // 由于无需等待其他warp因此可以立刻执行

        WMMA(warp_id)

RETURN $\boldsymbol{O}$

对循环融合和算子融合这两种技术进行分析. 先对循环融合进行分析,以一个二维矩阵上的运算为例. 假设已知两个二维矩阵 $\boldsymbol{A}_{1\,024\times 1\,024}$, $\boldsymbol{B}_{1\,024\times 1\,024}$,现在需要求矩阵 $\boldsymbol{C}_{1\,024\times 1\,024}$, C 中的每个元素等于 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 中相应元素的乘积再加上8. 实现这一过程的最朴素的方法是用三个循环来实现,依次完成元素初始化、对应位相乘和逐元素加8这三个操作. 这样的代码效率很低下,因为这三个部分的都需要从内存中读取 $\boldsymbol{C}$ 矩阵的所有元素,操作完成后再写回内存,对 $\boldsymbol{C}$ 矩阵的读写达到了3次. 但这三个部分的循环都是相同的两层循环,循环变量和迭代范围也相同,因此可以将这三部分的循环融合在一起,在一个循环内完成赋值和计算的全部过程. 变换后的代码不光是形式上更简洁,在性能上也有很大改进. 这两层循环遍历了 $\boldsymbol{C}$ 中的每个元素,然后一次性完成初始化、计算和赋值操作,再写回内存. 整个过程中只对 $\boldsymbol{C}$ 进行了一次读写,访存次数比变换前少了2/3.

在使用BGEMM算法之后,Winograd卷积四个部分的循环轴次序发生了较大变化. 在推理中,KTrans阶段是可以提前计算的,可以只分析其他三个阶段的循环融合. 容易找到 N 和ntiles维度是这三个阶段的共同循环轴,然而在融合的 N 循环和ntiles循环内,本质就是在一个切片上的Winograd卷积,计算阶段也退化成了批量矩阵向量乘法,由引入BGEMM算法带来的计算强度的提升也因此消失. 因此,依靠循环融合对Winograd卷积的kernel进行融合是不合适的.

而算子层面的融合在本质上等价于数学上的融合,考虑Winograd卷积kernel的融合,引入BGEMM后的二维Winograd卷积可表示为矩阵形式:

$$\boldsymbol{O} = \boldsymbol{A}^{\mathrm{T}}\{\mathrm{Re}[\mathrm{Re}(\boldsymbol{G}\boldsymbol{I}\boldsymbol{G}^{\mathrm{T}})\mathrm{Re}(\boldsymbol{F}_{\mathrm{Wino}})]\}\boldsymbol{A}$$

其中 $\mathrm{Re}(\cdot)$ 表示布局上的变换,为非线性变换,三处均为不同的布局变换. 暂无已知数学方法对式中嵌套的布局变换的矩阵运算进行化简,因此以算子融合的方式对引入BGEMM的二维Winograd卷积进行融合是不可行的. 但Winograd卷积的四个步骤存在数据相关性,存在利用数据局部性的可能性. 考虑到可以利用GPU中的缓存结构暂存每个阶段的中间运算结果,在进入下一阶段时直接从高速缓存中获取数据,因此本文提出了新的融合技术,以融合在数学上和算法实现上无

法直接融合的Winograd卷积kernel.

4.2.2　面向GPU内存层级的新融合策略

非融合的两个相邻算子的数据流如图5所示. 算子首先从DRAM中读取数据,经过LLC和共享内存读取到寄存器中. 在完成第一个算子的计算之后,将运算结果经过LLC和共享内存再放回DRAM. 到了下一个算子的计算时,再以相同的数据路径从DRAM中读取数据,完成计算后也以相同的数据路径写回. 因此对于非融合的两个算子,数据需要两次完整流过数据路径.

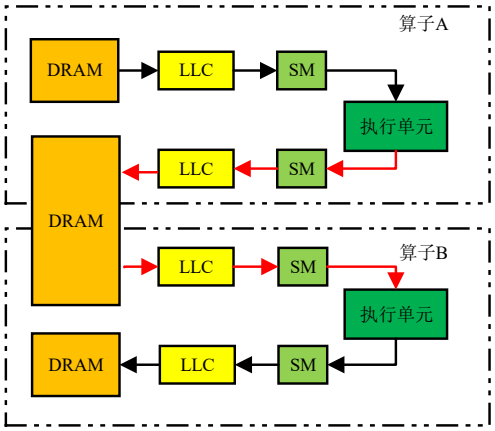


图5　非融合的两个相邻算子的数据流

而对于传统的算子或循环融合,前一步的中间运算结果不再重新写回内存. 第一个算子完成计算之后,直接将中间运算结果作为下一步运算的输入或临时存放到寄存器上,如图6所示. 下一个算子则不再需要从DRAM中重新读取,直接使用寄存器中的中间运算结果即可. 但根据上一小节的分析,可以发现无法直接用算子融合的方法直接融合Winograd卷积kernel,因此考虑新的融合方法.

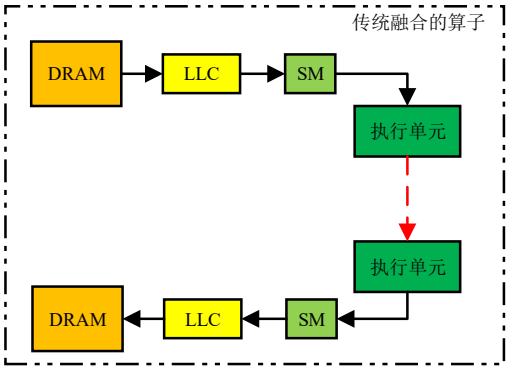


图6　传统融合算子的数据流

考虑这样一种数据路径:算子首先从DRAM中读取数据并交给第一个算子进行运算,再将中间运算结果暂存在Cache中,在下一个算子进行计算时,直接从Cache中读取中间运算结果作为输入. 那么根据GPU的多级内存层级,可以利用的Cache有两层,一层是LLC,另一层是共享内存. 利用LLC和共享内存的数据路径分别如图7和图8所示.

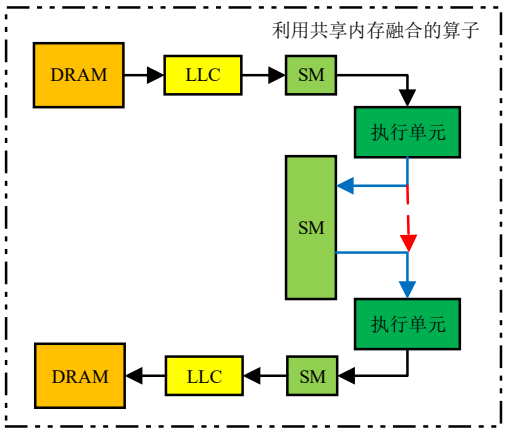


图7　利用共享内存融合的算子的数据流

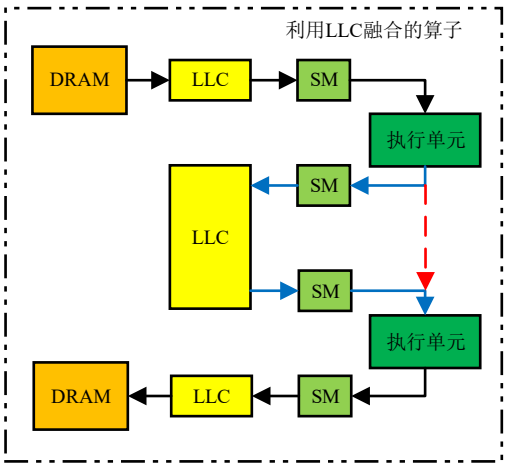


图8　利用LLC融合的算子的数据流

这两种融合方法与利用数据局部性有相似之处. 但利用数据局部性是隐式利用了数据的暂存,在Cache行被新读取的数据刷新之后,Cache上的数据就会失效,无法实现持久的驻留. 因此,本文提出的是显式使用内存层级,为每个kernel在运行时显式分配缓存空间. 如此一来,分配的Cache空间就不会被新访问的数据所刷新,在kernel运行时实现长久的驻留. 将前一个算子的运算结果存放在Cache上分配的空间中,在后一个算子需要时直接从相应的Cache中读取即可.

4.2.3　PKF技术

考察Winograd卷积在推理中的kernel融合,针对这种无法将全部四个kernel融合到一起的情况,本文提出了融合部分kernel的算法,即将KTrans视为提前执行的部分,然后融合ITrans+BGEMM或融合BGEMM+OTrans. 考虑到Sync-free BGEMM-TC算法中,K维度映

射到了线程块的索引维度，而C则映射到了线程块的迭代维度，每个线程块都会对完整的C维度进行计算，因此C维度上的融合更适合Sync-free BGEMM-TC算法，所以本文采用了ITrans+BGEMM的融合策略．由于这种融合策略仅融合了部分kernel，因此本文将这种策略命名为部分计算核融合（Partial Kernel Fusion，PKF）技术．

分别对两种利用不同GPU内存层级的PKF技术进行分析．首先是利用共享内存的PKF技术，记为PKF-Shmem．在GPU的CUDA编程模型中，可以通过__shared__关键字来显式使用共享内存．而共享内存是在SM上使用的，一个SM上同时只能运行一个warp，因此对共享内存内的使用是warp级别的，需要在相应的维度申请合适的共享内存大小．前文分析指出，WMMA API对矩阵规模有严格限制，因此限制了ntiles，C和K上切分的因子．以ntiles为例，ntiles的切分因子越大，单个线程块中分配的warp和线程数就会越多，更充分地利用计算资源；但同时也意味着卷积核需要重复读取更多次，浪费不必要的带宽资源．结合Sync-free BGEMM-TC算法，利用共享内存应用PKF技术的Winograd卷积算法如算法4所示．

使用__shared__关键字可以显式使用共享内存，使用__device__关键字可以使用DRAM，但没有其他函数或声明方式是可以直接在LLC上申请空间的．换言之，LLC对于设备上的代码来说是透明的．所有对内存的访问在没有命中L1 Cache（新的GPU上完成了L1 Cache和共享内存的统一）的情况下都会向LLC发起访问请求，而所有的写访问也都会经过LLC．通过指定编译选项-dlcm=cg可以使全局访问不再在共享内存中缓存但在LLC中缓存，但这种方法禁用了共享内存所以反而不利于性能提升．而使用线程同步指令可以确保所有到DRAM的写访问都在LLC中可见，这为本文实现利用LLC的融合提供了一种新的思路．在使用的数据适应LLC大小的情况下，可以最大化利用LLC读写数据．在ITrans完成之后，数据会经过共享内存和LLC写到DRAM上，在ITrans之后添加一个线程同步指令，就可以确保写访问的数据已经在LLC上可见，BGEMM就可以直接从LLC上访问数据，而不用向DRAM请求数据．因此，利用LLC的PKF技术在算法上比利用共享内存的更简单一些，只需要合并ITrans和BGEMM的实现，并在二者之间插入一个线程同步指令即可．

4.3 设计空间探索

以$\mathrm{F}(m\times m, r\times r)$为例，Winograd卷积的输出切片大小为$(m,m)$，输入切片大小为$(m+r-1, m+r-1)$．输出切片为互不重叠的张量，这对卷积输出张量的尺寸提出了要求，即输出张量的HW维度必须被m整除．通过对卷积进行分解，可以应用多种不同参数的$\mathrm{F}(m\times m, r\times r)$突破这一限制，不满足条件的卷积也可以等价为多个满足条件的卷积，因此本文只需考虑零填充、单位步长的情况，即输入输出尺寸相等．因此，Winograd卷积的切片数量计算可以简化为$\mathrm{ntiles}=HW/m^2$．而H_{tile}和W_{tile}即均可表示为$m+r-1$，对于最典型的$\mathrm{F}(2\times 2, 3\times 3)$，$\mathrm{F}(4\times 4, 3\times 3)$和$\mathrm{F}(6\times 6, 3\times 3)$，该值分别为4，6和8．算法将$W_{\mathrm{tile}}$维度映射在了warp_id上，因此每个线程块包括$m+r-1$个warp，共$32(m+r-1)$个线程．

算法4 利用共享内存应用PKF技术的Sync-free BGEMM-TC Winograd卷积

输入：输入张量$\boldsymbol{I}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{ntiles}, C)$和卷积核$\boldsymbol{F}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, K, C)$

输出：输出张量$\boldsymbol{O}(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{ntiles}, K)$

```
// 融合的kernel ITrans+BGEMM
// 划分一个批次的GEMM为TB_M × TB_N × TB_K规模的矩阵乘法
// 映射到线程块的x索引维度
FOR n_outer 0 → ntiles/TB_M
    // 映射到线程块的y索引维度
    FOR k_outer = 0 → K/TB_N
        Shared shmem_F[H_tile, W_tile, TB_N, C] = LOAD(F);
        Shared shmem_I[H_tile, W_tile, TB_M, TB_K];
        FOR n_inner = 0 → TB_M
            FOR c = 0 → C
                shmem_I = InputTrans(I[:][:][n_inner][c]);
    // 作为线程块的迭代维度
    FOR c_outer = 0 → C/TB_K
        // 作为warp的迭代维度
        FOR h = 0 → H_tile
            // 映射到warp的索引维度
            FOR warp_id = 0 → W_tile
                Fragment A, B, C;
                // Warp加载和WMMA,调用Tensor Intrinsics
                A, B = LOAD(shmem_I, shmem_F)
                C = WMMA(A, B)
                // …
                // kernel OTrans
                // …
RETURN O
```

首先对基于共享内存的PKF进行分析．每个线程块负责的矩阵乘规模为为$\mathrm{TB}_M\times\mathrm{TB}_N\times\mathrm{TB}_K$，其中$\mathrm{TB}_M$和$\mathrm{TB}_K$分别映射在了线程块的$xy$索引维度，而$\mathrm{TB}_N$为线程块的迭代维度．因此线程块每次迭代需要访问的Winograd卷积核大小为$(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{TB}_N, \mathrm{TB}_K)$，特征映射域输入尺寸为$(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{TB}_M, \mathrm{TB}_K)$．经过ITrans阶段，得到的Winograd域中间计算结果大小也为$(H_{\mathrm{tile}}, W_{\mathrm{tile}}, \mathrm{TB}_M, \mathrm{TB}_K)$．经过BGEMM阶段，线程块得到

的Winograd域输出尺寸为($H_{\text{tile}}, W_{\text{tile}}, \text{TB}_M, \text{TB}_N$). 因此,需要的共享内存大小为$H_{\text{tile}}W_{\text{tile}}\text{TB}_M\text{TB}_K$个元素,对于半精度浮点数FP16和INT8,分别为$2H_{\text{tile}}W_{\text{tile}}\text{TB}_M\text{TB}_K$字节和$H_{\text{tile}}W_{\text{tile}}\text{TB}_M\text{TB}_K$字节. TB_M和TB_K分别对应于ntiles和C这两个维度的切分因子,所以这两个切分因子直接决定了对共享内存的占用. 又由于WMMA API的限制,TB_M必须为8,16,32的倍数,TB_K必须为16的倍数.

基于LLC的PKF在内存占用的分析上与基于共享内存的PKF类似,但使用的缓存空间变成了LLC. 但需要注意的是,LLC是由所有SM共享的缓存,因此要想数据占用大小适应LLC的大小,应计算出平均每个SM可用的LLC大小. 线程块的数量由映射到线程块索引维度上的值大小决定,各个索引维度上的值相乘的结果即为线程块数量总和. 在算法中,ntiles轴和K轴切分之后的外层循环映射到了线程块的xy轴上,因此线程块的总数量$N_{\text{TB}}=\text{ntiles}/\text{TB}_M\cdot K/\text{TB}_N=(\text{ntiles}\cdot\text{ntiles})/(\text{TB}_M\cdot\text{TB}_N)$. 但GPU上同时启动的线程块的数量是有限的,上限为SM的个数,记为N_{SM}. 当$N_{\text{TB}}<N_{\text{SM}}$时,LLC最多由$N_{\text{TB}}$个SM共享,而当$N_{\text{TB}}\geqslant N_{\text{SM}}$时,LLC由$N_{\text{SM}}$个SM共享.

假设共享内存和LLC的大小分别为V_{Shmem}和V_{LLC},单位为字节(B). 那么对于基于共享内存的PKF,线程块每次迭代所占用的共享内存不能超过V_{Shmem},即$V_e H_{\text{tile}}W_{\text{tile}}\text{TB}_M\text{TB}_K\leqslant V_{\text{Shmem}}$,其中$V_e$为单个中间运算结果的数据类型的大小. 而对于基于LLC的PKF,所有运行中的线程块所占用的LLC的大小之和不能超过V_{LLC},即$V_e H_{\text{tile}}W_{\text{tile}}\text{TB}_M\text{TB}_K\text{Min}\{N_{\text{TB}},N_{\text{SM}}\}\leqslant V_{\text{Shmem}}$. 因此,两种PKF算法实现的设计空间必须尽可能满足相对应的不等式,防止超过缓存大小限制带来的缓存丢失. 需要注意的是,基于共享内存的PKF也应满足$N_{\text{TB}}\gg N_{\text{SM}}$以最大化GPU利用率. 整理以上设计空间限制条件,就可以为该实例得到参数($\text{TB}_M, \text{TB}_N, \text{TB}_K$)有限的可选集合.

现通过与原始Winograd算法、现有BGEMM算法对比,对PKF算法的算法复杂性进行分析. 从空间复杂度来说,三种算法在输入输出上元素个数一致,仅在布局上有所区别. 但PKF算法采用了占用内存空间更少的低精度数据类型,即FP16和INT8,因此在内存使用上分别缩小为原有的一半和四分之一. 而在性能度量上,并行算法区别于传统串行算法,不简单使用算法的时间复杂性来描述,而是使用工作量、通信开销和并行效率等指标进行阐述. 在工作量上,三个算法总的运算次数是一致的,仅仅与输入规模有关,而与并行度无关;在通信开销上,由于原始Winograd卷积在EWMM运算后再在C维度上进行规约累加,因此线程间单次通信量为$H_{\text{tile}}W_{\text{tile}}$,而原始BGEMM算法和PKF算法中,线程协作完成矩阵乘法,降低了线程间通信开销,同时PKF算法中同步指令的消除也进一步缩减了开销;而在并行效率上,原始Winograd卷积在CUDA Core上进行EWMM运算,对GPU的计算能力利用较低,即使提高并行度也难以覆盖访存、通信开销,而原始BGEMM算法提高了算术强度,降低了访存、通信开销相对计算的比例,PKF算法通过利用Tensor Core的计算能力进一步提升了并行效率,并通过对设计空间进行探索,最大化了对资源的利用. 综上分析,采用PKF算法可以为Winograd卷积的计算带来显著性能提升.

4.4　基于TVM的Winograd卷积解耦合实现

4.4.1　基于TVM的原型实现

TVM[23]是一个端到端的深度学习编译器,可以为各种硬件后端生成优化的低级代码. 与大多数传统编译器不同,TVM的输入是用compute和schedule描述的算法,而不是一般的高级编程语言代码. 通过对compute和schedule进行编译可以生成特定硬件加速器的高级语言代码或低级的机器代码. TVM引入了一种张量表达式(tensor expression)语言来定义各种运算符,还提供了一些程序转换原语来对默认调度进行变换. TVM在设计上受启发于Halide中分离compute和schedule的做法,这种思想为自动优化和生成代码提供了便利. TVM使用schedule来将张量表达式映射到低级代码,通过指定如何进行计算来定义compute的实现. 可以使用各种方法(比如AutoTVM[27]使用了基于学习的方法)搜索调度空间来找到compute的高性能schedule.

利用TVM提供的张量表达式和调度原语,可以定义Winograd卷积的comptue并指定调度,但要实现本文提出的算法还存在两点困难,分别为同步指令的消除和阶段的融合. Sync-free BGEMM的调度使用基本的调度原语是可以实现的,该算法的提出是为了避免不必要的线程同步指令,但使用TVM的调度原语会使生成CUDA代码时自动在该位置插入内存同步指令,使算法失去Sync-free这一特性. 而阶段的融合问题出在线程块内的warp数量上. 前面的分析中指出,算法在BGEMM阶段每个线程块有W_{tile}个warp,而在ITrans阶段的计算则只能在C维度进行切分并指定内层循环绑定到warp的索引维度. 对于典型的$F(4\times4,3\times3)$,W_{tile}为6,ITrans却不能指定warp数量也为6,否则会引起不整除的情况. 因此,BGEMM阶段和ITrans阶段的warp索引维度大小一般都是不同的,而这会导致无法使用TVM原语指定两个阶段的融合. 考虑到通过直接修改TVM源码实现算法代价很大并可能影响其正常工作,本文提出先使用TVM生成基本具备BGEMM调度并且映射到Tensor Core运算的原型CUDA代码,再通过代码重构器将Sync-free和PKF技术应用在CUDA代码上以生成最终代码.

原型的实现分为两步:一是通过定义compute来给定Winograd卷积的基本运算,二是根据本文提出的算法定义Winograd卷积的schedule. compute的定义上并无特殊之处,在此不做赘述,下面着重描述schedule的实现细节. 首先是ITrans阶段的调度. compute的定义中ITrans阶段包括三个部分,分别为零填充、输入张量切片和输入张量变换,前两步可以直接使用compute_inline原语在第三步中内联计算,因此只需对输入张量变换阶段的schedule进行定义. 这里需要引入对内存层级的显式使用. 首先将切片读取到共享内存中,再将其读取到SM的寄存器即本地存储上. 分别在ntiles维度、C维度进行切分,前者切分得到的外层映射到线程块索引上,后者切分得到的内层映射到warp内的线程索引维度上,而ntiles维度上切分得到的内层,即进一步切分以映射到warp索引维度上. 从ITrans的计算阶段向外层推导,可得到将切片读取到共享内存上的调度. 加载到共享内存的调度中设置一偏移量以避免bank冲突. 变换阶段的schedule定义完成之后,将输入张量切片和零填充这两个步骤依次内联到输入张量变换中.

在BGEMM阶段的调度实现上,依然遵循着从输出到输入的顺序. 首先对输出到内存的部分进行调度,根据算法2和3,W维度作为warp的索引维度,ntiles维度和K维度上进行切分映射到线程块的索引维度上,warp内部则使用向量化的数据访问方式. 将结果写到共享内存的部分,一个warp每次会按照WMMA API写一个相应大小运算结果,同样引入一个偏移量以防止bank冲突. 在计算部分,一个warp在一次迭代中会计算一个wmma_m×wmma_n×wmma_k大小的矩阵乘法,在此之前也会从共享内存读取相应的矩阵到WMMA空间. 而从内存读取到共享内存的部分中,同样将W维度作为warp的索引维度,warp内部使用向量化的数据访问方式,根据数据类型可以设置向量化访问相应的元素个数. 最后将WMMA计算映射到Tensor Intrinsic(即WMMA API提供的计算指令)上. 通过定义Tensor Core上的compute模式匹配compute实现,再使用tensorize原语将相应的计算替换为张量化的Tensor Intrinsic即可. 最后,调用tvm. build(·)生成底层代码实现,并输出为CUDA源代码,即可得到算法的Winograd卷积原型.

4.4.2 基于代码重构技术的PKF优化

本文提出了一种基于代码重构的低成本的、解耦合的重构器——PKF-Reconstructor. 首先基于第4.4.1节设计实现的compute和schedule原语在TVM上构建PKF Winograd的原型IRModule(Intermediate Representation Module,中间表示模块). 然后,使用TVM为该原型生成中间CUDA代码. 最后,使用PKF-Reconstructor从中间代码生成最终的PKF Winograd CUDA代码并将其编译为GPU工作负载. 这种实现思路避免了对TVM代码库本身的修改,同时将调度优化和Sync-free PKF技术解耦合,降低了成本的同时也利于调试和改进. 这也导致了一个严重的缺点,即这种方法只能生成单个优化的Winograd卷积算子,无法直接应用于完整的CNN模型. 但在保持输入输出数据布局一致的情况下,单个算子的性能变化足以验证算法的效果.

PKF-Reconstructor的核心为正则表达式匹配,从原型CUDA代码提取出需要的代码并进行处理以得到PKF Sync-free的Winograd卷积的kernel实现,再插入算子测试模板并修改参数得到测试用例最终代码. PKF-Reconstructor具体的工作流程如下:

(1)通过正则表达式匹配分离的ITrans,KTrans,BGEMM和OTrans的kernel源码;

(2)在BGEMM的kernel代码中匹配并移除warp加载数据后自动插入的线程同步指令;

(3)为融合的ITrans+BGEMM的kernel生成对应的kernel函数名和参数列表,并将ITrans的kernel源码作为融合的kernel主体;

(4)从BGEMM的kernel代码中匹配绑定到线程块的位置,将其作用域内的代码插入到融合的kernel中对应的位置;

(5)如果是基于Shmem的实现,匹配部分需要修改的迭代变量,重新计算其值并修改以确保结果的正确性,否则跳过这一步;

(6)将融合的kernel,KTrans和OTrans插入测试模板代码的相应位置得到测试用例最终代码;

(7)修改模板中负责定义工作负载分配的相关参数,重新计算并修改迭代变量以获得正确的PKF Winograd;

(8)调用nvcc编译测试代码并运行.

此外,在实验过程中发现,可以在不影响程序正确性的情况下进一步移除部分线程同步指令. 因此,可在PKF-Reconstructor编译之前增添一个步骤:匹配代码中所有的线程同步指令,逐个尝试删除,删除后进行编译并测试. 如果程序输出结果与正确结果一致,则保留该删除并继续尝试;否则,撤销该删除并继续尝试. 在完成所有线程同步指令的删除测试之后,正式编译测试代码并测试,给出最终的测试结果.

5 实验分析

本节在GPU上对一组实际应用的卷积算子进行了测试,评估几种PKF Winograd卷积实现的性能. 主要的实验设置如下:

(1)硬件平台. NVIDIA RTX 2080Ti GPU,DRAM大小为11 GB,LLC大小为5.5 MB,共享内存大小为每SM各64 KB,共有68个SM,每个SM包含8个Tensor Core.

(2)软件平台. 基于Apache TVM 0.7.dev[28]实现了基本的Winograd卷积和优化的Winograd卷积原型,并将其用于生成中间CUDA C++代码;PKF-Reconstructor使用Python语言编写,在Python 3.8.4上成功运行;GPU工作负载通过NVCC v11.0编译并链接.

(3)工作负载. 工作负载为现实世界CNN模型中最常见的卷积算子. 本文对MXNet Zoo[29]中的Inception-V3, ResNet-V1, ResNet-V2和DenseNet等网络的卷积层进行了统计和分析,选择了其中频率最高的$C=K=64$,$R=S=3$的卷积核作为工作负载. 该卷积核参数具有代表性,而统计和分析表明,出现频率最高的28个卷积核参数中,有26组满足算法需求. 为测试算法在大规模特征映射上的适应性,输入特征映射的尺寸从基本的224变化到超分辨率网络中常见的960不等.

(4)基线. 本文使用GPU供应商NVIDIA提供的神经网络计算库cuDNN v8.5来生成基线Winograd卷积工作负载;为了进行更全面的比较,TVM上的原生Winograd卷积实现(记为Vanilla TVM)也在测试范围内作为对照组. 每个工作负载重复测试100次并取平均值作为测试结果.

通过4.3节对实现设计空间的分析,对于FP16上的算法,仅有$TB_M=16$和$TB_M=32$这两种情况在设计空间中,而这两种参数选择和两种PKF技术实现相组合可以得到四种不同的实现. 这里通过短横线在方法名后附加TB_M的值来区分不同的方法. 对cuDNN进行评估时,融合的Winograd卷积实现和不融合的Winograd卷积实现都进行了测试,取较快的作为测量结果. TVM上有四种Winograd卷积的实现,同样取其中的最佳性能代表TVM原生Winograd卷积实现的测试结果. 同时,在应用PKF-Reconstructor之前的原型(记为Tuned TVM)也作为对照组进行实验,以评估PKF技术的有效性. 最终的测试结果如图9所示.

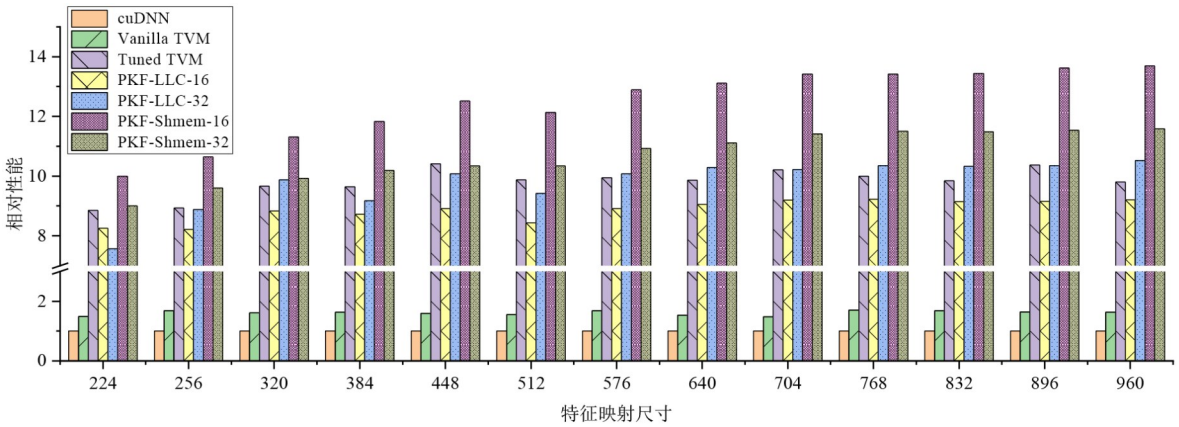


图9　Winograd卷积实现在FP16上的相对性能

实验结果表明,与cuDNN相比,FP16上的四种PKF实现都取得了可观的性能. 其中PKF-Shmem-16表现最好,实现了9.99到13.69倍的加速. 通过图9可以直观地注意到,PKF-Shmem技术带来的改进随着特征映射大小的增长而增加,而PKF-LLC技术没有表现出这种趋势. 与cuDNN相比,原生TVM有47.54%~70.50%的效率提升,但与四种PKF实现和原型实现相差甚远. 与原型相比,PKF-LLC甚至有性能下降,因为这种融合方法需要额外的线程同步. 而PKF-LLC-32则更好地利用了计算资源,在计算量变大时表现出融合带来的优势. PKF-Shmem的性能表现突出,因为这种实现可以更好地利用高带宽的共享内存,而不会引入同步开销. PKF-Shmem-32实现没有获得与PKF-Shmem-16相同的性能增益. 通过使用nvprof分析PKF-Shmem-32的kernel执行情况,可以发现其LLC的命中率反常地远高于其他的实现. 这种通常的命中率增加是由于重复访问Winograd域的卷积核,这种重复的访问严重浪费了LLC的带宽. 使用相同的实验配置可以在INT8上进行测试,最终的测试结果如图10所示.

可以看到,INT8上的PKF技术同样获得了很大的性能提升,但与FP16的效果相比并没有较大差别. 其原因在于使用INT8数据类型会使BGEMM阶段的输出结果数据类型变为INT32,而FP16数据类型在BGEMM阶段的输出是FP16类型,前者每个元素占用4个字节,而后者仅占用2个字节,因此前者会占用更多的带宽,从而在OTrans阶段花费更多的时间. 而另一点与FP16不同的是,INT8上PKF-LLC的平均表现要略强于FP16. 前面分析到计算过程中会重复访问Winograd域

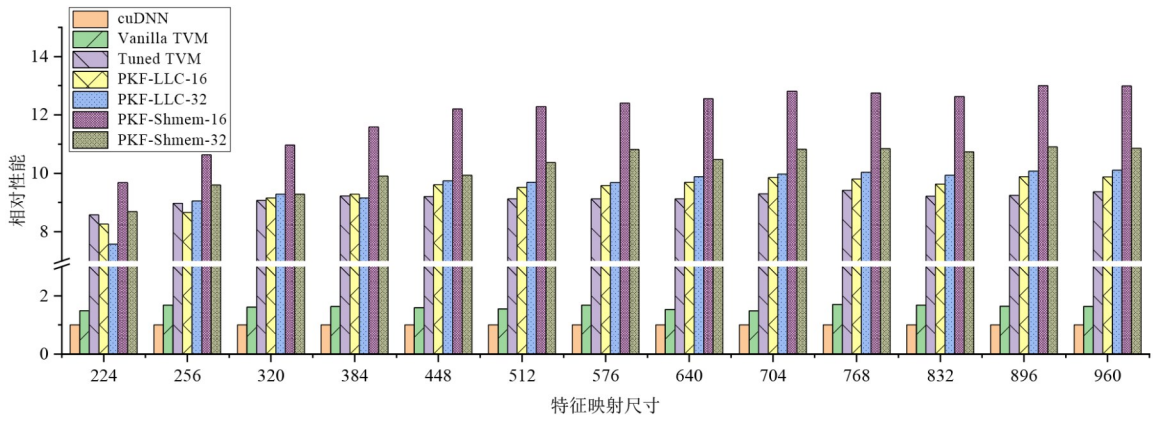


图 10 Winograd 卷积实现在 INT8 上的相对性能

卷积核，浪费 LLC 的带宽，而使用 INT8 数据类型的 Winograd 域卷积核可以将带宽占用降低一半，进而为 LLC 上的融合节约更多的带宽. 总体而言，INT8 上的 PKF 技术相对 cuDNN 上的 Winograd 卷积实现也带来了 7.58~13.00 倍的性能提升. 利用本文在两个不同内存层级的缓存上提出的 PKF 方法，可以进一步扩展 TVM 这类深度学习编译器的搜索空间，并为寻找更好的解决方案提供更多可能性.

6 总结与展望

本文面向现代 GPU 体系结构对 Winograd 卷积进行了优化，提出利用 Tensor Core 进行低精度计算和利用 GPU 内存层级进行部分融合. 实验表明，本文提出的方法相比 cuDNN 上的 Winograd 卷积实现有 7.58~13.69 倍的性能提升. 当然，还有诸多优化技术未应用到本文的实现中，比如共享内存的 shuffle，double buffer 等. 但这些优化技术与本文提出的算法是相互正交的，可以进一步结合以优化工作负载. 不过，毋庸置疑的是，本文提出的新颖的 PKF 技术带来了可观的性能提升，对于 Winograd 卷积之外的工作负载在理论上也是有效的. 另外，多级内存层级不仅在 GPU 中存在，在 CPU 等其他体系结构中也是类似的. 因此 PKF 技术不仅拓宽了设计空间的维度，同时也有望推广到其他体系结构上. 另一方面，低精度和量化技术通常与剪枝技术相结合，GPU 上已有高效稀疏矩阵的运行算法，也可考虑在算法中进一步结合剪枝技术进一步减少乘法计算量以获得更大的性能提升.

参考文献

[1] LAVIN A, GRAY S. Fast algorithms for convolutional neural networks[C]//2016 IEEE CVPR. Las Vegas: IEEE, 2016: 4013-4021.

[2] MENG L, BROTHERS J. Efficient Winograd convolution via integer arithmetic[EB/OL]. (2019) [2021]. http://arxiv.org/abs/1901.01965.

[3] GONG Y, LIU B, GE W, et al. ARA: Cross-layer approximate computing framework based reconfigurable architecture for CNNs[J]. Microelectronics Journal, 2019, 87: 33-44.

[4] FERNANDEZ-MARQUES J, WHATMOUGH P N, MUNDY A, et al. Searching for Winograd-aware quantized networks[J]. Proceedings of Machine Learning and Systems, 2020, 2: 14-29.

[5] LIU Z G, MATTINA M. Efficient residue number system based Winograd convolution[C]//Computer Vision-ECCV 2020. Cham: Springer, 2020: 53-68.

[6] ZHANG W, LIAO X, JIN H. Fine-grained scheduling in FPGA-based convolutional neural networks[C]//2020 IEEE 5th ICCCBDA. Chengdu: IEEE, 2020: 120-128.

[7] BARABASZ B. Quantaized Winograd/Toom-Cook convolution for DNNs: Beyond canonical polynomials base[EB/OL]. (2020)[2021]. http://arxiv.org/abs/2004.11077.

[8] LI G, LIU L, WANG X, et al. Lance: Efficient low-precision quantized Winograd convolution for neural networks based on graphics processing units[C]//ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). Barcelona: IEEE, 2020: 3842-3846.

[9] HAN Q, HU Y, YU F, et al. Extremely low-bit convolution optimization for quantized neural network on modern computer architectures[C]//49th International Conference on Parallel Processing-ICPP. New York: ACM, 2020: 1-12.

[10] SABIR D, HANIF M A, HASSAN A, et al. TiQSA: Workload minimization in convolutional neural networks using tile quantization and symmetry approximation[J]. IEEE Access, 2021, 9: 53647-53668.

[11] CAO Y, SONG C, TANG Y. Efficient LUT-based FPGA accelerator design for universal quantized CNN inference [C]//2021 2nd Asia Service Sciences and Software Engi-

neering Conference. Macao: ACM, 2021: 108-115.
[12] GHAFFAR M M, SUDARSHAN C, WEIS C, et al. A low power in-DRAM architecture for quantized CNNs using fast Winograd convolutions[C]//The International Symposium on Memory Systems. Washington: ACM, 2020: 158-168.
[13] WU D, FAN X, CAO W, et al. SWM: A high-performance sparse-Winograd matrix multiplication CNN accelerator[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2021, 29(5): 936-949.
[14] YAO Y, LI Y, WANG C, et al. INT8 Winograd acceleration for Conv1D equipped ASR models deployed on mobile devices[EB/OL]. (2020) [2021]. http://arxiv. org/abs/2010.14841.
[15] HUANG D, ZHANG X, ZHANG R, et al. DWM: A decomposable winograd method for convolution acceleration[J]. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, 34(4): 4174-4181.
[16] JIANG J, CHEN X, TSUI C Y. A reconfigurable Winograd CNN accelerator with nesting decomposition algorithm for computing convolution with large filters[EB/OL]. (2021) [2021]. https://arxiv.org/abs/2102.13272v1.
[17] VINCENT K, STEPHANO K, FRUMKIN M, et al. On improving the numerical stability of Winograd convolutions [C]//5th International Conference on Learning Representations (ICLR 2017). Toulon: OpenView, 2017: 1-4.
[18] HONG B, RO Y, KIM J. Multi-dimensional parallel training of Winograd layer on memory-centric architecture [C]//2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Fukuoka: IEEE, 2018: 682-695.
[19] JIA L, LIANG Y, LI X, et al. Enabling efficient fast convolution algorithms on GPUs via MegaKernels[J]. IEEE Transactions on Computers, 2020, 69: 986-997.
[20] YAN D, WANG W, CHU X. Optimizing batched Winograd convolution on GPUs[C]//Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego: ACM, 2020: 32-44.
[21] WINOGRAD S. Arithmetic complexity of computations [EB/OL]. (1980) [2021]. https://epubs. siam. org/doi/abs/10.1137/1.9781611970364.
[22] ABADI M, BARHAM P, CHEN J, et al. TensorFlow: A system for large-scale machine learning[C]//Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016: 265-283.
[23] CHEN T, MOREAU T, JIANG Z, et al. TVM: An automated end-to-end optimizing compiler for deep learning [C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18). Savannah: USENIX Association, 2018: 17.
[24] KENNEDY K, ALLEN J R. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach [M]. San Francisco: Morgan Kaufmann Publishers Inc., 2001.
[25] GELASHVILI R, SHAVIT N, ZLATESKI A. L3 fusion: Fast transformed convolutions on CPUs[EB/OL]. (2019) [2021]. http://arxiv.org/abs/1912.02165.
[26] NVIDIA. GPU performance background user guide[EB/OL]. (2021)[2021]. http://docs.nvidia.com/deeplearning/frameworks/dl-performance-gpu-background/index.html.
[27] CHEN T, ZHENG L, YAN E, et al. Learning to optimize tensor programs[EB/OL]. (2018) [2021]. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html.
[28] Apache. Apache TVM[EB/OL]. (2021) [2021]. https://tvm.apache.org/.
[29] Apache. Apache MXNet[EB/OL]. (2021) [2021]. https://mxnet.apache.org/versions/1.8.0/.

作者简介

童　敢　男,1995年出生,安徽宣城人. 现为国防科技大学计算机学院硕士研究生. 主要研究方向为面向计算体系结构的算法优化.
E-mail: gantong_nudt@163.com

黄立波　男,1983年出生,湖南邵阳人. 现为国防科技大学计算机学院副研究员. 主要研究方向为计算机体系结构.
E-mail: libohuang@nudt.edu.cn

吕雅帅　男,1981年出生,河北邯郸人. 现为华为海思技术专家. 主要研究方向为编译优化与处理器体系结构.
E-mail: freelancer_lys@163.com