软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(6):2668−2686 [doi: 10.13328/j.cnki.jos.007102]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

AutoConfig: 面向深度学习编译优化的自动配置机制*

张洪滨[1,2], 周旭林[1,2], 邢明杰[2], 武延军[2], 赵 琛[2]

[1](中国科学院大学, 北京 100049)

[2](中国科学院 软件研究所, 北京 100190)

通信作者: 武延军, E-mail: yanjun@iscas.ac.cn

摘 要: 随着深度学习模型和硬件架构的快速发展, 深度学习编译器已经被广泛应用. 目前, 深度学习模型的编译优化和调优的方法主要依赖基于高性能算子库的手动调优和基于搜索的自动调优策略. 然而, 面对多变的目标算子和多种硬件平台的适配需求, 高性能算子库往往需要为各种架构进行多次重复实现. 此外, 现有的自动调优方案也面临着搜索开销大和缺乏可解释性的挑战. 为了解决上述问题, 提出 AutoConfig, 一种面向深度学习编译优化的自动配置机制. 针对不同的深度学习计算负载和特定的硬件平台, AutoConfig 可以构建具备可解释性的优化算法分析模型, 采用静态信息提取和动态开销测量的方法进行综合分析, 并基于分析结果利用可配置的代码生成技术自动完成算法选择和调优. AutoConfig 创新性地将优化分析模型与可配置的代码生成策略相结合, 不仅能保证性能加速效果, 还能减少重复开发的开销, 同时可以简化调优过程. 在此基础上, 进一步将 AutoConfig 集成到深度学习编译器 Buddy Compiler 中, 对矩阵乘法和卷积的多种优化算法建立分析模型, 并将自动配置的代码生成策略应用在多种 SIMD 硬件平台上进行评估. 实验结果可验证 AutoConfig 在代码生成策略中完成参数配置和算法选择的有效性. 与经过手动或自动优化的代码相比, 由 AutoConfig 生成的代码可达到相似的执行性能, 并且无需承担手动调优的重复实现开销和自动调优的搜索开销.

关键词: 深度学习编译器; 编译优化; 代码生成; 自动配置机制

中图法分类号: TP314

中文引用格式: 张洪滨, 周旭林, 邢明杰, 武延军, 赵琛. AutoConfig: 面向深度学习编译优化的自动配置机制. 软件学报, 2024, 35(6): 2668–2686. http://www.jos.org.cn/1000-9825/7102.htm

英文引用格式: Zhang HB, Zhou XL, Xing MJ, Wu YJ, Zhao C. AutoConfig: Automatic Configuration Mechanism for Deep Learning Compilation Optimization. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2668–2686 (in Chinese). http://www.jos.org.cn/1000-9825/7102.htm

AutoConfig: Automatic Configuration Mechanism for Deep Learning Compilation Optimization

ZHANG Hong-Bin[1,2], ZHOU Xu-Lin[1,2], XING Ming-Jie[2], WU Yan-Jun[2], ZHAO Chen[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)

[2](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Deep learning compilers have been widely employed with the rapid development of deep learning models and hardware architectures. At present, the compilation optimization and tuning methods of deep learning models mainly rely on high-performance operator libraries and automatic compiler tuning. However, facing various target operators and adaptation requirements of several hardware platforms, high-performance operator libraries should conduct multiple implementations for different architectures. Additionally, existing auto-tuning schemes face challenges in substantial search overheads and interpretability. To this end, this study proposes AutoConfig, an automatic configuration mechanism for deep learning compilation optimization. Targeting different deep learning workloads and multiple

* 基金项目: 国家重点研发计划 (2022YFB4401402)
本文由“编译技术与编译器设计”专题特约编辑冯晓兵研究员、郝丹教授、高耀清博士、左志强副教授推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-29

hardware platforms, AutoConfig builds interpretable performance analysis models, conducts a thorough assessment via static information extraction and dynamic overhead measurement, and automates algorithm selection and configuration tuning for code generation. The key innovation of this study is combining the optimization analysis model and a configurable code generation strategy, which ensures a performance acceleration effect and reduces repeated development overheads with the simplified tuning process. Furthermore, this study integrates AutoConfig into a deep learning compiler Buddy Compiler, builds analysis models for convolution and matrix multiplication optimization, and evaluates the optimization on multiple SIMD hardware platforms. Experimental results indicate that AutoConfig effectively completes parameter configuration and algorithm selection in the code generation strategy. Additionally, compared with the codes by manual or automatic optimization, the codes generated by AutoConfig can yield comparable performance without both the repeated manual tuning implementation overheads and auto-tuning search overheads.

Key words: deep learning compiler; compilation optimization; code generation; automatic configuration mechanism

深度学习在处理图像[1]、语音[2]、文本[3]等各个场景中得到了越来越广泛的应用. 为了取得更优异的表现性能, 深度学习模型的设计日益精妙和复杂. 举例而言, 早期经典的图像分类模型 AlexNet[1]有 600 万参数, 如今流行的大语言模型 ChatGPT[4]的参数量已经达到 1 750 亿, 参数量的飞跃式增长为深度学习模型在硬件设备中高效落地带来重要挑战.

目前支持深度学习模型落地主要有两类技术. 一类是通过开发高性能算子库来加速模型[5−7]. 这些算子库通常使用编译器内建函数或汇编级别指令来手动实现算子的核心逻辑, 可以在特定的计算场景中提供充分的算子优化机会. 然而要在大部分计算场景中都获得较好的性能, 需要在算子库中为不同规模的算子输入重复编写程序逻辑, 这个开发流程费时费力. 而且程序逻辑中对优化算法和优化参数的选择也与硬件平台强相关, 这使得一套程序逻辑无法被不同的硬件平台直接复用, 因此这类技术依赖于手动调优, 通常不具有跨平台的普适性.

另一类技术是构建支持深度学习模型的编译优化框架. 这些框架的优化流程与处理高级语言程序的传统编译器 (如 LLVM[8]) 有共通之处, 它们将深度学习模型的落地视作一个编译优化的过程: 首先采用某种高层抽象的语言来表示模型, 然后将高层抽象逐步下降到硬件级别中间表示, 同时在下降的不同阶段结合多种策略对模型进行优化. 与调用高性能算子库相比, 深度学习编译优化技术能够从全局的角度发掘模型的优化空间, 发挥跨硬件平台做自动调优的潜力. 然而现有深度学习编译优化框架的自动调优方式通常基于搜索策略. 当面对程序逻辑复杂且具有大量调优参数的算法时, 庞大的搜索空间会导致调优过程的开销变得难以承受, 而且参数选取的过程也不具备可解释性.

针对上述问题, 本文提出了 AutoConfig, 一种面向深度学习编译优化的自动配置机制. AutoConfig 由重写模式 (rewrite pattern) 和优化分析模型两个部分组成. 重写模式提供可配置的代码生成策略, 以驱动优化分析模型进行参数选取和算法选择. 优化分析模型可以根据硬件平台信息确定调优参数范围, 并使用基准程序从计算、访存和特殊指令开销等多个方面量化模型中的权重系数, 以指导优化算法的选择. 通过综合考虑计算负载特点和硬件特征等要素, AutoConfig 能够完成调优参数选取、优化算法选择和自动代码生成, 只需一次优化实现就能适配多种硬件平台. 本文将 AutoConfig 集成在深度学习编译器 Buddy Compiler[9]中, 并针对实际的深度学习计算负载场景, 在多种 SIMD 平台上对 AutoConfig 的参数配置及优化算法的选择进行探究, 并将基于优化分析模型的代码生成策略与 TVM[10]的自动调优策略进行跨平台的性能比较. 实验结果验证了 AutoConfig 作为全新的深度学习编译优化开发范式的实用性和有效性.

本文第 1 节介绍相关工作. 第 2 节介绍基础知识, 包括深度学习计算负载及其优化方法、深度学习编译优化的基本流程和 AutoConfig 的实现所依托的编译基础设施. 第 3 节介绍 AutoConfig 的设计思路、模块组成、使用方式和生态集成. 第 4 节介绍 AutoConfig 中的优化分析模型, 并以矩阵乘法和卷积为例, 针对不同的优化算法进行建模和分析. 第 5 节介绍 AutoConfig 中的静态信息提取和动态开销测量策略. 第 6 节呈现实验设计、结果和分析. 第 7 节给出结论.

1 相关工作

在深度学习模型诞生的早期, 人们通过开发高性能算子库来加速模型的推理过程. cuDNN[5]是一个由 NVIDIA 开发的深度学习加速库, 它提供了一系列深度学习计算负载实现, 采用启发式方法为负载实现选择最佳的算法, 并使用 CUDA 技术来加速, 可以在 NVIDIA GPU 上实现高效的深度学习推理. oneDNN[6]是由 Intel 开发的用于深度神经网络的数学内核库, 它可以发挥 Intel 处理器体系结构和多核处理器的优势, 实现深度学习计算负载的高性能

计算. MIOpen[7]是由 AMD 开发的数学和科学计算库, 专为深度学习在 AMD 设备上的并行计算而设计. 高性能算子库的开发通常针对某类特殊硬件, 实现上一般依赖手工调优. 这种调优方式尽管在某些具体场景能取得良好的效果, 但是不具备跨平台迁移的能力.

近年来, 随着摩尔定律趋于终结, 为了应对深度学习领域对高算力的需求, 越来越多计算场景引入了 FPGA、CGRA、TPU[11]等深度神经网络加速器进行加速. 为了将深度学习模型的计算模式高效映射到多样的硬件设备上, 基于深度学习编译优化的框架设计成为了研究重点. TVM 是目前工业界中较为常用的深度学习编译框架, 它将模型优化的过程视为对张量程序的变换, 可借助 autoTVM[12]、Ansor[13]等基于代价模型和启发式算法的搜索策略优化算子执行逻辑. 但是在面对程序逻辑复杂、调优参数众多的算法时, 庞大的搜索空间会使调优开销变得不可承受, 且调优方式也不具备理论上的可解释性. 除了以搜索策略为主的编译框架, 还有其他的工作如 ROLLER[14]和 SparTA[15]. ROLLER 可基于硬件平台完成张量程序的分块调优, 并采用递归算法实现代码生成, SparTA 则提出了一种充分利用稀疏性的端到端模型优化通路. 这两种框架通过针对某种特定优化方式或利用数据特性来从新的视角发掘优化的机会, 但对于其他通常情况则存在应用的局限.

2 基础知识

本节首先以卷积和矩阵乘法为例介绍深度学习模型计算负载及其优化方法, 然后概述深度学习编译优化的基本流程, 最后介绍 AutoConfig 的实现所依托的编译基础设施 MLIR[16].

2.1 深度学习计算负载及其优化方法

2.1.1 矩阵乘法及其优化方法

近年来, 随着大语言模型的迅速发展, Transformer[17]已成为自然语言处理领域中极具影响力的经典模型. 注意力机制是 Transformer 中的关键模块, 它能够有效处理长序列和目标序列中的依赖关系. 该模块通过计算序列中不同位置之间的相似度, 并为每个位置分配一个权重系数, 从而实现对不同位置的加权关注. 在具体实现中, 需要首先计算查询向量和键向量之间的相似度, 然后将相似度与值向量相乘并加权求和, 最终得到加权后的值向量. 因为此类计算模式在注意力机制中广泛存在, 所以优化矩阵乘法是高效执行 Transformer 模型的关键[18].

矩阵乘法的一种在算法级别的优化方法是 Strassen 方法[19], 它通过减少乘法操作的次数来降低计算的理论时间复杂度. 在工业界中, 矩阵乘法的优化策略主要集中在提高计算效率和减少资源消耗上, 包括利用多线程和多核处理器做并行计算、将大矩阵划分成小块做分块计算、优化矩阵在内存中的排布来减少缓存未命中次数等高性能优化方法.

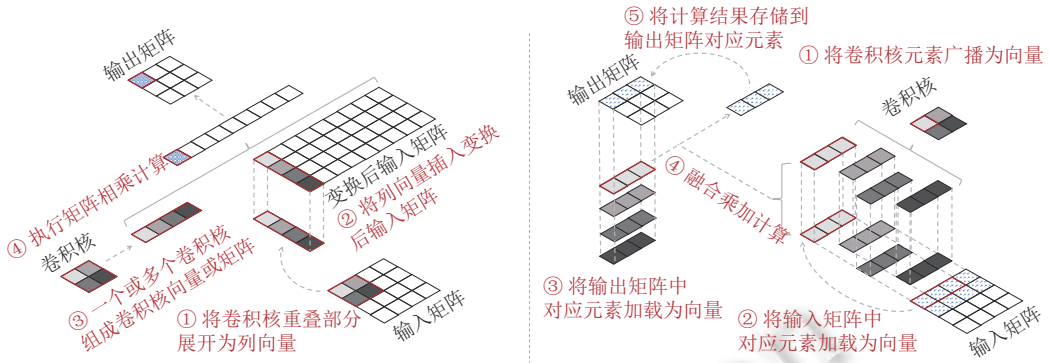
2.1.2 卷积层及其优化方法

卷积神经网络是在计算机视觉领域中广泛使用的模型. 它通过卷积模块来提取图像的局部特征. 卷积层是卷积模块的核心, 它通过在输入数据上滑动一个固定大小的卷积核来提取图像的局部信息. 卷积层中可以包含多个卷积核, 每个卷积核可以提取不同的特征. 在深度学习模型中, 卷积还有空洞卷积、深度可分离卷积等不同的变种, 它们根据输入的张量表示也可分为“NHWC”和“NCHW”等不同的数据排列方式. 不同的卷积类型、输入尺寸和卷积核大小、数据排列方式会影响计算负载优化的策略.

卷积的优化方法有很多种, 包括从矩阵乘法的视角看待卷积[20]并应用 Strassen 方法做优化、基于 FFT 和 Winagrad 的变换方法[21]和 Im2Col 算法[22]等. 其中 Im2Col 算法是目前最通用的优化算法之一, 其主要思想是先将卷积核重叠部分展开和变换输入矩阵, 然后利用高性能的矩阵乘法实现来加速卷积运算, 如图 1(a) 所示. 需要注意的是变换的开销由于涉及到访存, 因此不可忽略.

2.1.3 Broadcast 向量化算法

在执行卷积和矩阵乘法等计算负载时, 可以通过改变原始算法的操作顺序来换取更好的访存模式, 以为向量化提供充分的空间. 本文采用了一种基于广播操作的优化算法 (Broadcast 算法), 它的计算逻辑是从一个矩阵中顺序取出向量, 从另一个矩阵中顺序取出标量并使用广播操作构造出向量, 从而达到向量化计算的效果, 如图 1(b) 所示. 在具体实现时, Broadcast 算法会将需要跨行的矩阵计算转换为访存模式更友好的多次迭代单行计算, 并专注于对内层循环做向量化. 本节的后续内容将基于该算法和其他算法采用优化分析模型进行建模、比较和验证.



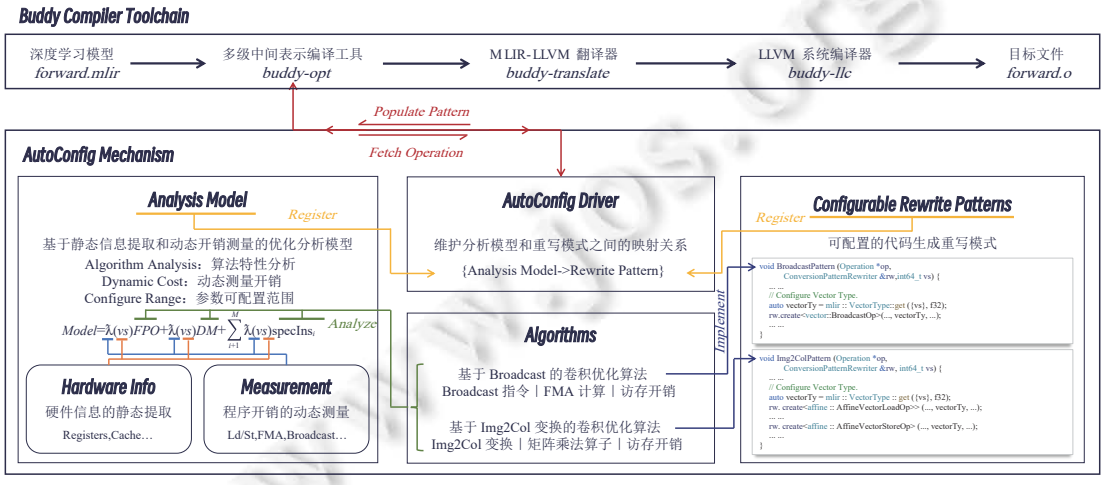
(a) lm2Col+GEMM优化算法 (b) 基于广播操作的优化算法

图 1 卷积优化算法图示

2.2 深度学习编译优化的基本流程

深度学习编译优化的基本流程如图 2(a) 所示. 首先, 来自深度学习模型中的计算负载会通过框架前端引入成为各种中间表示. 接着, 在中间表示的各个层级会进行编译优化, 优化后的结果随后被送到编译器和工具链中完成硬件代码生成. 在这个过程中, 中间表示层级是优化和调优的关键部分[23].

(a) AutoConfig 机制的概述



(b) AutoConfig 机制的设计、实现与集成

图 2 AutoConfig 的概述、设计、实现与集成

现有的优化和调优策略一般分为编译器的自动调优和调用手动优化高性能库两个大类, 二者的特性决定了它们各自适合不同的场景. 自动调优在循环优化上有出色的效果, 在同类硬件上可以复用基础设施避免重复开发, 但是由于缺乏可解释性, 这种优化和调优方法在新型硬件架构上需要准确的代价模型和耗时的搜索调优过程才能发挥作用. 手动优化能够最大化地利用硬件特性, 可以达到极致性能表现, 但往往需要使用平台特定的编程模型或接口, 即使同类硬件也需要重复开发优化, 造成大量的工程和维护开销.

2.3 编译基础设施 MLIR

MLIR[16]是一种现代编译基础设施, 它致力于提供模块化、可复用、可扩展的多层中间表示. 目前, MLIR 已经逐渐形成一个编译生态, 尤其是在深度学习编译优化领域, 许多前端框架和后端硬件提供了对 MLIR 的编译支持[24−26]. MLIR 的特点在于其所抽象出的多层中间表示. 每一层中间表示称作方言 (Dialect), 这是 MLIR 扩展机制的核心概念. MLIR 提供一系列内置的核心方言, 包括高层级面向领域计算的方言 (TOSA Dialect, Linalg Dialect). 循环变换级别的方言 (Transform Dialect, Affine Dialect, SCF Dialect). 硬件抽象级别的方言 (MemRef Dialect, Vector Dialect). 特定后端级别的方言 (LLVM Dialect, GPU Dialect). 在每一种方言内可以定义操作 (operation)、类型 (type)、属性 (attribute), 来提供特定抽象表示的语义.

MLIR 提供了强大的扩展机制和自动化能力, 用户可以使用声明式的方法灵活地扩展出自定义方言. 所有的内置方言均可与自定义方言兼容, 并且可以复用所有的内置编译下降通路与优化. 这种扩展能力还使得 MLIR 支持与其他已有的编译器对接, 便于支持更多的后端和硬件平台.

3 AutoConfig 概述

本节概述 AutoConfig 机制的设计思路、模块组成、使用方式和生态集成.

3.1 AutoConfig 的设计思路

深度学习编译器间的差异主要体现在优化方式和调优策略的设计方面. 如图 2(a) 所示, 目前领域内广泛使用如下两种优化和调优策略: (1) 计算与调度分离的编译优化方式, 结合编译器的自动调优机制; (2) 图级别编译器生成特定优化算子, 结合手动调优的高性能算子库. 前者通过提供优化与调优编程接口, 将调优的复杂度交给编译器的使用者; 后者通过使用硬件平台特定的编程模型来编写高性能库, 将调优的复杂度交给高性能库的编写者.

本文提出的 AutoConfig 是一种新的面向深度学习计算负载的编译优化和调优方法, 其核心设计思路是编写可配置的代码生成重写模式, 结合优化分析模型自动配置调优过程, 将调优的复杂度交给编译器的设计者和开发者. 这种设计思路可以利用编译器桥接软硬件的天然优势, 获取计算负载的语义信息和计算平台硬件信息, 实现可配置的编译优化和调优机制. 如图 2(a) 所示, AutoConfig 采用优化分析模型自动配置调优策略, 并选择合适的优化算法进行代码生成, 有效地权衡了多种优化算法和调优策略. 相比自动调优和手动优化, AutoConfig 的设计能够达成一次优化实现适配多种硬件平台的目标, 并且具有性能可解释、可迁移的特性.

可配置的代码生成重写模式是 AutoConfig 优化方法的基础. 通过采用粗颗粒度重写的优化策略, AutoConfig 可以在保障优化通用性和高性能的同时, 将优化算法实现为可配置的编译优化. 这种策略集成到编译工具链中, 使得不同目标硬件平台的代码生成可以进行灵活的配置. 此外, 重写模式的配置过程依赖于优化分析模型, 它可以在计算负载的编译时评估计算开销, 并根据此评估结果自动选择并配置最优的代码生成策略.

优化分析模型是 AutoConfig 调优方法的基础. 该模型对优化算法的总体计算开销进行建模, 将其分解为计算、访存和特殊指令的开销总和. 编译优化的开发者在实现代码生成重写模式时, 也需配套构建参数化的优化分析模型, 并根据硬件信息确定可配置参数取值范围. 在编译过程中, 优化分析模型通过具体的参数配置和动态测量的程序特性来确定总开销. 各优化算法对应的预测总开销是选择代码生成策略的关键依据, 以确保最终的编译优化结果能够最小化计算开销.

值得强调的是, AutoConfig 中的各模块构成了可以服务于任意编译优化和调优的基础设施, 并具有良好的可

扩展性. 在当前领域特定计算和体系结构发展的黄金时代, 各领域计算负载到多种硬件平台的编译和优化需求日益增长. AutoConfig 中的硬件信息收集策略可以针对特定硬件进行定制, 用户可根据需要扩展默认提供的动态测量程序集合, 以及划定给定程序的子集. 无论是优化分析模型的定义还是配置规则的注册, AutoConfig 都提供了通用的基础设施, 以满足灵活多变的调优需求. AutoConfig 的设计能够帮助开发者解耦编译优化的开发、分析、调优的过程, 并为更精细的分工提供了可能性. 这种设计使得开发者可以专注于特定子模块进行深入开发, 并通过复用现有组件降低开发成本.

3.2 AutoConfig 的模块组成

AutoConfig 的架构由驱动器、可配置的代码生成重写模式和优化分析模型这 3 大核心组件构成, 它们共同协作为编译优化算法服务. 其中, 优化分析模型抽象出优化算法的特征, 而代码生成重写模式负责表达算法的优化逻辑. 如图 2(b) 所示, 分析模型与重写模式之间存在直接的映射关系, 由 AutoConfig 驱动器维护这一映射, 并确保其无缝集成到深度学习编译工具链中.

AutoConfig 驱动器是核心控制模块, 负责调用各模块接口进行协同工作, 并且对接深度学习编译工具链. 当深度学习编译工具执行搭载 AutoConfig 的编译优化 Pass 时, AutoConfig 驱动器在所有已注册的分析模型中筛选出总开销最低的一个. 随后, 它利用选定的分析模型和相对应的重写模式来生成优化代码.

优化分析模型融合了静态信息提取与动态开销测量这两大环节, 并结合编译优化开发者对优化算法的特征分析, 最终形成一个综合的评估函数. 模型的分析是一个动静结合的过程: 静态信息主要来源于对优化算法及硬件平台的特性提取; 动态信息来自于算法特征在目标硬件平台上的执行开销. 如图 2(b) 所示, 对优化算法的静态分析用来确定计算次数、访存频率和特殊指令的发生次数, 而硬件信息的静态提取用于明确配置项取值范围. 动态开销测量模块根据这一配置范围获取具体测量结果作为各部分开销的权重. 硬件信息的精确提取和分析对于明确可配置的范围至关重要, 这不仅可以减少动态测量的开销, 也能够高效地选择配置项. 总的来说, 优化分析模型的设计与评估是 AutoConfig 准确性的关键, 本文第 4 节将详尽阐述以向量计算尺寸为调优点的优化分析模型构建过程.

代码生成重写模式是对编译优化算法逻辑的直接实现. 在代码生成方面, AutoConfig 采用 MLIR 作为中间表示, 并利用 MLIR 基础设施提供的各层级操作接口实现重写模式. AutoConfig 要求在这些重写模式中嵌入可配置的参数, 它们的设置依据来自前述优化分析模型的结果. AutoConfig 的设计思路强调将这些可配置参数与特定的重写模式相绑定, 而非与整体的编译优化 Pass 绑定, 这样的设计提升了调优的精细度和灵活性, 使得每一次优化都可以根据特定的需求和条件进行个性化调整与配置.

3.3 AutoConfig 的使用方式

AutoConfig 旨在服务于编译优化的开发者, 提供了一系列接口以辅助优化和调优的开发. 区别于使用固定策略开发编译优化, AutoConfig 要求开发者在创建重写模式时明确定义可配置项及其可能的取值范围. 此外, 用户需要针对优化算法构造分析规则和相应的动态测量程序集合. 这些分析规则、动态测量程序集合和可配置项取值范围共同构成优化分析模型. 针对每种计算负载的不同优化方式, 需要独立定义相应的分析模型和重写模式, AutoConfig 驱动器负责将它们配对并完成注册. 集成到 MLIR 优化 Pass 中的 AutoConfig 驱动器, 在触发时根据计算负载和硬件平台信息评估每一组分析模型的开销. 随后 AutoConfig 开始执行自动配置, 它通过整合计算负载的尺寸信息、分析模型和硬件信息, 全面评估所有可能的算法与配置选择, 从而选择出最佳策略进行优化代码生成.

如算法 1 所示, 本文以可配置参数 *vs* 为例展示针对计算负载 op 的两种编译优化算法的自动配置和代码生成过程. 首先, 为可配置参数 *vs* 设置可选的配置数值. 这里可以采用统一的配置集合以适应所有硬件后端, 或者根据各种硬件特性定制专门的配置集合. 例如, 对于 x86 的 AVX512 硬件平台, 可选配置集合可以是{64, 128, 256}, 而对于 Arm Neon 硬件平台, 则可以是{16, 32, 64}. 在选择了特定的配置集合后, 可以配置动态测量程序子集以适应所需的开销函数, 以此来降低配置成本.

算法 1. AutoConfig 机制的使用.

```
1. function MatchAndRewrite(op)
2.     // 以可配置参数 vs 为例, 设置可选配置数值
3.     configRange ← setRange(16,32,64,128,256)
4.     // 设置开销函数子集, 默认使用开销函数全集
5.     costSet ← setCostSet(BroadcastCost,FMACost,MemoryCost,...)
6.     // 定义优化算法重写模式 1
7.     pat1 ← rewritePattern1(op,...)
8.     // 分析优化算法构造分析规则 1
9.     rule1 ← func(op,...)
10.    // 初始化分析模型 1
11.    model1 ← initAnalyModel(rule1,costSet,configRange)
12.    // 定义优化算法重写模式 2
13.    pat2 ← rewritePattern2(op,...)
14.    // 分析优化算法构造分析规则 2
15.    rule2 ← func(op,...)
16.    // 初始化分析模型 2
17.    model2 ← initAnalyModel(rule2,costSet,configRange)
18.    // 初始化 AutoConfig 对象
19.    autoConfig ← initAutoConfig()
20.    // 将分析模型 1 和重写模式 1 注册到 AutoConfig 对象中
21.    autoConfig.register(model1,pat1)
22.    // 将分析模型 2 和重写模式 2 注册到 AutoConfig 对象中
23.    autoConfig.register(model2,pat2)
24.    // AutoConfig 对象选择合适的算法和配置进行代码生成
25.    autoConfig.populate()
26. end function
```

进一步地, 开发者需针对不同优化算法定义相应的重写模式和分析模型. 这些重写模式以 MLIR 为中间表示, 利用其多层级的 MLIR 方言来丰富语义表达. 优化分析模型结合分析规则、动态测量程序集合以及可配置参数的取值范围, 共同定义了编译优化的配置空间以供编译优化过程中的搜索和决策使用. 在用户接口方面, 本文提出的方法使用 rewritePattern 函数实现可配置的重写模式, 并使用 lambda 表达式来约定优化分析模型. 其中, rewritePattern 函数接受目标操作以及可配置参数, 并在函数体内使用各个操作的构造函数实现编译优化的代码生成逻辑. 同时, 为了构建分析模型, lambda 表达式通过其捕获列表传递动态测量程序集合, 通过参数列表传递可选配置数值集合. 该模型在 lambda 函数体内部完成构建, 详细的构建过程将在本文的第 4 节中展开讨论.

在定义了多种优化算法的重写模式和分析模型之后, 开发者需要初始化一个 AutoConfig 对象, 这一对象作为驱动器, 负责将每对重写模式和分析模型进行注册. AutoConfig 对象的 populate 函数负责为每个重写模式选择最优的配置项, 然后对不同的重写模式进行比较和评估, 最终选择出执行开销最小的重写模式和最佳配置项, 将其整合进编译优化 Pass 及其工具链.

3.4 AutoConfig 的生态集成

本文将 AutoConfig 集成到了 Buddy Compiler 中, 展示了 AutoConfig 对深度学习编译器的适配能力及其在优化深度学习计算任务方面的调优能力. Buddy Compiler 是一个基于 MLIR 的特定领域编译器, 它致力于打造面向深度学习领域的软硬件协同编译生态. 如图 2(b) 所示, Buddy Compiler 中的多级中间表示编译工具 buddy-opt 会

调用 AutoConfig 的各个模块对关键的深度学习操作进行分析和配置, 随后选择适当的代码生成策略进行编译优化, 并将代码编译下降后翻译至 LLVM IR, 最终生成针对目标硬件的汇编代码. Buddy Compiler 支持面向 CPU SIMD/Vector、GPU、加速器的代码生成, 通过集成 AutoConfig 的优化与调优方法, 它能够实现面向多种硬件后端的性能迁移和优化复用, 从而打造出高效通用的编译通路.

除了在 Buddy Compiler 中的集成之外, AutoConfig 更重要的贡献是为 MLIR 生态提供了新的调优方法和思路. MLIR 是深度学习编译器的主流生态之一, MLIR 上游提供 PDL 和 Transform 方言作为基础对高层中间表示进行调度和优化, 例如循环分块、顺序调整、向量化等. 基于 MLIR 的深度学习编译器 IREE 除了采用上游提供的基础方言之外还通过自定义方言进行调度, 并且使用强大的运行时环境实现多种平台的部署. 不同于使用特定方言进行调优和调度, AutoConfig 的创新点在于使用多层级的方言 (例如 Vector 方言, Affine 方言等) 编写可配置的优化算法重写模式, 并为优化算法建立分析模型, 搭配硬件信息收集策略提供的开销函数, 可以在编译时确定优化重写模式的配置参数, 从而实现面向特定计算负载和硬件的代码生成. 同时, AutoConfig 与 MLIR 生态中的其他调优方式并不冲突, 使用上游调度方言或者自定义方言的方式也可以集成 AutoConfig 使得其调优过程具备可解释性和可预测性.

4 优化分析模型

AutoConfig 的可解释性源于其优化分析模型. 该分析模型可以对优化算法的计算特征进行建模, 从而近似预测其执行开销. 本节提出一种向量尺寸可配置的优化分析模型建模方法, 可以在编译时根据高层中间表示中计算负载的输入尺寸和 SIMD 硬件平台特征来预测计算负载优化算法的期望开销, 从而有效指导 AutoConfig 进行代码生成. 值得注意的是, AutoConfig 并不耦合与某个特定的优化分析模型, 而是开放出可注册的接口, 用户可以按需注册自己的优化分析模型.

4.1 模型总体描述

在深度学习编译优化中, 对优化算法的选择以及算法内的参数配置都会对性能产生影响. 这些影响主要体现在计算开销、访存开销和特殊指令执行开销 3 个方面. 一般来说, 优化算法与朴素算法相比具有更低的计算开销, 但是由于对算法的优化通常会引入额外的内存排布操作和特殊指令, 这会带来额外的访存开销和特殊指令执行开销. 设 $Cost_{\text{All}}$ 为一个优化算法的理论总开销, 则有:

$$Cost_{\text{All}} = Cost_{\text{Arith}} + Cost_{\text{Memory}} + Cost_{\text{SpecIns}} \tag{1}$$

其中, $Cost_{\text{Arith}}$ 表示计算开销, 在 SIMD 浮点计算场景中主要指融合乘加操作 (FMA) 的执行开销. $Cost_{\text{Memory}}$ 表示访存开销, 主要指访问内存、缓存和寄存器组等存储模块所需的开销. $Cost_{\text{SpecIns}}$ 则涵盖了除了计算指令和访存指令以外其他特殊指令的执行开销. 不同的优化算法通常会引入不同的特殊指令, 比如 Broadcast 向量化算法会引入广播操作指令, 针对卷积优化的 Im2Col 算法会引入维度变换指令等. 朴素算法一般不引入特殊指令, 它的理论总开销只由计算开销和访存开销构成.

为了衡量这些开销, 设优化算法中包含的浮点运算指令个数 (floating point operations) 为 FPO, 内存访问指令个数 (data movement) 为 DM, 所包含的特殊指令种数为 M, 且第 i 条特殊指令的个数为 Num_{SpecIns_i}, 则上述公式可进一步描述为:

$$Cost_{\text{All}} = \lambda_{\text{Arith}} FPO + \lambda_{\text{Memory}} DM + \sum_{i=1}^{M} \lambda_{\text{SpecIns}_i} Num_{\text{SpecIns}_i} \tag{2}$$

其中, λ_{Arith}、λ_{Memory} 和 λ_{SpecIns} 分别表示计算开销、访存开销和特殊指令开销的重要性系数. 以计算开销为例, 由于它的大小与浮点运算指令个数成正比, 因此 λ_{Arith} 的含义就是单位运算指令开销. 重要性系数与优化算法的参数配置和执行平台紧密相关, 优化算法的参数调优越有效, 越能充分发挥执行平台的硬件特点, 则重要性系数越小. 在实际优化中, 该系数将通过第 5.2 节的动态测量流程确定.

基于上述对一个优化算法理论总开销的建模方式, 进一步提出优化加速比的概念, 以综合比较不同优化算法所带来的性能差异, 并作为 AutoConfig 选择最优算法的依据. 优化加速比的定义是: 相对于某个基准算法, 在相同的编译优化流程和硬件执行环境下, 优化算法的理论总开销与基准算法的比值. 具体而言, 对于优化算法 A 和优化算法 B 而言, A 相较于 B 的优化加速比可表示为:

$$SpeedUp_{\mathrm{All}}=\frac{Cost^{A}_{\mathrm{All}}}{Cost^{B}_{\mathrm{All}}} \tag{3}$$

在优化分析时, 还可以在计算、访存、特殊指令等不同方面定义局部优化加速比 $SpeedUp_{\mathrm{Arith}}$、$SpeedUp_{\mathrm{Memory}}$ 和 $SpeedUp_{\mathrm{SpecIns}}$, 它们能够帮助分析不同算法在不同方面的优劣情况. 以计算方面为例, 当 A 和 B 两种算法的计算开销重要性系数保持一致时, 其局部优化加速比可表示为:

$$SpeedUp_{\mathrm{Arith}}=\frac{Cost^{A}_{\mathrm{Arith}}}{Cost^{B}_{\mathrm{Arith}}}=\frac{\lambda_{\mathrm{Arith}}FPO^{A}}{\lambda_{\mathrm{Arith}}FPO^{B}}=\frac{FPO^{A}}{FPO^{B}} \tag{4}$$

此时通过分别统计两种算法的浮点运算指令个数, 就可以求得两者在计算方面的局部优化加速比. 第 4.2 节会借助该指标来探究不同算法在计算、访存方面的优化表现.

4.2 算法特性分析

4.2.1 标量算法与向量算法的特性分析

矩阵乘法是一种典型的深度学习计算负载. 设两个输入矩阵分别为 $M\times K$ 的矩阵 A 和 $K\times N$ 的矩阵 B, 输出矩阵为 $M\times N$ 的矩阵 C, 下面基于优化分析模型, 对朴素的标量算法和向量化的 Broadcast 算法做建模分析.

标量算法采用 3 层嵌套循环完成计算逻辑, 总的循环次数为 MNK. 在每一次循环中, 会进行一次 FMA 操作, 因此其浮点运算指令个数为 MNK. 此外在每一次循环中, 默认会涉及到对矩阵 A、B 和 C 各一次的读操作和对矩阵 C 的一次写操作, 但是在实际实现时可以把对矩阵 C 的读和写操作提取到长度为 K 的内层循环外面, 因此内存访问指令个数为 $2MN+2MNK$. 标量算法没有特殊指令执行开销.

基于 Broadcast 算法的向量化实现可通过参数 vs 来调节向量化长度, 以对长度为 N 的最内层循环做向量化. 在 Broadcast 算法中, 总的循环次数为 $MNK\frac{1}{vs}$, 因此浮点运算指令个数为 $MNK\frac{1}{vs}$, 每个 FMA 操作同时对 vs 个元素进行乘加运算. 在进入最内层循环之前, 需要使用一种标量广播的特殊指令将矩阵 A 的单个元素广播成长度为 vs 的向量数组, 该指令共需调用 MK 次. 在每次运算的过程中, 分别需要对矩阵 B 和 C 进行一次长度为 vs 的向量读操作, 将其与广播后的向量数组进行乘加运算, 并在最内层循环结束后将结果向量写回矩阵 C. 因此该算法的内存访问指令个数为 $MK+3MNK\frac{1}{vs}$.

设将向量化长度设置为 vs 时, 执行一条向量 FMA 指令的计算开销为 $Cost_{\mathrm{FMA}}[vs]$, 以向量数组的形式访问内存的访存开销为 $Cost_{\mathrm{Memory}}[vs]$, 将标量操作视为 $vs=1$ 时的向量操作, 则对标量算法与向量算法的特性分析结果如表 1 所示. 从中可以看出, 向量算法通过引入了特殊的广播指令并改变了原先的访存模式来为向量化提供优化空间. 一般来说 vs 越大, 则向量化越充分. 但是当 vs 过大时, 一方面负载的输入尺寸规模可能较小, 可向量化的空间不足. 另一方面向量化所带来的计算便利可能难以弥补寄存器溢出的开销. 因此硬件平台信息会对向量化长度的配置形成约束, 而基于优化分析模型也容易探究不同向量化长度对算法在计算、访存等方面上的影响.

表 1 标量算法与向量算法的特性分析

分类	标量算法	向量算法	局部优化加速比
浮点运算指令个数	MNK	$MNK\frac{1}{vs}$	$\frac{1}{vs}\times\frac{Cost_{\mathrm{FMA}}[vs]}{Cost_{\mathrm{FMA}}[1]}$
内存访问指令个数	$2MN+2MNK$	$MK+3MNK\frac{1}{vs}$	$\frac{K+3NK\frac{1}{vs}}{2N+2NK}\times\frac{Cost_{\mathrm{Memory}}[vs]}{Cost_{\mathrm{Memory}}[1]}$
特殊指令个数	0	MK	∞

4.2.2 Im2Col 与 Broadcast 算法的特性分析

深度学习模型里的卷积层通常处理的是四维张量, 这些张量会带有 batch、channel 和 feature 等维度. 设 N 表示 batch, C 表示 channel, F 表示 feature, $H_i \times W_i$ 、 $H_k \times W_k$ 和 $H_o \times W_o$ 分别表示张量二维切片中的输入、卷积核和输出所对应的尺寸, 下面以数据排布形式为 *NCHW_FCHW* 的卷积层为例 (即输入张量的数据排布为 *NCHW*, 卷积核张量的数据排布为 *FCHW*, 输出张量的数据排布为 *NFHW*), 对 Im2Col 与 Broadcast 两种算法做优化分析.

Im2Col 算法首先通过 Im2Col 策略, 对卷积层的输入进行数据重排, 并对卷积核进行维度变换, 然后通过矩阵乘法操作得到结果矩阵张量, 最后将该张量再进行维度变换恢复成卷积层的输出. 由于维度变换直接对张量进行原地操作, 不涉及浮点计算和内存访问, 因此主要关注该算法的数据重排开销和矩阵乘法操作开销. 在数据重排的过程中, 需要进行 $NCH_kW_kH_oW_o$ 次迭代, 每次迭代会读取输入中的一个元素, 再存储到矩阵 B 中, 该部分的访存开销为 $2NCH_kW_kH_oW_o$, 不涉及计算开销. 在矩阵乘法操作中, 开销等价于 N 次固定规模矩阵乘法的开销. 设每次矩阵乘法的两个输入矩阵和一个输出矩阵分别为 A、B、C, 则 A 矩阵的行数 $A_{\text{row}} = F$, B 矩阵的行数 $B_{\text{col}} = H_oW_o$, A 矩阵的列数 $A_{\text{col}} = CH_kW_k$. 若采用针对矩阵乘法的 Broadcast 向量算法进行计算, 则一次向量化矩阵乘法的迭代次数为 $FCH_oW_oH_kW_k\frac{1}{vs}$, 浮点运算指令个数为 $FCH_oW_oH_kW_k\frac{1}{vs}$, 内存访问指令个数为 $FCH_kW_k + 3FCH_oW_oH_kW_k\frac{1}{vs}$, 共需要用到 FCH_kW_k 条标量广播的特殊指令. 综合 Im2Col 策略和所有矩阵乘法操作, Im2Col 算法的总浮点运算指令个数为 $NFH_oW_oCH_kW_k\frac{1}{vs}$, 总内存访问指令个数为 $2NCH_kW_kH_oW_o + NFCH_kW_k + 3NFH_oW_oCH_kW_k\frac{1}{vs}$, 总共需要用到 $NFCH_kW_k$ 条标量广播的特殊指令.

若直接采用 Broadcast 算法, 在朴素算法的基础上对内层循环 W_o 进行向量化, 则迭代次数为 $NFCH_oH_kW_kW_o\frac{1}{vs}$. 总浮点运算指令个数等于迭代次数, 内存访问指令个数为 $NFCH_oH_kW_k + 3NFCH_oH_kW_kW_o\frac{1}{vs}$, 标量广播的特殊指令使用次数为 $NFCH_oH_kW_k$.

对 Im2Col 算法和 Broadcast 算法的特性分析如表 2 所示. 可以看出, 当输入宽度 W_o 较大时, Broadcast 算法在访存上更占优势. 而当输入高度 H_o 较大时, Im2Col 算法的优化性能更好. 这是因为 Broadcast 算法直接对 W_o 所对应的内层循环进行向量化, 而 Im2Col 算法会通过将卷积层转换成矩阵乘法操作来对 CH_kW_k 所对应的内层循环进行向量化. 所以在同样的向量化长度下, 虽然两种优化算法所需的浮点运算指令个数相同, 但是 Im2Col 算法需要花费额外的访存开销进行数据重排, 与此同时 Broadcast 算法需要花费较多的特殊指令执行开销. 在这种情况下便需要优化分析模型基于输入尺寸对算法的选择做出合理的判断.

表 2 Im2Col 与 Broadcast 算法的特性分析

分类	Im2Col算法	Broadcast算法	Im2Col相对Broadcast的局部优化加速比
浮点运算指令个数	$NFH_oW_oCH_kW_k\frac{1}{vs}$	$NFCH_oH_kW_kW_o\frac{1}{vs}$	1
内存访问指令个数	$2NCH_kW_kH_oW_o + NFCH_kW_k + 3NFH_oW_oCH_kW_k\frac{1}{vs}$	$NFCH_oH_kW_k + 3NFCH_oH_kW_kW_o\frac{1}{vs}$	$\frac{F + 3FW_o\frac{1}{vs}}{2W_o + F + 3FW_o\frac{1}{vs}}$
特殊指令个数	$NFCH_kW_k$	$NFCH_oH_kW_k$	H_o

5 静态信息提取与动态开销测量

优化分析模型是 AutoConfig 可解释性的来源, 其准确性由静态信息的提取和动态开销的测量共同决定. 静态信息提取阶段收集计算负载的尺寸信息和硬件平台数据, 旨在确定可配置参数的范围, 可以看作是对配置空间的精化和剪枝, 从而降低调优开销. 动态开销测量阶段通过在指定优化平台上测量示例程序, 以最终确定对应计算、

访存和特殊指令的权重系数, 为优化分析模型提供量化数据来选择合适的优化算法, 进而完成代码生成. 为了聚焦于 AutoConfig 的设计思想, 本文只使用单指令多数据流 SIMD 作为示例进行论述.

5.1 硬件信息的静态提取

静态信息提取根据计算负载的优化分析模型的需求进行采集, 结合这些硬件信息以及低层编译器的代码生成策略可以分析出加速指令的使用和额外的开销. 面向 CPU 的优化需要利用 SIMD 的加速能力并且使用 Cache 降低访存开销, 因此本文展示的静态信息提取主要包括收集 SIMD 寄存器堆的尺寸和 Cache 的信息. 配合生成的 LLVM 代码和汇编代码进行静态分析.

5.1.1 SIMD 寄存器关键信息提取与分析

SIMD 计算单元的关键信息包括 SIMD 寄存器长度和 SIMD 寄存器堆的容量. SIMD 寄存器的长度标志着一条 SIMD 指令能够操作的元素个数. SIMD 寄存器堆的容量标志着一个 SIMD 中间表示语句能够使用的最大非溢出尺寸. 使用这两个硬件信息能够结合编译器使用的 SIMD 抽象尺寸分析出计算的内存溢出次数.

编译策略选择的主要的挑战在于向量中间表示参数和实际机器关键信息之间存在差距. MLIR 和 LLVM IR 的向量抽象提供了灵活的语义, 并且根据后端硬件平台生成 SIMD 代码, 这意味着中间表示层面有更强的向量语义的表达能力来承载计算负载的编译优化, 同时能够避免在多个后端重复实现优化算法. 然而, 物理机器上的 SIMD 寄存器的长度和数量是固定的, 需要生成多条 SIMD 指令来弥合向量中间表示到 SIMD 器件的鸿沟. 一旦中间表示层面使用的向量抽象尺寸超过了 SIMD 寄存器堆的容量, SIMD 寄存器数量不足以完成所有的计算, 此时汇编代码生成器就会将生成访存指令, 将寄存器中的元素溢出到内存, 腾挪出空闲寄存器完成计算, 当溢出的元素被后续计算使用时, 再将这部分元素从内存中取回到 SIMD 寄存器, 这就造成了访存操作的额外开销.

在向量中间表示的转换过程中, 为了兼容多种后端硬件平台, 向量抽象的长度和类型不会发生改变. 图 3 展示了使用融合乘加操作 (FMA) 给出的具体分析示例. 示例函数由 MLIR 编写, 接受 3 个内存抽象的中间表示 (memref) 作为参数, 从这 3 个内存抽象中加在向量中间表示, 对 3 个向量进行融合乘加运算, 并将输出的向量存储到目标内存中间表示中. 示例使用两个不同长度的向量表示, 长度分别为 128 和 1 024, 向量中的每个元素为 32 位浮点数. 在从 MLIR 编译至 LLVM IR 之前, 在 MLIR 层级还需要将不同方言的操作改写为低层级方言中的等价操作这个过程向量表示的尺寸不会改变. 最终, 所有低层级方言中的操作都被翻译为 LLVM IR. 图 3(a2) 和图 3(b2) 描述了融合乘加函数的 LLVM IR 代码. 图 3 高亮了 MLIR 和 LLVM IR 中的向量类型. 在这个编译过程中, 中间表示的转换和翻译会重写各层级的方言和操作, 但这些向量抽象的长度和类型从未改变.

但是, 与 MLIR 和 LLVM IR 所抽象的向量中间表示参数不同, 汇编代码级别的参数与实际机器关键信息强相关. 此处选择 AVX512 扩展来展示编译结果. AVX512 的寄存器堆包括 32 个 zmm SIMD 寄存器, 每个寄存器的宽度为 512 位. 图 3(a3) 展示了向量抽象长度为 128 的融合乘加函数的汇编代码. 汇编代码清楚地表现出了 SIMD 指令的加载、计算和存储的过程. 加载和存储操作使用 vmovups 指令, 融合乘加操作使用 vfmadd213ps 指令. 在汇编代码中, 每个 vfmadd213ps 指令需要两个 zmm 寄存器. 所以向量抽象类型为 vector<128xf32> 的融合乘加操作需要 16 个 zmm 寄存器 (4096/512×2) 就可以完成计算. 当向量类型为 vector<1024xf32> 时, 理论上需要 128 个 zmm 寄存器, SIMD 寄存器堆但只有 32 个可用. 图 3(b3) 的汇编代码展示了寄存器首先用于进行加载操作, 直至 32 个 zmm 寄存器用尽. 此后指令 vfmadd213ps 进行融合乘加的计算. 由于没有空闲寄存器可用, 需要将计算结果溢出到内存中, 将寄存器腾挪出来负责后续计算. 最后进行存储操作时, 溢出的计算结果被重新加载回寄存器然后存储到目标缓冲区. 因此, 由于汇编代码和硬件平台指令集绑定的, 不合理的向量化配置会降低执行性能.

综上所述, 选择向量配置实际是在权衡迭代开销和溢出开销. 如果使用保守的向量配置策略, 寄存器堆可以承载所有的计算负载需求, 但是需要多次迭代才能完成所有的计算, 即增加了迭代的开销. 如果使用激进的向量配置策略, 寄存器堆需要溢出到内存才可以承载所有的计算负载, 这增加了访存开销, 但是可以用更少次数的迭代完成全部计算, 节省了循环迭代的开销. 这种权衡关系可以指导参数配置范围的选择.

图 3 不同向量配置下的融合乘加计算负载与 SIMD 寄存器的使用情况

5.1.2 Cache 信息提取与分析

优化算法的访存模型对实际性能有重要的影响, 因此编译优化生成代码对 Cache 的利用十分关键. 本节对 L1 数据 Cache 进行信息收集和分析, 主要关注 Cache 容量和 Cache 块尺寸. 这些信息可以辅助编译器制定循环分块或者循环顺序的策略, 尽可能利用 Cache 降低访存开销.

以实验平台 Intel(R) Xeon(R) Gold 5218R CPU 为例, L1 数据 Cache 的容量为 32 KB, 而每个 Cache 块的大小为 64 字节. 这个 Cache 块的大小可以看作与 vector<16xf32> 向量抽象尺寸相当, 这与 AVX512 的寄存器长度恰好一致. 当使用 vector<16xf32> 的整数倍作为向量抽象时, 数据之间并不会存在 Cache 命中或未命中的相互影响. 正如 第 5.1.1 节所提及的, 本文所选择的配置项均为 16 的整数倍. 因此, 对于本文这种以向量化为核心的优化策略, Cache 的状态不会对优化分析模型的准确性产生显著的影响. 对于访存密集型的优化任务, Cache 相关的可配置项是非常重要的, 并应当纳入优化分析模型. 针对这种优化任务, Cache 容量的抽象表示和 Cache 的结构都是必须进行详细提取和分析的要素. 以该实验平台为例, 访存操作一共可以在 L1 数据 Cache 中存放 vector<8192xf32>, 以及对应的二维向量表示, 例如 vector<8x1024xf32>、vector<4x512xf32> 等. L1 数据 Cache 采用 8 路组相连的设计, 每一路 64 个 Cache 块. 因此, 在优化过程中还需要关注访存操作的抽象长度是否会超出每一路的 Cache 块数量. 为了达到高效的访存模式, 高层代码优化策略可以根据以上分析采用特定的循环分

块和循环顺序策略.

5.2 程序开销的动态测量

动态开销测量针对优化所在平台确定计算、访存和特殊指令开销的权重系数, 并传入优化分析模型来得到最终的执行时间表达式. 为此, AutoConfig 维护了一个动态测量的程序集合, 其中的每个实例程序都体现了某一种程序特性. 该集合允许用户进行删改和添加, 使得用户可以根据其优化需求来调整程序集合. AutoConfig 提供的集合能够覆盖大部分与性能有关的 MLIR 操作, 并且这些动态测量的示例程序都源自实际的测试和优化片段. 这些示例程序的种类和迭代次数都是可配置的, 用户可据此定制合适的示例, 确保得到的测量值既准确又可靠, 从而避免随机误差的影响.

本节以 Broadcast 向量化算法的开销表示为例阐述动态测量流程, 涉及的测量开销如表 3 所示. 其中包括访存开销 $Cost_{\text{Memory}}[vs]$ 、SIMD 计算开销 $Cost_{\text{FMA}}[vs]$ 和表示广播指令的特定操作开销 $Cost_{\text{Broadcast}}[vs]$, 它们分别对应了优化分析模型的 3 项权重系数 λ_{Arith} 、 λ_{Memory} 和 λ_{SpecIns} . 具体来说, 在 SIMD 浮点计算场景中, λ_{Arith} 是单条 FMA 指令的执行时间, λ_{Memory} 则是单条访存指令的执行时间. 而 λ_{SpecIns} , 在对 Broadcast 向量化算法的分析中, 等价于单条广播操作指令的执行时间, 因为该优化算法涉及的特殊指令只有广播指令. 其他优化算法中特殊指令也同理.

表 3 Broadcast 向量化算法的开销表示

开销表示	开销描述	示例程序描述
$Cost_{\text{Memory}}[vs]$	将 vs 个元素进行访存操作	MemCopy示例程序的执行时间
$Cost_{\text{FMA}}[vs]$	vs个数据进行融合乘加操作的开销	FMA示例程序与MemCopy示例程序执行时间的差分
$Cost_{\text{Broadcast}}[vs]$	vs个数据进行广播操作的开销	BroadcastFMA示例程序与FMA示例程序执行时间的差分

为了量化这些开销, 本节使用动态测量程序集合中的特定子集在多个目标硬件平台上进行了实验. 其中访存开销的测量选择内存拷贝示例程序, 其中包括使用向量方言的内存加载和存储指令. FMA 开销的测量选择了 FMA 示例程序, 该程序结合了访存指令和融合乘加指令. 表示广播指令的特殊指令开销的测量选择了 BroadcastFMA 的示例程序, 其中包括访存指令、广播指令和融合乘加指令. 以上示例程序均使用 2 的 20 次方作为计算负载尺寸, 计算负载迭代 10 000 次, 通过测量总的执行时间来确定 $Cost_{\text{Memory}}[vs]$ 、 $Cost_{\text{FMA}}[vs]$ 和 $Cost_{\text{Broadcast}}[vs]$ 的量化数值.

表 4 是在 AVX512 平台下的实验测量结果, 从中可以看出, 访存指令开销 $Cost_{\text{Memory}}[vs]$ 平均为 FMA 指令开销 $Cost_{\text{FMA}}[vs]$ 和表示广播指令的特定操作开销 $Cost_{\text{Broadcast}}[vs]$ 的 20 倍左右. 在同样的硬件平台下, 这里得出的开销比值与权重系数 λ_{Arith} 、 λ_{Memory} 和 λ_{SpecIns} 之间的比值是一致的, 因此这些数据可以为优化分析模型提供权重数值, 进而分析出整体算法的性能表现, 并选择合适的算法和配置进行代码生成.

表 4 开销表示的测量结果 (ms/每 10 000 次迭代)

测量开销	vs=1	vs=16	vs=32	vs=64	vs=128	vs=256
$Cost_{\text{Memory}}[vs]$	12 028.5	1 648	1 751.5	1 779.5	1 989.5	2 345.5
$Cost_{\text{FMA}}[vs]$	644	70	142	83	102	52
$Cost_{\text{Broadcast}}[vs]$	738	175	209	204	303	154

6 实验设计、结果和分析

本节面向实际的深度学习计算负载场景, 在 AVX512 和 ARM Neon 两种 SIMD 平台上, 对 AutoConfig 的参数配置及优化算法的选择进行探究, 并将基于分析模型的代码生成策略与 TVM 的自动调优策略进行跨平台的性能比较. 其中, 在第 6.2 节会探究向量化长度参数的选取对优化性能的影响, 指出基于不同硬件平台的特性来确定参数配置的意义. 在第 6.3 节会利用硬件信息和动态开销数据来计算不同算法的预测加速比, 以验证 AutoConfig 的算法选择是否准确. 在第 6.4 节将 AutoConfig 与基于自动调优策略的 TVM 进行跨平台的代码生成性能比较.

AutoConfig 基于 Buddy Compiler 实现, 在代码生成阶段使用了开源编译器框架 LLVM, 版本为 17.0.0; 在对比实验中采用的 TVM 版本为 v0.14.0.

6.1 优化场景的选择

本文选取了一个经典的卷积神经网络 EfficientNet (模型文件来自 https://coral.ai/models/image-classification/), 并针对其中的关键深度学习计算负载即不同输入尺寸的卷积层进行优化. 卷积神经网络是以卷积层作为主要模块的深度学习模型. 对于卷积神经网络而言, 靠近输入部分的卷积层其功能主要在于从输入中提取初步特征, 因此卷积的输入尺寸较大而通道数较少. 处于模型中间部分或靠近输出部分的卷积层其功能主要在于从上一层输出的特征中进一步提取深层信息, 因此卷积的输入尺寸较小而通道数较多.

EfficientNet 是经典的卷积神经网络模型, 本节基于 EfficientNet 完成实验探究. 表 5 对该模型文件中与计算和访存相关的粗粒度计算负载进行了统计, 可以看出卷积 (conv_2d_nhwc_hwcf) 的计算占所有计算负载的大多数 (38/64), 可见通过从模型中的不同部分选取了尺寸多样的卷积作为优化对象具有一定的代表性, 能够反映优化算法的选择对整体性能的影响.

表 5 EfficientNet 高层计算负载统计

高层计算负载	出现次数
batch_matmul	1
depthwise_conv_2d_nhwc_hwcm	11
conv_2d_nhwc_hwcf	38
collapse_shape	12
expand_shape	2

6.2 基于硬件信息的参数配置

为了验证硬件信息对优化算法参数选取的影响, 在 AVX512 和 ARM Neon 平台中分别以 {16, 32, 64, 128, 256} 的数值设置向量化长度, 并在不同的卷积优化场景中计算优化加速比, 随后对于这两个平台分别选取平均实际加速比最大的向量化长度设置, 将相关数据记录在表 6 和表 7 中. 为了方便对比, 也将适合 AVX512 的向量化长度配置应用于 ARM Neon 平台中进行测试, 将相关数据记录在表 8 中. 图 4 是面向 ARM Neon 平台时, 选取不同向量化参数 *vs* 的 Broadcast 算法优化卷积的执行时间比较.

表 6 基于 AVX512 平台, 向量化长度为 64 时的最佳算法选择探究

卷积尺寸	标量执行时间 (ms)	Im2Col执行时间 (ms)	Broadcast执行时间 (ms)	实际加速比	预测加速比	算法选择结果是否正确
(1×1536×7×7, 192×1536×1×1)	75.3	2.2	14.5	6.74	2.51	√
(1×192×7×7, 1536×192×1×1)	76.9	2.1	14.7	7.14	2.83	√
(1×48×30×30, 384×48×3×3)	647	16.9	32.5	1.91	1.58	√
(1×32×114×114, 96×32×3×3)	1 726	76.9	46.6	0.66	0.83	√
(1×64×58×58, 64×64×3×3)	581	19.9	18.0	9.91	1.84	√

表 7 基于 ARM Neon 平台, 向量化长度为 16 时的最佳算法选择探究

卷积尺寸	标量执行时间 (ms)	Im2Col执行时间 (ms)	Broadcast执行时间 (ms)	实际加速比	预测加速比	算法选择结果是否正确
(1×1536×7×7, 192×1536×1×1)	53	2.9	28.0	10.00	1.51	√
(1×192×7×7, 1536×192×1×1)	49.2	4.5	26.7	5.88	1.57	√
(1×48×30×30, 384×48×3×3)	461	15.4	63.9	4.17	1.15	√
(1×32×114×114, 96×32×3×3)	1 229	49.7	47.8	0.97	0.94	√
(1×64×58×58, 64×64×3×3)	1 211	50.1	47.7	0.95	0.94	√

一方面, 从图 4 中可以看出, 在 ARM Neon 平台上选择 16 作为向量化长度能够获得更小的执行时间和更准确的预测结果, 这说明向量化长度参数的选取会对优化性能产生关键影响. 另一方面, 根据表 6, 在 AVX512 平台

上选择 64 作为向量化长度是更优的参数配置, 但是根据表 7 和表 8, 该参数对 ARM Neon 平台来说并不是最优的. 这说明采用较大的向量化长度不一定在每个平台上都能得到较好的执行性能, 还需要考虑硬件平台里 SIMD 寄存器个数和 Cache 容量的限制. 实践中发现, 在面向 AVX512 的硬件后端时, {32,64,128} 是合适的参数项配置范围. 而在面向 ARM Neon 的硬件后端时, 参数选择 {16,32,64} 可以得到理想的调优效果, 这反映了参数的可取范围受硬件信息的约束 (举例而言, 不同平台有不同的指令向量长度. AVX512 的指令向量长度为 512 位, 而 ARM Neon 的指令向量长度只有 128 位). 由此可见, 合理的参数配置能够提高算法选择的精确度, 同时提升代码的执行性能.

表 8　基于 ARM Neon 平台, 向量化长度为 64 时的最佳算法选择探究

卷积尺寸	标量执行时间 (ms)	Im2Col执行时间 (ms)	Broadcast执行时间 (ms)	实际加速比	预测加速比	算法选择结果是否正确
(1×1536×7×7, 192×1536×1×1)	53	11.5	89.5	7.69	1.86	√
(1×192×7×7, 1536×192×1×1)	49.2	11.2	88.5	7.69	2.02	√
(1×48×30×30, 384×48×3×3)	461	19.9	192	10.00	1.28	√
(1×32×114×114, 96×32×3×3)	1 229	45.9	144	3.12	0.88	×
(1×64×58×58, 64×64×3×3)	1 211	46.9	147	3.12	0.87	×

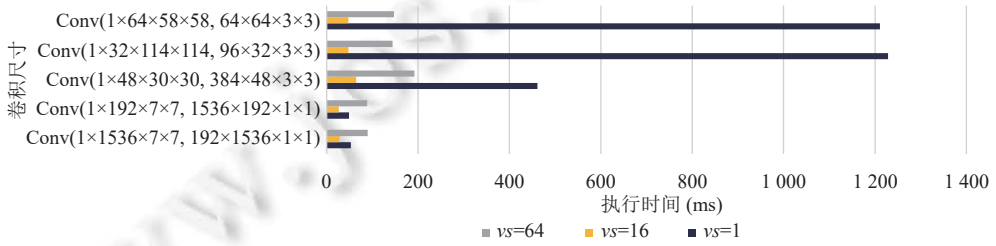


图 4　ARM Neon 平台下不同向量化参数 *vs* 的 Broadcast 算法优化卷积的执行时间 (ms)

6.3　优化分析模型的最佳算法选择

为了验证 AutoConfig 进行算法选择的有效性, 首先基于硬件信息和动态开销求出 Im2Col 相对于 Broadcast 的预测加速比. 预测加速比大于 1 表示在优化分析模型中 Im2Col 的执行性能优于 Broadcast, 此时 AutoConfig 会选择 Im2Col 算法实施代码生成策略. 反之选择 Broadcast 算法实施代码生成策略. 然后在不同的硬件平台上基于不同的配置参数, 分别显性指定 Im2Col 和 Broadcast 算法进行代码生成, 测量两种算法所得到优化代码各自的实际执行时间, 计算出 Im2Col 相对于 Broadcast 的实际加速比, 最后在不同的卷积优化场景中 AutoConfig 的预测加速比进行验证, 实验结果如表 6、表 7 和表 8 所示. 图 5 是在 AVX512 平台下, 确定最佳参数配置 *vs*=64 时的最佳算法选择结果. 图 5 中计算得出了 Im2Col 和 Broadcast 算法的相对执行时间占比, 一个算法对应的柱形部分越短, 则该算法的性能与另一个算法相比越好, 此时选择该算法是最佳的策略.

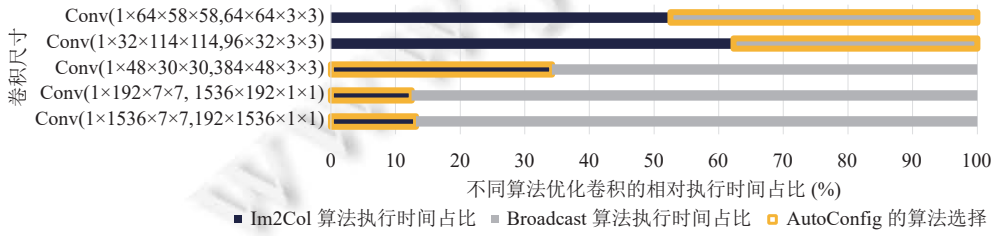


图 5　不同算法优化卷积的相对执行时间占比

根据图 5, 结合表 6–表 8 对算法选择结果的正确性验证, 可以看出 AutoConfig 在多种硬件平台和不同向量化长度配置下, 能够有效地完成代码生成策略中的算法选择, 这体现在对于大部分卷积优化场景, AutoConfig 能够预测得到实际性能更好的优化算法. 尽管如此, AutoConfig 的分析模型也有其局限性, 这会导致它在某些情况下 (如处理表 8 中后两个卷积尺寸时) 的算法选择结果存在偏差. 具体而言, 在第 4.1 节的建模中笼统地将各种访存操作带来的开销视为同一类开销 $Cost_{\text{Memory}}$, 但实际上, $Cost_{\text{Memory}}$ 还应包括 Cache 未命中的代价、从内存中预取数据的代价、数据从寄存器中溢出的代价等. 在实际的访存中预取数据的操作是穿插在运算操作之间来隐藏延迟的, 它与读取 Cache 操作的频率共同反映了算法对内存和缓存的利用程度, 而寄存器溢出的代价则反映了算法对寄存器组的使用情况. 由此可见, 提高优化分析模型对硬件行为的抽象程度和描述精度是实现更加实用的自动配置机制的关键.

6.4 AutoConfig 的代码生成性能验证

TVM 采用搜索的方式进行自动调优, 搜索的过程中需要通过枚举优化参数并实际执行程序来确定最佳的优化策略. 与 TVM 相比, AutoConfig 进行配置的所需开销主要体现在动态开销的测量上. 基于静态硬件信息和动态开销测量的结果, AutoConfig 能够直接借助代价模型完成参数配置和算法选择, 实现代码生成的过程. 针对深度学习计算负载场景, 本节将 AutoConfig 与基于自动调优策略的 TVM 进行跨平台的性能比较.

在 AVX512 平台和 ARM Neon 平台上, 对不同尺寸的矩阵乘法 (Matmul) 和卷积 (Conv) 实施调优的实验结果如表 9 和表 10 所示. 可以看出 AutoConfig 和 TVM 的优化相比于原始程序取得了较明显的性能提升, 且 AutoConfig 的优化加速比和所生成优化代码的绝对执行时间与 TVM 的优化在一个数量级内, 具有可比性.

表 9 AVX512 平台下 AutoConfig 与其他调优机制的性能对比

负载尺寸	TVM基准性能 (ms)	TVM调优性能 (ms)	TVM调优加速比	AutoConfig基准性能 (ms)	AutoConfig调优性能 (ms)	AutoConfig调优加速比
Matmul(64×65536, 65536×256)	2 518	189	**13.3**	4 248	242	**17.6**
Matmul(1024×1024, 1024×1024)	3 483	95	**36.7**	3 801	218	**17.4**
Conv(1×1536×7×7, 192×1536×1×1)	160	2.8	**57.1**	75.3	2.2	**34.2**
Conv(1×192×7×7, 1536×192×1×1)	159.5	0.9	**177.2**	76.9	2.1	**36.6**
Conv(1×48×30×30, 384×48×3×3)	184	2.4	**76.7**	647	16.9	**38.3**
Conv(1×32×114×114, 96×32×3×3)	445.1	10.3	**43.2**	1 726	46.6	**37.0**
Conv(1×64×58×58, 64×64×3×3)	153	1.4	**109.3**	581	18	**32.3**

进一步的, 为了评估 AutoConfig 相比 TVM 在调优时的性能优势, 本文定义单位加速比所需调优开销, 表示优化算法在与标量算法相比实现性能提升时, 平均每单位倍数的性能提升所需的调优开销, 该指标直观反映了调优行为对生成代码加速的贡献力度. 基于表 10 使用单位加速比所需调优开销, 对 AutoConfig 和 TVM 在卷积优化场景进行描述, 实验结果如图 6 所示. 结果表明, 与 TVM 的自动调优方式相比, AutoConfig 无需通过耗时较长的搜索方式确定最优参数, 只需要完成硬件信息提取和动态开销测量, 即可依托代价模型实现可自动配置的代码生成, 且由生成的代码可达到与自动调优相似的执行性能.

表 10 ARM Neon 平台下 AutoConfig 与其他调优机制的性能对比

负载尺寸	TVM基准性能 (ms)	TVM调优性能 (ms)	TVM调优加速比	AutoConfig基准性能 (ms)	AutoConfig调优性能 (ms)	AutoConfig调优加速比
Matmul(64×65536, 65536×256)	6 194	134	**46.2**	6 455	140	**46.1**
Matmul(1024×1024, 1024×1024)	1 300	76	**17.1**	3 889	127	**30.6**
Conv(1×1536×7×7, 192×1536×1×1)	234.1	8.2	**28.5**	53	11.5	**4.6**
Conv(1×192×7×7, 1536×192×1×1)	228.6	7.4	**30.9**	49.2	11.2	**4.4**
Conv(1×48×30×30, 384×48×3×3)	299.5	3.1	**96.6**	461	19.9	**23.2**
Conv(1×32×114×114, 96×32×3×3)	349.4	9.4	**37.2**	1 229	45.9	**26.8**
Conv(1×64×58×58, 64×64×3×3)	125.6	4.6	**27.3**	412	16.8	**24.5**

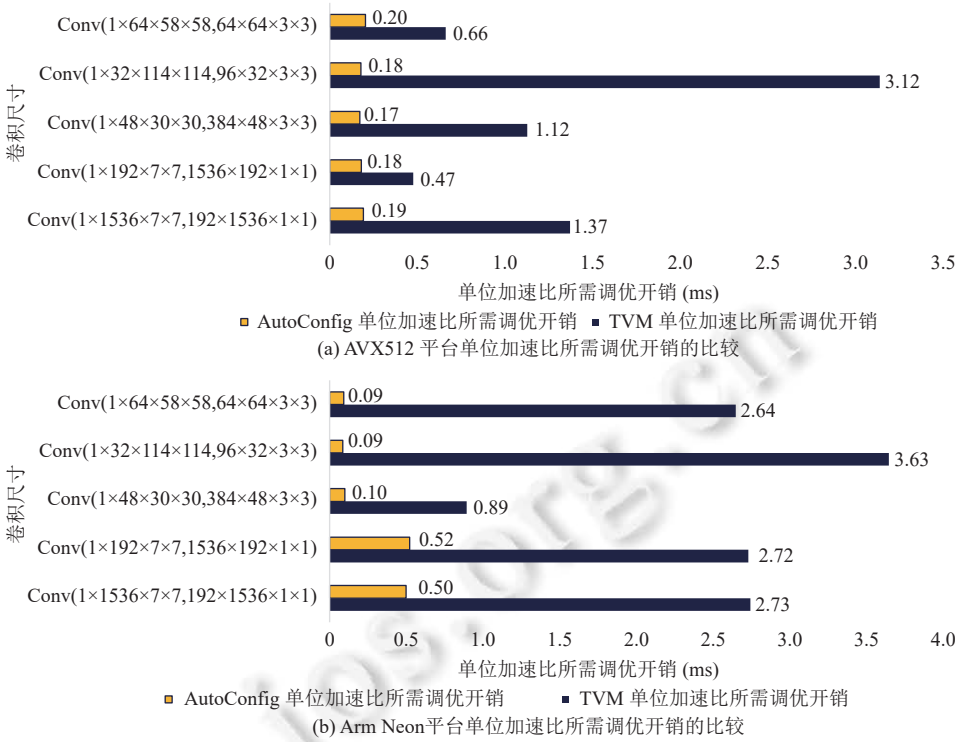


图 6 单位平台加速比所需调优开销的跨平台比较

7 总 结

本文提出了 AutoConfig, 一种面向深度学习编译优化的自动配置机制. 针对不同的深度学习计算负载和硬件平台, 该机制构建了具备可配置的代码生成重写模式和可解释性的优化分析模型, 通过分析静态提取的信息和动态测量的开销来确定最佳的参数配置与优化算法, 从而进行代码生成. 本文还将 AutoConfig 集成到深度学习编译器 Buddy Compiler 中, 旨在达成一次优化实现适配多种硬件平台的目标. 通过对深度学习模型中的卷积和矩阵乘法的优化实验, 本文验证了 AutoConfig 自动配置优化和调优方法能够解决当前深度学习编译器调优开销大, 优化效果可解释性差的问题. AutoConfig 生成的优化代码可以达到与编译器自动调优相似的性能表现, 同时避免了重复实现和反复调优.

本文期望 AutoConfig 成为自动配置编译优化的基础设施, 并拥有开放的使用模式. 本文阐述的优化分析模型和动静融合的调优策略并非与 AutoConfig 的基础设施紧密耦合. 用户可以提供不同的优化分析模型和调优机制来驱动 AutoConfig, 从而实现更加高效的编译优化. 例如, 用户可以针对其他深度学习场景进行代码生成, 可以采用更多样的优化算法进行比较和调优, 也可以从硬件平台收集更细颗粒度的信息并建立理论模型来提高性能预测的准确性, 还可以集成 Amdahl 模型、Roofline 模型等不同的分析模型进行针对性的调优等. 因此, 本文更长远的意义在于提供了全新的编译优化开发范式, 将编译优化解耦为开发、分析和调优的过程, 为编译优化提供了新的研究方向.

References:

[1] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Proc. of the 25th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: ACM, 2012. 1097–1105. [doi: 10.5555/2999134.2999257]

[2] Radford A, Kim JW, Xu T, Brockman G, McLeavey C, Sutskever I. Robust speech recognition via large-scale weak supervision. In: Proc.

of the 40th Int'l Conf. on Machine Learning. Honolulu: ACM, 2023. 1182. [doi: 10.5555/3618408.3619590]

[3] Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano PF, Leike J, Lowe R. Training language models to follow instructions with human feedback. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: NeurIPS, 2022. 27730–27744.

[4] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2020. 159. [doi: 10.5555/3495724.3495883]

[5] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. CuDNN: Efficient primitives for deep learning. arXiv:1410.0759, 2014.

[6] Li JH, Qin ZN, Mei YJ, Cui JZ, Song YF, Chen CY, Zhang YF, Du LS, Cheng XH, Jin BH, Ye J, Lin E, Lavery D. OneDNN graph compiler: A hybrid approach for high-performance deep learning compilation. arXiv:2301.01333, 2023.

[7] Khan J, Fultz P, Tamazov A, Lowell D, Liu C, Melesse M, Nandhimandalam M, Nasyrov K, Perminov I, Shah T, Filippov V, Zhang J, Zhou J, Natarajan B, Daga M. MIOpen: An open source library for deep learning primitives. arXiv:1910.00078, 2019.

[8] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2004 Int'l Symp. on Code Generation and Optimization. San Jose: IEEE, 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]

[9] Zhang HB, Xing MJ, Wu YJ, Zhao C. Compiler technologies in deep learning co-design: A survey. Intelligent Computing, 2023, 2: 0040. [doi: 10.34133/icomputing.0040]

[10] Chen TQ, Moreau T, Jiang ZH, Zheng LM, Yan E, Cowan M, Shen HC, Wang LY, Hu YW, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: ACM, 2018. 579–594. [doi: 10.5555/3291168.3291211]

[11] Jouppi NP, Young C, Patil N, *et al*. In-datacenter performance analysis of a tensor processing unit. In: Proc. of the 44th ACM/IEEE Annual Int'l Symp. on Computer Architecture. Toronto: Association for Computing Machinery, 2017. 1–12. [doi: 10.1145/3079856.3080246]

[12] Chen TQ, Zheng LM, Yan E, Jiang ZH, Moreau T, Ceze L, Guestrin C, Krishnamurthy A. Learning to optimize tensor programs. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montreal: ACM, 2018. 3393–3404. [doi: 10.5555/3327144.3327258]

[13] Zheng LM, Jia CF, Sun MM, Wu Z, Yu CH, Haj-Ali A, Wang YD, Yang J, Zhuo DY, Sen K, Gonzalez JE, Stoica I. Ansor: Generating high-performance tensor programs for deep learning. In: Proc. of the 14th USENIX Conf. on Operating Systems Design and Implementation. ACM, 2020. 49. [doi: 10.5555/3488766.3488815]

[14] Zhu H, Wu R, Diao Y, Ke S, Li H, Zhang C, Xue J, Ma L, Xia Y, Cui W, Yang F, Yang M, Zhou L, Cidon A, Pekhimenko G. ROLLER: Fast and efficient tensor compilation for deep learning. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX, 2022. 233–248.

[15] Zheng NX, Lin B, Zhang QL, Ma LX, Yang YQ, Yang F, Wang Y, Yang M, Zhou LD. SparTA: Deep-learning model sparsity via tensor-with-sparsity-attribute. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX, 2022. 213–232.

[16] Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O. MLIR: Scaling compiler infrastructure for domain specific computation. In: Proc. of the 2021 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). Seoul: IEEE, 2021. 2–14. [doi: 10.1109/CGO51591.2021.9370308]

[17] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: ACM, 2017. 6000–6010. [doi: 10.5555/3295222.3295349]

[18] Kim S, Hooper C, Wattanawong T, Kang M, Yan RH, Genc H, Dinh G, Huang QJ, Keutzer K, Mahoney MW, Shao YS, Gholami A. Full stack optimization of transformer inference: A survey. arXiv:2302.14017, 2023.

[19] Stothers AJ. On the complexity of matrix multiplication [Ph.D. Thesis]. Edinburgh: The University of Edinburgh, 2010.

[20] Cong J, Xiao BJ. Minimizing computation in convolutional neural networks. In: Proc. of the 24th Int'l Conf. on Artificial Neural Networks and Machine Learning. Hamburg: Springer, 2014. 281–290. [doi: 10.1007/978-3-319-11179-7_36]

[21] Lavin A, Gray S. Fast algorithms for convolutional neural networks. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). Las Vegas: IEEE, 2016. 4013–4021. [doi: 10.1109/CVPR.2016.435]

[22] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing. In: Proc. of the 10th Int'l Workshop on Frontiers in Handwriting Recognition. La Baule: University of Rennes, 2006.

[23] Li MZ, Liu Y, Liu XY, Sun QX, You X, Yang HL, Luan ZZ, Gan L, Yang GW, Qian DP. The deep learning compiler: A comprehensive

survey. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(3): 708–727. [doi: 10.1109/TPDS.2020.3030548]

[24] Katel N, Khandelwal V, Bondhugula U. MLIR-based code generation for GPU tensor cores. In: Proc. of the 31st ACM SIGPLAN Int'l Conf. on Compiler Construction. Seoul: Association for Computing Machinery, 2022. 117–128. [doi: 10.1145/3497776.3517770]

[25] Vasilache N, Zinenko O, Bik AJC, Ravishankar M, Raoux T, Belyaev A, Springer M, Gysi T, Caballero D, Herhut S, Laurenzo S, Cohen A. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. arXiv:2202.03293, 2022.

[26] Hu PC, Lu M, Wang L, Jiang GY. TPU-MLIR: A compiler for TPU using MLIR. arXiv:2210.15016, 2023.

张洪滨(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为编译技术.

武延军(1979－), 男, 博士, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 系统安全.

周旭林(2001－), 男, 硕士生, CCF 学生会员, 主要研究领域为编译技术.

赵琛(1967－), 男, 博士, 博士生导师, CCF 高级会员, 主要研究领域为编译技术, 操作系统, 网络软件.

邢明杰(1980－), 男, 高级工程师, CCF 专业会员, 主要研究领域为编译技术.