基于 TVM 的 Winograd 自动性能优化

陈疆[1, 2, 3]，朱泓霖[3]，孟金涛[2*] 魏彦杰[2]

[1] (南方科技大学，深圳 518055)

[2] (中国科学院深圳先进技术研究院，深圳 518055)

[3] (深圳市腾讯计算机系统有限公司，深圳 518063)

摘要 卷积神经网络(CNNs)作为深度学习的典型代表，是计算机视觉等任务最常用的神经网络，然而卷积运算通常占整个 CNNs 运行时的 90%以上，成为 CNNs 的性能瓶颈。另外，由于当下硬件的复杂性及工作负载的多样性，之前工作的一些特定优化往往缺乏性能可移植性。对此我们提出了 BlazerML，一个基于 TVM 模板代码自动生成的开源卷积计算库，可以自动生成任意输入形状的高性能卷积实现，BlazerML 是基于 Winograd 算法实现的，因为该算法是快速卷积算法中性能最高的算法。实验结果表明 BlazerML 显著优于当下最先进的开源库。在 x86 CPU 上运行常见的深度学习网络前向推理分别比 OnnxRuntime、MNN 和 TVM 社区版本快 1.18~2.47、1.18~2.27 和 1.01~1.66 倍。在 ARM CPU 上运行常见深度学习网络的单层推理分别比速 ACL 和 FastConv 快 1.26~6.11、1.04~4.28 倍。

关键词 深度学习；卷积神经网络；快速卷积算法；Winograd 算法；TVM；自动性能优化

中图分类号 TP 399 文献标志码 A doi: 10.12146/j.issn.2095-3135.20240202001

Winograd Automatic Performance Optimization Based on TVM

CHEN Jiang[1,2,3], ZHU Honglin[3], MENG Jintao[2*] WEI Yanjie[2]

[1](Southern University of Science and Technology, city Shenzhen 518055, China)

[2](Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, city Shenzhen 518055, China)

[3](Shenzhen city Tencent computer system Co.Ltd., city Shenzhen 518063, China)

Abstract Convolutional Neural Networks (CNNs), as a quintessential representation of deep learning, are the most commonly used neural networks in tasks such as computer vision. However, convolution operations typically account for over 90% of the runtime in CNNs, becoming a bottleneck for performance. Additionally, due to the complexity of current hardware and the diversity of workloads, specific optimizations in previous work often lack performance portability. To address this, we introduce BlazerML, an open-source convolution computation library based on auto-generated code templates from TVM, capable of automatically generating high-performance convolution implementations for any input shape. BlazerML is implemented based on the Winograd algorithm, known for its high performance in fast convolution algorithms. Experimental results demonstrate that BlazerML significantly outperforms current state-of-the-art open-source libraries. On x86 CPUs, running common deep learning network forward inferences, it is faster by 1.18~2.47, 1.18~2.27, and 1.01~1.66

来稿日期：2024-02-02 修回日期：2024-03-21

基金项目：广东省重点领域研发计划资助项目(2021B0101310002)；国家自然科学基金项目(62272449)；深圳市基础研究项目(RCYX20200714114734194, KQTD20200820113106007, ZDSYS20220422103800001)；中国科学院青年创新促进会(Y2021101)

作者简介：陈疆，硕士研究生，研究方向为高性能计算；朱泓霖，腾讯 AI Lab 工程师；孟金涛（通讯作者），副研究员，研究方向为高性能计算，E-mail：jt.meng@siat.ac.cn；魏彦杰，研究员，研究方向为生物信息学与高性能计算交叉领域，聚焦蛋白质折叠、基因大数据分析等研究。

times compared to OnnxRuntime, MNN, and the TVM community version, respectively. On ARM CPUs, for single-layer inference of common deep learning networks, it surpasses ACL and FastConv by 1.26~6.11 and 1.04~4.28 times, respectively.

Keywords Deep Learning; Convolutional Neural Networks; Fast Convolution Algorithms; Winograd Algorithm; TVM; Automatic Performance Optimization

Funding This work was supported by the Key Research and Development Project of Guangdong Province (2021B0101310002), National Natural Science Foundation of China (62272449), Shenzhen Basic Research Fundation (RCYX20200714114734194, KQTD20200820113106007, ZDSYS20220422103800001), and Youth Innovation Promotion Association, CAS (Y2021101)

引 言

近年来，随着深度学习的迅速发展，卷积神经网络(CNNs)[1]已经成为处理复杂视觉任务的核心技术，广泛应用于图像识别、物体检测、自动驾驶等多个领域。这些模型依赖于大量的数据和复杂的数学运算，尤其是在训练阶段，需要庞大的计算资源进行支持。然而，随着深度学习技术的广泛应用，如自动驾驶车辆、智能监控和个性化医疗等领域，模型推理的效率和能效成为了研究的焦点。模型推理主要发生在数据中心或边缘设备上，这些场景对计算效率、能耗和实时性有着严格的要求。

尽管 GPU 和专用 AI 加速器在处理深度学习任务方面展现出了卓越的性能，但它们在成本、能耗和可用性方面存在局限。相比之下，CPU 作为一种更为通用的计算平台，在多种环境中都有广泛的部署，从而成为推理任务的一个重要平台。CPU 的普遍可用性意味着在不增加特定硬件的前提下，能够在现有的基础设施上部署深度学习模型。此外，CPU 的能效和性能在过去几年里有了显著提升，特别是在 x86 和 ARM 架构上，这使得它们成为运行推理任务的有力候选者。

然而，在 CNNs 的计算中，90%以上的计算量集中在卷积层的实现上，因此卷积层的计算性能几乎决定了整个卷积神经网络的性能。此外，卷积也不仅仅局限于卷积神经网络，在很多其他网络模型中也都有涉及。由此可见，卷积性能的好坏对整个深度学习领域都有着非凡的影响。当下有很多基于专家传统手工优化的方法虽然能够提高卷积运算的效率，但面对不断演变的硬件架构和多样化的应用需求，这些方法往往缺乏灵活性和可扩展性。因此，探索一种能够自动适应不同硬件和数据特性的性能优化方法，成为了当下的一个紧迫需求。

当前实现卷积计算的方法主要有直接卷积[2]、基于通用矩阵乘(GEMM)[3]、快速傅里叶变换(FFT)以及 Winograd[4]4 种。直接卷积就是按照卷积定义，通过在输入张量上滑动卷积核，并在每个位置计算卷积核与输入小块的点积来工作，常表现出较差的性能。基于 GEMM 的算法也称为 Im2col，这种方法通过格式转换将输入张量映射成行或列优先矩阵，将卷积操作转换为 GEMM 操作。使用这种矩阵格式，卷积操作可以作为单一矩阵乘法执行，以利用高度优化的 GEMM 操作，该操作使用高度优化的基础线性代数程序集(BLAS)库加速。然而，Im2col 可能会增加内存占用，并且张量到矩阵的转换可能导致形状不规则、常常不能获得最佳的性能。FFT 的核心思想是在频域进行卷积运算，减少了浮点计算量。它在大卷积核场景下可以有很好的性能，但是目前卷积运算中的卷积核都比较小，导致性能较差。最后就是 Winograd 算法，其核心思想是引入更多的加法来代替乘法计算，在大多数场景下都有不错的表现，使得其在学术界与工业界均受到了广泛的关注和研究。

目前，研究工作主要集中在算法的泛化、拓展及其在各种体系结构上的实现。在算法优化方面，采用数学方法来突破 Winograd 算法的局限性显得尤为关键，但这一过程需要专业数学家的深入证明和精确推导。因此，探索针对硬件友好的优化方法成为未来研究的一个重要方向。然而，要在不同硬件特性的设备上高效实现 Winograd 算法仍然是一项挑战，考虑到为各种硬件单独开发适配的 Winograd 算法既不经济又不实用，这一点尤为突出。

此外，深度学习模型的多样性也带来了额外的复杂性。每个模型中输入图像的大小、输入输出通道的尺寸都有所不同，导致不同卷积层的参数在形状和尺寸上差异显著。由于不同形状和尺寸的矩阵需要完全不同的最优计算实现方法，因此开发一个高性能的 Winograd 算法还需要解决在不同卷积层输入形状上的性能可移植性问题。

在这种背景下，开发一个既能自动适应不同输入形状，又能在各种硬件上提供高性能的 Winograd 算法变得极为重要。这不仅有助于提高算法的通用性和效率，而且还能在不同的应用和硬件环境中实现最佳性能。

2 相关原理与技术

2.1 卷积计算原理

在 CNNs 中，卷积层的作用是处理输入图像并通过卷积核提取特征。这个过程通常涉及接收一批输入图像 I，这些图像有 N 张，每张图像包含 C 个输入通道，尺寸为 $H \times W$，并以 NCHW 格式表示。同时，卷积层使用一组卷积核 $\boldsymbol{G}$，具有 K 个输出通道和 C 个输入通道，每个卷积核的尺寸为 $R \times S$，数据格式可简化为 KCRS。在进行卷积计算时，特别是在前向传播过程中，当 N 等于 1 时，输出图像 O 数据格式可简化为 NEFK 即，这些图像有 N 张，每张图像包含 C 个输入通道，尺寸为 $E \times F$，其中的每个元素都是通过特定的计算过程得到的如式 1，具体计算过程如图 1。

$$\boldsymbol{O}_{k,x,y} = \sum_{c=1}^{C} \sum_{u=1}^{R} \sum_{v=1}^{S} \boldsymbol{I}_{c,x+u,y+v} \cdot \boldsymbol{G}_{k,c,u,v} \tag{1}$$

其中，$\boldsymbol{O}$ 为输出张量，$\boldsymbol{I}$ 为输入张量，$\boldsymbol{G}$ 为卷积核。

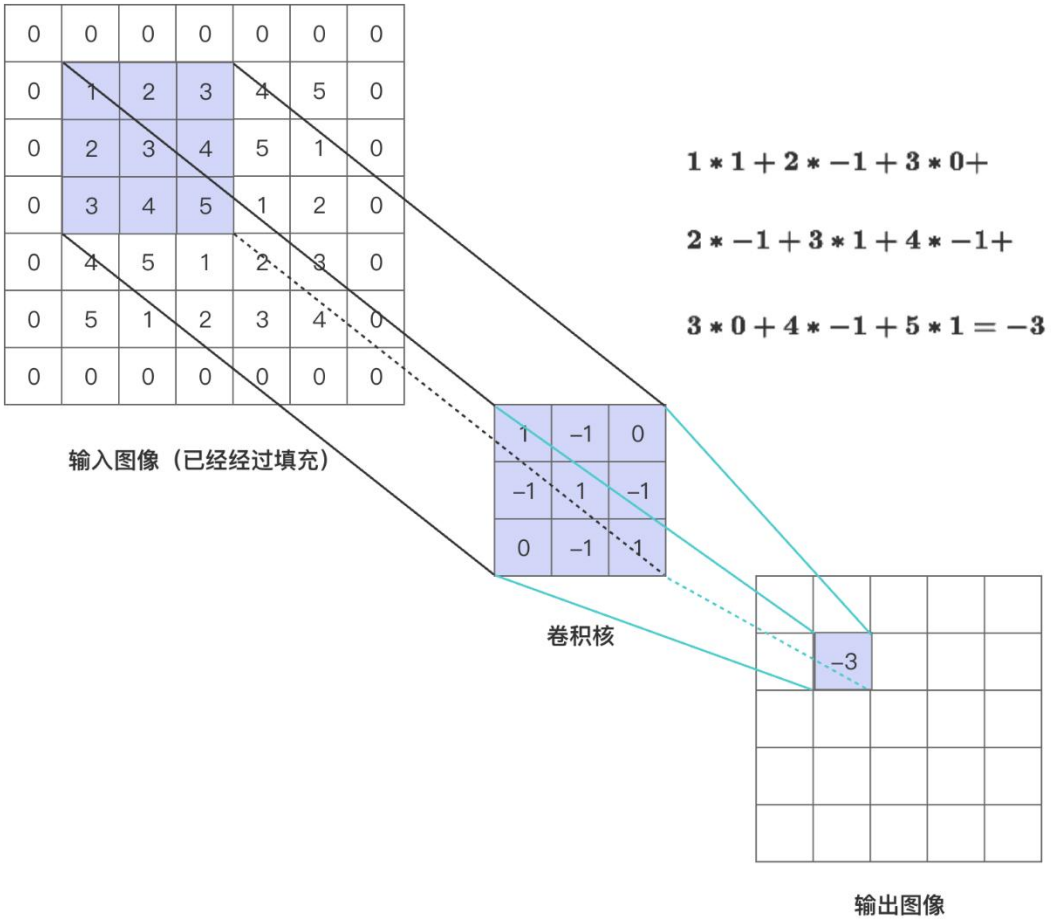


图 1　卷积计算过程

Fig. 1　The process of convolution

这个计算过程包括在输入图像 I 中寻找与卷积核尺寸 $R\times S$ 相匹配的区域，也就是局部感受野。接着，执行卷积计算，即将输入图像中的每个局部感受野中的元素与卷积核逐元素相乘，并对结果求和。这个累加的结果就形成了输出图像 O 中对应位置的元素值。卷积核在输入图像上的滑动遵循从左至右、从上至下的顺序，每次滑动的距离被称为步长(stride)。当卷积核从输入图像的左上角滑动到右下角，完成一次完整的遍历后，就生成了一张输出图像。这个过程可以用公式 1 表示。

通过这种方式，卷积层能够从输入图像中提取有用的特征，并将它们传递到神经网络的后续层。

2.2 Winograd 快速卷积算法原理

1980 年，Winograd 提出了一种面向有限脉冲响应(FIR)滤波器的最小乘法算法[5]。该算法指出，由 r 拍的 FIR 滤波器生成 m 个输出，即 $F(m, r)$，所需的最少乘法数量为 $\mu(F(m, r))=m+r-1$。以 $F(2, 3)$为例，所需的乘法数量从原本的 6 次减少到 4 次，即 $\mu(F(2, 3))=2+3-1=4$ 次。

2015 年，这种 Winograd 最小滤波算法首次被应用于 CNNs 中[4]，目的是减少卷积运算中的乘法次数，从而提升算子性能。如果用矩阵的形式表示 Winograd 最小滤波算法，其公式为式 2：

$$\boldsymbol{O}=\boldsymbol{A}^{\mathrm{T}}\left[\left(\boldsymbol{G}\boldsymbol{g}\right)\odot\left(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}\right)\right] \tag{2}$$

其中，$\boldsymbol{g}$ 为卷积核，$\boldsymbol{d}$ 为输入，$\boldsymbol{O}$ 为输出，$\boldsymbol{G}$ 表示卷积核变换矩阵，$\boldsymbol{B}^{\mathrm{T}}$ 表示输入变换矩阵，⊙表示矩阵的逐元素乘法(哈达玛积)，$\boldsymbol{A}^{\mathrm{T}}$ 表示输出变换矩阵。其中矩阵 $\boldsymbol{A}$，$\boldsymbol{B}$ 和 $\boldsymbol{G}$，都是根据 $\boldsymbol{d}$ 和 $\boldsymbol{g}$ 的维度，推导出来的常数矩阵，通过嵌套一维最小滤波算法 F(m, r)，可以得到二维的最小滤波算法 F(m×m, r×r)的公式 3 ：

$$\boldsymbol{O}=\boldsymbol{A}^{\mathrm{T}}\left[\left(\boldsymbol{G}\boldsymbol{g}\boldsymbol{G}^{\mathrm{T}}\right)\odot\left(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}\boldsymbol{B}\right)\right]\boldsymbol{A} \tag{3}$$

在二维最小滤波算法中，所需的乘法数量为(m+r-1)^2，相比于传统卷积算法需要的 m×m×r×r 次乘法。以 F(2×2, 3×3)而言，乘法次数从 36 降低到了 16 次，减少了 2.25 倍。

根据式 3 的计算过程，可以将 Winograd 算法的实现拆分成分为四个独立的阶段：输入变换、卷积核变换、哈达玛积和输出变换，如图 2 所示是一个 F(2×2, 3×3)的完整实现过程，图 2 中展示的只是单输入通道的情况。

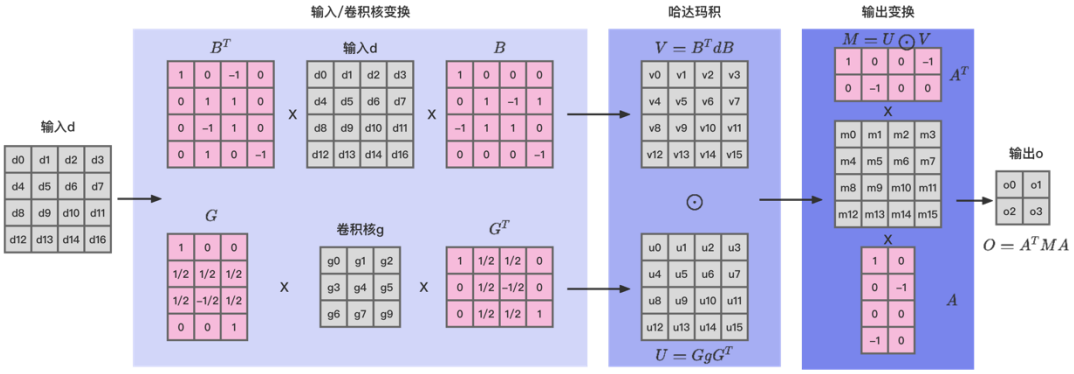


图 2　Winograd F(2×2, 3×3)计算过程

Fig. 2　The process of Winograd F(2×2, 3×3)

当输入通道大于 1 时，则需要在第三步哈达玛积后将每个输入通道上的结果进行求和，因此可以将这一步转换成批量矩阵乘法(BatchGEMM)[6]或者张量矩阵乘法(TensorGEMM)[7]，两种不同矩阵乘法的区别见图 3，可以看到 TensorGEMM 相比与 BatchGEMM 的核心区别就是将 BatchGEMM 中原来的 Batch 维度放入 GEMM 运算中最内层并进行向量化运算，不再是单个元素间的运算，而是向量间运算。

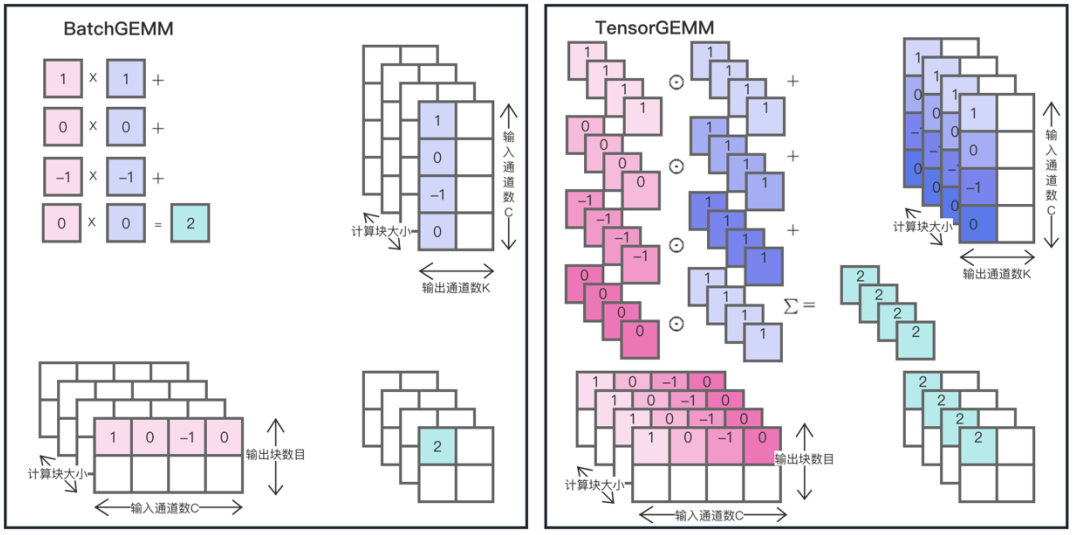


图 3　BatchGEMM 和 TensorGEMM 过程

Fig. 3　The process of BatchGEMM and TensorGEMM

2.3 深度学习编译器原理

随着深度学习技术的快速发展，模型变得越来越复杂，同时应用场景也越来越多样化。这些模型需要在各种硬件设备上高效运行，包括但不限 CPU、GPU、FPGA 和专用的神经网络处理器(NPU)。每种硬件都有其独特的特性和优化需求。

在这样的背景下，深度学习编译器应运而生。它的主要目的是为了简化和优化模型在不同硬件平台上的部署和执行过程。传统上，深度学习模型是针对特定的硬件和框架手动优化的，这种方法不仅耗时耗力，而且难以适应迅速发展的多样化硬件环境。

深度学习编译器通过提供一个统一的转换和优化流程，使得同一个模型可以自动适配和优化到多种硬件上。它接收由各种不同训练框架（如 TensorFlow[8]、PyTorch[9]等）生成的模型，然后将这些模型转换成一种统一的中间表示（通常是计算图或 Graph IR）。这个计算图是对模型结构和运算的一种高级抽象，它能够被不同的硬件平台理解和执行。

接下来，深度学习编译器会对这个计算图进行优化，包括但不限于操作融合、内存优化和硬件特定的调整。这个过程类似于传统编译器中的代码优化阶段。

最后，编译器将优化后的计算图转换为针对特定硬件的执行代码，确保在不同的硬件平台上都能实现高效的运行。当前发展前景较好的工作主要有 Tiramisu[10]、Halide[11]、MLIR[12]和 TVM[13]。

TVM，作为首个全面的深度学习自动编译与代码生成解决方案，极大地简化了将高级框架（如 TensorFlow、PyTorch）开发的深度学习网络高效部署到各种硬件后端（包括 CPU、GPU 及 FPGA 加速器）的过程。TVM 的设计巧妙地融合了内存访问、线程模式与新兴硬件元语，构建了广阔的搜索空间以纳入各种可能的手工优化，从而快速生成优化的部署代码。这使 TVM 在性能上能媲美甚至超越主流硬件供应商的库，并且具备适应新兴专用加速器后端的能力。

虽然，通过 TVM 可以简化优化难度，降低多设备支持的成本，扩大优化空间。但同时还存在一个“最后一公里”问题：我们往往希望编译过程可以自动、无干预的进行，这就需要 TVM 可以对深度学习网络进行自动优化。无需手动指定优化方式，就可以自动地将调度原语应用到给定的深度学习网络中，生成高性能的代码。Auto-tuning 模块提供了一种解决上述问题的方法。

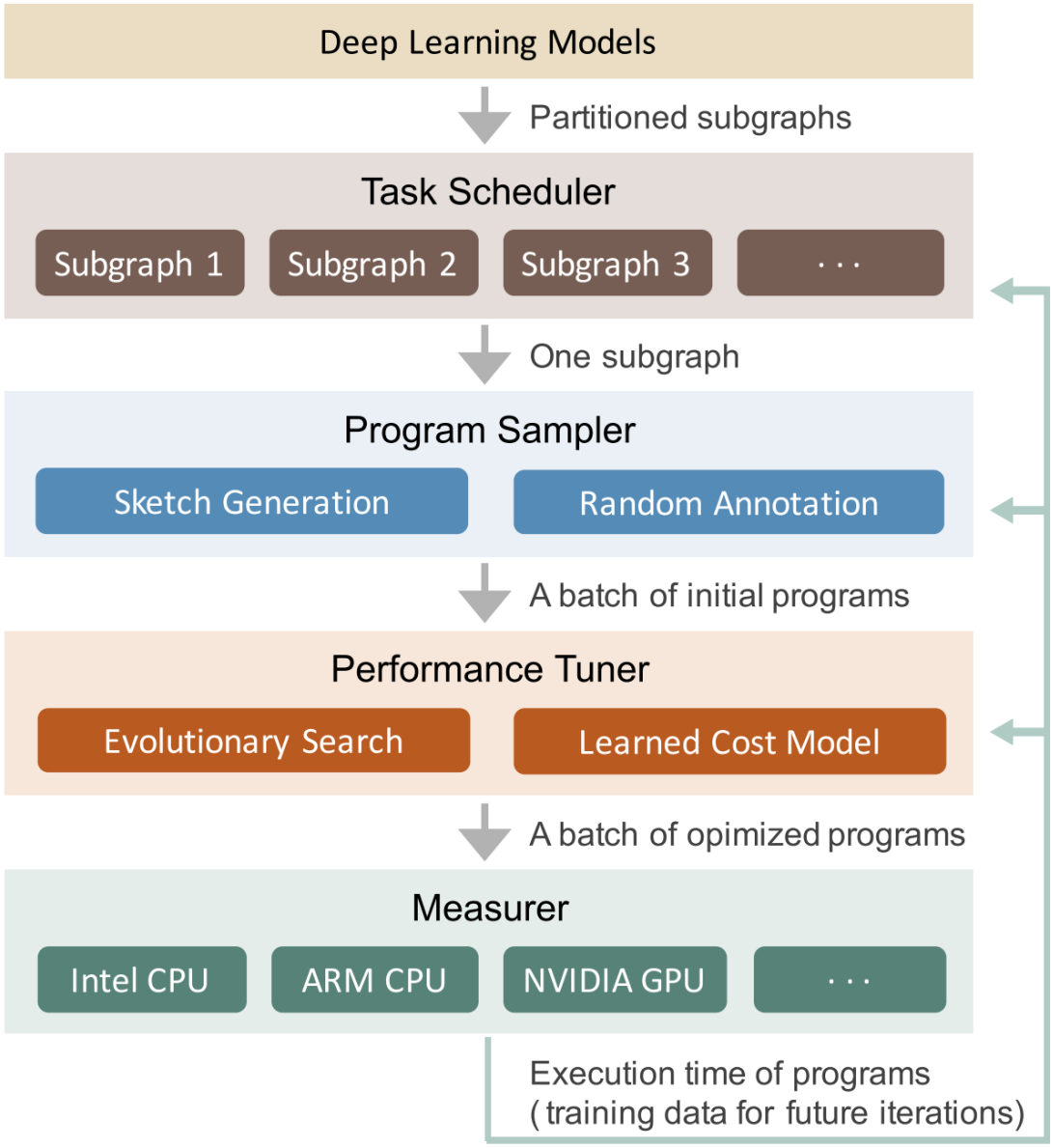
2018 年，陈天奇等人进一步推出了第一代 Auto-tuning 模块 AutoTVM[14]，这是一个利用机器学习优化底层算子编译的高级框架。AutoTVM 通过机器学习技术精准预测程序空

间中各种配置的代价，从而在庞大的搜索空间中高效地选取最优实现。它的出色之处在于模型的可迁移性，即通过历史优化记录预测新目标的代价，显著缩短搜索时间。

接着郑怜悯等人开发了 Ansor[15]，解决了如何自动化构造大规模搜索空间及如何提高搜索效率的两大核心问题。Ansor 通过层次化搜索空间和增加采样的策略，显著提升了搜索效率。它将深度学习模型拆分成小子图，并为每个子图生成高效的张量程序。Ansor 的架构分为程序采样器、性能调整器和任务调度器，共同作用于优化深度学习网络的各个子图，实现高效且精确的深度学习编译。图 4 是 Ansor 的整体架构。

图 4　Ansor 架构图

Fig. 4　The architecture of Ansor



3 基于 TVM 的 Winograd 自动性能优化方法

3.1 实验动机

在进行深度学习模型的性能优化时，尤其是对卷积运算的优化，我们通常从算法和实现两个层面进行思考和改进。算法层面的优化旨在通过精心设计的算法减少计算量，提高效率。因此，选择 Winograd 算法就是为了减少卷积运算中的乘法操作次数，从而理论上降低了运算量，这对于加速卷积网络的运算有重要意义。

在实现层面，优化的核心是确保算法能够高效地在具体的硬件上执行。这就要求我们深入理解和考量硬件架构的特点，包括处理器的计算能力、内存的访问速度和带宽等。同时，我们还需要针对算法的计算过程进行优化，比如调整数据的存储和访问模式，使之更好地适应硬件的特性，例如通过数据重用减少内存访问次数，或者通过并行计算充分利用多核处理器的计算资源。

此外，算法和硬件之间的协同优化也非常关键。通过理解算法的计算特性和硬件的执行特点，我们可以调整和优化算法的计算流程，使之更好地匹配硬件架构，如调整计算顺序以降低缓存失效，或者优化线程分配策略以提高并行度。这种协同优化能够使算法在特定硬件上达到最佳性能，充分发挥出硬件的计算潜力。

总的来说，通过在算法层面选择高效的计算方法，以及在实现层面针对特定硬件特性进行细致的优化，我们可以显著提高卷积运算甚至整个深度学习模型的运行效率。

一位精通高性能计算的工程师，通过细致的手工优化，一定能够达到近乎极致的性能，但其付出的时间人力成本也是巨大的。因此在当今复杂多变的计算领域，自动性能优化揭示了其不容忽视的价值。在这个计算资源丰富的时代，我们完全有潜力利用这些资源来应对所有架构和工作负载的挑战。

基于此我们提出了基于 TVM 的 Winograd 自动性能优化方法，通过精心设计我们的初始 Winograd 模版，利用 TVM 框架的 Auto-tuning 功能对 Winograd 算法进行精细的自动性能优化。这使得 Winograd 算法在不同设备上都能通过 Auto-tuning 获得优异的性能表现。这得益于 TVM 的设计巧妙地融合了内存访问、线程模式与新兴硬件元语，构建了广阔的搜索空间以纳入各种可能的手工优化。针对不同 CPU 架构，我们只需给定目标架构 Target，TVM 就会去根据目标架构 Target 选择相应的搜索空间进行搜索，例如在 x86 架构下会使用 AVX512 向量化指令，在 ARM 架构下则会使用 Neon 向量化指令。还可以通过指定线程模式来选择最大使用的线程数量，来提高算法的并行度。通过这种方式，我们不仅可以显著提升深度学习中核心卷积计算的效率，而且还确保了算法在各种硬件环境下都能达到最佳性能，从而为深度学习应用的发展提供了强大的计算支持。

3.2 实验方法

3.2.1 数据排布的选择

目前流行的深度学习框架中，数据一般都是 4 维的，因此有很多种不同的数据布局，典型的是像 Caffe[15]和 Pytorch 这样的 NCHW 的布局以及像 TensorFlow 这样的 NHWC 布局。它们的逻辑存储和物理存储如图 5，虽然存储的数据实际上是相同的，但是不同的顺序会导致数据访问特性不一致，所以即使进行相同的操作 ，相应的计算性能也会不同。

在 NCHW 格式中，通道数（C）是紧接在批量大小（N）之后的。这意味着，在进行卷积操作时，需要跨越通道维度来访问内存。这可能导致内存访问的不连续性，因为内存地址可能会在不同的通道之间跳跃。这种不连续的内存访问可能会导致缓存未命中和内存访问延迟，从而降低程序的性能。

相比之下，NHWC 格式将通道数（C）放在最后。这意味着，在进行卷积操作时，可以连续地访问内存，因为内存地址会按照高度和宽度的顺序连续变化。这种连续的内存访问可以提高缓存的命中率和内存访问的效率，从而提高程序的性能。

因此我们的 Winograd 模版是基于 NHWC 格式的，后续实验部分 4.2.1 我们也会针对这两种数据格式进行一个简单的分析以证明我们的结论。

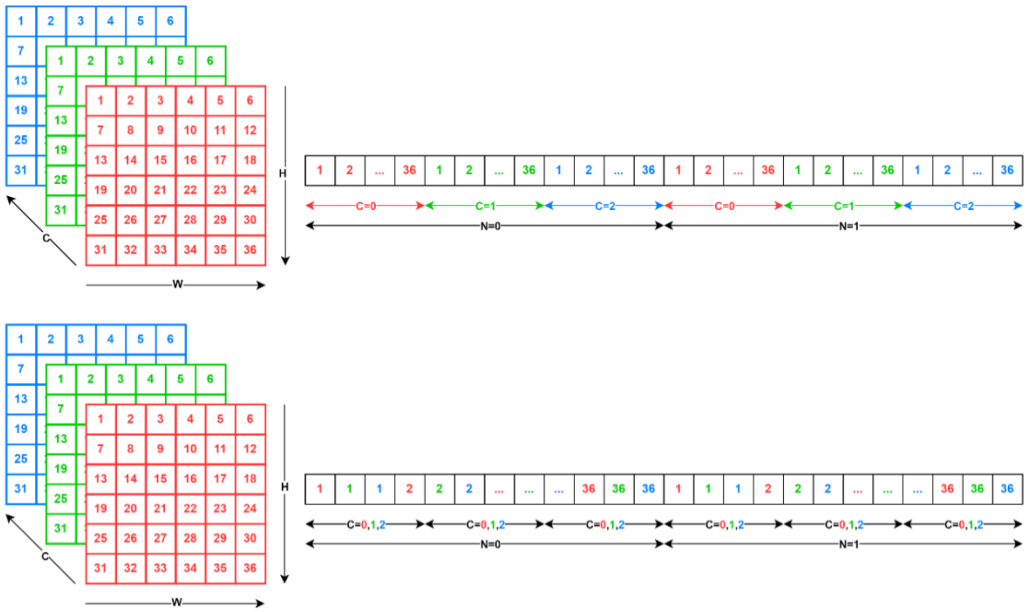


图 5　NCHW 和 NHWC 的逻辑和物理存储

Fig. 5　The logical and physical storage of the NCHW and NHWC

3.2.2 三个流程的 GEMM 实现方式选择

如第二章所述，Winograd 整个算法可以分成 4 个阶段，分别是输入变换、卷积核变换、哈达玛积及输出变换。通常在卷积神经网络推理过程中，卷积核是已知且恒定的，因此，我们可以提前算好卷积核变换后的张量，另外，哈达玛积通常实现 GEMM 可以有两种选择分别是 BatchGEMM 和 TensorGEMM，它们的计算过程区别见图 3。结合我们对输入数据排布选择的是 NHWC 格式，在进行 Winograd 算法的第三步哈达玛积时需要在输入通道上进行累加，为了保证最小的数据移动开销，我们将第三阶段的哈达玛积转换成 BatchGEMM 的实现，保证了输入通道数无论是在输入变换过程中还是输出变换过程中始终在最内维。由于中间采用了 BatchGEMM 实现，那么前后的输入变换和输出变换我们采用 TensorGEMM 实现，这是因为，哈达玛积和输入输出变换过程中的累加维度并不在同一维度上，哈达玛积累加维度在输入通道这一维，而输入输出变换的累加维度是常数矩阵的宽这一维，对应到输入即 NHWC 中 HW 所在的维度，如果输入输出变换和哈达玛积使用同一种 GEMM 实现方式就势必会引入一个新的数据排布变换带来的开销，因此，最好的实现方式就是在输入变换、哈达玛积、输出变换三个过程中交错使用不同的 GEMM 实现方式，比如我们 Winograd 模版所采用的 TensorGEMM+BatchGEMM+TensorGEMM 实现。当然还有另一种 BatchGEMM+TensorGEMM+BatchGEMM 实现，但是这种并不适合 NHWC 的输入格式，因为在进行 TensorGEMM 运算时输入通道并不在最内维，所以输入输出变换过程中还会引入一个数据排布变换的开销。我们 Winograd 模版的完整计算如下式 4—8，其中式 4、5 对应输入变换的两个 TensorGEMM，7、8 对应输出变换的两个 TensorGEMM，式 6 对应哈达玛积转换成的 BatchGEMM，通过这 5 个式子可以看到，在整个计算流程中，我们只有 5 个计算，刚好对应 Winograd 流程的最小执行步骤数，另外在整个计算过程中我们的数据排布基本保持不变，省去了不必要的数据排布变换开销。

$$\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}[ts][ts][TN][C]=\boldsymbol{B}^{\mathrm{T}}[ts][ts\_r]\times\boldsymbol{d}[N][C][H][W] \tag{4}$$

$$\boldsymbol{V}=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}\boldsymbol{B}[ts][ts][TN][C]=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{d}[ts][ts\_r][TN] \tag{5}$$

$$\boldsymbol{M}[ts][ts][TN][K]=\boldsymbol{V}[ts][ts][TN][C\_r]\times\boldsymbol{U}[ts][ts][C\_r][K] \tag{6}$$

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{M}[m][ts][TN][K]=\boldsymbol{A}^{\mathrm{T}}[m][ts\_r]\times\boldsymbol{M}[ts\_r][ts][TN][K] \tag{7}$$

$$\boldsymbol{O}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{A}[N][E][F][K]=\boldsymbol{B}^{\mathrm{T}}[ts][ts\_r]\times\boldsymbol{d}[N][C][H][W] \tag{8}$$

其中，***d*** 为输入张量，数据格式为 NHWC，*N* 为输入批量大小，*C* 为输入通道数，*HW* 为输入长宽。***O*** 为输出张量，数据格式为 NEFK，*K* 为输出通道大小，*EF* 为输出长宽。***U*** 为提前计算好的卷积核变换后的张量，数据格式为 ts ts CK 。*ts* 是 Winograd 计算过程中每次切出来的矩阵长宽大小，在 F(*m*×*m*, *r*×*r*)中，*ts=m+r-1*，*TN* 是 Winograd 计算过程中每次切出来的矩阵数目，在 F(*m*×*m*, *r*×*r*)中，*TN*=(*E*/*m*)×(*F*/*m*)，所有带有*_r* 的参数是指在这个维度进行 GEMM 的累加求和，我们这里假设了 *N* 为 1，*Pad* 为 1，*Stride* 为 1。

3.2.3 不同参数模版的动态选择

Winograd 参数的通用表示为 F(*m*×*m*, *r*×*r*)意味着我们可以有多种不同的参数选择，对于 *r*×*r* 目前 CNNs 最常见的是 3×3，另外还有 5×5，1×1 等，当下常见 CNNs 网络中不同卷积核大小占比见下表 1，当 1×1 时 Winograd 没有加速效果，因此这里我们主要考虑 3×3。另外可以看到 VGG-16 中全是 3×3 的卷积核，所以使用 Winograd 算法后，VGG-16 网络应该有更好的加速效果，后续实验主要选择 VGG-16 网络。

表 1　常见 CNN 网络中不同卷积核占比

Table 1　The proportion of different convolution kernels in common CNN networks

CNN 常见网络	1×1 卷积核	3×3 卷积核	5×5 卷积核	其他卷积核
VGG-16[16]	0	100	0	0
ResNet-50[17]	68.5	29.6	0	1.9
Inception-V4[18]	40.9	16.1	0	43
Inception-V3[19]	43.2	17.9	3.2	35.7
GoogLeNet[20]	64.9	17.5	15.9	1.7
MobileNet-V1[21]	93.3	6.7	0	0

根据前文 Winograd 快速卷积算法原理我们可以得到在 F(*m*×*m*, *r*×*r*)不同 *m*, *r* 参数情况下的理论加速比公式 9。

$$\text{理论加速比}=\frac{m\times m\times r\times r}{(m+r-1)^2} \tag{9}$$

但是不同 *m*, *r* 也会带来输入张量的扩张以及卷积核张量的扩张，扩张的公式见式 10，式 11，这其中需要有针对的去做选择。我们这里列出常见 *m*×*m* 即切块大小在 *r*×*r*=3×3 即卷积核大小为 3×3 情况下的理论加速比及输入张量和卷积核张量的扩张倍率，见下表 2。

$$\text{输入扩张}=\frac{(m+r-1)^2}{m\times m} \tag{10}$$

$$\text{卷积核扩张}=\frac{(m+r-1)^2}{r\times r} \tag{11}$$

表 2　不同切块大小对 Wiongrad 的影响

Table 2　The impact of different tile sizes on Wiongrad

切块大小(m×m)	理论加速比	输入张量扩张	卷积核扩张

2×2	2.25×	4×	1.78×
4×4	4×	2.25×	4×
6×6	5.06×	1.78×	7.11×
8×8	5.76×	1.56×	11.11×

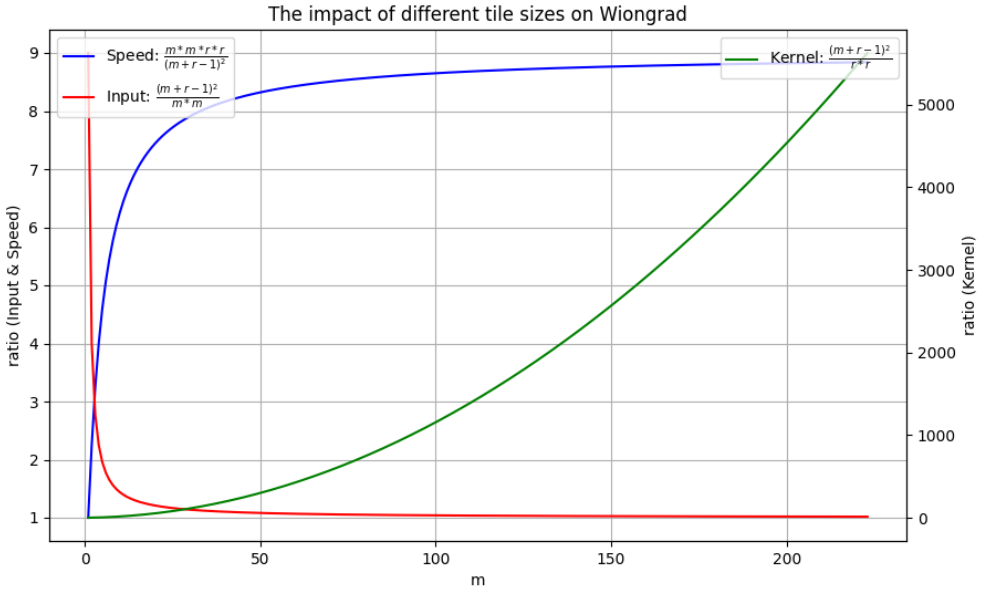
根据公式 9 可知，在 $r\times r$=3×3 的情况下当 $m>>r$ 时，理论加速比最大可接近 9 不能再大，即使输入张量基本没变，但是卷积核的扩张非常大。关于不同 m 下，理论加速比，输入张量扩张及卷积核扩张的趋势见图 6。m 在这里从 1 取到了 224，因为目前 CNNs 中输入张量常见的最大长宽为 224，比如 VGG-16 的第一层。

从图 6 可以看到并非 m 越大越好，随着 m 的增大理论加速比增大的速率急速下降，并且卷积核的扩张倍数急剧上升。对此我们首先挑选了 m=2,4,6 去实现我们的 Wiongrad 模板，这里没有考虑奇数情况是因为 Winograd 计算过程中常数矩阵的特殊性，相邻两行或两列之间存在着一些规律。但由于 m=2 时 2.25 倍的加速比被输入输出变换给掩盖了，实际情况下相比于直接卷积并没有较大优势，所以我们主要实现了 m=4 和 6 两种 Winograd 模板。

图 6　不同切块大小 m 对 Wiongrad 的影响

Fig. 6　The impact of different tile sizes on Wiongrad

根据上式 4—8，我们列出我们的模板在整个 Winograd 流程中的浮点运算次数(FLOPs)及访存次数(MACs)如下式 12—13。



$$FLOPs=\left(TN\times C\times ts^3\times 2\right)+\left(TN\times C\times K\times ts^2\right)+\left(TN\times K\times ts\times m\times\left(ts+m\right)\right) \tag{12}$$

$$
\begin{aligned}
MACs = &\left(1\times C\times H\times W + ts\times ts + 2\times ts\times ts\times TN\times C\right)+\\
&\left(ts\times ts\times TN\times C + ts\times ts + 2\times ts\times ts\times TN\times C\right)+\\
&\left(ts\times ts\times C\times K + ts\times ts\times C\times TN + 2\times ts\times ts\times K\times TN\right)+\\
&\left(ts\times ts\times K\times TN + ts\times m + 2\times m\times ts\times K\times TN\right)+\\
&\left(ts\times m\times TN\times K + ts\times m + 2\times 1\times K\times E\times F\right)
\end{aligned}
\tag{13}
$$

其中所有参数见式 4—8 定义，同样我们这里假设了 *N* 为 1，*Pad* 为 1，*Stride* 为 1。

下表 3 列出 VGG-16 中不同 *m* 对应我们模板的 FLOPs 及 MACs，作为我们选择不同参数模板的依据。

理想情况下我们的最优选择是 FLOPs 及 MACs 最小。如果存在 FLOPs 小但 MACs 大优先考虑 FLOPs。即我们的选择公式为。

$$
Cost\left(F_{f63}, M_{f63}, F_{f43}, M_{f43}\right)\begin{cases}
f63 & \text{if} \quad F_{f63} < F_{f43}\\
f43 & \text{if} \quad F_{f63} > F_{f43}\\
f63 & \text{if} \quad F_{f63} = F_{f43} \text{ and } M_{f63} < M_{f43}\\
f43 & \text{if} \quad F_{f43} = F_{f43} \text{ and } M_{f63} > M_{f43}\\
f63 / f43 & \text{if} \quad F_{f43} = F_{f43} \text{ and } M_{f63} = M_{f43}
\end{cases}
\tag{14}
$$

式中F_{f63}就是 F(6×6, 3×3)下对应的 FLOPs，F_{f43}是 F(4×4, 3×3)下对应的 FLOPs，M_{f63}是 F(6×6, 3×3)下对应的 MACs，M_{f43}是 F(4×4, 3×3)下对应的 MACs。

表 3 VGG-16 中不同切块大小 *m* 对 FLOPs 及 MACs 的影响

Table 3 The impact of different tile sizes m on FLOPs and MACs in VGG-16

C	K	H/W	F63-FLOPs	F43-FLOPs	F63-MACs	F43-MACs
3	64	224	168.57	**147.82**	**39.30**	44.74
64	64	224	**1070.55**	1194.59	**76.44**	89.26
64	128	112	**487.96**	553.94	**28.94**	33.21
128	128	112	**913.81**	1059.72	**39.14**	45.15
128	256	56	**480.05**	508.18	**17.62**	17.64
256	256	56	**925.70**	992.28	25.04	**24.64**
256	512	28	**449.74**	485.30	16.15	**12.95**
512	512	28	**882.28**	958.56	27.20	**20.58**
512	512	14	317.62	**313.00**	20.40	**12.98**

基于式 14 及表 3，在 VGG-16 中第一层和最后一层我们选择 F(4×4, 3×3)的 Winograd 模板，其他层我们选择 F(6×6, 3×3)的 Winograd 模板。

4 实验分析与评估

4.1 实验平台及环境

为了证明我们工作的可迁移性，我们随机挑选了一台 x86 CPU 服务器 Intel(R) Xeon(R) Platinum 8255C 及一台 ARM CPU 服务器 Ampere(R) Altra(R) Neoverse-N1 其相关硬件特性如下表 4。

表 4　实验中使用的 CPU

Table 3　The list of CPU used in the experiment

CPU	核心数	主频(GHz)	L1 缓存(Bytes)
Intel(R) Xeon(R) Platinum 8255C	24	24@2.50	24@32K
Ampere(R) Altra(R) Neoverse-N1	35	35@2.80	35@64K
CPU	L2 缓存	L3 缓存	CPU 架构
Intel(R) Xeon(R) Platinum 8255C	24@1024K	24@35.75MB-	x86
Ampere(R) Altra(R) Neoverse-N1	35@512K	35@32MB-sha	ARM

根据表 1 可知，VGG-16 中都是 3×3 的卷积核，所以所有卷积计算都可以使用 Winograd 算法进行加速，后续实验主要使用 VGG-16 网络。VGG-16 网络中的相关参数见上表 3。

4.2 实验结果

4.2.1 数据排布对性能影响

第三章 3.2.1 节我们提到过，不同的数据排布会导致数据访问特性不一致，导致计算性能有所差异。对应我们最终使用输入数据排布为 NHWC 的 Winograd 模板，我们同样的实现了一套输入数据排布为 NCHW 的 Winograd 模板，我们在 x86 CPU 上对 VGG-16 的一些层进行一个简单测试，结果如图 7，我们这里保证了整个运算过程中的 FLOPs 是一样的，对比了 VGG-16 网络中间 4 层不同输入形状下 NCHW 和 NHWC 下数据排布的每秒浮点计算量(GLOPS)，GFLOPS 的计算方式如下式 15

$$\mathrm{GFLOPS} = \frac{\mathrm{FLOPs} \times 10^{-9}}{\mathrm{cost}} \tag{15}$$

式中 cost 是实际推理的运行时间，GFLOPS 是衡量处理器的算力和执行效能的重要参数。越高表示计算效率越高。下文所有图中纵坐标是指使用我们 BlazerML-NCHW 的 GFLOPS 除使用其他实现的 GFLOP 得来的加速比。因此所有代表 BlazerML-NCHW 的柱子都为 1，即 BlazerML-NCHW 的 GFLOPS 除自身。

从图 7 可以看出在所有输入形状下，NHWC 都优于 NCHW，因此可以验证我们的推断，输入数据 NHWC 的排布相比与 NCHW 的排布有更好的性能，所以接下来的实验，我们只使用输入数据为 NHWC 排布的 Winograd 模板。

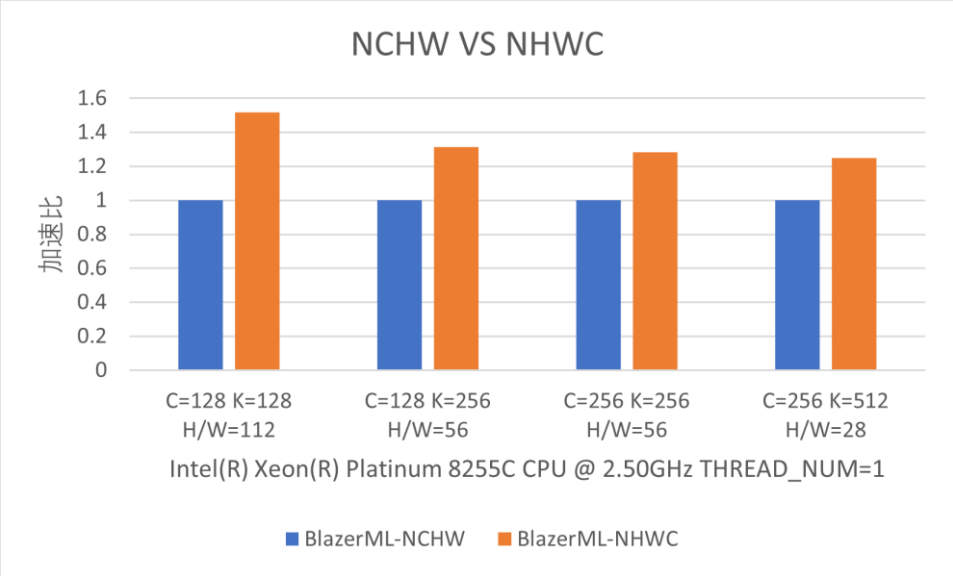


图 7　输入数据排布对 Wiongrad 的影响

Fig. 7　The impact of input data layout on Wiongrad

4.2.2 ARM CPU 上逐层性能分析

我们将我们的方法与 ARM CPU 上另外两个热门的深度学习计算库中的 Winograd 算法实现进行了性能比较，计算库包括 FastConv[22]这是一个经过专家手工调优的基于 C++模板代码自动生成的高性能开源卷积计算库，以及 ARM NN(https://github.com/ARM-software/armnn)一个专门针对 ARM 平台的高效推理引擎。我们这里跟 FastConv 论文中的提到的所有网络包括 VGG-16、ResNet-50、Inception-V4 及 Densenet-121[23]中所有卷积核大小为 3×3 的所有层进行了对比，结果如图 8。

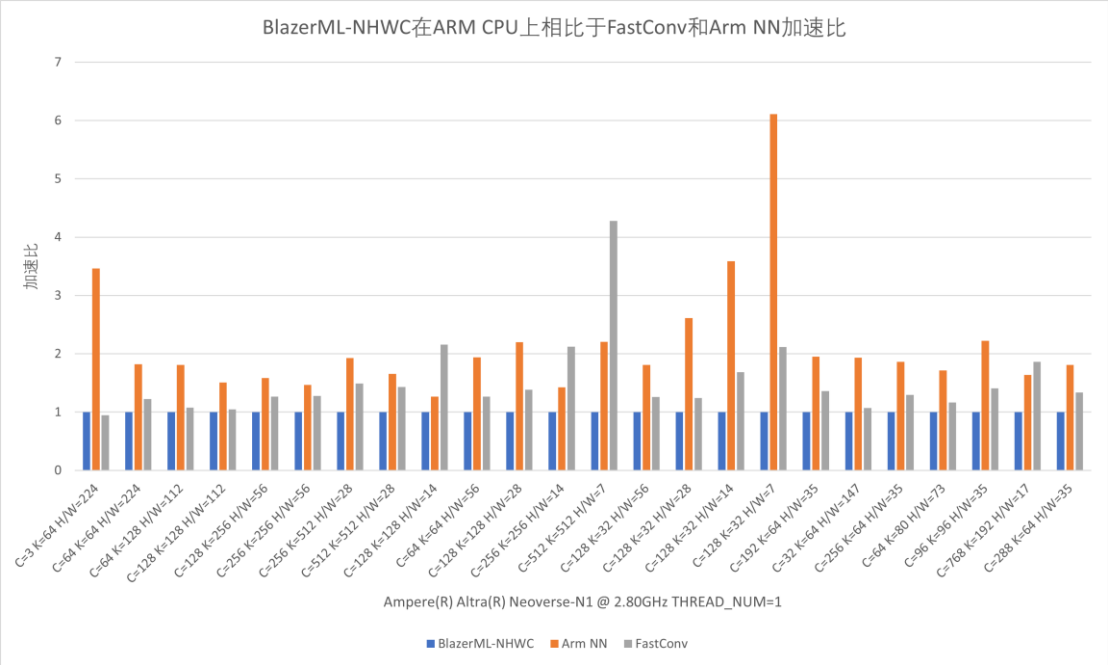


图 8　与 FastConv、ARM NN 中的 Winograd 算法进行性能比较的实验结果

Fig. 8　Speedup comparison against Winograd from FastConv, ARM NN inference engine.

从图 8 可以看到，我们的实现在 ARM CPU 上相比与 Arm NN 有 1.27 倍到 6.11 倍的加速比，相比于经过专家手工调优的 FastConv，在不考虑 VGG-16 的第一层即 C=3，K=64，H/W=224 的情况下我们有 1.04 倍到 4.28 倍的加速比，这里之所以 VGG-16 的第一层不及 FastConv 是因为 C 和 K 太小导致输入变换和输出变换过程的开销没有办法被 C 和 K 均摊掉，但 FastConv 对输入变换及输出变换在乘常数矩阵计算时，由于常数矩阵的特殊性质，做了合并同类项来减少输入变换和输出变换过程中计算量的特殊优化原因。

4.2.3 x86 CPU 上逐层性能分析

我们将我们的方法与 x86 CPU 上 SOTA(state-of-the-art)水平的 MNN[24]进行了性能比较，MNN 是由阿里巴巴开发的轻量级深度神经网络引擎。这里我们同样随机挑选了一些常见网络中的一些层，例如 VGG-16、Densenet-121 及 ResNext50[25]，结果如图 9

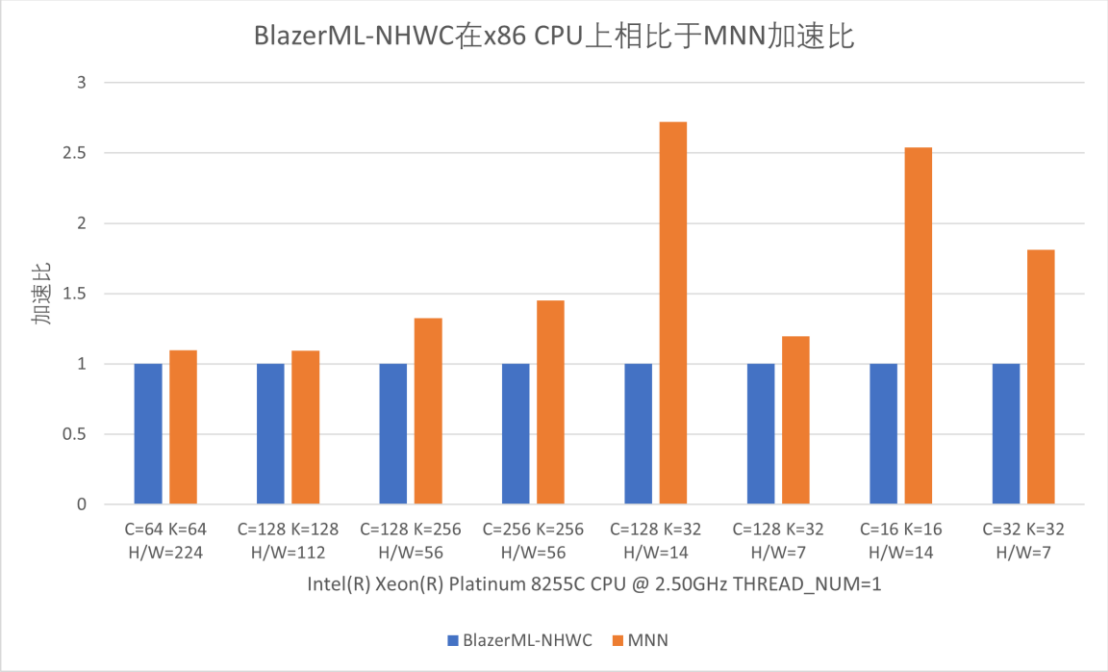


图 9　与 MNN 中的 Winograd 算法进行性能比较的实验结果

Fig. 9　Speedup comparison against Winograd implementation from MNN.

从图 9 可以看出即使是相比于当下 SOTA 的 MNN 我们在所选输入下依然有 1.10 倍到 2.72 倍的加速比。

4.2.4 x86 CPU 上全网络性能分析

为了证明我们 Winograd 实现在 CNNs 推理中的加速效果，我们也在 x86 CPU 上进行了一个全网络的性能测试，挑选了一些常见 CNN 网络如与当下一些常用推理框架进行对比，我们分别对比了 MNN，OnnxRuntime(https://github.com/microsoft/onnxruntime)和 TVM 社区版，其结果如图 10 所示，可以看到在常见 CNN 网络推理性能中，我们的 Winograd 实现都具有一定的加速比，尤其是在 VGG-16 这个全部可使用 Winograd 来加速卷积运算的网络中，我们相比与 TVM 社区版有 1.66 倍的加速比，相比于 OnnxRuntime 我们有 2.47 倍的加速比。而相比与 MNN 我们仅有 1.19 倍的加速比，这是因为 VGG-16 是公认的最适合使用 Winograd 的网络，因为其全是 3×3 的卷积核，所以各推理框架都会首选 VGG-16 去进行专业优化，这也证明了一个问题，当下对 Winograd 的优化只把重点放在了一些常见网络上，并没有考虑一些不常见网络，因为人的精力毕竟是有限的，没有办法针对每个工作负载都去做特定优化，但我们的工作解决了这一弊端，我们利用当下无限的算力去自动搜索每种工作负载的最优实现以达到任何情况下的理想性能。

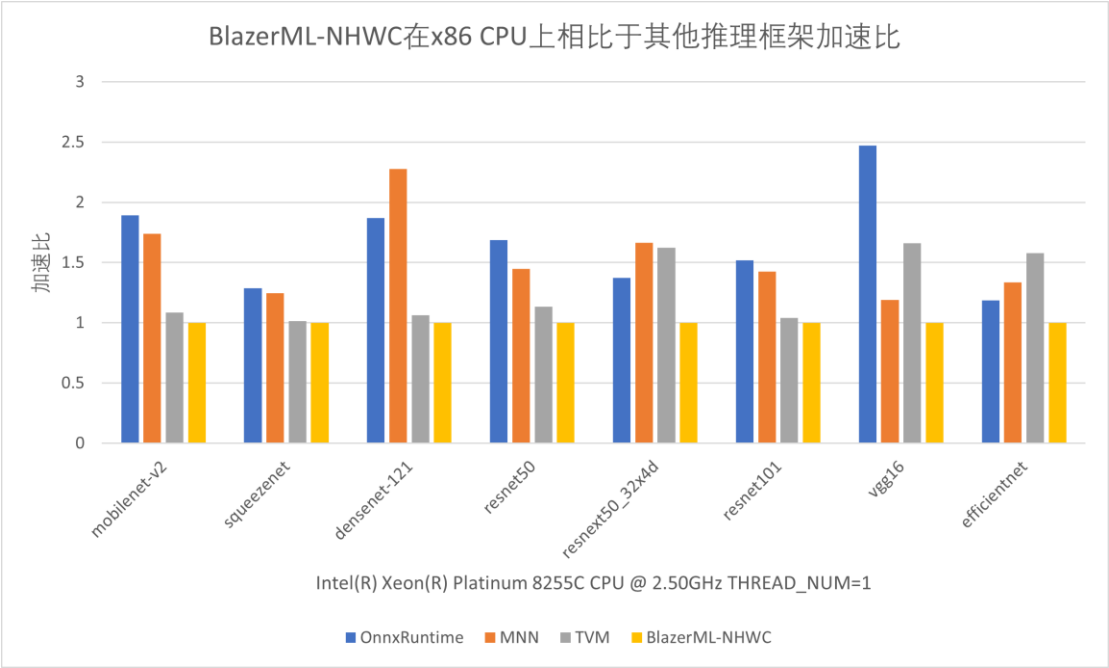


图 10　与其他推理框架进行性能比较的实验结果

Fig. 10　Experimental results comparing performance with other inference frameworks

4.2.5 x86 CPU 上可拓展性分析

之前的实验都是单线程的，为了证明我们 Winograd 实现在 CNNs 推理中多线程场景下依然具有不错的性能即可拓展性，我们在 x86 CPU 上进行了一个 VGG-16 全网络的多线程性能测试及 VGG-16 中某一层的多线程性能测试，结果如图 11 所示。

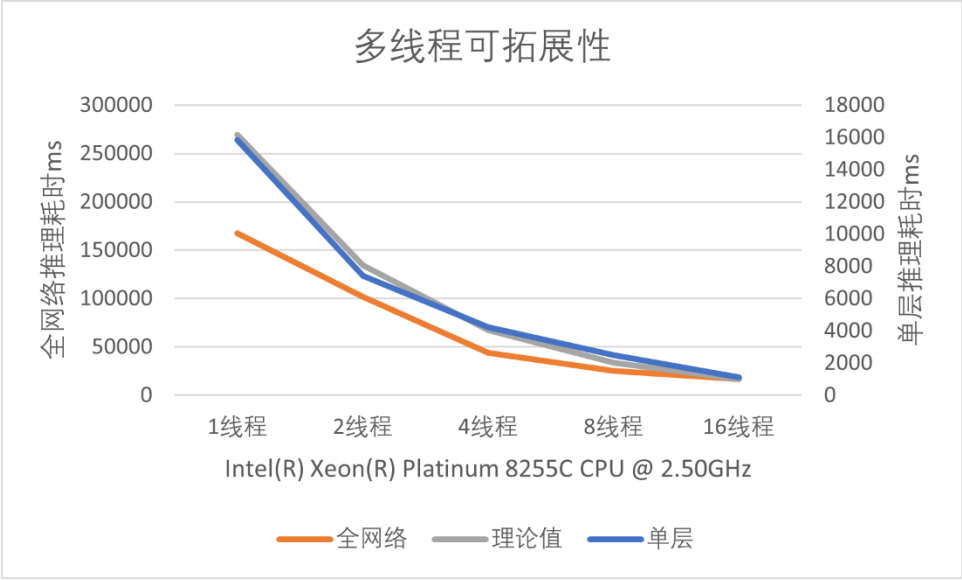


图 11　多线程可拓展性实验结果

Fig. 11　Multi-thread scalability experimental results

根据图 11 可知我们选择了最大到 16 线程，因为这台 x86 CPU 最大 24 核心，为了方便看出可拓展性我们只成倍增加线程数到 16，图中灰色的线是以 16 线程为 1，随着线程数成倍减少依次增加，即 1 线程时为 16。左边纵轴是全网络的推理耗时，右边纵轴是单层的推理耗时。可以看到单层推理耗时曲线与理论相仿，全网络推理耗时曲线变换趋势也与理论相仿，即无论是全网络还是单层，我们的 Winograd 实现都具有很好的可拓展性。

5 结论

本文提出了一种基于 TVM 的 Winograd 自动性能优化方法，通过合理选择输入数据排布，交错使用 BatchGEMM 和 TensorGEMM 设计了一个高性能的 Winograd 同样模板，并根据不同参数下计算量与访存量来动态选择不同 Winograd 模板。接着使用 TVM 的 Auto-tuning 模块来对 Winograd 算法进行精细的自动性能优化，以达到在不同硬件环境不同工作负载下都能获得优秀的性能。结果表明在 CNNs 推理时，无论是在 x86 还是 ARM CPU 上，无论是在全网络还是单层网络，我们的实现相比于开源框架都具有一定的加速比，并且在多线程场景下依然具有可拓展性。

关于 Winograd 性能的优化，依然存在着许多可以挖掘的地方，比如数据的排布不仅仅只有 NCHW 和 NHWC 两种排布，还有一些其他的数据排布。不同数据排布下又有不同的高性能实现方式，关于计算量和访存量的分析也不一样，是否有更合适的参数选择逻辑。另外我们当前的模板没有办法像手工优化一样，做到对每个变换乘常数矩阵阶段提取公因式的操作，因为受限于 Ansor 的写法，但是 TVM 的下一代 Auto-tuning 模块已经出现，我们可以做更进一步的细致优化以持续提高 Winograd 的性能，这将是我们的下一步工作。

参考文献

[1] Gu J, Wang Z, Kuen J, et al. Recent advances in convolutional neural networks[J]. Pattern recognition, 2018, 77: 354-377.

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

[3] Chetlur S, Woolley C, Vandermersch P, et al. Efficient primitives for deep learning[J]. nature, 2014, 1410.

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

[5] Winograd S. Arithmetic complexity of computations[M]. Siam, 1980.

[6] Li D, Huang D, Chen Z, et al. Optimizing massively parallel winograd convolution on arm processor[C]//Proceedings of the 50th International Conference on Parallel Processing. 2021: 1-12.

[7] Lan H, Meng J, Hundt C, et al. FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 31(3): 580-594.

[8] Abadi M, Barham P, Chen J, et al. {TensorFlow}: a system for {Large-Scale} machine learning[C]//12th USENIX symposium on operating systems design and implementation (OSDI 16). 2016: 265-283.

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

[10] Baghdadi R, Ray J, Romdhane M B, et al. Tiramisu: A polyhedral compiler for expressing fast and portable code[C]//2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2019: 193-205.

[11] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. Acm Sigplan Notices, 2013, 48(6): 519-530.
[12] Lattner C, Amini M, Bondhugula U, et al. MLIR: Scaling compiler infrastructure for domain specific computation[C]//2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2021: 2-14.
[13] 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). 2018: 578-594.
[14] Chen T, Zheng L, Yan E, et al. Learning to optimize tensor programs[J]. Advances in Neural Information Processing Systems, 2018, 31.
[15] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. 2014: 675-678.
[16] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.
[17] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 1-9.
[18] Szegedy C, Ioffe S, Vanhoucke V, et al. Inception-v4, inception-resnet and the impact of residual connections on learning[C]//Proceedings of the AAAI conference on artificial intelligence. 2017, 31(1).
[19] Xia X, Xu C, Nan B. Inception-v3 for flower classification[C]//2017 2nd international conference on image, vision and computing (ICIVC). IEEE, 2017: 783-787.
[20] Zhong Z, Jin L, Xie Z. High performance offline handwritten chinese character recognition using googlenet and directional feature maps[C]//2015 13th international conference on document analysis and recognition (ICDAR). IEEE, 2015: 846-850.
[21] Qin Z, Zhang Z, Chen X, et al. Fd-mobilenet: Improved mobilenet with a fast downsampling strategy[C]//2018 25th IEEE International Conference on Image Processing (ICIP). IEEE, 2018: 1363-1367.
[22] Meng J, Zhuang C, Chen P, et al. Automatic generation of high-performance convolution kernels on arm cpus for deep learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(11): 2885-2899.
[23] Iandola F, Moskewicz M, Karayev S, et al. Densenet: Implementing efficient convnet descriptor pyramids[J]. arXiv preprint arXiv:1404.1869, 2014.
[24] Jiang X, Wang H, Chen Y, et al. Mnn: A universal and efficient inference engine[J]. Proceedings of Machine Learning and Systems, 2020, 2: 1-13.
[25] Xie S, Girshick R, Dollár P, et al. Aggregated residual transformations for deep neural networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 1492-1500.