软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007649] [CSTR: 32375.14.jos.007649]

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

深度学习编译优化技术综述*

林泓宇[1,2], 刘 洋[1,2], 罗浩然[5], 李林海[1,2,3,4], 曹行行[1,2,3,4], 王梦娜[1,2], 徐家豪[1,2], 李泉毅[1,2,3,4], 张洪滨[1,2], 邢明杰[1,2], 武延军[1,2]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](中国科学院大学 计算机科学与技术学院, 北京 100049)

[3](中国科学院大学南京学院 信息学院, 江苏 南京 211135)

[4](中科南京软件技术研究院 科学技术部, 江苏 南京 211135)

[5](College of Computing and Data Science, Nanyang Technological University, Singapore 639798, Singapore)

通信作者: 张洪滨, E-mail: hongbin2019@iscas.ac.cn

摘 要: 随着深度学习的快速发展, 深度学习框架和硬件相关的研究成为推动其发展的重要方向, 其中框架为开发者提供了便捷的深度学习模型构建方式, 而硬件则提供了强大的计算能力. 然而, 当前多样化的深度学习框架与硬件平台之间的适配性差、兼容性不足, 导致了性能和可扩展性问题. 为解决这一挑战, 深度学习编译技术应运而生, 它通过一系列的中间表示及其对应的自动化转换, 将不同框架的模型高效地映射到特定后端设备上的可执行代码, 并在此过程中应用多种深度学习编译优化技术, 如算子融合、内存优化、自动调优等, 在解决可拓展性问题的同时, 显著提升模型的计算效率. 首先梳理深度学习编译的基本概念和整体流程, 随后总结并分类深度学习编译中的常见优化技术, 最后探讨该领域面临的挑战以及未来发展方向.

关键词: 深度学习; 深度学习编译; 编译优化; 大语言模型编译; 高性能计算; RISC-V

中图法分类号: TP314

中文引用格式: 林泓宇, 刘洋, 罗浩然, 李林海, 曹行行, 王梦娜, 徐家豪, 李泉毅, 张洪滨, 邢明杰, 武延军. 深度学习编译优化技术综述. 软件学报. http://www.jos.org.cn/1000-9825/7649.htm

英文引用格式: Lin HY, Liu Y, Luo HR, Li LH, Cao HH, Wang MN, Xu JH, Li QY, Zhang HB, Xing MJ, Wu YJ. Survey on Deep Learning Compilation Optimization Technologies. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7649.htm

Survey on Deep Learning Compilation Optimization Technologies

LIN Hong-Yu[1,2], LIU Yang[1,2], LUO Hao-Ran[5], LI Lin-Hai[1,2,3,4], CAO Hang-Hang[1,2,3,4], WANG Meng-Na[1,2], XU Jia-Hao[1,2], LI Quan-Yi[1,2,3,4], ZHANG Hong-Bin[1,2], XING Ming-Jie[1,2], WU Yan-Jun[1,2]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China)

[3](School of Information Science, University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)

[4](Department of Science and Technology, Nanjing Institute of Software Technology, Nanjing 211135, China)

[5](College of Computing and Data Science, Nanyang Technological University, Singapore 639798, Singapore)

Abstract: With the rapid development of deep learning, research on deep learning frameworks and hardware has become a crucial direction for advancing the field. Frameworks provide developers with convenient tools for building deep learning models, while hardware delivers powerful computational capabilities. However, limited adaptability and insufficient compatibility between diverse deep learning frameworks and hardware platforms often lead to performance and scalability issues. To address this challenge, deep learning compilation technology has emerged. Models from different frameworks are efficiently mapped to executable code for specific backend devices through

* 收稿时间: 2025-09-02; 修改时间: 2025-11-28; 采用时间: 2026-02-03; jos 在线出版时间: 2026-08-12

a series of intermediate representations and automated transformations. During this process, various deep learning compilation optimization techniques, such as operator fusion, memory optimization, and auto-tuning, are applied. These not only resolve scalability issues but also significantly improve the computational efficiency of the models. This study first reviews the basic concepts and overall process of deep learning compilation, then summarizes and categorizes the common optimization techniques in deep learning compilation, and finally discusses the challenges faced by the field and potential future development directions.

Key words: deep learning (DL); deep learning compilation; compiler optimization; large language model (LLM) compilation; high performance computing; RISC-V

随着深度神经网络 (deep neural network, DNN) 以及以 Transformer[1]为代表的大语言模型 (large language model, LLM) 在计算机视觉 (computer vision, CV)、自然语言处理 (natural language processing, NLP)、强化学习 (reinforcement learning, RL) 等领域的广泛应用, 深度学习 (deep learning, DL) 已从算法范式演进为支撑智能时代的核心技术之一. 例如, 卷积神经网络 (convolutional neural network, CNN) 在图像识别中的突破[2], 递归神经网络 (recurrent neural network, RNN) 在语言建模和机器翻译中的优越表现[3], AlphaGo 等强化学习模型在博弈任务中的卓越能力[4], 以及 Transformer 结构在语言建模与多模态理解中的统治地位, 都凸显了深度学习在不同智能任务上的跨领域统治力. 然而, 随着模型规模、输入序列长度及推理并发需求的持续攀升, 计算框架、编译器与硬件体系面临前所未有的性能、能效与可扩展性压力, 传统手工优化方式已难以满足此类复杂系统需求.

如图 1 所示, 为了提升深度学习模型的开发效率, 学术界与工业界提出了多种深度学习框架, 如 PyTorch[5]、TensorFlow[6]、Jax[7]、PaddlePaddle[8]等. 这些框架为模型训练和推理提供了丰富的高层抽象与运行时优化, 使研究人员和工程师能够专注于模型设计, 而无需直接操作底层硬件. 然而, 框架的多样化也带来了严重的互操作性挑战, 例如模型转换过程的语义不一致、算子集不统一、动态形状处理能力不足等问题, 使得跨平台部署面临巨大开销. ONNX[9]作为统一模型交换格式, 在一定程度上缓解了互操作性问题, 但其静态图特性与算子集更新滞后于框架生态, 使其在高动态性模型 (如大语言模型推理、动态序列任务) 中仍存在局限. 此外, 各类推理优化框架 (如 TensorRT[10]、OpenVINO[11]、TVM[12]) 利用算子融合、张量重排、自动调优等技术实现了显著加速, 特别是在处理大模型推理时通过 FlashAttention[13]、PagedAttention[14]等机制有效减少内存访问开销.

早期的深度神经网络主要依赖于 CPU 等通用计算平台进行计算, 但此类平台对深度学习典型的数据流计算不够友好. 随着模型复杂度与数据规模的不断攀升, 深度学习计算对高吞吐、低时延与高能效的需求迅速提高, 推动 GPU、NPU、FPGA (field programmable gate array) 与 ASIC (application specific integrated circuit) 等专用加速硬件的快速发展. 例如, GPU 凭借其成熟的软件生态和强大的并行计算能力, 成为深度学习训练和推理的主流平台, 典型代表有 NVIDIA 的 Ampere 和 Hepper 架构; FPGA 通过硬件可重构逻辑能够针对特定任务进行深度优化, 且具有灵活性高、设计周期短的优势, 适用于高能效与低时延的应用场景; 而 ASIC 是定制的 AI 专用芯片, 通过专用矩阵乘单元、片上互连与优化内存体系进一步突破性能瓶颈, 典型代表包括 Google TPU[15]、Amazon Inferentia[16]、Hisilicon NPU 和 Qualcomm Cloud AI 100 等.

虽然计算框架与硬件的发展极大推动了人工智能的技术进步, 但框架多样性和硬件异构性也导致了组合爆炸式的适配问题. 传统方式多依赖高度优化的基础线性代数子程序库 (basic linear algebra subprograms, BLAS) 作为框架和硬件之间的桥梁, 例如 cuDNN[17]、cuBLAS[18]、Eigen[19]、MKL[20]和 OpenBLAS[21]. 然而, 随着深度学习模型深度、宽度与序列长度不断增加, 其算子形态、内存访问模式与执行路径也持续演化, 使传统依赖手工优化的算子库逐渐难以适配大型模型的快速迭代节奏. 首先, 跨硬件可移植性差, 不同硬件在并行度、访存结构、向量宽度、片上缓存等方面差异巨大, 使得算子实现难以复用, 开发成本与维护成本随平台数量急剧增加. 其次, 新算子涌现速度远超人工优化能力, Transformer、Diffusion、LLM 中不断涌现新算子, 而传统的手工优化方式难以跟上新算子的迭代速度, 影响了模型的部署效率. 此外, 新硬件生态增长迅速, 异构加速器 (NPU、DSP、RISC-V 等) 架构特点显著不同, 使传统通用编译器 (如 LLVM) 在表达深度学习计算语义、调度空间与设备特性方面受到限制. 因此, 研究人员提出了深度学习编译 (deep learning compilation) 技术, 以自动化方式将高层框架中的计算表示转换为适用于特定硬件的高效可执行代码, 从而在提升计算效率的同时, 提高系统的灵活性与可扩展性.

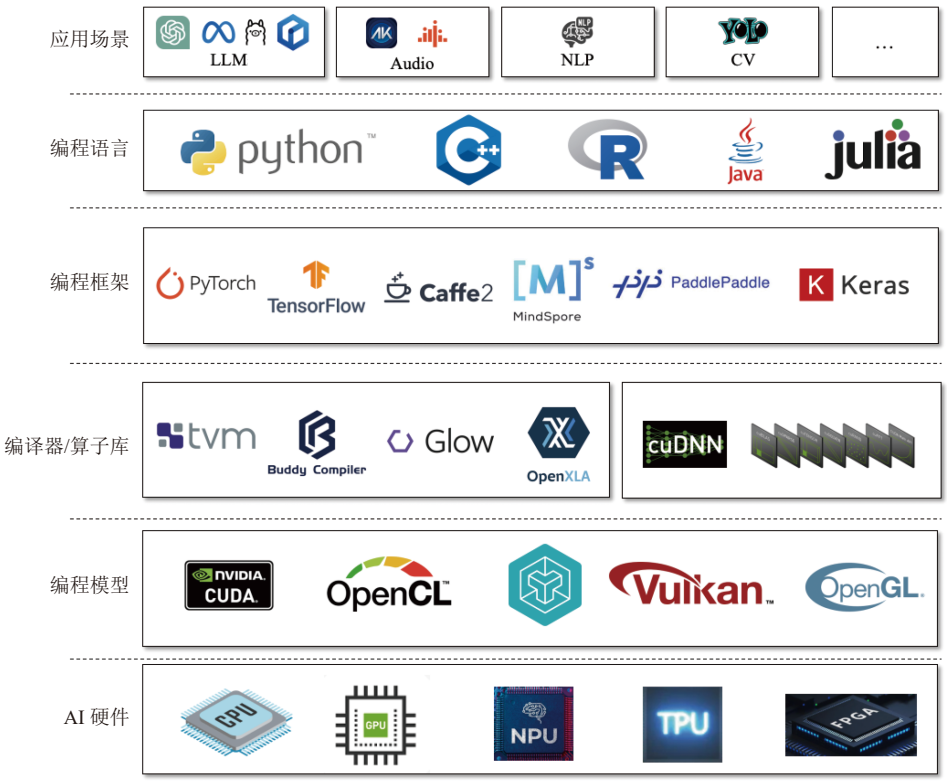


图 1 AI 全栈架构图

深度学习编译的核心思想是在统一且可扩展的中间表示 (intermediate representation, IR) 体系下, 将框架表示的深度学习模型自动生成特定后端硬件上可执行的内核代码, 并在此基础上应用多种编译优化技术, 从而在解决了灵活性和可拓展性问题的同时实现了性能上的提升. 当前, 深度学习编译技术已成为人工智能计算的重要研究方向, 并催生了一系列深度学习编译器, 如 TVM[12]、XLA[22]、TorchInductor[23]、Hidet[24]等. 这些编译器通过不同的优化策略, 在计算图优化、算子调度、自动调优等方面取得了显著进展.

Li 等人[25]对深度学习编译器的整体架构、基本原理、核心概念、评价指标等进行了较为全面的整理、对比和总结. Zhang 等人[26]详细阐述了近年来有关深度学习编译器和深度学习硬件协同设计的工作, 特别是这两种技术的结合. Xing 等人[27]对编译器框架在领域特定语言、中间表示、优化策略和自动调度方法方面的差异进行了深入比较. 然而, 面向深度学习编译优化技术的研究综述至今仍然存在很大空白, 一方面是因为与深度学习相关的领域发展迭代迅速, 整个研究体系需要不断整合层出不穷的各种新技术; 另一方面是因为深度学习编译领域目前缺少一个统一的技术框架和标准, 呈现一种百家争鸣的状态, 各方之间需要更多的交流和互鉴.

因此本文尝试构建跨模型、跨系统、跨硬件的统一优化技术视角, 对深度学习编译优化技术的最新进展进行系统性综述, 主要贡献如下.

(1) 总结并分类当前主流的深度学习编译优化技术, 分析其关键方法和实现机制.

(2) 讨论不同编译优化策略的优劣势, 并对比它们在不同深度学习编译器中的应用.

(3) 探讨当前深度学习编译领域面临的挑战, 并展望未来可能的发展方向.

本文第 1 节介绍深度学习编译器的整体架构和工作流程. 第 2、3 节详细讨论深度学习编译中的优化技术, 包括计算图优化、算子优化、调度优化等, 如图 2 所示. 第 4 节系统分析当前代表性编译器系统的设计与优化路径. 第 5 节总结全文并展望未来研究方向.

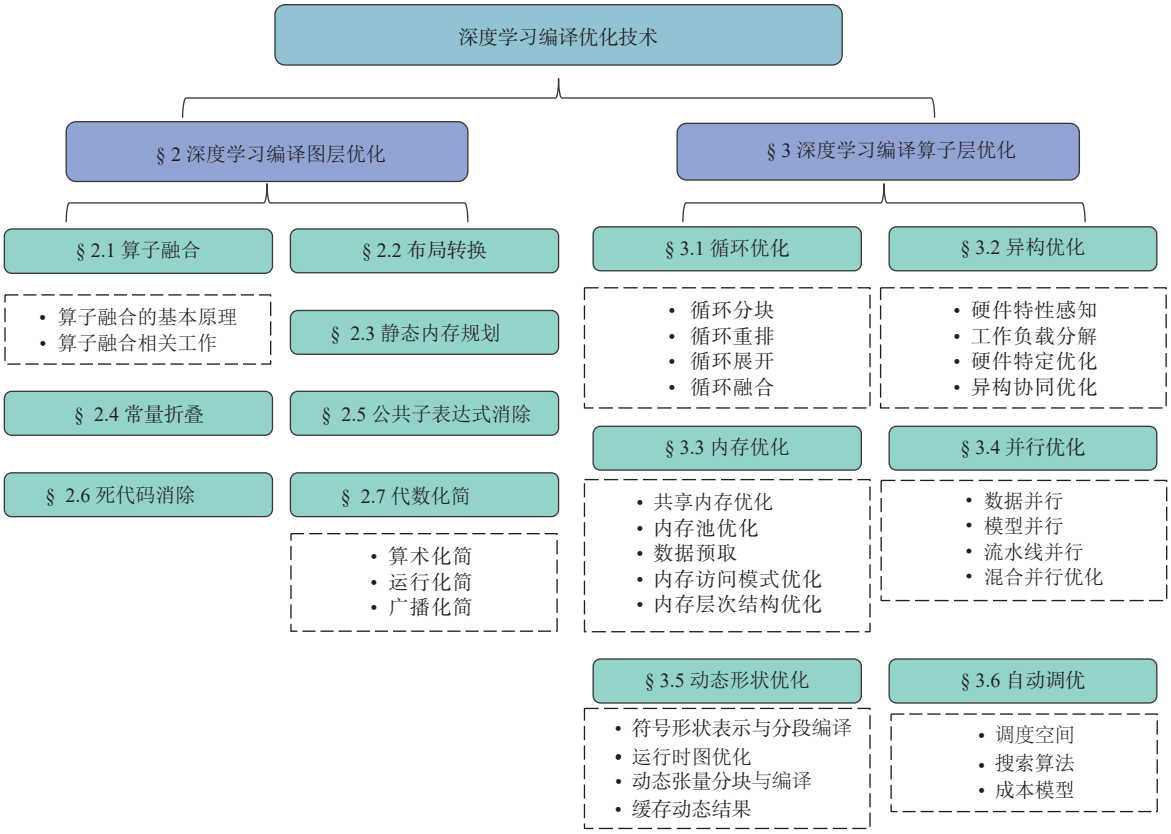


图 2　深度学习编译优化技术分类

1　深度学习编译器的整体流程

借鉴传统编译领域中分层优化的思想, 深度学习编译器主要可以分为前端 (面向深度学习模型的计算图层) 和后端 (面向深度学习硬件的算子层) 两部分, 如图 3 所示, 其中每一个部分都有独立的中间表示以及基于中间表示的特定优化技术, 从而构成了一个端到端的深度学习编译优化体系.

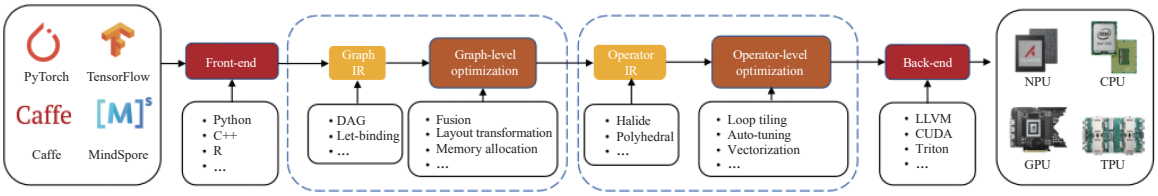


图 3　深度学习编译器的总体流程图

深度学习编译器能够应用在训练和推理两种场景中, 首先深度学习编译器的前端将基于深度学习框架表示的神经网络模型转换成计算图, 然后执行一系列的硬件无关的前端图层优化; 在此基础上, 深度学习编译器再将计算图下降到以循环为核心的算子表示, 进行硬件相关的后端算子层编译优化; 最后输出能够在深度学习硬件上高效执行的代码.

2 深度学习编译图层优化技术

深度学习模型中的数据流关系通常被表示为计算图, 计算图是一个有向无环图 (directed acyclic graph, DAG), 其中的节点表示神经网络中的算子 (operator), 边表示算子之间的数据依赖关系, 即张量 (tensor) 的流动. 计算图作为深度学习编译器中的高层中间表示描述了神经网络的计算结构, 使得深度学习编译器可以在执行模型前就能够掌握各层之间的依赖关系和数据流向, 从而为后续的一系列图层编译优化技术奠定基础.

深度学习编译器的前端负责将输入的神经网络模型转换为计算图, 并在此基础上执行一系列图级别的优化. 前端优化聚焦于计算图的整体拓扑结构, 而不涉及特定算子的底层实现, 其主要目的是通过算子间的融合、化简、转换等技术, 优化计算图的计算和存储开销, 并为算子层编译优化提供高效的计算表示, 总结如表 1 所示.

表 1 深度学习编译图层优化技术总结

技术/方法	描述	应用场景
算子融合 (operator fusion)	将多个具有数据依赖的算子合并为一个复合算子, 减少内存访问和计算开销	移动端实时图像处理 (如融合卷积+ReLU+池化操作, 降低延迟)
布局转换 (layout transformation)	根据硬件特性优化张量的存储格式 (如NCHW、NHWC), 减少内存访问冲突和计算冗余	GPU训练卷积网络 (采用NHWC布局适配cuDNN计算模式)
静态内存规划 (static memory planning)	通过一次性预分配内存, 利用就地计算复用输入张量、内存共享机制优化生命周期, 减少内存占用与动态分配开销	嵌入式多任务系统 (预分配内存块供多个模型复用, 避免频繁申请释放)
常量折叠 (constant folding)	在编译阶段计算常量表达式, 并用计算结果替换计算图中相关算子, 减少运行时计算开销	工业传感器数据处理 (固化标定参数计算, 消除冗余算术指令)
公共子表达式消除 (common sub-expression elimination)	在计算图中搜索相同的计算子图, 并共享其计算结果, 减少冗余计算	图神经网络推理 (缓存共享子图的节点特征计算结果)
死代码消除 (dead code elimination)	移除计算图中不影响程序执行结果的算子, 避免为无用节点分配空间和执行计算	模型压缩后部署 (移除剪枝丢弃的通道对应计算分支)
代数化简 (algebraic simplification)	通过数学规律重构计算图, 消除冗余计算与重复操作, 优化算子顺序与张量广播策略	计算图编译优化 (合并线性变换层系数, 简化矩阵乘加操作)

前端的优化技术大致可以分为两类.

(1) 神经网络特定的优化: 算子融合、布局转换、内存分配等.

(2) 基于传统编译的优化: 常量折叠、公共子表达式消除、死代码消除、代数简化等.

2.1 算子融合 (operator fusion)

本节聚焦于算子融合这一深度学习编译优化中的关键图层优化技术. 首先, 详细介绍算子融合的基本原理, 即如何通过合并多个存在数据依赖关系的算子, 以优化计算效率, 减少内存访问开销, 并提升硬件资源的利用率. 然后探讨算子融合的主流方法, 以及算子融合在主流深度学习编译器中的设计与应用.

2.1.1 算子融合的基本原理

算子融合[28]的核心思想是将具有数据依赖关系的多个神经网络算子组合为一个等价的复合算子, 从而增加数据复用, 减少内存访问次数, 并提高计算效率和优化硬件资源利用率.

算子融合的输入是深度学习模型中基本算子构成的原始计算图, 输出是融合算子构成的优化计算图. 融合后的算子实现通常为硬件设备上的内核, 比如 CPU 上的 Intrinsic 代码或者 GPU 上的 CUDA 代码等. 这些内核可能是由编译器自动生成的, 可能来自人工算子库 (cuDNN、cuBLAS), 也可能是由两者混合得到的.

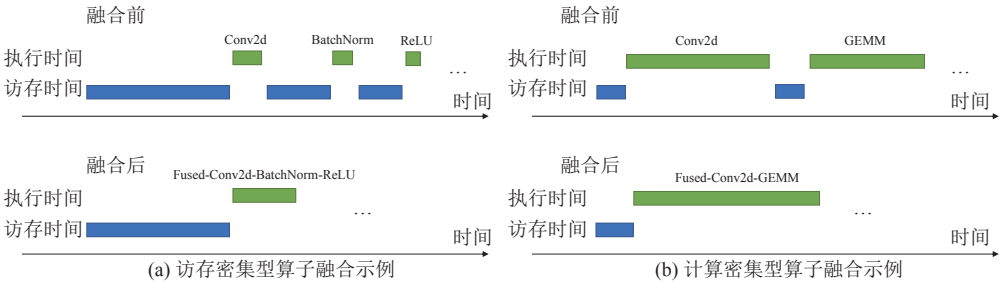
算子的计算密度指的是指单位访存量下所需的计算量, 单位是 FLOPs/byte, 用于衡量程序在计算与内存访问之间的相对密集程度. 根据计算密度的高低, 可以将算子分为两种类别: 一类是访存密集型算子, 这类算子的计算密度较低, 性能瓶颈主要受限于内存带宽. 常见的访存密集型算子有逐元素运算的算子等. 另一类是计算密集型算子, 这类算子的计算密度较高, 性能瓶颈主要受限于处理器的计算能力. 常见的计算密集型算子有矩阵乘法、卷积算子等. 针对上述两种类型的算子, 现有的融合技术也主要可以分为访存密集型算子融合和计算密集型算子融合.

(1) 访存密集型算子融合

访存密集型算子 (如激活函数、归一化、池化等) 计算量较小, 但需要频繁地进行内存读取和写入, 其性能通常受限于内存带宽和访存延迟.

为了解决访存密集型算子所面临的内存墙问题, 即处理器计算能力的提升速度远高于内存带宽的增长速度, 访存密集型算子融合通过将计算图中多个存在数据依赖且需要频繁访存的算子合并为一个算子实现, 使数据在片上缓存或寄存器中直接传递, 避免中间数据的频繁写回内存, 从而提升了访存的局部性和缓存的复用率, 并提升了模型的执行效率.

如图 4(a) 所示, 在融合前, Conv2d、BatchNorm 和 ReLU 作为独立算子分别执行: Conv2d 首先读取输入数据进行卷积运算并将结果写回内存, 随后 BatchNorm 读取 Conv2d 的输出并对数据进行标准化, 再次写入内存, 最后 ReLU 读取 BatchNorm 处理后的数据进行处理并写回内存. 这种独立执行方式产生了多次访存操作, 算子间的频繁数据交换带来了较大的访存开销, 而内存访问延迟和带宽限制了整体计算效率. 例如通过将 Conv2d、BatchNorm 和 ReLU 融合为一个复合算子 Fused-Conv2d-BatchNorm-ReLU, 在一个算子中直接完成所有运算步骤, 数据可以在算子内部通过缓存或寄存器直接传递, 在独立执行时需要 3 次写回与 3 次读取; 而融合后仅需 1 次写回与 1 次读取, 从而显著减少访存开销, 提升了模型的性能.



(a) 访存密集型算子融合示例 (b) 计算密集型算子融合示例

图 4 算子融合示意图

(2) 计算密集型算子融合

计算密集型算子 (如矩阵乘法、卷积计算等) 计算量较大, 其性能主要受限于计算单元的算力.

为了解决上述问题, 计算密集型算子融合通过将计算图中多个需要大量运算的算子合并为一个算子实现, 减少了中间步骤的冗余计算, 提高并行计算效率.

如图 4(b) 所示, 将相邻的 Conv2d 和 GEMM 进行融合, 直接在处理器的一个算子中完成所有计算操作. 一方面, 减少了算子间切换的开销, 因为数据不再需要在不同的算子间频繁传递, 消除了中间不必要的访存操作. 另一方面, 融合后的算子计算量增加, 可以更加充分利用硬件的并行计算能力, 实现更高的并行度, 从而提升系统的整体执行效率.

图 4(a) 为访存密集型算子融合的示例, 通过将 Conv2d、BatchNorm 和 ReLU 融合为一个复合算子 Fused-Conv2d-BatchNorm-ReLU, 减少了多次内存读写操作. 图 4(b) 为计算密集型算子融合的示例, 通过将 Conv2d 和 GEMM 融合为一个复合算子 Fused-Conv2d-GEMM, 减少了中间步骤的冗余计算, 提高了计算资源的利用率.

算子融合通过优化内存访问和计算资源的利用率, 有效提升了深度学习模型的执行性能, 许多先进的编译框架基于算子融合的思想, 提出了改进的优化策略, 进一步推动了深度学习编译技术的发展. 接下来, 我们将介绍目前主流的算子融合相关工作, 探索它们如何利用算子融合的技术来提升性能.

2.1.2 算子融合相关工作

随着深度学习模型的规模和复杂度不断提升, 以算子库为代表的传统高性能计算优化手段已无法满足实际需求. 因此, 算子融合技术逐渐成为加速深度学习工作负载的关键优化手段. 算子融合经历了从手工融合到自动融合的过程, 在被广泛应用于深度学习之前, 手工内核融合[29]等优化方法已在 GPU 编程中广泛采用, 开发者通过提前

编写好融合规则和融合后的算子, 将多个计算内核的任务手动整合到一个内核中, 以降低内核启动开销和全局内存访问负担, 从而提高计算效率. 虽然手工融合的方式在部分常见模式下能够取得不错的效果, 但是这种方式人工开销较大, 同时缺乏通用性和泛化性.

为了解决手工融合泛化性差的问题, 以 XLA、TVM 等为代表的深度学习编译器开始转向基于模式匹配的自动融合优化技术, 其核心思想是通过遍历整张计算图来匹配可融合的子图并进行模式替换. XLA[22]的研究表明, 在长链、以访存为主的逐元素算子场景下, 通过激进的算子融合, 与未融合执行相比, 最高可实现 10.56 倍的加速. 相比之下, TVM[12]的实验结果显示, 算子融合在图级层面通常可带来约 1.2–2 倍的性能提升, 而在内核级层面, 针对 NVIDIA Titan X 上的部分卷积层可实现最高约 3 倍的加速. 但这些编译器的优化模式相对固定, 会错过一些潜在的优化机会, 难以充分利用硬件性能.

随着神经网络拓扑结构的复杂化, 遍历策略的时间复杂度将呈指数式增长, 从而导致融合搜索空间规模爆炸的问题[30]. 为此 DNNVM[31]提出将依赖于多个算子或由不同算子组成的算子设置为屏障, 并假设融合的算子组永远不会包含这些屏障, 从而缩减了搜索空间的规模. 然而, 对于 SqueezeNet[32]等分支较多的网络, 这种方法将错过大量潜在的融合机会. Zheng 等人[33]提出在不违反数据依赖关系的前提下, 通过将复杂模型转换为线性拓扑并应用动态规划算法来寻找最优的算子融合结果, 但是这种方法在构建线性结构时会引入大量时间开销.

针对算子融合选择受限导致性能优化不充分的问题, Jia 等人[34]提出了一个基于放宽算子融合规则的优化方案, 该方案通过设计松弛的图替换策略, 在算子之间允许临时性能下降的中间状态, 并结合回溯搜索和图分割技术, 从而探索了更多的融合选择. 为了减少设计算子融合规则的工作量, TASO[35]第 1 个提出使用自动生成图替换来优化 DNN 计算图, 同时借助形式化验证技术确保融合过程的正确性. DNNFusion[36]提出了基于分类的算子融合, 其核心思想是将算子及其组合进行分类, 并为不同类型的算子组合设定对应的融合规则, 最后根据不同算子组合的融合优化效果来选择最佳的融合方案. NeoFlow[37]通过引入基于表达式的表示和自动梯度计算, 提供了灵活的算子融合框架, 尤其在面对新兴深度学习模型时, 能够自动生成适用于新算子的融合策略, 填补了之前工作在新型算子支持上的空白. Substation[38]提出了一种以数据移动为核心的融合策略, 通过对训练过程中的数据流图进行全局优化, 从而提高了模型的训练效率.

随着高性能硬件的快速发展, 单个算子无法充分利用硬件提供的并行性, 从而导致实际性能与峰值性能之间存在很大差距. 为了解决多核芯片与单算子并行度不匹配的并行墙问题, 许多工作在不同的启发式指导下探索增加算子间的并行性. MetaFlow 将匹配特定模式的多个算子融合为一个更大的算子, 以提升并行度, 使硬件利用率提升 10%–25%, 实现 1.12–1.41 倍的整体加速. Tang 等人[39]提出了一种贪心策略, 直接在 CPU 上执行所有可用的算子, 以最大化资源利用率. 这些方法虽然能够在一定程度上实现算子的局部并行化, 但并不能得到整个网络的全局最优调度. 为了进一步增加并行度, IOS[40]提出了两种并行策略: 算子合并 (将相同类型的算子合并为一个算子) 和并发执行 (将计算图分为多段, 各段之间按序执行, 再将同一段中的算子划分为多个组, 有依赖的算子会被划分在同一组, 不同组的算子可以并发执行), 并结合算子类型、算子形状和硬件设备等因素自动选择合适的并行策略. Rammer[41]过将计算过程中互不依赖、低计算量子图进行打包, 统一调度为一个计算内核并发执行, 从而提升硬件资源的利用率.

针对访存密集型算子融合中存在的低效率问题以及不规则形状问题, AStitch[42]和 FusionStitching[43]均致力于通过更有效的内存管理和数据重用策略来提高 GPU 资源的利用率, 并结合 JIT 编译优化根据特定任务的需求实时生成融合策略和代码. FusionStitching 主要通过对算子依赖关系的处理以及对数据重用的支持, 从而扩大算子融合的范围 FusionStitching 通过在 GPU 上减少内核启动和访存, 在部分算子子图上可实现 1.3–2.0 倍加速. AStitch 则进一步引入了多维度的优化空间, 从依赖性、内存层次结构以及并行性的联合角度出发, 提出了 4 种算子拼接策略: 独立的 (Independent)、本地的 (Local)、区域的 (Regional) 和全局的 (Global), 每种策略将数据存放于不同的内存空间, 从而拓展了融合的范围. AStitch 在处理 Transformer 层中的逐元素算子时, 通过跨算子拼接减少访存, 实现 1.24–1.52 倍加速.

随着硬件专用性的持续提升, 计算核心的速度和外部内存之间的差距愈发显著, 导致许多计算密集型算子的

性能受制于内存带宽, 为此 Chimera[44]提出了一种优化框架, 通过算子间融合 (优化计算块执行顺序以最小化数据移动量) 和算子内优化 (使用硬件特定的可替换微内核优化计算过程), 生成高效的融合内核, 这一框架不仅提高了数据复用率, 还充分利用了不同硬件架构的特点, 使卷积-GEMM 混合子图可获得 1.3–1.8 倍加速.

为了实现不同算子融合方式的协同组合, Apollo[45]提出了一个多层规约式融合架构, 其主要思想是将不同的融合方式实现为多个层次, 同时覆盖访存密集型和计算密集型算子的两种融合策略, 不仅在图层间进行算子融合, 还在每一层内部寻找并行计算的机会, 从而得到最佳的融合组合方式.

针对深度学习的动态特性, MAGPY[46]提出了一种框架, 通过监控程序执行状态生成更完整的计算图. 该框架通过记录执行过程中的状态信息, 通过记录执行时的状态信息, 并根据程序状态生成参考图, 从而实现自动化的计算图转换, 解决了在动态语言 (如 Python) 中融合效果不佳的问题.

2.2 布局转换 (layout transformation)

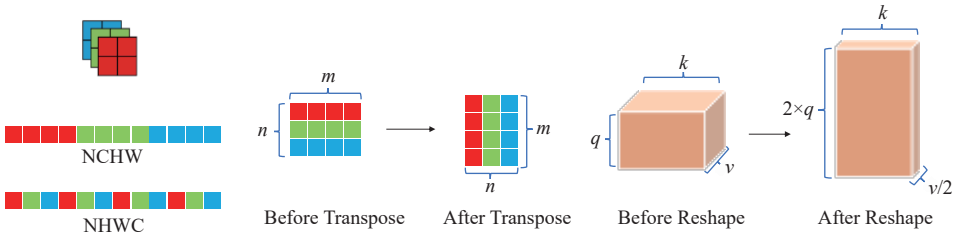
在深度学习计算中, 张量的存储格式对计算效率有着直接影响. 张量的本质是一个多维数组, 通常用于表示深度学习模型的多维数据. 例如, 卷积神经网络的特征图就是一个四维张量, 其主要维度有: (1) N: Batch, 特征图的批次数. (2) H: Height, 特征图的高度, 即垂直高度方向的像素个数. (3) W: Width, 特征图的宽度, 即水平宽度方向的像素个数. (4) C: Channels, 特征图中的通道数.

根据四维数据存储的先后顺序不同, 张量的存储格式主要可以分为 NCHW 和 NHWC 两种情况.

• NCHW 中同一个通道的数据值连续排布, 更适合对每个通道的数据独立运算的操作, 如池化. NCHW 计算时需要的存储更多, 但访存与计算的控制逻辑相对简单, 更适合内存带宽较大且并行计算能力强的 GPU.

• NHWC 中不同通道中同一位置的元素顺序存储, 更适合对不同通道的同一位置数据做相同运算的向量化操作, 如卷积. NHWC 计算的时延较低, 需要的内存空间也很小, 但计算控制灵活且复杂, 更适合具有 SIMD 指令优化的 CPU.

不同深度学习框架会按照不同的维度顺序来存储特征图数据: 以 NPU/GPU 为基础的 PyTorch 和 MindSpore 默认使用 NCHW 格式, 而 TensorFlow 以及面向移动端部署的 TFLite 则采用了 NHWC 格式. 尽管两种布局下的数据实际上都是线性存储的, 但是不同的存储顺序会导致不同的数据访问特性, 因此即使进行同样的运算, 相应的计算性能也会有差异. 与此同时, 一些深度学习编译器依赖于特定于硬件的库来实现更高的性能, 而这些库可能需要特定的布局. 此外, 一些边缘设备通常配备异构计算单元, 不同的单元可能需要不同的数据布局以更好地利用数据, 因此需要深度学习编译器提供一种跨各种硬件执行布局转换的方法, 从而将数据布局转化为后端设备友好的形式, 如图 5 所示.



(a) 两种数据布局 (b) 张量转置操作 (c) 张量重塑操作

图 5 布局转换示意图

布局转换是提升模型性能的重要技术之一, 广泛应用于图像识别等领域. 转换的具体方式是找到在计算图中存储张量的最佳数据布局, 然后将布局转换节点 (Transpose、Reshape 等) 插入图中. 除了满足硬件平台对不同张量布局的优化需求, 布局转换还能减少内存访问冲突, 并通过优化数据排列降低计算冗余. cuDNN v8[3]对卷积算子在 NHWC 布局下可启用 Tensor Core 加速, 使部分推理任务吞吐量提升 1.3–1.8 倍.

需要注意的是, 冗余的布局转换会显著降低深度神经网络的执行性能, 因为每次转换都会引入额外的内存操

作, 增加带宽消耗, 破坏数据局部性, 导致计算效率下降, 这种问题常出现在诸如 Transformer 等复杂模型中. 因此, 减少冗余的布局转换对于提高 DNN 的效率至关重要. SmartMem[47]通过将 DNN 中常见的算子根据输入/输出布局分类, 同时考虑硬件内存特性, 从而消除了冗余的布局转换, 使 Transformer 模型端到端推理延迟降低 15%–22%.

2.3 静态内存规划 (static memory planning)

深度学习模型涉及大量张量存储, 合理的内存管理对计算效率至关重要. 深度学习模型所占用的静态内存空间指的是需要在模型初始化时一次性申请完的内存空间, 从而避免推理时频繁的申请操作. 例如网络中的权重参数、常量和输入输出等.

静态内存规划技术包括如下两类.

● 就地计算 (in-place computation): 如图 6(a) 所示, 对于逐元素操作, 可以直接在输入张量上进行计算并更改张量的内容, 而无需额外分配新的内存. 例如, 某些激活函数 (如 ReLU) 可直接覆盖输入数据, 而无需创建新的张量存储结果. 当结合张量生命周期分析和静态内存复用技术 (例如 MODeL[48]) 时, 可以进一步降低整体内存需求 30% 以上, 部分结构甚至可节省 50%–70% 的内存.

图 6 就地计算和内存共享示例图

● 内存共享 (memory sharing): 如图 6(b) 所示, 通过分析计算图中的张量生命周期, 两个操作可以复用相同大小的内存块, 从而减少整体内存占用. 例如, TVM 采用 Graph Planner 机制, 在编译时合理分配内存池, 能减少 20%–60% 的 GPU 全局内存占用, 使部分模型可在具有更小显存的设备上运行.

2.4 常量折叠 (constant folding)

常量折叠是图层优化中最基础且最有效的静态优化技术之一, 其核心思想是在编译阶段提前计算所有可静态求值的常量表达式, 并将其结果直接写入计算图中, 从而减少运行时的算术开销并缩短推理延迟. 与传统编译器类似, 深度学习编译器中的常量折叠主要作用于加法、乘法、广播、转置、维度推导等可在编译时确定结果的算子.

如图 7(a) 所示, 两个常量参与的算术表达式可在编译时直接化简为单一常量张量, 执行时无需重复计算. 此外, 在现代深度学习编译器中, 常量折叠还常与算子融合结合使用: 例如 Conv+Bias+Scale 等组合可在编译阶段将常量线性变换提前折叠到卷积权重中, 从而减少一次访存与一次加法操作, 提高整体性能.

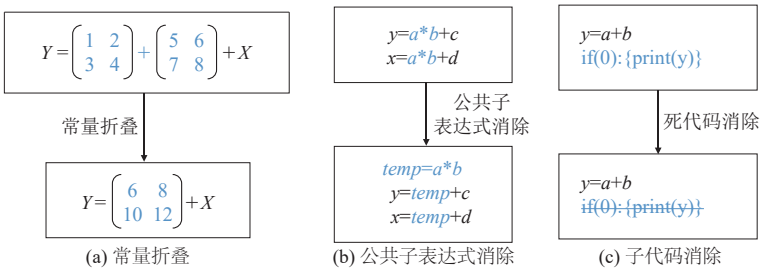


图 7 常量折叠、公共子表达式消除、死代码消除示意图

根据 TVM、XLA 与 TensorRT 的实践经验, 常量折叠可带来 3%–12% 不等的推理加速, 尤其在存在大量常量权重计算 (如 LayerNorm 参数折叠、卷积 Bias 融合、静态 Rshape/Transpose) 时效果显著. 例如, 在 BERT 结构中, 常量折叠可移除约 2%–5% 的节点, 并使推理延迟降低 2%–7%. 近年来, 一些系统 (如 BladeDISC[49]、TensorRT) 进一步提出跨子图的全局常量折叠, 能够在更大范围内检测冗余常量子图, 减少重复计算, 提升端到端推理效率. 这些进展表明常量折叠在现代深度学习编译优化中仍然保持关键地位.

2.5 公共子表达式消除 (common sub-expression elimination)

公共子表达式消除旨在识别计算图中重复出现的相同子图结构, 并仅计算一次, 将其结果复用到多个使用点. 与传统编译中的公共子表达式消除不同, 深度学习中的公共子表达式消除更关注跨张量运算的结构相似性 (如多处出现相同的线性变换、相同的 *Broadcast*+*Add* 序列等).

如图 7(b) 所示, 表达式 *a*b* 在两个后续计算中被重复使用, 通过提取公共表达式, 可减少一次乘法计算, 降低运行时开销. 对于图神经网络和多分支模型, 公共子表达式消除的效果尤为显著. 例如, 当多个分支使用相同节点特征转化算子时, 可将节点特征的线性变换结果缓存并复用.

为了进一步提高公共子表达式消除的适用性, 近年来多个系统提出了图等价性检测技术, DNNFusion[36]表明, 编译期的公共子表达式消除在深度模型中具有重要价值: 在融合代码生成阶段, 通过数据流树中的公共子树识别, 可避免重复计算同一中间结果; 同时, 其数学性质驱动的图重写能够系统性地消除冗余与暴露融合机会. 在 GPT-2 上, 图重写能使可融合层减少约 18%. 进一步地, DNNFusion 通过模式匹配与代数变换识别语义等价子图, 在 ResNet、MobileNet、R-CNN 及 Transformer 等模型上实现了最多 8.8 倍的融合机会, 并带来最高约 9.3 倍的端到端加速, 融合率相较现有框架提升 1.3–8.8 倍.

2.6 死代码消除 (dead code elimination)

死代码消除旨在移除计算图中对最终输出没有贡献的算子及其中间张量, 以减少不必要的计算与内存消耗. 死代码通常不是由模型编写时产生, 而是由前端优化 (布局转换、融合、化简)、子图重写、算子替换等过程引入的, 因此死代码消除通常作为近端优化的最后一步执行.

如图 7(c) 所示, 恒为 false 的分支, 以及后续未被任何算子消费的张量 *y*, 可安全移除. 在真实的深度学习编译任务中, 死代码常见于: (1) 剪枝后无效通道或特征分支被保留但永不使用; (2) 融合或化简后被替代的旧算子残留在图中; (3) 自动布局转换生成但随后被优化掉的冗余 Transpose/Reshape; (4) 自动调优阶段探索产生的临时候选算子未被采用.

在 Transformer、Diffusion、GNN 等模型的编译优化中, 死代码消除可减少 5%–15% 的算子数量[50]. BladeDISC 进一步提出形状不可达性分析来识别动态形状条件下的不可执行分支, 使死代码消除能够覆盖更多复杂场景, 提升整体推理吞吐量. 总体来看, 死代码消除不仅减少了执行时间, 也显著降低了显存峰值, 是深度学习编译图层优化中不可或缺的步骤.

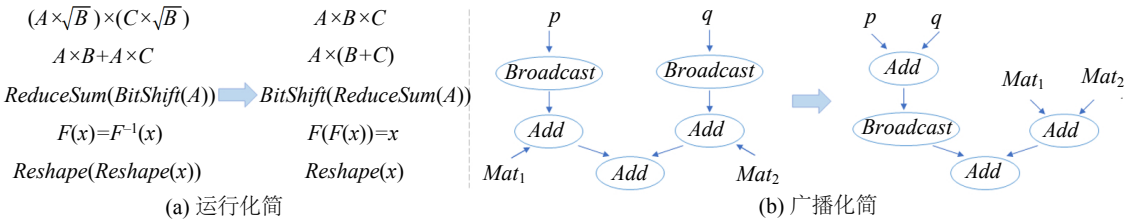
2.7 代数化简 (algebraic simplification)

代数化简的目的是利用交换律、结合律等规律调整计算图中算子的执行顺序, 或者删除不必要的算子, 从而提高模型的计算效率, 典型系统如 DNNFusion 和 Unity[50]. 深度学习编译器中的代数化简可以通过子图替换的方式完成.

2.7.1 算术化简

DNNFusion 基于 3 种算数运算法则的图重写对计算图进行化简, 从而减少 12%–25% 的算术操作量, 实现 1.10–1.32 倍的整体性能提升

(1) 结合律化简: 如图 8(a) 中第 1 行所示, 可以将两个 $\sqrt{B}$ 算子合并为一个 B 算子, 从而消除了一次乘运算.



(a) 运行化简 (b) 广播化简

图 8 算数化简示意图

(2) 分配律化简: 如图 8(a) 中第 2 行所示, 2 个 *Mul* 算子和 1 个 *Add* 算子的组合可以用 1 个 *Add* 算子和 1 个 *Mul* 算子来替换, 从而消除了 1 个 *Mul* 算子.

(3) 交换律化简: 在保证两个算子交换位置合法性的前提下, 通过交换算子位置来减少计算量. 如图 8(a) 中第 3 行所示, 通过交换 *BitShift* 算子和 *ReduceSum* 算子的位置, 可以减少 *BitShift* 算子的计算量.

2.7.2 运行化简

通过运行化简可以减少执行时冗余的算子或者算子对.

(1) 对合算子化简: 针对逆函数等于自身的函数, 如取反、倒数、逻辑非、矩阵转置等. 如图 8(a) 中第 4 行所示, 连续 2 次 F 操作可以相互抵消.

(2) 幂等算子化简: 针对连续作用在某一元素上多次与一次的效果相同的算子. 如图 8(a) 中第 5 行所示, 对同一数据的两次 *Reshape* 操作可以由一次 *Reshape* 操作完成.

2.7.3 广播化简

针对多个形状不同的张量, 化简为计算所需的最小广播运算数量. 如图 8(b) 所示, 在左侧的原始计算图中, 输入 p 和 q 分别进行 *Broadcast* 操作, 然后执行 *Add* 生成矩阵 Mat_1 和 Mat_2, 最后通过额外的 *Add* 相加, 每个分支需要独立进行广播, 导致了重复计算. 在右侧的优化图中, 首先 p 和 q 直接相加得到中间结果, 然后对这个结果执行一次 *Broadcast*, 再分别与 Mat_1 和 Mat_2 的计算结果进行 *Add* 操作. 该优化通过将加法前置, 先进行加法运算再广播, 减少了广播操作的冗余, 显著降低了计算量和内存占用. 整体上看, 广播化简的方法在处理大型张量时, 能够有效提升性能. 在 Transformer 中, TVM 通过将广播前置可减少 20%–40% 的广播节点, 将推理延迟减少 8%–15%.

3 深度学习编译算子层优化技术

算子的核心在于其计算与调度, 其中计算 (computation) 指的是算子的形式定义, 即算子对应的算法流程. 而调度 (schedule) 指的是算子的具体实现, 即如何执行算子所规定的计算. 同一算子只有一种明确的计算定义, 但可以有多种不同的调度方式. 为了简化张量程序优化, Halide[51]提出了一种将张量程序的计算定义与计算调度解耦的编程范式, 从而使深度学习编译器可以结合具体的应用场景和设备资源为算子选择最高效的实现方式.

在将深度学习模型转换为图层 IR, 并实现硬件无关的图层编译优化后, 深度学习编译器会再将计算图下降 (lowering) 到以循环为核心的算子层 IR, 然后执行算子层的一系列编译优化. 由于针对不同的后端硬件架构, 不同的算法实现在性能上会有显著差异, 因此算子层优化聚焦于改进算子内部的具体实现 (输入、输出、循环和计算逻辑等), 从而生成在特定硬件上性能最好的算子. 总结如表 2 所示.

表 2 深度学习编译算子层优化技术

技术	描述	应用场景
循环优化 (loop optimization)	循环分块、循环展开、循环融合等技术优化循环结构, 提升数据局部性和并行性	大规模数据计算, 密集计算任务加速 (通过块和展开减少循环层级开销)
异构优化 (heterogeneous optimization)	分解深度学习任务到 CPU、GPU 等硬件, 利用各自特性进行任务分配、硬件特定优化及协同调度, 提升计算效率与降低功耗	跨平台计算任务 (协同多硬件加速复杂工作流)
内存优化 (memory optimization)	共享内存优化、内存池优化、数据预取等技术减少内存访问延迟, 提升内存带宽利用率	高吞吐数据处理系统 (减少内存碎片化, 提升批量任务吞吐量)
并行优化 (parallelization)	任务并行、数据并行、模型并行和流水线并行等技术充分利用多核、多线程硬件资源	分布式计算框架 (扩展计算资源, 处理超大规模任务)
动态形状优化 (dynamic shape optimization)	通过符号形状表示、运行时分块调度及缓存机制, 在保持模型灵活性的同时, 动态调整计算图与内存分配策略	动态输入推理系统 (自适应可变输入尺寸)
自动调优 (auto-tuning)	构建调度空间、应用搜索算法, 自动化探索特定的最优代码调度策略, 降低人工调优成本, 提升跨平台计算效率	跨硬件部署框架 (自动适配不同硬件后端)

3.1 循环优化 (loop optimization)

深度学习模型中的大部分计算任务都涉及对多维数组 (张量) 的操作, 而这些操作通常通过循环结构来实现. 因此, 循环优化是深度学习编译器中至关重要的优化技术之一. 循环优化的核心目标是在保证程序语义不变的前提下, 通过调整循环的迭代顺序、规模和结构来挖掘程序的数据局部性计算和并行性, 从而充分利用底层硬件架构的特性, 提升深度学习模型的执行效率. 具体而言, 循环优化技术包括循环分块 (loop tile)、循环重排 (loop reorder)、循环展开 (loop unroll) 和循环融合 (loop fuse) 等. 图 9 展示了循环优化的几种常见技术及其效果.

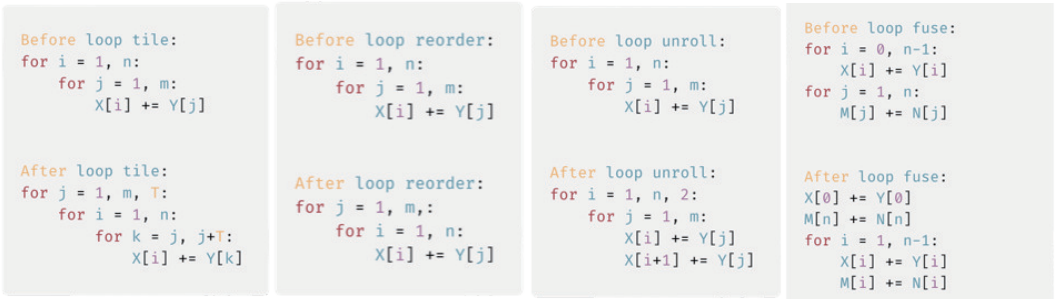


图 9 循环优化示意图

3.1.1 循环分块 (loop tiling)

循环分块是一种利用缓存局部性来优化数据访问的重要技术. 当处理大规模数据集时, 数据访问的局部性较差, 容易导致缓存未命中 (cache miss), 从而增加内存访问延迟. 循环分块通过将大规模数据集划分为多个小块 (tiles), 使得每个小块的数据能够完全加载到缓存中, 并在缓存中被多次复用, 从而减少内存访问的开销.

具体实现时, 循环分块将外层循环分解为多个嵌套的内层循环, 确保内层循环所访问的数据块大小适合缓存容量. 通过这种方式, 数据块在被多次复用后才被替换出缓存, 从而提高了数据的局部性, 减少了内存访问延迟. 此外, 由于块内的循环迭代规模较小, 不同块之间的迭代可以利用多个处理单元并行执行, 从而进一步提升程序的并行度.

循环分块的关键在于确定分块的大小和形状. 分块形状通常由程序的依赖关系决定, 常见的分块形状包括矩形和平行四边形等. 分块大小的选择则可以通过性能调优或运行时动态调整来确定. 过小的分块会增加循环控制和同步的开销, 而过大的分块则可能导致数据冲突和缓存失效, 因此需要在两者之间进行权衡.

3.1.2 循环重排 (loop reordering)

循环重排通过调整多重嵌套循环的执行顺序来优化程序的局部性和并行性. 具体而言, 循环重排将可向量化的循环移动到循环嵌套的最内层, 从而充分利用硬件的向量指令集和缓存, 实现对多个数据元素的并行处理. 此外, 循环重排还将可并行化的循环移动到循环嵌套的最外层, 以增加程序的并行粒度, 使得多个处理单元能够同时执行独立的循环计算, 从而加快程序的执行速度.

循环重排的关键在于保证循环变换前后的语义一致性, 即变换后的循环结构不会改变程序的依赖关系和执行结果. 通过合理的循环重排, 编译器能够显著提升程序的局部性和并行性, 从而优化深度学习模型的执行效率.

3.1.3 循环展开 (loop unrolling)

循环展开通过将循环体中的部分或全部迭代展开为多个独立的指令, 从而减少循环控制的开销. 循环展开特别适用于循环次数已知且数据之间没有依赖关系的情况. 通过展开循环, 编译器可以生成大量针对每个数据的独立指令, 从而为具有多个功能单元的处理器提供指令级并行性, 并优化指令流水线的调度.

循环展开的优点是能够减少循环控制的开销, 并提高指令级并行性. 然而, 过度的循环展开可能导致代码膨胀, 增加缓存冲突和内存消耗. 因此, 编译器需要在循环展开的深度和代码膨胀之间进行权衡, 以找到最优的展开策略.

3.1.4 循环融合 (loop fusion)

循环融合通过将多个嵌套循环中的操作合并到一个循环中, 形成一个统一的迭代空间. 这种优化技术能够缩短生产数据和使用数据之间的距离, 减少循环迭代次数, 从而提高数据的局部性和缓存的利用率. 循环融合特别适用于多个循环之间存在数据依赖关系的情况, 通过合并循环, 可以减少中间数据的存储和传输开销, 从而加快程序的运行速度.

循环融合的关键在于维护循环间的依赖关系, 确保融合后的循环结构不会改变程序的语义. 此外, 循环融合还需要考虑融合后的循环体积是否过大, 以避免增加缓存冲突和内存消耗. 通过合理的循环融合, 编译器能够显著提升深度学习模型的执行效率.

3.2 异构优化 (heterogeneous optimization)

随着深度学习模型的规模不断扩大, 单一硬件平台往往难以满足其计算需求. 异构优化通过充分利用多种硬件平台 (如 CPU、GPU、FPGA/ASIC、TPU 等) 的计算特性, 将深度学习任务分解并分配到最合适的硬件上执行, 从而显著提升计算效率并降低功耗. 异构优化的核心目标是通过硬件特性感知、工作负载分解、硬件特定优化以及异构协同优化, 显著提升深度学习模型的计算效率, 并降低功耗.

3.2.1 硬件特性感知 (hardware awareness)

深度学习编译器首先需要对各种后端硬件的特性进行深入分析, 以便合理分配计算任务. 不同硬件平台在计算能力、内存带宽、功耗等方面存在显著差异, 因此编译器需要根据硬件的特性选择最优的计算策略. 具体说明如下.

• CPU 特性: 具有较强的通用计算能力, 适合数据预处理等串行计算任务, 或条件判断等涉及复杂控制逻辑的任务.

• GPU 特性: 具有高吞吐量和强大的并行计算能力, 适合大规模矩阵运算和卷积操作等任务.

• FPGA/ASIC 特性: 具有高度的灵活性和可定制性, 能够针对特定任务进行硬件级别的优化. FPGA 适合低功耗、低延迟的应用场景, 而 ASIC 则能够为深度学习提供极致的性能.

• TPU 特性: 专为深度学习设计的硬件, 通过专用的矩阵乘法单元和高效的张量处理能力, 能够显著加速深度学习模型的训练和推理, 特别适合张量计算.

3.2.2 工作负载分解 (workload partitioning)

在硬件特性感知的基础上, 深度学习编译器将深度学习计算任务分解为不同的子任务, 并根据硬件特性将其分配给最合适的设备. 工作负载分解的目标是最大化硬件资源的利用率, 同时减少数据传输和同步的开销. 具体策略包括如下 3 种.

• 计算密集型任务如卷积操作和矩阵计算通常分配给 GPU 或 TPU, 以利用其强大的并行计算能力.

• 数据预处理等低吞吐计算任务通常分配给 CPU, 以利用其高效的串行处理能力.

• 稀疏计算或特定优化算子任务可以分配给 FPGA 或 ASIC, 以利用其硬件可重构性和定制化优势.

3.2.3 硬件特定优化 (hardware-specific optimization)

在任务分配完成后, 深度学习编译器需要结合硬件架构和任务特性, 生成硬件特定的优化代码. 硬件特定优化的目标是充分利用硬件的计算资源, 减少内存访问延迟, 并提升计算效率. 具体优化策略包括如下 4 种.

• CPU 优化: 利用 SIMD 指令集 (如 AVX、NEON) 并行执行张量运算, 以提升计算效率. 此外, 编译器还可以通过循环展开、向量化等技术优化 CPU 的计算性能.

• GPU 优化: 通过优化全局内存访问, 利用 GPU 的线程并行和 Warp 协作机制, 最大化硬件利用率. 编译器还可以通过共享内存优化、数据预取等技术减少 GPU 的内存访问延迟. 例如 Gandiva[52]、Tiresias[53]、THEMIS[54]、Gavel[55]通过各种技术实现了对于 GPU 集群调度的优化.

• TPU 优化: 对 TPU 执行静态优化, 自动生成支持 TPU 矩阵乘法指令的代码, 以充分发挥其张量计算能力. 编译器还可以通过张量分块、内存层次优化等技术进一步提升 TPU 的性能.

• NPU 优化: 李帅江等人[56]提出一种面向昇腾处理器的高性能同步原语自动插入方法, 通过引入“虚拟同步资源”的抽象将同步原语的插入和物理同步资源的选择进行解耦.

3.2.4 异构协同优化 (heterogeneous co-optimization)

除了针对单一硬件的优化, 深度学习编译器还需要考虑不同硬件之间的协同工作. 异构协同优化的目标是通过任务调度和数据传输优化, 减少硬件之间的通信开销, 并最大化整体系统的计算效率. 具体策略包括如下 2 种.

• 任务调度优化: 通过动态任务调度算法, 将计算任务分配给最合适的硬件, 并在任务执行过程中实时调整任务分配策略, 以应对硬件负载的变化.

• 数据传输优化: 通过数据压缩、流水线传输等技术, 减少硬件之间的数据传输开销. 此外, 编译器还可以通过数据预取和数据重用技术, 减少数据传输的等待时间.

3.3 内存优化 (memory optimization)

在深度学习模型的推理和训练过程中, 内存访问效率是影响整体性能的关键因素之一. 后端硬件的内存空间通常是分层的 (如寄存器、共享内存、全局内存等), 不同层次的内存具有不同的特性 (如大小、速度等). 因此, 深度学习编译器需要通过优化数据的存储方式以及数据在不同层次存储器之间的传递, 以最大化硬件资源的利用率, 减少内存访问延迟, 并提高模型的执行效率. 常见的内存优化方法包括共享内存优化、内存池优化以及数据预取等.

3.3.1 共享内存优化 (shared memory optimization)

共享内存是 GPU 架构中的一种片上内存, 其访问速度远快于全局内存, 但容量有限. 深度学习编译器可以通过将频繁访问的数据划分为适合共享内存的更小切片, 并将其存储在共享内存中, 以减少对全局内存的重复访问, 从而提升数据访问的局部性和缓存的复用率.

Halide 通过定义多种内存优化相关的原语来优化内存访问, 例如 cache_read 将数据先加载到片上共享内存, 然后再参与运算, 从而减少数据访问时间, storage_align 将数据与共享内存的 Bank 大小对齐, 防止多个线程访问同一 Bank, 从而减少 Bank 冲突, 进一步提升并行计算的效率, prefetch 用于实现数据预取, 提前将未来可能需要的数据加载到缓存中, 以减少等待时间.

3.3.2 内存池优化 (memory pool optimization)

深度学习模型在推理和训练过程中涉及大量张量的分配和释放, 频繁的动态内存分配会导致显著的开销. 内存池优化通过预分配内存块 (称为内存池) 并在计算期间重用这些内存块, 从而最小化动态内存分配的开销, 同时最大化内存资源的重用率.

具体而言, 深度学习编译器不再为每个操作动态分配和释放内存, 而是从预分配的内存池中重用内存. 这种方法特别适用于内存管理成为瓶颈的大规模深度学习模型, 能够显著减少内存碎片化, 并提高内存访问的效率.

3.3.3 数据预取 (data prefetching)

数据预取是一种通过提前将未来可能需要的数据加载到缓存中, 以减少内存访问延迟的技术. 深度学习编译器可以通过分析计算图的数据访问模式, 预测未来可能访问的数据, 并在计算过程中提前将这些数据加载到高速缓存中. 这种方法能够有效减少内存访问的等待时间, 尤其是在处理大规模数据集时, 能够显著提升模型的执行效率.

3.3.4 内存访问模式优化 (memory access pattern optimization)

深度学习编译器还可以通过优化内存访问模式来提升内存带宽的利用率. 例如, 对于 GPU 等并行计算设备, 编译器可以通过调整数据的存储布局 (如将数据存储为连续的内存块) 或优化循环结构 (如循环分块、循环展开等), 以减少内存访问的冲突和不规则访问, 从而提升内存带宽的利用率.

3.3.5 内存层次结构优化 (memory hierarchy optimization)

现代硬件通常具有多级内存层次结构 (如 L1、L2、L3 缓存、全局内存等). 深度学习编译器可以通过分析数据访问的局部性, 将频繁访问的数据放置在高速缓存中, 而将不频繁访问的数据放置在低速内存中, 从而最大化内

存层次结构的利用率.

3.4 并行优化 (parallelization)

并行优化是深度学习编译器中提升模型推理和训练效率的关键技术之一. 随着深度学习模型的规模不断扩大, 单核计算能力已无法满足其计算需求, 因此充分利用硬件的多核、多线程以及加速器资源成为提升性能的重要手段. 并行优化的核心目标是通过各种层级的并行技术, 最大化硬件资源的利用率, 从而提升计算效率.

3.4.1 数据并行 (data parallelism)

数据并行是指将大批量数据集划分后分配到多个处理单元, 每个处理单元都保存模型的副本并执行相同的计算逻辑来处理不同的数据子集, 最后通过全局同步机制汇总结果. 数据并行的优势在于能够显著提升大规模数据集的处理效率.

硬件的 SIMD 指令集 (如 x86 的 AVX[57]、ARM 的 NEON[58]) 通过将标量操作转化为矢量操作, 在单个时钟周期内对多个数据元素执行相同操作, 从而减少循环开销和数据处理时间, 最大化单一硬件的执行效率. SIMD 常用于深度学习中的卷积操作、矩阵乘法、激活函数等批量数据计算场景.

3.4.2 模型并行 (model parallelism)

模型并行是指将模型参数分割, 将同一层的计算 (水平模型并行) 或不同层的计算 (垂直模型并行) 分配到不同设备上执行. 每个设备保存模型参数的子集, 并仅计算其分配的前向和后向传递部分. 模型并行特别适合无法完全放入单个设备内存的超大规模模型 (如 GPT、BERT).

Megatron-LM[59]是由 NVIDIA 开发的一个大语言模型训练框架, 其核心思想是通过高效的模型并行和张量并行技术, 结合通信优化和计算优化, 显著提升大语言模型训练的效率和可扩展性. Megatron-LM 一方面将大语言模型中单个层 (如 Transformer 的注意力机制和前馈网络) 的参数进行切分, 通过水平模型并行减少单个设备的内存占用. 另一方面将模型的多个层分布到不同设备上计算, 通过垂直模型并行减少单个设备的计算负载.

3.4.3 流水线并行 (pipeline optimization)

流水线并行是指将模型的计算图划分为不同的阶段, 这些阶段以流水线的方式在不同设备上执行. 每阶段计算完成后立即将结果发送到下一个设备, 从而减少数据依赖造成的等待时间, 最大化资源的利用率.

DeepSpeed[60]是由微软开发的一个深度学习优化库, 专注于高效训练大规模模型. 其流水线并行优化思想体现在将每个训练批次 (Batch) 进一步划分为多个微批次 (Micro-batch). 通过流水线调度, 使不同设备同时处理不同的微批次, 隐藏通信和计算延迟. Mesh-TensorFlow[61]是 Google 开发的一个分布式深度学习框架, 其将模型的层按顺序切分到多个设备上, 并通过流水线调度隐藏通信和计算延迟, 从而通过流水线调度最大化设备利用率.

3.4.4 混合并行优化 (hybrid parallel optimization)

在实际应用中, 单一的并行优化策略往往难以满足复杂模型的需求, 因此混合并行优化成为提升性能的重要手段. 混合并行优化通过结合各种并行技术, 从而最大化硬件资源的利用率.

Alpa[62]提供了一个统一的并行策略组合框架, 并将并行优化分为两个层次: (1) 算子间并行, 使用图划分算法将计算图分解为多个子图, 从而优化不同算子之间的并行执行 (如流水线并行); (2) 算子内并行, 使用分块、切分等技术优化单个算子的计算 (如数据并行、模型并行). Alpa 的自动化框架显著降低了分布式深度学习的实现难度, 同时提供了高效的性能和资源利用率.

GSPMD[63]是 Google 开发的一种分布式机器学习并行化框架, 其核心优化思想是通过基于注解的自动化张量切分 (如按行切分、按列切分) 和灵活的并行策略组合 (如数据并行、模型并行、流水线并行), 实现高效的分布式训练和推理, 同时提供了良好的可扩展性.

3.5 动态形状优化 (dynamic shape optimization)

在深度学习模型中, 动态形状 (dynamic shape) 是指数据的维度在运行时可能发生变化, 例如批量大小、序列长度等. 传统的编译器通常针对静态形状生成高效代码, 而动态形状的引入使得编译器需要在运行时确定模型的计算、内存分配和调度策略. 动态形状优化通过符号形状表示、运行时图优化、动态张量分块等技术, 实现在运

行时生成特定低级代码的动态编译 (dynamic compilation), 从而在保持灵活性的同时, 提高计算效率.

3.5.1 符号形状表示与分段编译 (symbolic shape representation and piecewise compilation)

符号形状表示是动态形状优化的基础技术之一. 深度学习编译器通过使用符号变量表示动态维度, 从而在编译阶段生成多个优化的执行路径. 这种方法允许编译器在运行时根据实际的输入形状选择最优的执行路径, 从而避免重复编译的开销.

• 符号执行优化: TensorFlow XLA 通过符号执行优化动态形状的操作, 编译器在编译阶段生成多个可能的执行路径, 并在运行时根据输入形状选择最优路径.

• 分段计算图生成: TVM 支持动态形状的分段计算图生成, 编译器将计算图划分为多个静态形状的子图, 并在运行时根据输入形状动态组合这些子图.

3.5.2 运行时图优化 (runtime graph optimization)

在运行时确定输入形状后, 深度学习编译器可以通过动态调整计算图和操作的调度, 进一步提升计算效率. 运行时图优化的核心思想是根据实际的输入形状, 动态优化计算图的执行顺序和内存分配.

• 运行时跟踪优化: TorchDynamo[64]通过运行时跟踪优化动态计算图, 编译器在运行时根据输入形状动态调整计算图的执行顺序, 并生成高效的执行计划.

• 可缓存的优化结果: ONNX Runtime 为动态输入形状生成可缓存的优化结果, 编译器在运行时根据输入形状缓存优化后的计算图, 从而减少重复编译的开销.

3.5.3 动态张量分块与编译 (dynamic tensor tiling and compilation)

针对长序列或大维度的动态形状, 深度学习编译器可以通过动态分块输入数据以降低计算开销. 动态张量分块的核心思想是将大规模张量划分为多个小块, 并在运行时根据硬件资源动态调整分块大小.

• 长序列的动态分块: HuggingFace Accelerate[65]支持长序列的动态分块与并行优化, 编译器在运行时根据输入序列长度动态调整分块大小, 同时并行处理每个分块.

• 动态分块与编译: BladeDISC 通过动态分块技术优化动态形状的计算, 编译器在运行时根据输入形状动态生成分块策略, 并生成高效的执行代码.

3.5.4 缓存动态结果 (caching dynamic result)

对于重复的动态形状计算结果, 深度学习编译器可以通过缓存机制减少重新编译的开销. 缓存动态结果的核心思想是将相同输入形状的计算结果缓存起来, 并在后续计算中直接复用.

• 动态形状缓存: BladeDISC 通过缓存动态形状的计算结果, 减少重复编译的开销. 编译器在运行时根据输入形状缓存优化后的计算图, 并在后续计算中直接复用缓存结果.

• 符号形状缓存: TVM 通过符号形状缓存优化动态形状的计算, 编译器在运行时根据符号形状缓存优化后的计算图, 并在后续计算中直接复用缓存结果.

3.6 自动调优 (auto-tuning)

自动调优是深度学习编译器中提升计算效率的关键技术之一, 旨在通过自动化方式为特定硬件平台生成高效的代码. 针对某个特定的张量算子或算法, 人工优化的方式需要相关的硬件知识和调优经验, 同时耗费较大的人力成本. 而自动调优的优势在于基于给定的算子描述和目标硬件平台, 依据设置的调度原语、规则或模板, 能够自动生成目标硬件平台上和人工调度相近乃至更好的方案, 从而显著减少调优时间和人力成本.

自动调优的基本流程包括调度空间的生成、搜索算法的应用以及成本模型的评估, 如图 10 所示. 其中, 调度空间是自动调优过程中所能够探索的调度所组成的空间, 搜索算法指的是搜索调度空间的方式, 成本模型则用于评估不同调度策略的运行效率. 自动调优的核心思想是通过搜索算法在调度空间中寻找最优的调度策略, 并结合成本模型评估调度的性能.

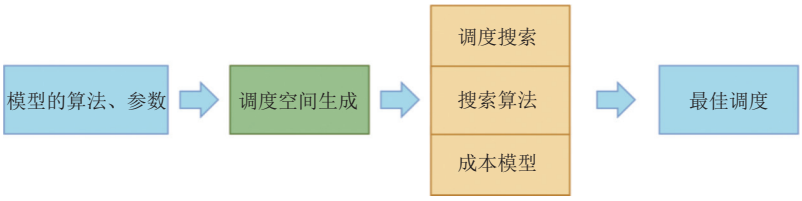


图 10　自动调优的基本流程

3.6.1　调度空间 (schedule space)

调度空间是指自动调优过程中能够探索的所有可能的调度组合. 调度空间生成是自动调优的第 1 步, 决定了搜索算法的探索范围. 调度空间生成方式可以分为以下 3 种, 如图 11 所示.

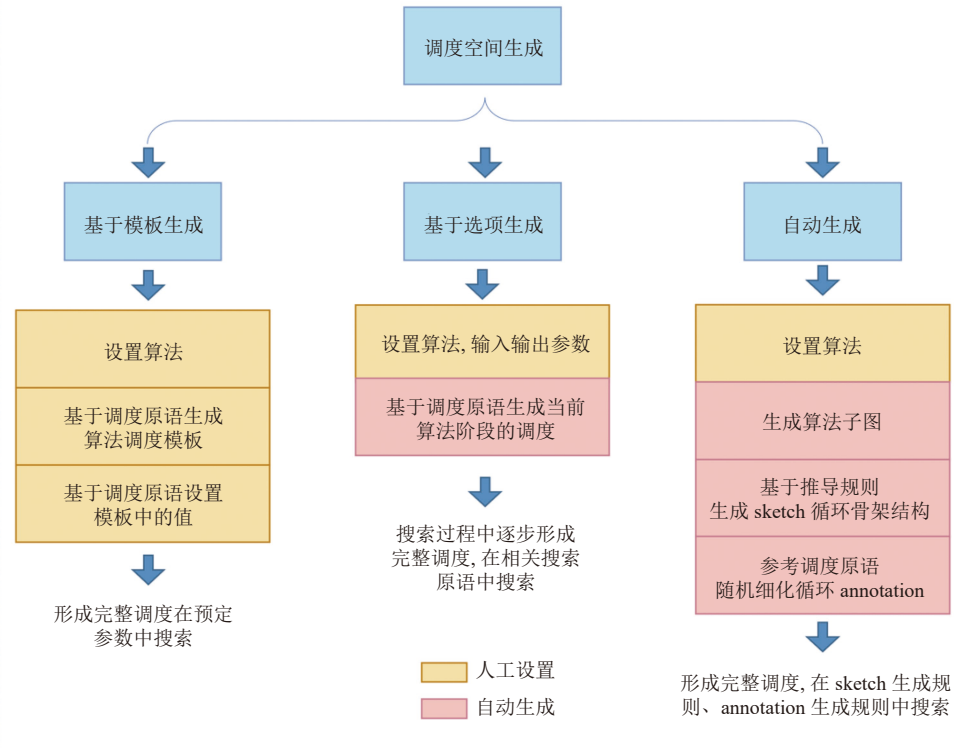


图 11　调度空间的生成方法分类

(1) 基于模板的调度空间生成: AutoTVM[66]、AdaTune[67]和 Chameleon[68]通过为每个算子定义模板, 并将模板中的可调参数作为搜索对象, 生成调度空间. 常见的参数如循环分块大小、循环展开次数等. 然而, 基于模板的调度空间生成依赖于人工经验, 难以探索模板之外的优化空间. 此外, 人工设置模板需要花费大量的人力, 随着算法和硬件设备数量的增加, 这种方式变得愈加低效.

(2) 基于选项的调度空间生成: ProTuner[69]通过迭代扩展节点, 并根据设置的调度选项生成调度空间. 例如, 可以依据生产者-消费者关系和数据传递的局部性原理, 生成循环嵌套的组合选项. 然而, 选项的设置限制了调度空间的范围, 同时基于流水线的迭代构建方式容易造成误差的累积.

(3) 自动生成的调度空间: Ansor[70]将程序的高层次循环骨架抽象为草图, 将低层次的调度选项 (分块、并行、矢量化展开等) 抽象为注释, 通过递归应用推导规则来生成可能的草图, 从而实现调度空间的分层构造. 自动生成

的方式能够探索人工经验之外的优化空间, 同时减少误差的累积.

理想的调度空间能够包括全部的调度组合和硬件相关参数. 但如果调度空间过大, 可能会给搜索带来困难. 为此, 需要对调度空间进行优化.

一种优化方法是在调度空间生成过程中加入一些限制. Halide 通过限制每个函数使用的调度原语的数量, 控制调度空间的规模. FlexTensor[71]通过对调度原语的使用限制、参数的设置限制、特定硬件的调度决策修正来修剪生成的调度空间, 从而避免搜索到一些次优的调度设计.

另一种优化方法是对生成的调度空间进行重排. 一维列表表示的搜索空间维度较小, 难以表示局部性的调度特征, 且相应的搜索步数容易较长, FlexTensor 通过重排形成高维空间, 能够更有效地探索局部区域, 缩短搜索时间.

3.6.2 搜索算法

搜索算法用于在调度空间中寻找最优的调度策略. 常见的搜索算法如下.

(1) 基于生物进化的算法: 遗传算法和进化搜索是模仿自然选择过程的启发式算法. 例如, Halide 利用基于领域特定知识的遗传算法寻找较好的搜索起始点, 并通过交叉、突变、随机和精英主义来生成下一代调度策略. Ansor 在每次迭代中, 根据成本模型的预测选择性能较高的调度策略, 并通过变异和交叉对所选策略进行微调来生成下一代.

(2) 基于物理退火的算法: 模拟退火和上下文模拟退火是在较大搜索空间中逼近全局最优解的启发式算法. 例如, AutoTVM 通过模拟退火算法从搜索空间中的一个随机点出发, 不断生成新的候选调度, 并通过成本模型判断新的候选调度的优劣. AdaTune 通过上下文模拟退火算法修改了贪婪采样的固定概率, 将其替换成一个上下文因子, 从而兼顾调度的质量和多样性.

(3) 基于贪心策略的算法: 贪心算法通过局部最优选择逐步构建全局最优解. 例如, Mullapudi 等人[72]使用贪心算法进行算法流水线转换过程中的搜索, 通过先验知识对分块策略进行指导, 并优先在数据重用率较大的方向上进行划分. Adams 等人[73]通过束搜索策略维护一个候选状态列表, 并根据成本模型选择成本最低的前 k 个状态作为下一轮搜索的候选状态.

(4) 基于强化学习的算法: 强化学习算法通过策略网络和价值网络形成调度策略. 例如, Chameleon 采用了 actor-critic 模式, 通过策略网络提供决策方向, 并通过价值网络评估决策的好坏. ProTuner 通过蒙特卡洛树搜索逐步构建调度策略, 并通过成本模型获取奖励信息. FlexTensor 使用模拟退火算法辅助搜索新的器点, 然后通过 Q-learning 算法预测在搜索空间中移动的最佳方向.

3.6.3 成本模型

成本模型用于评估不同策略或调度的运行效率. 常见的成本模型如下.

(1) 硬件测量: Halide 早期通过对调度算法的编译和运行来评估调度的优劣. FlexTensor 和 AutoConfig[74]在 CPU 和 GPU 上通过测量来获得实际成本. 硬件测量能够得到准确的运行时间, 但对于数量较大的候选调度来说, 硬件测量的效率较低.

(2) 集成学习模型: AutoTVM 从模型中提取入代码循环结构的特征, 并基于特征进行预测. 类似的, Ansor 通过提取算术特征和内存访问特征, 使用 XGBoost[75]进行成本预测. AdaTune 中则采用随机森林模型进行成本预测.

(3) 神经网络模型: AutoTVM 通过可训练的神经网络 TreeGRU 来进行预测. Ragan-Kelley 等人[51]通过人工设计将一个调度进行算法相关和调度相关的特征化, 并通过神经网络输出预测系数.

4 现有的典型系统

随着深度学习技术的快速发展, 深度学习编译器作为连接深度学习框架与硬件平台的关键桥梁, 逐渐成为学术界和工业界的研究热点. 近年来, 多种主流的深度学习编译器得到了广泛关注和应用. 这些编译器通过不同的优化策略, 在计算图优化、算子调度、自动调优等方面取得了显著进展. 本节将介绍几种典型的深度学习编译器系统, 并分析其核心思想、优化技术以及在实际应用中的表现.

4.1 传统深度学习编译优化系统

TVM[76]是一个开源的端到端深度学习编译器框架, 旨在通过自动化的方式优化深度学习模型在不同硬件平台上的运行效率. TVM 的核心思想是通过图级和算子级的中间表示及其转换, 将高层框架中的计算表示转换为适用于特定硬件的高效可执行代码. TVM 还采用了一种新颖的基于学习的成本建模方法来自动优化低级程序以适应不同的硬件特性.

XLA[77]是 Google 为其 TensorFlow 框架设计的编译器, 旨在通过静态计算图优化和专用硬件 (TPU) 支持, 显著加速了 TensorFlow 构建的模型的训练和推理速度. 随着 XLA 的发展, 它也被集成到其他先进的机器学习框架中, 如 PyTorch 和 JAX. 与此同时, XLA 还降低了在特定领域中手动管理编译器的复杂性, 在单个编译器工具链中简化了部署流程.

IREE[78]是 Google 开发的针对机器学习模型的通用编译器框架, 旨在为各种硬件提供高效、跨平台的机器学习模型执行环境. IREE 基于 MLIR[59]框架构建, 能够将不同机器学习框架中的计算图转换为 MLIR 中间表示, 并进一步生成针对特定硬件的优化代码, 从而具备很强的灵活性和可扩展性. IREE 支持多种硬件平台, 包括移动设备、边缘计算和云端推理, 并通过自动化的优化流程降低了手动调优的复杂度.

ROLLER[79]是由微软研究院联合多所高校推出的针对深度学习张量编译的快速高效优化框架, 其核心优化思想是通过层次化优化和基于规则的搜索策略来显著减少编译时间, 同时保持高性能. ROLLER 将张量计算分解为多个层次 (如算子级别、子图级别、图级别), 并在每个层次上结合基于规则的推导来快速确定优化策略, 从而减少全局搜索空间的复杂度.

Hidet[24]是由多伦多大学推出的一个深度学习编译器, 其核心思想是将调度过程嵌入到张量程序表示中, 并通过任务映射原语允许开发者在更细粒度上操纵张量程序, 这种新方法通过允许开发者在更细粒度 (例如, 允许程序语句级) 上定义计算的顺序, 从而丰富可表达的优化. Hidet 还构建了一个高效的以硬件为中心的调度空间, 该空间与程序的输入规模无关, 从而大大减少了调优时间.

TorchInductor[80]是由 Meta 推出的基于 PyTorch 2[81]的深度学习编译器, 是 TorchDynamo 的默认后端编译器, 可以基于 PyTorch 中的模型计算图生成高性能的后端代码. TorchInductor 通过运行时定义的中间表示支持 Python 语言的动态特性, 从而消除了大量模板代码, 能够以简洁的方式编写优化规则. TorchInductor 还集成了 OpenAI 开发的编程框架 Triton[82], 能够在降低算子开发复杂度的同时, 提供不逊于 CUDA 的性能.

4.2 基于大语言模型的编译优化系统

近年来, 随着大语言模型 (LLM) 的突破性发展, 研究者开始探索将编译器与优化系统与 LLM 结合的新范式—即“神经编译 (neural compilation)”或“LLM 驱动的编译/优化 (LLM-based compilation/optimization)”. 这种方法与传统编译器优化有本质不同, 其目标不仅限于一般应用程序, 而是面向大规模张量运算、高维模型推理、异构硬件以及模型量化/部署等复杂场景.

例如, Meta Large Language Model Compiler (LLM Compiler)[83,84]是由 Meta AI 提出的一组预训练 LLM 模型, 专门针对编译器中间表示与汇编的优化任务. 该模型在高达 5460 亿个 LLVM-IR 和汇编代码 tokens 上训练, 并通过指令微调 (instruction fine-tuning) 来提升其对编译器优化流程的理解. 研究表明, 该模型在代码尺寸优化任务中, 可实现“77% 的自动调优潜力”. 此外, 它还展示了从 x86_64/ARM 汇编反汇编回 LLVM-IR 的能力.

除了优化 IR 和汇编, 还出现了将 LLM 用于基于前端语言生成汇编 (或低级代码) 的尝试. 比如, LEGO-compiler[85]提出了一种“分块翻译+可验证工作流+自我修正”的神经编译系统. 它将高层语言程序切分为多个可管理的小块, 然后逐块翻译成汇编, 再通过验证确保正确性. LEGO-Compiler 在多个基准上表现优异, 其编译代码的成功率和规模扩展性显著高于此前的尝试.

也有研究关注将 LLM 用于优化决策: 例如, REASONING COMPILER[86]框架将编译优化视为一个顺序且上下文感知的决策过程, 由 LLM 提出变换建议 (如 tiling、fusion、vectorization 等) 并结合蒙特卡洛树搜索来选择优化序列, 从而在不需要额外训练的情况下, 显著提升优化效率和样本利用率.

另一个具有代表性的工作是 VecTrans[87], 它利用 LLM 来重构那些传统编译器难以识别为可矢量化的代码区域, 使其更加适合传统编译器的处理. 通过这种方法, VecTrans 成功将一些在 Clang/GCC/其他编译器下无法自动向量化的代码转化为可向量化形式, 并在多个基准上实现平均约 2.02 倍的性能加速.

还有一些更广泛面向编译器自动调优和优化 pass 选择的新兴工作. Compiler-R1[88]提出了一种强化学习 (RL) 驱动的 LLM 编译器优化框架, 通过构建高质量推理数据集+两阶段端到端训练 (supervised fine-tuning+RL), 使 LLM 能够自主探索并选择优化 pass 序列以最小化中间表示指令数. 实验证明, 在多个基准上, Compiler-R1 相较于传统的 opt-Oz 优化, 平均可减少约 8.46% 的 IR 指令数.

更进一步, AwareCompiler[89]提出了一种“知识-数据驱动 (knowledge-data driven)”的智能体编译优化框架. AwareCompiler 通过结构化的编译器领域知识+程序上下文数据+混合训练的设计, 使得 LLM 在考虑程序语义与编译环境的基础上, 能够生成上下文感知的优化 pass 序列. 该方法不仅在优化效率和性能上明显优于基线, 而且在正确性和优化可行性上也表现优越. 实验结果显示 AwareCompiler 在标准基准上取得了比传统方法及现有基于 LLM 的优化方法更好的效果与效率.

这些工作共同证明: 基于大语言模型的编译优化不仅是对传统编译器技术的补充, 更代表了一种面向未来系统/模型/硬件的新路径. 与传统方法相比, 它具有如下优势: (1) 更适用于张量计算/深度学习/模型推理等高维、高复杂度场景; (2) 具备灵活性和可适应性: 通过“学习+推理+自我修正”机制, 可以处理传统编译器难以覆盖的新语言结构、新 IR、新硬件架构; (3) 支持跨语言/跨架构的编译流程, 降低人工维护编译器后端的成本.

当然, 基于 LLM 的编译/优化也面临挑战, 如正确性、可验证性、基准与评估标准不足、对于复杂/大规模项目的扩展性以及与异构硬件的适配等问题. 也有工作 [90]对这些问题进行尝试, 比如针对从 IR 到汇编的神经编译评测基准以及通过“self-evolving prompting+iterative self-debugging”提高正确率和性能.

因此, 在面向现代大语言模型的系统设计、模型部署、性能优化 /硬件适配等研究中, 将传统编译/优化方法与大语言模型感知编译优化相结合, 将会是一个极具前景的方向.

5 未来的研究方向

经过多年的发展, 深度学习编译器及其优化技术已经在计算图优化、算子自动调度、自动代码生成、异构硬件适配等方面取得显著进展, 有效提升了模型训练和推理的效率, 极大地提高了生产力. 然而, 随着模型规模的指数级增长、硬件架构的多样化以及场景需求的复杂化, 现有编译优化策略在可组合性、可扩展性和可移植性等方面仍存在明显不足. 未来深度学习编译技术的发展, 需要更系统化的方法、更统一的软件栈设计以及更高层次的自动化能力. 下面结合当前技术现状及最新研究成果, 对可能的研究方向进行展望.

5.1 深度学习编译优化技术的选择、组合与集成

深度学习编译优化技术的快速发展为模型的高效执行提供了丰富的手段. 然而, 这些优化技术之间往往存在复杂的依赖和制约关系, 同时不同硬件架构的特性也使得优化策略的选择和权衡变得更加复杂. 此外, 由于不同编译器通常采用不同的 IR, 导致优化技术的通用性和可移植性受到限制. 因此, 如何高效地选择、组合和集成优化技术成为深度学习编译领域亟待解决的关键问题.

5.1.1 优化技术的选择与组合

深度学习编译优化技术的选择和组合是一个复杂的多目标优化问题. 由于不同优化技术之间可能相互制约, 甚至在某些情况下产生负面影响, 因此需要设计高效的搜索机制来找到最优的优化组合. 具体而言, 优化技术的选择和组合面临以下挑战.

(1) 深度学习算子优化涉及算子融合、循环变换、内存布局、向量化等多个维度, 它们之间存在高度耦合. 例如, 对卷积算子进行融合后, 其循环结构和缓存行为都将发生改变, 导致调度空间急剧增大. 如 Ansor 自动调优框架所指出的, 深度学习调度组合的搜索空间规模巨大, 即使采用基于机器学习的搜索也可能需要成百上千次的运行采样. 这导致自动化优化在大语言模型或嵌入式场景中不够高效.

(2) 搜索空间呈指数级增长, 且优化策略跨层影响难建模: 如 Halide 和 TVM 所展示的那样, 调度与计算的解耦能够增强优化自由度, 但目前多数编译器仅在局部进行启发式优化, 缺乏跨层级的联合建模能力.

(3) 硬件多样性导致策略难以复用: GPU、TPU、NPU 存在完全不同的计算模型 (SIMT vs. SIMD vs. systolic array). 某些优化策略在 GPU 上能提升性能, 但在 SIMD CPU 上反而造成寄存器溢出和访存冲突. 例如针对 NVIDIA GPU 的 Warp-level schedule 在 Ascend NPU 上不可用, 而 RISC-V Vector 的 VL 可变特性又要求动态调度机制. 因此, 优化策略的可移植性仍然较弱.

为了应对这些挑战, 研究者可以探索以下方向.

(1) 构建结构化搜索空间: 为减少搜索空间复杂度, 可从考虑如下类型的高效搜索空间.

• 基于算子结构的分层搜索: 将算子按语义 (如卷积、GEMM、element-wise) 划分不同搜索空间, TVM MetaSchedule 的分层搜索策略已初步验证其可行性.

• 基于图拓扑的可分解搜索: AStitch 提出利用依赖图的多级拼接, 可有效减少融合搜索复杂度.

• 启发式引导的约束搜索: 融入缓存大小、利用率上界等约束, 可显著减少非法搜索候选.

(2) 跨层协同优化框架: 未来编译器需要打通 Graph IR、Tensor IR、Loop IR, 使优化能够跨层级共同决定. 如基于统一代价模型的跨层优化, 引入多目标 (延迟、能耗、内存) 联合优化, 或者采用可微编译方式, 使优化策略可通过梯度搜索.

(3) 发展语义增强优化器: 未来编译器可利用算子语义、张量形状、数据布局等高层信息进行更可靠的优化推理. 例如, 基于 shape 等式推理判断融合是否会导致广播开销恶化; 基于算子代数特性 (如可交换、可分解性) 推断合法的优化路径; 利用类型系统描述特殊张量属性 (如 block-sparsity、KV-cache).

(4) 发展可泛化的硬件感知优化: 未来的优化器需要更好地适配异构硬件. 例如 GPU 的 warp 同步限制, CPU 的 cache locality, NPU 的 systolic array 映射等. 研究者可将上述硬件特性抽象为统一的代价模型模板, 使优化策略具备更高的可移植性.

5.1.2 基于不同 IR 的优化技术的融合与集成

深度学习编译器通常基于不同的 IR 实现, 例如 TVM 的 TensorIR[91]、 TensorFlow 的 MLIR-HLO[92]、Halide 的 HalideIR[93]等. 这些 IR 的设计目标和表达能力各不相同, 这种多样性虽然提升了灵活性, 但也导致优化技术在不同编译器之间难以复用和集成.

为此, Google 提出了 MLIR 作为一种通用的编译器基础设施, 旨在解决 IR 之间的互操作性问题. MLIR 的核心优势在于其架构的可扩展性和灵活性. MLIR 允许开发者定义自己的 IR, 并通过统一的框架实现不同 IR 之间的转换和集成. 这种设计使得多种优化技术可以在 MLIR 的框架下共存, 从而提高了优化技术的通用性和可移植性. 目前已有部分深度学习编译器基于 MLIR 开发自己的 IR 和优化框架. 例如, IREE 专注于高效推理, Buddy Compiler[26]则致力于为新兴硬件架构提供支持. 这些实践表明, MLIR 在促进优化技术融合和集成方面具有巨大潜力.

MLIR 为解决 IR 碎片化问题提供了有力工具, 为了在优化技术的选择、组合与集成之间取得平衡, 可以设计一种基于 MLIR 的综合优化框架, 结合自动化技术选择和集成. 具体而言, 该框架可以包括以下研究方向.

(1) 构建统一的张量语义 IR, 并支持完整的 shape algebra、layout algebra、broadcast semantics、dynamic shape constraints 等. StableHLO 与 MHLO 已开始尝试标准化, 但仍缺乏跨框架共识.

(2) 基于程序验证理论发展可验证的 IR 转换系统, 如 SMT solver、IR equivalence checker、rewrite rule formalization 等. 确保跨 IR 转换的正确性, 避免错误传播到内核代码生成阶段.

(3) 自动 Dialect 融合框架, MLIR 的优势在于其可扩展性, 但仍缺乏自动化工具用于推断 Dialect 之间可共享的优化规则、自动生成转换模式、自动检测冗余 Dialect 等. 未来可基于类型系统与模式推理构建“Dialect Auto-Fusion Engine”.

(4) 构建动态形状友好的 IR, BladeDISC 提出的动态形状 IR 展示了方向性突破, 但仍有优化空间, 如 dynamic symbolic dimension propagation、hybrid static/dynamic lowering、runtime shape specialization cache 等. 未来 IR 需

要原生支持动态形状, 以满足 LLM、扩散模型等需求.

5.2 面向新型负载的深度学习编译系统设计

深度学习系统的演进很大程度上是由深度学习负载所驱动. 从早期的卷积神经网络, 到近年来以 GPT[94]、DeepSeek[95]、LLaMA[96]为代表的大语言模型, 这些新型负载一方面极大拓展了人工智能的应用边界, 另一方面也对现有深度学习编译系统在计算图表达、内存管理、分布式并行和动态形状支持等方面提出了前所未有的要求. 现有编译技术在传统中小规模模型上已相对成熟, 但大语言模型这类长序列、超大参数量、强依赖分布式训练与推理的场景下仍存在明显短板, 需要在系统结构与编译优化策略上进行重新审视和设计.

5.2.1 大语言模型对深度学习编译系统的挑战

大语言模型的典型特征包括: 参数规模跨越数十亿到数千亿量级、训练和推理过程中存在大量稠密矩阵运算和通信操作、计算图具有显著的动态性 (如可变序列长度、KV Cache 增长、条件分支等). 这些特征在编译层面主要带来以下挑战, 同时也揭示了未来若干重要研究方向.

(1) 计算与通信耦合增强, 传统单机优化难以满足需求

Megatron-LM、DeepSpeed、Mesh-TensorFlow 等系统表明, 大语言模型的高效训练依赖于数据并行、张量并行、流水线并行甚至专家并行等多种并行策略的组合, 跨设备通信 (如 AllReduce、AllGather) 的开销在总时间中占比显著. 当前主流编译器更多关注单算子或单设备上的计算优化, 对“计算-通信”一体化建模和优化支持不足.

未来研究方向之一是构建计算-通信协同优化的编译框架: 在图级 IR 中显式建模通信算子, 将算子融合、计算排布与通信重叠纳入统一的代价模型, 自动决定在不同并行策略下的切分方式、通信拓扑和调度次序, 从而实现端到端的延迟和带宽优化.

(2) 动态形状和运行时图结构变化削弱静态编译效果

在大语言模型负载中, 输入序列长度、批大小、专家激活数等经常在运行时变化, 这导致静态形状假设失效. TorchDynamo 通过运行时跟踪捕获 Python 程序的动态图结构, BladeDISC 针对动态形状提出了分段编译与缓存机制, 但在对应场景中仍存在两个不足: 一是动态形状条件分支复杂度高, 导致图优化开销增加; 二是针对长序列、KV Cache 的动态分块策略尚不够精细.

未来可重点探索动态形状优先的 IR 与编译方案: 在 IR 层面引入更细粒度的符号形状约束, 支持对序列长度、窗口大小等参数的代数推理; 对于常见大语言模型结构 (如 Multi-Head Attention、RMSNorm、KV Cache 更新) 构建专门的动态形状模板, 实现“模式化动态编译”; 利用运行时信息自适应调整分块大小和缓存策略, 形成静态-动态混合优化路径.

(3) 内存与算力不匹配导致训练/推理受限, 需要更强的内存感知编译

ZeRO[97]、ZeRO-Infinity[98]针对大语言模型训练中的内存瓶颈提出了参数分片、通信重叠和异步 offloading 等技术, 但这些策略大多在框架或运行时层面实现, 与编译器的算子级、图级优化尚未充分结合. 与此同时, 推理场景下的 KV Cache 管理、激活重计算策略也与编译器生成的调度高度相关.

未来研究可以在以下方向深入: 在编译器 IR 中引入“内存级别”与“驻留时长”的显式标注, 使得算子调度与内存分配可以协同优化; 面向 KV Cache、MoE 路由等特定数据结构设计编译期可见的缓存策略, 通过生命周期分析与复用规划降低峰值显存; 与 ZeRO 系列方法协同, 将参数分片和算子划分统一到编译器的图重写与调度框架之中, 实现对显存-带宽-计算的多目标优化.

(4) 系统规模扩大后, 编译开销本身成为瓶颈

在超大模型和大规模集群场景下, 单次编译可能涉及数万级算子和复杂的分布式并行划分, XLA、TVM 等编译器的优化时间和代码生成时间不容忽视.

因此, 未来还需要研究可增量与可重用的编译技术: 对常见子图 (如 Transformer block) 进行模板化编译和版本缓存; 利用哈希、语义特征等机制识别图中重复结构, 避免重复优化; 探索低成本的近似优化策略, 在保证足够性能的前提下显著缩短编译时间.

5.2.2 大语言模型在深度学习编译系统中的应用

除了作为编译优化对象, 大语言模型本身也开始被用作“编译助手”或“自适应优化器”, 即将大语言模型引入编译流程, 帮助完成代码生成、调度搜索、性能预测等任务. 这一方向为深度学习编译带来了新的可能性, 但也需要更加系统化的设计. 具体而言, 大语言模型可以在以下几个方面发挥作用.

(1) 大语言模型驱动的自动代码生成与内核专用化

已有工作尝试利用大模型生成 CUDA/Triton 内核或调度模板, 使其根据高层计算描述和硬件特性自动生成高效代码. 这一思路与 TVM、Halide 等框架的“调度脚本”理念相契合, 但现有尝试往往仍停留在“人机协同”的交互式阶段, 缺乏可验证性和鲁棒性.

未来可在 3 个方面深入: 一是建立编译领域专用的代码语料库与指令微调策略, 使大语言模型对 IR、调度原语、硬件约束有更准确的内在表示; 二是将大语言模型的代码生成结果纳入形式化验证或测试框架中 (例如类似 TASO[30]对等价变换的验证思想), 防止错误优化; 将大语言模型生成的调度方案与现有自动调优系统 (如 Ansor、MetaSchedule) 结合, 使其作为搜索初始点或搜索策略的高层“先验”.

(2) 大语言模型辅助的代价模型与性能预测

现有自动调优多依赖于基于特征工程或轻量模型的代价预测[75,76], 这在复杂硬件环境下可能难以捕捉所有影响因素. 大语言模型在多模态、跨域知识融合方面的优势, 使其有潜力作为“高层性能顾问”, 例如基于算子类型、形状、布局和目标硬件描述预测调度好坏.

未来可以探索: 将 IR 片段、硬件参数等编码为序列输入大语言模型, 通过其输出的性能排序或评分指导搜索; 结合历史调优数据, 将大语言模型作为可迁移的“元代价模型”, 对新硬件、新算子提供零样本或少样本性能预测; 将大语言模型的预测与传统基于统计学习的代价模型融合, 构成层次化成本评估体系.

(3) 大语言模型参与的自适应编译与在线优化

在在线推理或长时间训练任务中, 硬件负载、输入分布可能发生变化, 静态编译策略无法长期保持最优

未来可研究大语言模型参与的在线编译反馈闭环: 利用运行时监控数据 (如计算延迟、显存占用) 构造自然语言或结构化提示, 指导大语言模型自动提出“重新编译”建议; 将大语言模型产出的优化方案限制在一组形如“启用/禁用某种融合、调整分块大小范围”的高层决策空间, 由底层编译器负责安全落地; 在云平台场景中, 通过大语言模型实现跨任务、跨模型的经验迁移, 使编译策略能够持续演进.

5.2.3 面向大语言模型的深度学习编译器设计

大语言模型的普及使得“通用编译器+若干特定优化”的模式逐渐显露不足. 越来越多的系统实践表明, 需要从体系结构到编译流程进行“面向 LLM 的重新设计”, 构建专门面向大模型训练和推理的编译系统.

(1) 分布式并行优先的编译架构

Megatron-LM、DeepSpeed、Alpa、GSPMD 等工作展示了多种并行方式与自动划分算法, 但现有编译器通常将“并行划分”视作优化流程外部的配置项, 而非 IR 和优化的一等公民.

未来可构建以并行策略为核心抽象的编译框架: 在 IR 中显式表达张量分布、通信模式和并行层级, 将数据并行、张量并行、流水线并行、专家并行统一到同一个抽象层; 在编译流程中引入“并行计划生成”阶段, 通过图划分和代价模型自动生成并行方案, 并驱动后续算子划分、通信插入和调度生成; 让并行抽象与硬件拓扑描述紧密耦合, 实现跨集群、跨机架的性能优化.

(2) KV Cache 与长序列优化友好的编译机制

在 LLM 推理场景中, KV Cache 的管理已经成为关键优化点之一. 现有系统多在框架或手写内核层面处理缓存分块、分页管理和重用, 而编译器对这一过程的掌控力有限.

面向未来, 可以设计 KV Cache 感知的编译器组件: 在 IR 中显式表示 KV Cache 的生命周期、访问模式和存储层级, 将其与常规张量区分对待; 对长序列解码过程引入可配置的分块策略和截断策略, 使编译器能够生成在不同 batch size、不同并发度下表现稳定的执行计划; 探索基于 profile 信息自动调整缓存布局、压缩方式、预取策略等.

(3) 训练-推理一体化的编译优化

Orca[99]、ZeRO、ZeRO-Infinity 等工作主要聚焦训练端的内存与通信优化, 而推理端则更多依赖算子库与工程手段. 随着 LLM 服务持续在线、模型频繁更新, 训练–推理一体化的编译架构将具有重要意义.

研究人员可以考虑如下方向: 在训练阶段收集梯度分布、激活稀疏性等统计信息, 为推理阶段的剪枝、量化和算子重写提供依据; 通过联合设计训练时的张量布局和推理时的执行计划, 降低模型部署和热更新的成本; 在统一的编译 IR 上同时支持“训练视图”和“推理视图”, 实现跨场景共享优化成果.

(4) 可扩展的安全与可靠性保证机制

随着 LLM 编译优化的复杂度上升, 错误优化带来的后果更加严重, 可能引入数值不稳定、精度下降甚至安全漏洞.

未来的大模型编译器需要在以下方面增强可靠性: 对关键算子 (如归一化、Softmax) 的重写采用形式化等价验证或数值边界验证; 为动态形状与分布式并行相关的变换引入断言与运行时检查, 避免错误传播; 研究适用于 LLM 场景的“回滚与降级机制”, 在优化失败或环境变化时能够安全回退到保守策略.

5.3 面向新型硬件的深度学习编译优化

RISC-V[100]作为一种基于精简指令集的开放指令集架构, 凭借其简洁、灵活和可扩展的设计, 逐渐成为嵌入式系统中的主流选择. 考虑到 RISC-V 有望与 x86、ARM 三分全球处理器市场, 其开放性、通用性和可扩展性为深度学习处理器的设计提供了更多可能性. 因此, 研究面向 RISC-V 的深度学习编译优化技术具有重要的学术价值和实际意义.

5.3.1 RISC-V 架构的特点与优势: 开放性与可扩展性驱动的新编译生态

RISC-V 作为近年快速发展的开放指令集架构, 凭借模块化扩展机制、可自定义指令能力以及适用于低功耗和高性能场景的可调节设计, 逐渐成为学术界与工业界关注的重点. 与 x86/ARM 不同, RISC-V 允许对基础 ISA 进行可裁剪和可生长式扩展 (如 V-extension、B-extension、F-extension), 为未来构建深度学习专用加速器提供了天然优势.

RISC-V 在深度学习场景中具有以下潜在优势.

(1) 可定制性强, 适合构建领域专用硬件: 开发者可根据模型特征增添向量、矩阵运算扩展指令, 使硬件特性与编译器调度机制深度耦合. 已有原型如“承影”[101]即展示了基于 RISC-V 的定制加速能力.

(2) 生态开放, 便于软硬件协同设计: 相比 ARM 授权机制的限制, RISC-V 的开放性降低了新型编译器-加速器协作的门槛, 更利于建立面向深度学习的开源体系, 如 OpenXLA、TVM RISC-V 后端等.

(3) 对边缘推理友好: RISC-V 基于精简指令与可扩展流水线, 有利于构建超低功耗的边缘推理芯片, 适配 IoT、机器人、车辆计算等场景.

尽管硬件仍处于快速演进阶段, 这些特性已为构建新型编译栈奠定良好基础, 是未来研究的重要着力点.

5.3.2 面向 RISC-V 的深度学习编译优化挑战: 模型差异、硬件限制与生态不成熟

尽管 RISC-V 在嵌入式系统中展现出巨大潜力, 但面向 RISC-V 的深度学习编译优化仍面临诸多挑战.

(1) 执行模型差异导致现有调度策略不可直接迁移

主流 GPU 基于 SIMT 模型, 调度逻辑、warp 管理、memory coalescing 等结构决定了现有 CUDA/Triton 内核的许多优化策略 (如 thread tiling、warp shuffle). 然而 RISC-V Vector (RVV) 基于 SIMD 模型+动态向量长度特性, 即向量长度不固定而由硬件决定, 这使得现有编译器在调度模式上必须重新设计.

(2) 硬件生态不成熟, 缺少真实芯片导致编译优化难以验证

目前许多 RISC-V 加速器仍处于 FPGA 或模拟器阶段, 例如“承影”[86] GPU 尚未流片. 这导致模拟器的性能行为难以真实反映物理芯片特性且编译器难以构建可靠的性能模型与代价模型, 影响自动调优效果. 因此, 相比 GPU 的成熟生态 (cuBLAS、cuDNN、TensorRT), RISC-V 的软硬件闭环仍较弱.

(3) 编译工具链不完善, 缺少统一标准化 IR

LLVM 对 RISC-V 的支持快速增长, 但尚未完全优化针对 RVV 的自动向量化. 与此同时, 不同加速器厂商扩

展了不同的自定义 ISA, 缺乏可移植抽象, 这使得上层框架难以统一管理算子生成和调度.

(4) ISA 快速演进导致优化策略脆弱化

随着 RISC-V 的 V-extension、P-extension 等不断更新, 编译器的调度规则易随规范变化而失效, 需要更灵活、更可验证的自动化生成机制.

5.3.3 面向 RISC-V 的深度学习编译优化技术: 突破方向与系统性设计

为了克服上述挑战, 研究者可以从算法级、编译级、硬件级和软硬件协同这 4 个层面推动 RISC-V 深度学习编译技术发展.

(1) 算法级优化: 为 RISC-V 架构特性重新设计模型计算方式

• 轻量化模型结构设计: RISC-V 边缘芯片特别适合部署 MobileNet、ShuffleNet、TinyBERT 等轻量模型, 使其运行更高效.

• 量化/剪枝友好的数据布局: RISC-V Vector 在处理子字节 (4-bit、8-bit) 量化时, 需要特殊的对齐与分块策略, 可在模型训练端注入布局感知优化.

• 面向 RVV 结构的注意力机制改造: 例如使用块稀疏注意力减少不规则访存, 提高 SIMD 利用率.

(2) 编译级优化: 为 RISC-V 构建新的调度模式与向量化框架

• 向量长度无关调度生成: RISC-V Vector 的向量长度在不同硬件上可能不同, 因此编译器要生成可适配不同向量长度的代码, 例如: 基于掩码的循环拆分、自动向量读写等.

• 针对自定义 ISA 的自动映射技术: 借鉴 Halide、TVM TensorIR 的算子级重写机制, 自动构建从 IR 到 ISA 的映射模板, 使编译器无需手写大量向量指令序列.

• 向量级内存层次优化: 为适应 RISC-V 芯片常见的浅缓存结构, 需增强以下优化能力, 如软件预取、缓存感知分块、向量寄存器重用模型.

• RISC-V 专用算子融合机制等: 不同于 GPU 的 warp-level fusion, RISC-V 可以探索基于寄存器组的融合, 避免频繁访存.

(3) 硬件特定优化: 结合硬件流水线与加速模块自动优化

• 基于硬件拓扑的并行策略生成: 对多核 RISC-V 芯片, 需要自动生成任务级并行计划.

• 针对 NPU 模块的算子映射: 许多 RISC-V SoC 会整合 DSP/NPU 模块, 需自动进行算子拆分、量化转换以及数据搬移优化.

• V-extension + systolic array 的混合调度: 随着越来越多 RISC-V 加速器采用 systolic 结构, 需要在编译器处理张量重排、分块映射、多阶段流水线调度等.

(4) 软硬件协同设计: 结合 RISC-V 的开放性, 未来编译器可不再被动适配硬件, 而是参与硬件设计阶段, 形成完整闭环.

• 自动生成 ISA 扩展模板: 通过分析算子模式自动建议硬件指令扩展, 提高硬件利用率.

• 协同构建代价模型与性能预测器: 将模拟器性能数据反馈给编译器, 使自动调优更加高效.

• 硬件可解释性与可验证性增强: 使用形式化方法确保自定义指令与高层 IR 之间的语义一致性, 提高系统可靠性.

5.4 面向云、端、侧场景的深度学习编译优化

随着云计算、边缘计算及端侧智能设备的普及, 深度学习部署场景呈现“云-边-端”多层架构, 而底层硬件也从传统的 CPU/GPU 逐渐扩展到 NPU、FPGA、RISC-V 等多种异构平台. 硬件的多样化推动了深度学习编译技术持续演进, 但也带来一系列挑战: ISA 多元化、硬件能力差异大、生态成熟度参差不齐、动态负载多样性强等. 如何构建具有高可移植性、高性能、低功耗和高可靠性的新型编译优化框架, 是未来深度学习系统研究的重要方向.

(1) 云端场景: 分布式计算与通信优化主导性能

在云端场景中, 计算资源丰富, 但通信和存储开销较大. 因此, 编译优化技术需要重点关注分布式计算和通信

优化. 例如, 基于编译器的通信算子重排以减少全局同步; 利用流水线重叠通信与计算; 通过图级切分实现更高效的张量并行.

(2) 边缘场景: 低功耗与动态负载优化

边缘设备通常具有有限的计算资源和严格的功耗限制. 因此, 编译优化技术需要关注计算效率和功耗优化. 例如, 低精度量化的自动化 IR 重写; 运行时感知的性能调优, 使模型能够自适应频率变化、温度变化等设备状态; 内存复用、流水级调度等轻量优化策略.

(3) 终端设备: 极低延迟与稳态实时性要求

终端设备 (如智能手机、物联网设备) 通常具有极低的功耗预算和严格的内存限制. 因此, 编译优化技术需要关注内存优化和实时性. 例如, 更加激进的算子融合与内核生成; 节点粒度的内存分配与激活压缩; 稳态实时性调度, 例如结合操作系统调度器优化计算图执行.

6 总 结

随着深度学习模型和深度学习硬件的迅速迭代, 愈加凸显了深度学习编译优化技术的重要性. 现阶段大部分的深度学习编译优化技术可以分为面向深度学习模型的图层优化以及面向深度学习硬件的算子层优化, 并构成了端到端的深度学习编译优化体系. 各种深度学习编译技术不仅在模型和硬件之间搭建了桥梁, 同时使模型在性能、资源利用率、可拓展性、易用性等方面有了显著提升, 对实际的生产生活产生了积极影响. 然而, 目前深度学习编译领域的发展仍然存在很多挑战, 首先是确定现有编译优化技术的最佳组合和顺序的问题; 其次是各种不同的深度学习编译器和编译优化技术之间的融合和集成的问题; 最后是新模型和新硬件的出现所带来的适配性和可拓展性的问题, 这些挑战也为该领域的进一步探索和研究提供了机遇. 未来的研究方向可以包括: (1) 深度学习编译技术的选择和组合, 设计高效的搜索机制以找到最优的优化组合; (2) 深度学习编译技术的借鉴和集成, 推动中间表示的标准化; (3) 面向新型负载的深度学习编译系统设计, 特别是针对大语言模型的优化; (4) 面向新型硬件的深度学习编译优化, 如 RISC-V 生态下的各类计算架构的编译优化技术.

References

[1] 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: Curran Associates Inc., 2017. 6000–6010. [doi: 10.5555/3295222.3295349]

[2] He KM, Zhang XY, Ren SQ, Sun J. Deep residual learning for image recognition. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). Las Vegas: IEEE, 2016. 770–778. [doi: 10.1109/CVPR.2016.90]

[3] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional Transformers for language understanding. In: Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: ACL, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

[4] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A, Chen YT, Lillicrap T, Hui F, Sifre L, van den driessche G, Graepel T, Hassabis D. Mastering the game of Go without human knowledge. Nature, 2017, 550(7676): 354–359. [doi: 10.1038/nature24270]

[5] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin ZM, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai JJ, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 721.

[6] Abadi M, Barham P, Chen JM, Chen ZF, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng XQ. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). Savannah: USENIX Association, 2016. 265–283.

[7] Frostig R, Johnson MJ, Leary C. Compiling machine learning programs via high-level tracing. In: Proc. of the 2018 Machine Learning Systems (MLSys) Conf. Stanford, 2018.

[8] Ma YJ, Yu DH, Wu T, Wang HF. PaddlePaddle: An open-source deep learning platform from industrial practice. Frontiers of Data & Computing, 2019, 1(1): 105–115 (in Chinese with English abstract). [doi: 10.11871/jfdc.issn.2096-742X.2019.01.011]

[9] ONNX. Open standard for machine learning interoperability. 2024. https://github.com/onnx/onnx

[10] TensorRT. NVIDIA developer. 2024. https://developer.nvidia.com/tensorrt

[11] OpenVINO™ Toolkit. Open visualinference and neural network optimization. 2018. https://docs.openvino.ai/

[12] 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 Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 579–594.

[13] Dao T, Fu DY, Ermon S, Rudra A, Ré C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022. 1189. [doi: 10.5555/3600270.3601459]

[14] Kwon W, Li ZH, Zhuang SY, Sheng Y, Zheng LM, Yu CH, Gonzalez J, Zhang H, Stoica I. Efficient memory management for large language model serving with PagedAttention. In: Proc. of the 29th Symp. on Operating Systems Principles. New York: ACM, 2023. 611–626. [doi: 10.1145/3600006.3613165]

[15] Jouppi N, Kurian G, Li S, Ma P, Nagarajan R, Nai LF, Patil N, Subramanian S, Swing A, Towles B, Young C, Zhou X, Zhou ZW, Patterson DA. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In: Proc. of the 50th Annual Int'l Symp. on Computer Architecture. New York: ACM, 2023. 82. [doi: 10.1145/3579371.3589350]

[16] Amazon Web Services. AWS inferentia. 2024. https://aws.amazon.com/cn/machine-learning/inferentia/

[17] cuDNN: CUDA Deep Neural Network library. NVIDIA developer. 2024. https://developer.nvidia.com/cudnn

[18] cuBLAS. NVIDIA developer: Basic linear algebra on NVIDIA GPUs. 2024. https://developer.nvidia.com/cublas

[19] Eigen: A C++ template libarary for linear algebra. 2010. http://eigen.tuxfamily.org

[20] Intel. Intel® oneAPI math kernel library (oneMKL). 2024. https://www.intel.com/content/www/cn/zh/developer/tools/oneapi/onemkl.html

[21] OpenBLAS. An optimized BLAS library. 2024. http://www.openmathlib.org/OpenBLAS/

[22] Amit Sabne. XLA: Compiling machine learning for peak performance. 2020. https://research.google/pubs/xla-compiling-machine-learning-for-peak-performance/

[23] PyTorch. TorchInductor GPU profiling. 2024. https://pytorch.org/docs/2.1/torch.compiler_inductor_profiling.html

[24] Ding YY, Yu CH, Zheng BJ, Liu YZ, Wang YD, Pekhimenko G. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 370–384. [doi: 10.1145/3575693.3575702]

[25] 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]

[26] 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]

[27] Xing Y, Weng J, Wang YS, Sui LZ, Shan Y, Wang Y. An in-depth comparison of compilers for deep neural networks on hardware. In: Proc. of the 2019 IEEE Int'l Conf. on Embedded Software and Systems (ICESS). Las Vegas: IEEE, 2019. 1–8. [doi: 10.1109/ICESS.2019.8782480]

[28] Boehm M, Reinwald B, Hutchison D, Sen P, Evfimievski AV, Pansare N. On optimizing operator fusion plans for large-scale machine learning in systemML. Proc. of the VLDB Endowment, 2018, 11(12): 1755–1768. [doi: 10.14778/3229863.3229865]

[29] Ashari A, Tatikonda S, Boehm M, Reinwald B, Campbell K, Keenleyside J, Sadayappan P. On optimizing machine learning workloads via kernel fusion. ACM SIGPLAN Notices, 2015, 50(8): 173–182. [doi: 10.1145/2858788.2688521]

[30] Moreira O, Popp M, Schulz C. Graph partitioning with acyclicity constraints. In: Proc. of the 16th Int'l Symp. on Experimental Algorithms (SEA). Dagstuhl, 2017. 30: 1–30: 15.

[31] Xing Y, Liang S, Sui L, Jia XJ, Qiu JT, Liu X, Wang YS, Shan Y, Wang Y. DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2020, 39(10): 2668–2681. [doi: 10.1109/TCAD.2019.2930577]

[32] Iandola FN, Han S, Moskewicz MW, Ashraf K, Dally WJ, Keutzer K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. arXiv:1602.07360, 2016.

[33] Zheng SX, Zhang XJ, Ou DL, Tang SB, Liu LB, Wei SJ, Yin SY. Efficient scheduling of irregular network structures on CNN accelerators. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2020, 39(11): 3408–3419. [doi: 10.1109/TCAD.2020.3012215]

[34] Jia ZH, Thomas J, Warszawski T, Gao MY, Zaharia M, Aiken A. Optimizing DNN computation with relaxed graph substitutions. In: Proc. of the 2nd Conf. on Machine Learning and Systems. Stanford, 2019. 27–39.

[35] Jia ZH, Padon O, Thomas J, Warszawski T, Zaharia M, Aiken A. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 47–62. [doi: 10.1145/3341301.3359630]

[36] Niu W, Guan JX, Wang YZ, Agrawal G, Ren B. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 883–898. [doi: 10.1145/3453483.3454083]

[37] Zheng SZ, Chen RZ, Jin YC, Wei AJ, Wu BY, Li XH, Yan SG, Liang Y. NeoFlow: A flexible framework for enabling efficient compilation for high performance DNN training. IEEE Trans. on Parallel and Distributed Systems, 2022, 33(11): 3220–3232. [doi: 10.1109/TPDS.2021.3138862]

[38] Ivanov A, Dryden N, Ben-Nun T, Li SG, Hoefler T. Data movement is all you need: A case study on optimizing Transformers. In: Proc. of the 4th Conf. on Machine Learning and Systems. San Jose, 2021. 711–732.

[39] Tang LP, Wang YD, Willke TL, Li K. Scheduling computation graphs of deep learning models on manycore CPUs. arXiv:1807.09667, 2018.

[40] Ding YY, Zhu LG, Jia ZH, Pekhimenko G, Han S. IOS: Inter-operator scheduler for CNN acceleration. In: Proc. of 4th Conf. on Machine Learning and Systems. San Jose: MLSys, 2021.

[41] Ma LX, Xie ZQ, Yang Z, Xue JL, Miao YS, Cui W, Hu WX, Yang F, Zhang LT, Zhou LD. RAMMER: Enabling holistic deep learning compiler optimizations with rTasks. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2020. 881–897.

[42] Zheng Z, Yang XD, Zhao PZ, Long GP, Zhu K, Zhu FW, Zhao WY, Liu XY, Yang J, Zhai JD, Song SL, Lin W. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 359–373. [doi: 10.1145/3503222.3507723]

[43] Zheng Z, Zhao PZ, Long GP, Zhu FW, Zhu K, Zhao WY, Diao LS, Yang J, Lin W. FusionStitching: Boosting memory intensive computations for deep learning workloads. arXiv:2009.10924v2, 2021.

[44] Zheng SZ, Chen SY, Song PD, Chen RZ, Li XH, Yan SG, Lin DH, Leng JW, Liang Y. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In: Proc. of the 2023 IEEE Int'l Symp. on High-performance Computer Architecture (HPCA). Montreal: IEEE, 2023. 1113–1126. [doi: 10.1109/HPCA56546.2023.10071018]

[45] Zhao J, Gao X, Xia RJ, Zhang ZC, Chen DS, Chen L, Zhang RW, Geng Z, Cheng B, Jin XF. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In: Proc. of the 5th Conf. on Machine Learning and Systems (MLSys 2022). Santa Clara, 2022. 1–19.

[46] Zhang C, Dong RC, Wang HJ, Zhong RX, Chen JK, Zhai JD. MAGPY: Compiling eager mode DNN programs by monitoring execution states. In: Proc. of the 2024 USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2024. 683–698.

[47] Niu W, Sanim MR, Shu ZH, Guan JX, Shen XP, Yin M, Agrawal G, Ren B. SmartMem: Layout transformation elimination and adaptation for efficient DNN execution on mobile. In: Proc. of the 29th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. La Jolla: ACM, 2024. 916–931. [doi: 10.1145/3620666.3651384]

[48] Steiner B, Elhoushi M, Kahn J, Hegarty J. MODeL: Memory optimizations for deep learning. In: Proc. of the 40th Int'l Conf. on Machine Learning. Honolulu: JMLR.org, 2023. 1352.

[49] Zheng Z, Pan ZF, Wang DL, Zhu K, Zhao WY, Guo TY, Qiu XF, Sun MM, Bai JJ, Zhang F, Du XY, Zhai JD, Lin W. BladeDISC: Optimizing dynamic shape machine learning workloads via compiler approach. Proc. of the ACM on Management of Data, 2023, 1(3): 206. [doi: 10.1145/3617327]

[50] Unger C, Jia ZH, Wu W, Lin SN, Baines M, Narvaez CEQ, Ramakrishnaiah V, Prajapati N, McCormick P, Mohd-Yusof J, Luo X, Mudigere D, Park J, Smelyanskiy M, Aiken A. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2022). Carlsbad: USENIX Association, 2022. 267–284.

[51] Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM, 2013. 519–530. [doi: 10.1145/2491956.2462176]

[52] Xiao WC, Bhardwaj R, Ramjee R, Sivathanu M, Kwatra N, Han ZH, Patel P, Peng X, Zhao HY, Zhang QL, Yang F, Zhou LD. Gandiva:

Introspective cluster scheduling for deep learning. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2018). Carlsbad: USENIX Association, 2018. 595–610.

[53] Gu JC, Chowdhury M, Shin KG, Zhu YB, Jeon M, Qian JJ, Liu HQ, Guo CX. Tiresias: A GPU cluster manager for distributed deep learning. In: Proc. of the 16th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2019). Boston: USENIX Association, 2019. 485–500.

[54] Mahajan K, Balasubramanian A, Singhvi A, Venkataraman S, Akella A, Phanishayee A, Chawla S. THEMIS: Fair and efficient GPU cluster scheduling. In: Proc. of the 17th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2020). Santa Clara: USENIX Association, 2020. 289–304.

[55] Narayanan D, Santhanam K, Kazhamiaka F, Phanishayee A, Zaharia M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2020). USENIX Association, 2020. 481–498.

[56] Li SJ, Zhang XY, Zhao JC, Tian XH, Shi XY, Xu XX, Cui HM. Automatic insertion method of high-performance synchronization primitives for ascend processors. Journal of Computer Research and Development, 2025, 62(8): 1962–1978 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202440093]

[57] AVX. Chess programming Wiki. 2025. https://www.chessprogramming.org/AVX

[58] ARM Ltd. The architecture for the digital world. 2025. https://www.arm.com/technologies/neon

[59] Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B. Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv:1909.08053, 2019.

[60] Rasley J, Rajbhandari S, Ruwase O, He YX. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proc. of the 26th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining (KDD). ACM, 2020. 3505–3506. [doi: 10.1145/3394486.3406703]

[61] Shazeer N, Cheng YL, Parmar N, Tran D, Vaswani A, Koanantakool P, Hawkins P, Lee H, Hong MS, Young C, Sepassi R, Hechtman B. Mesh-TensorFlow: Deep learning for supercomputers. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 10435–10444.

[62] Zheng LM, Li ZH, Zhang H, Zhuang YH, Chen ZF, Huang YP, Wang YD, Xu YZ, Zhuo DY, Xing EP, Gonzalez JE, Stoica I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2022). Carlsbad: USENIX Association, 2022. 559–578.

[63] Xu YZ, Lee H, Chen DH, Hechtman B, Huang YP, Joshi R, Krikun M, Lepikhin D, Ly A, Maggioni M, Pang RM, Shazeer N, Wang SB, Wang T, Wu YH, Chen ZF. GSPMD: General and scalable parallelization for ML computation graphs. arXiv:2105.04663, 2021.

[64] PyTorch. TorchDynamo. 2024. https://github.com/pytorch/pytorch/tree/main/torch/_dynamo

[65] HuggingFace. Accelerate. 2024. https://huggingface.co/docs/accelerate/index

[66] 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. Montréal: Curran Associates Inc., 2018. 3393–3404.

[67] Li MH, Zhang MJ, Wang C, Li MQ. AdaTune: Adaptive tensor program compilation made efficient. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1241.

[68] Ahn BH, Pilligundla P, Yazdanbakhsh A, Esmaeilzadeh H. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In: Proc. of the 2020 Int'l Conf. on Learning Representations. Addis Ababa: OpenReview.net, 2020.

[69] Haj-Ali A, Genc H, Huang QJ, Moses W, Wawrzynek J, Asanović K, Stoica I. ProTuner: Tuning programs with Monte Carlo tree search. arXiv:2005.13685, 2020.

[70] 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 Symp. on Operating Systems Design and Implementation (OSDI 2020). USENIX Association, 2020. 863–879.

[71] Zheng SZ, Liang Y, Wang S, Chen RZ, Sheng KW. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 859–873. [doi: 10.1145/3373376.3378508]

[72] Mullapudi RT, Adams A, Sharlet D, Ragan-Kelley J, Fatahalian K. Automatically scheduling Halide image processing pipelines. ACM Trans. on Graphics, 2016, 35(4): 83. [doi: 10.1145/2897824.2925952]

[73] Adams A, Ma K, Anderson L, Baghdadi R, Li TM, Gharbi M, Steiner B, Johnson S, Fatahalian K, Durand F, Ragan-Kelley J. Learning to optimize Halide with tree search and random programs. ACM Trans. on Graphics, 2019, 38(4): 121. [doi: 10.1145/3306346.3322967]

[74] 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 with English abstract). http://www.jos.org.cn/1000-9825/7102.htm [doi: 10.13328/j.cnki.jos.007102]

[75] Chen TQ, Guestrin C. XGBoost: A scalable tree boosting system. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD). San Francisco: ACM, 2016. 785–794. [doi: 10.1145/2939672.2939785]

[76] Apache. TVM. 2024. https://tvm.apache.org/

[77] TensorFlow. XLA. 2024. https://www.tensorflow.org/xla?hl=zh-cn

[78] IREE. IREE. 2024. https://iree.dev/

[79] Zhu HY, Wu RF, Diao YJ, Ke SB, Li HY, Zhang C, Xue JL, Ma LX, Xia YQ, Cui W, Yang F, Yang M, Zhou LD, 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 (OSDI 2022). Carlsbad: USENIX Association, 2022. 233–248.

[80] PyTorch. TorchInductor: A PyTorch-native compiler with define-by-run IR and symbolic shapes. 2024. https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747

[81] Ansel J, Yang E, He H, *et al*. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In: Proc. of the 29th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. La Jolla: ACM, 2024. 929–947. [doi: 10.1145/3620665.3640366]

[82] Tillet P, Kung HT, Cox D. Triton: An intermediate language and compiler for tiled neural network computations. In: Proc. of the 3rd ACM SIGPLAN Int'l Workshop on Machine Learning and Programming Languages. Phoenix: ACM, 2019. 10–19. [doi: 10.1145/3315508.3329973]

[83] Cummins C, Seeker V, Grubisic D, Elhoushi M, Liang YW, Roziere B, Gehring J, Gloeckle F, Hazelwood K, Synnaeve G, Leather H. Large language models for compiler optimization. arXiv:2309.07062, 2023.

[84] Cummins C, Seeker V, Grubisic D, Roziere B, Gehring J, Synnaeve G, Leather H. Meta large language model compiler: Foundation models of compiler optimization. arXiv:2407.02524, 2024.

[85] Zhang SM, Zhao JC, Xia CW, Wang Z, Chen YJ, Feng XB, Cui HM. LEGO-compiler: Enhancing neural compilation through composable chain of thought. In: Proc. of the 13th Int'l Conf. on Learning Representations. Singapore: OpenReview.net, 2025.

[86] Tang AS, Priebe C, Mahapatra R, Qin LH, Esmaeilzadeh H. REASONING COMPILER: LLM-guided optimizations for efficient model serving. In: Proc. of the 39th Annual Conf. on Neural Information Processing Systems. San Diego: OpenReview.net, 2025.

[87] Zheng ZC, Cheng L, Li L, Rocha RCO, Liu TY, Wei W, Zhang XW, Gao YQ. VecTrans: LLM transformation framework for better auto-vectorization on high-performance CPU. arXiv:2503.19449, 2025.

[88] Pan HL, Lin HY, Luo HR, Liu Y, Yao KC, Zhang LB, Xing MJ, Wu YJ. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. In: Proc. of the 39th Conf. on Neural Information Processing Systems. 2025.

[89] Lin HY, Pan HL, Luo HR, Li YC, Yao KC, Zhang LB, Xing MJ, Wu YJ. AwareCompiler: Agentic context-aware compiler optimization via a synergistic knowledge-data driven framework. In: Proc. of the 14th Int'l Conf. on Learning Representations. Rio de Janeiro: OpenReview.net, 2026.

[90] Fang HN, Wen YB, Bi J, Wang YH, He TH, Tang YL, Huang D, Guo JM, Zhang R, Guo Q, Chen YJ. QiMeng-NeuComBack: Self-evolving translation from IR to assembly code. arXiv:2511.01183, 2025.

[91] Feng SY, Hou BH, Jin HY, Lin WW, Shao JR, Lai RH, Ye ZH, Zheng LM, Yu CH, Yu Y, Chen TQ. TensorIR: An abstraction for automatic tensorized program optimization. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 804–817. [doi: 10.1145/3575693.3576933]

[92] TensorFlow. MLIR-HLO. 2025. https://github.com/tensorflow/mlir-hlo

[93] DMLC. HalideIR: Symbolic expression and statement module for new DSLs. 2025. https://github.com/dmlc/HalideIR

[94] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018. https://gds.techfak.uni-bielefeld.de/_media/teaching/2024summer/advanced_ml/improving_language_understanding_by_generative_pretraining.pdf

[95] DeepSeek-AI, Liu AX, Feng B, *et al*. DeepSeek-V3 technical report. arXiv:2412.19437, 2024.

[96] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A, Joulin A, Grave E, Lample G. LLaMA: Open and efficient foundation language models. arXiv:2302.13971, 2023.

[97] Rajbhandari S, Rasley J, Ruwase O, He YX. ZeRO: Memory optimizations toward training trillion parameter models. In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE Press, 2020. 20.

[98] Rajbhandari S, Ruwase O, Rasley J, Smith S, He YX. ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning. In: Proc. of the 2021 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. St. Louis: ACM, 2021. 59. [doi: 10.1145/3458817.3476205]

[99] Yu GI, Jeong JS, Kim GW, Kim S, Chun BG. ORCA: A distributed serving system for Transformer-based generative models. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2022. 521–538.
[100] RISC-V Int'l. RISC-V: The open standard RISC instruction set architecture. 2024. https://riscv.org/
[101] THU-DSP-LAB. 2024. https://github.com/THU-DSP-LAB/ventus-gpgpu/blob/master/docs/Ventus-GPGPU-doc.md

附中文参考文献

[8] 马艳军, 于佃海, 吴甜, 王海峰. 飞桨: 源于产业实践的开源深度学习平台. 数据与计算发展前沿, 2019, 1(1): 105–115. [doi: 10.11871/jfdc.issn.2096-742X.2019.01.011]
[56] 李帅江, 张馨元, 赵家程, 田行辉, 石曦予, 徐晓忻, 崔慧敏. 面向昇腾处理器的高性能同步原语自动插入方法. 计算机研究与发展, 2025, 62(8): 1962–1978. [doi: 10.7544/issn1000-1239.202440093]
[74] 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 [doi: 10.13328/j.cnki.jos.007102]

作者简介

林泓宇, 博士生, CCF 学生会员, 主要研究领域为编译技术, 操作系统.
刘洋, 硕士, 主要研究领域为编译技术.
罗浩然, 博士, 主要研究领域为知识图谱, 自然语言处理.
李林海, 硕士, 主要研究领域为编译技术.
曹行行, 博士生, 主要研究领域为编译技术.
王梦娜, 硕士生, 主要研究领域为编译技术.
徐家豪, 硕士生, 主要研究领域为编译技术.
李泉毅, 硕士生, 主要研究领域为编译技术.
张洪滨, 博士, 助理研究员, CCF 专业会员, 主要研究领域为编译技术.
邢明杰, 高级工程师, CCF 专业会员, 主要研究领域为编译技术.
武延军, 博士, 研究员, 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.