软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(6):2390−2410 [doi: 10.13328/j.cnki.jos.007619] [CSTR: 32375.14.jos.007619]

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

面向 RISC-V 架构的深度学习算子测试*

刘佳玮[1,2], 高 岩[1,2], 张犬俊[3], 尚 也[1,2], 房春荣[1,2], 陈振宇[1,2]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210046)

[2](南京大学 软件学院, 江苏 南京 210093)

[3](南京理工大学 计算机科学与工程学院 (人工智能学院、软件学院), 江苏 南京 210094)

通信作者: 房春荣, E-mail: fangchunrong@nju.edu.cn

摘 要: 随着边缘计算和端侧智能软件的发展, RISC-V 架构因其开源、模块化及低成本优势在学术界和产业界受到广泛关注. 然而, 将智能软件部署到 RISC-V 架构上面临诸多挑战. 智能软件的运行依赖于执行卷积、矩阵乘法、归一化等基本张量运算的深度学习算子, 一旦算子出现问题, 将直接影响上层大量智能软件的执行效率、准确性和可靠性, 因此算子质量的评估至关重要. 现有算子测试方法主要针对 x86 架构, 难以刻画不同计算复杂度对 RISC-V 架构下受限内存、功耗与系统资源使用行为的影响差异. 为此, 提出 RIVdoo, 一种面向 RISC-V 架构的深度学习算子质量评估方法. RIVdoo 通过基于算子输入空间复杂度的分组策略系统性地覆盖不同计算负载, 结合多维度指标评估精度、执行效率、内存占用及系统开销, 并引入复杂度放大系数的差分测试机制, 有效识别性能异常及架构适配问题. 在覆盖计算密集型、内存密集型和轻量型的 47 个算子的实验中, RIVdoo 揭示了不同算子库和优化策略在 RISC-V 架构下呈现出的显著性能权衡与适配差异: TFLite 以 1.5–2 倍内存开销换取 30%–50% 速度提升的策略在资源受限场景下面临内存瓶颈; TVM 的动态存储调度因 RISC-V 有限的 TLB 和缓存容量导致缺页率比 TFLite 高 60%–150%; RVV 向量化在部分算子和低复杂度场景下因软件模拟和启动开销导致性能劣化, 说明现有优化策略缺乏对 RISC-V 平台特性的针对性设计. 实验结果表明, 不同算子实现和优化策略在 RISC-V 架构下的运行行为具有明显的复杂度相关特征, 仅依赖输出正确性验证难以全面反映其真实部署表现. RIVdoo 为面向 RISC-V 架构的算子适配性分析与优化提供了一种系统化的评估手段.

关键词: 深度学习算子; 差分测试; RISC-V; 算子部署

中图法分类号: TP311

中文引用格式: 刘佳玮, 高岩, 张犬俊, 尚也, 房春荣, 陈振宇. 面向RISC-V架构的深度学习算子测试. 软件学报, 2026, 37(6): 2390–2410. http://www.jos.org.cn/1000-9825/7619.htm

英文引用格式: Liu JW, Gao Y, Zhang QJ, Shang Y, Fang CR, Chen ZY. Deep Learning Operator Testing for RISC-V Architecture. Ruan Jian Xue Bao/Journal of Software, 2026, 37(6): 2390–2410 (in Chinese). http://www.jos.org.cn/1000-9825/7619.htm

Deep Learning Operator Testing for RISC-V Architecture

LIU Jia-Wei[1,2], GAO Yan[1,2], ZHANG Quan-Jun[3], SHANG Ye[1,2], FANG Chun-Rong[1,2], CHEN Zhen-Yu[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210046, China)

[2](Software Institute, Nanjing University, Nanjing 210093, China)

[3](School of Computer Science and Engineering (School of Artificial Intelligence & School of Software), Nanjing University of Science & Technology, Nanjing 210094, China)

* 基金项目: 国家重点研发计划 (2024YFF0908004); 国家自然科学基金 (U24A20337, 62372228); 深港澳科技计划 (C 类) (SGDX20230821091559018)
本文由“RISC-V 与人工智能系统软件前沿进展”专题特约编辑武延军研究员、谢涛教授、侯锐研究员、宋威研究员、邢明杰高级工程师推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20, 2025-12-08, 2025-12-15; 采用时间: 2025-12-17; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-04-02

Abstract: With the rapid development of edge computing and intelligent end-side software, the RISC-V architecture attracts increasing attention in both academia and industry due to its open-source, modular, and low-cost characteristics. However, deploying intelligent software on the RISC-V architecture presents significant challenges. The execution of intelligent software relies on deep learning operators, such as convolution, matrix multiplication, and normalization. Once defects occur in these operators, the execution efficiency, accuracy, and reliability of a large number of upper-layer intelligent applications are directly affected, making operator quality evaluation critically important. Existing operator testing methods are primarily designed for x86 architectures and have difficulty characterizing the impact of varying computational complexity on RISC-V platforms under constraints such as limited memory capacity, power consumption, and system resources. To address this issue, this study proposes RIVdoo, a deep learning operator testing method tailored for the RISC-V architecture. RIVdoo systematically covers different computational workloads through a grouping strategy based on operator input-space complexity, evaluates accuracy, execution efficiency, memory usage, and system overhead using multidimensional metrics, and introduces a differential testing mechanism with complexity amplification factors to effectively identify performance anomalies and architectural adaptation issues. Experiments covering 47 operators across compute-intensive, memory-intensive, and lightweight categories demonstrate that RIVdoo reveals significant performance trade-offs and adaptation differences among existing operator libraries and optimization strategies on the RISC-V architecture. Specifically, TFLite trades 1.5×–2× memory overhead for 30%–50% performance improvement, which leads to memory bottlenecks in resource-constrained scenarios; TVM's dynamic storage scheduling incurs 60%–150% higher page fault rates than TFLite due to the limited TLB and cache capacity of RISC-V; RVV vectorization causes performance degradation for certain operators and low-complexity workloads because of software emulation and startup overhead, indicating that existing optimization strategies lack targeted design for RISC-V platform characteristics. The results demonstrate that the runtime behavior of different operator implementations and optimization strategies on the RIVdoo architecture exhibits strong complexity-dependent characteristics, and that output correctness alone is insufficient to reflect real deployment performance. RISC-V provides a systematic evaluation methodology for operator adaptability analysis and optimization on RISC-V platforms.

Key words: deep learning operator; differential testing; RISC-V; operator deployment

RISC-V 架构是一种基于精简指令集 (RISC) 思想的开源指令集架构, 随着 RISC-V 架构在边缘计算和物联网领域的快速发展, 其模块化设计、可扩展性和成本优势正逐渐获得学术界与产业界的广泛关注[1,2]. 特别是在资源受限的嵌入式场景中, RISC-V 架构凭借低成本和灵活性, 为复杂软件的高效部署提供了新的可能性[3]. 与此同时, 基于深度学习模型的复杂软件在视觉识别、语音处理和智能控制等领域的广泛应用[4], 使得在基于 RISC-V 架构的平台上部署神经网络模型成为亟需解决的问题[5,6].

然而, 深度学习模型的执行高度依赖底层算子库的性能与稳定性, 算子是实现特定张量运算的基本功能单元, 例如卷积 (Conv2D)、矩阵乘法 (MatMul)、归一化 (BatchNorm) 等[7,8]. 模型的执行过程是由大量算子按计算图依次调用完成的, 因此算子的性能、精度与稳定性直接决定了整个模型在目标平台上的运行质量[9,10]. 若算子实现存在性能瓶颈或资源利用低效, 则即便模型结构先进, 也难以在硬件上获得预期效果. 当前主流算子库 (如 TVM、TensorFlow lite (TFLite) 等) 主要面向 x86 与 ARM 架构进行优化, 对 RISC-V 架构的支持尚不完善, 其在算子层面的执行性能、资源利用效率以及运行稳定性仍存在诸多未知因素. 因此, 针对 RISC-V 架构的算子测试与质量评估成为构建可用深度学习生态的基础性问题.

与成熟的 x86 和 ARM 生态相比, RISC-V 架构上的算子部署面临若干独特挑战. 首先, RISC-V 架构在编译器优化、运行时库支持以及硬件加速单元方面尚处于发展阶段, 可能导致算子实际性能与理论预期存在显著差距[11]. 其次, RISC-V 架构通常运行在内存容量有限、功耗约束严格以及散热能力不足的环境中, 使得算子执行效率不仅影响计算性能, 也直接关系到系统稳定性与寿命[12,13]. 此外, RISC-V 架构上的部署多依赖静态交叉编译方式, 虽然简化了依赖管理, 但增加了移植复杂度并可能引入新的性能损失[14−16].

这些 RISC-V 架构的独特挑战使得传统的深度学习算子测试方法暴露出两个关键局限性. 针对资源约束和热管理挑战, 现有算子测试方法采用统一的参数配置策略, 无法区分不同运算复杂度对有限内存、功耗预算和散热能力的差异化影响, 缺乏系统性的输入空间划分来识别复杂度相关的资源利用模式和热行为变化. 针对编译优化不足和架构特性差异挑战, 现有算子测试方法主要围绕 x86 和 ARM 架构开展研究, 其测试指标主要关注通用性能问题, 而对 RISC-V 架构特有的约束 (如精简指令集效率、静态编译代码膨胀、缓存层级限制等) 缺乏针对性评估维度, 无法有效识别这些架构特有因素导致的算子质量问题.

针对上述问题, 本文提出一种面向 RISC-V 架构的深度学习算子质量评估方法 RIVdoo. 针对资源约束环境下

的复杂度敏感性问题, RIVdoo 设计了基于运算复杂度的输入空间分组策略, 通过系统性地覆盖从轻负载到高负载的不同计算场景, 能够精确识别算子在各种工作负载下对有限内存、功耗预算和散热能力的差异化影响, 从而揭示传统统一测试方法无法发现的资源利用规律和性能临界点. 针对 RISC-V 架构带来的特性差异问题, RIVdoo 构建了多维度测试分析体系, 从精度、执行效率、内存占用和系统调用开销等层面全面评估算子在 RISC-V 架构的适配质量, 并通过基于四分位距 (interquartile range, *IQR*) 的复杂度放大系数的差分测试方法, 对不同复杂度下度量指标的变化模式进行统计分析, 有效识别算子实现的性能异常及架构适配问题. 通过上述 3 大模块的协同作用, RIVdoo 实现了在资源受限、架构特有约束下对深度学习算子复杂度敏感性与软硬件交互行为的系统性评估, 为算子优化和平台调优提供了量化依据.

为了验证 RIVdoo 方法在 RISC-V 架构深度学习算子测试中的有效性, 本文针对 47 个深度学习算子开展大规模测试, 覆盖计算密集型、内存密集型和轻量型这 3 种主要计算模式, 能够充分反映深度学习模型在不同资源需求和计算特征下的表现. 实验结果表明, RIVdoo 在实验中系统性地揭示了不同算子库与优化策略在 RISC-V 平台上的行为差异与性能权衡特征: TFLite 的预编译策略在 RISC-V 平台上相比 TVM 具有 30%–50% 的速度优势但以 1.5–2 倍的内存开销为代价, TVM 的动态存储调度策略在归约类算子 (Softmax、Mean) 上因 RISC-V 有限的 TLB (32–64 项) 和缓存容量 (2 MB L2) 导致缺页率比 TFLite 高 60%–150%, RVV 向量化优化在计算密集型算子 (Conv2D、MatMul) 的高复杂度场景下性能提升 30%–50%, 但在除法、取模等缺乏硬件加速支持的算子以及低复杂度场景下因软件模拟开销和向量化启动成本反而导致性能下降 10%–30%, 以及 TVM 的频繁小块内存分配机制在低复杂度场景下导致系统调用开销占比超过 30%. 上述结果从执行效率、内存占用、存储访问和系统行为等多个维度, 揭示了算子在 RISC-V 架构上的资源敏感性特征与平台相关的行为差异, 同时验证了 RIVdoo 能够在无需依赖运行时反馈的条件下, 有效识别传统精度验证方法难以暴露的效率、内存与系统层面的潜在适配风险, 为算子优化和平台调优提供了可复现、可量化的实验依据.

综上所述, 本文的主要贡献总结如下.

(1) 提出面向 RISC-V 架构的深度学习算子测试理论, 实现对不同工作负载下资源利用模式和性能临界点的量化评估, 并通过多维度指标体系和复杂度放大系数差分测试, 系统性地揭示算子在 RISC-V 架构下可能呈现的异常行为模式与潜在适配问题.

(2) 基于 TVM 和 TFLite 框架开发了 RIVdoo 测试工具 (https://github.com/yanyanyyy720/OperatorTestForRiscv), 实现了算子在高、中、低复杂度场景下的自动化测试生成、性能监控和差分分析, 能够有效支持算子库在 RISC-V 架构上的适配性评估.

(3) 开展大规模算子测试实验, 覆盖多类深度学习算子和不同参数配置, 从执行效率、内存峰值、缺页率及系统开销等角度, 系统刻画算子在 RISC-V 架构上的平台相关行为特征与资源敏感性规律, 为算子优化与平台调优提供可靠的数据依据.

本研究不仅为 RISC-V 架构上深度学习生态的健康发展提供了专门化的测试和评估工具, 更为新兴指令集架构上的智能软件质量保证建立了系统性的方法论基础, 在推动开源硬件生态与人工智能技术融合发展方面具有重要理论价值和实践意义.

本文第 1 节回顾深度学习算子测试相关工作和 RISC-V 算子部署研究背景. 第 2 节详细阐述提出的面向 RISC-V 架构的深度学习算子测试方法 RIVdoo. 第 3 节介绍实验设计方案, 包括测试算子选择、参数分组策略和评估指标体系. 第 4 节展示并分析实验结果, 重点讨论 RISC-V 架构特有的算子性能特征和资源利用模式. 第 5 节总结全文并展望未来研究方向.

1 相关工作和研究背景

1.1 深度学习算子测试

深度学习算子是构成神经网络计算图的基础数学运算单元, 包括卷积运算、矩阵乘法、池化操作、激活函数、

批归一化等核心计算模块[17]. 这些算子不仅决定了神经网络的计算语义, 更直接影响模型的训练收敛性和推理准确性[18]. 深度学习算子测试是指对这些基础运算单元进行功能正确性、数值精度和跨平台一致性验证的系统性方法[19]. 与传统软件测试不同, 深度学习算子测试面临独特的技术挑战: 首先是测试预言问题, 即缺乏明确的预期输出作为算子计算正确性的判断标准[20]; 其次是数值精度问题, 深度学习计算涉及大规模张量运算、多层网络的链式求导和复杂的数值优化过程, 这些计算特有的高维度、长序列和深层级联特性使得浮点误差在网络传播过程中呈指数级放大, 微小的算子实现差异可能在经过多次迭代和层级传递后导致模型收敛失败或推理结果的显著偏差[21]; 再次是输入空间的组合爆炸问题, 张量维度、数值范围和参数配置的排列组合使得穷举测试在计算上不可行[22,23]. 深度学习算子中的错误会通过上层网络的运算进一步被传播放大, 导致模型训练失败或推理结果错误, 最终造成上层智能软件系统功能异常、业务逻辑紊乱和服务质量恶化, 严重时甚至可能引发系统崩溃或安全事故, 对依赖人工智能技术的现代软件生态造成连锁性损害[24,25].

现有的深度学习算子测试方法主要围绕差分模糊测试来开展. 如图 1 所示, 典型的深度学习算子测试工作流程包含从测试输入生成到结果验证的完整技术体系[17,21]. 具体而言, 张量形状初始化与种子集收集为算子测试提供了符合规范约束的输入基础, 负责根据算子接口规范生成有效的张量维度配置和初始数据样本; 测试语料生成与变异方法应用通过智能化的输入扩展策略系统性地探索算子的输入空间, 采用覆盖率引导的模糊测试等方法最大化测试覆盖率; 算子执行阶段实现了跨框架、跨硬件的一致性验证, 为后续验证算子在不同计算环境下的行为一致性分析提供了实验数据; 精度评估与差分测试构建了多层次的正确性判断体系, 通过相对误差分析等方式解决测试预言问题. 这一完整的测试流程体现了现代深度学习算子测试的核心方法论, 其中每个环节都针对深度学习算子测试的特定挑战提供了专门化的解决方案, 共同构成了保障深度学习框架质量的重要技术基础.

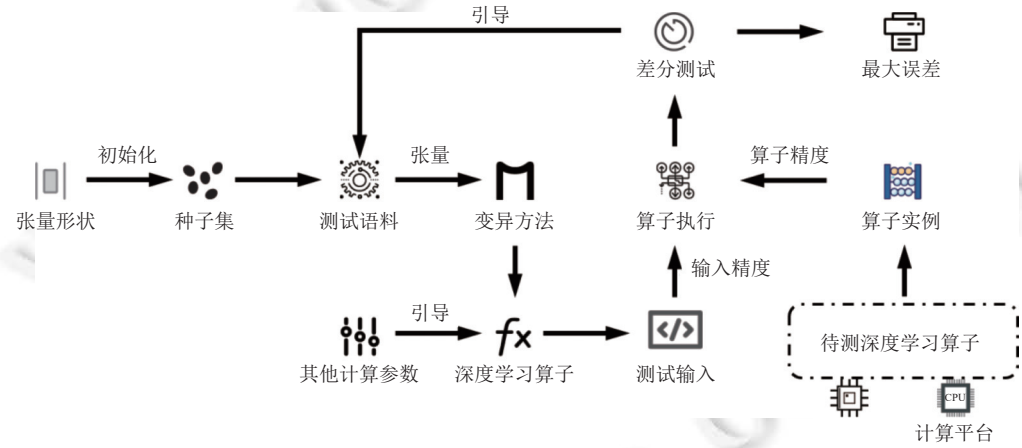

图 1 深度学习算子的一般测试流程

张量形状初始化作为算子测试的起始环节, 承担着根据算子输入规范生成符合约束条件的张量维度配置的重要任务. 现有方法采用基于约束求解的生成策略, 通过分析算子的应用程序接口文档和实现代码, 自动推导出满足维度兼容性要求的张量形状组合[22]; 这一环节需要处理动态形状推导、多输入张量的维度匹配和边界条件覆盖等复杂问题. 特别是对于支持广播机制的算子, 形状初始化过程必须考虑不同维度张量之间的广播规则, 确保生成的测试用例能够触发算子的各种执行路径[23,26].

种子集收集环节通过多种策略获取具有代表性的输入数据, 为后续测试语料的生成奠定了重要基础. 现有方法主要依赖随机采样和人工构造的方式来收集种子数据[27,28], 同时结合历史测试数据中的错误模式来指导种子选择策略[29,30].

测试语料生成是整个算子测试流程的核心环节, 负责基于收集的种子数据产生大规模、多样化的测试输入. 张量模糊测试方法采用近似最近邻等引导算法实现覆盖率引导的模糊测试, 通过维护覆盖率等反馈机制来优化输

入生成策略[23]; 梯度模糊测试利用反向传播等算法计算的梯度等反馈信息生成对抗性扰动, 专门用于测试算子在边界条件下的数值稳定性[8]; 基于变形关系的测试方法通过数学等价变换生成满足特定性质的测试用例, 这种方法特别适用于具有明确数学定义的算子测试[21]. 变异方法应用环节通过系统性的输入变换操作扩展测试用例的覆盖范围, 是提升算子测试效果的关键技术. 现有方法采用张量形状变异、数值范围修改、边界值测试、类型转换、内存布局变化和组合变换等策略来系统性地探索输入参数空间[9].

算子执行环节负责在多种计算环境中运行测试用例并收集执行结果, 是验证算子跨平台一致性的重要阶段. 现有方法支持跨框架验证能力, 能够在 TensorFlow、PyTorch 等主流深度学习框架上并行执行相同测试用例, 通过对比不同框架的执行结果来发现实现差异和潜在错误[31,32]. 精度评估是验证算子数值正确性的关键环节, 现有方法建立了多维度的精度指标评估体系, 包括统计误差分布特征分析、变形关系违反检测和形式化验证技术的应用[21]. 差分测试作为算子测试流程的核心验证手段, 通过比较不同实现或不同配置下的算子执行结果来识别潜在的功能异常和实现不一致性. 现有方法一般通过跨框架的输出差异分析来检测算子的实现不一致性和兼容性问题[32].

然而, 现有算子测试方法主要针对资源丰富的 x86 和 ARM 架构设计, 在面对 RISC-V 等新兴架构时暴露出明显的局限性.

1.2 面向 RISC-V 架构的深度学习算子部署

RISC-V 架构作为开源指令集架构, 凭借其模块化设计、可扩展性和无专利限制等优势, 正在成为边缘计算和物联网领域的重要选择[33]. 然而, 在 RISC-V 架构上部署深度学习算子面临着与成熟 x86 和 ARM 生态系统截然不同的技术挑战和部署要求. 深度学习算子在 RISC-V 架构上的部署不仅需要解决基本的功能移植问题, 更需要应对指令集差异、编译工具链不成熟、硬件资源限制和性能优化等多方面的技术难题.

在 RISC-V 架构上部署深度学习算子通常采用静态交叉编译的方式, 这一部署流程与传统架构存在显著差异. 首先, 算子源代码需要通过 RISC-V 工具链进行交叉编译, 将针对 x86 或 ARM 架构编写的算子代码转换为 RISC-V 机器码. 由于 RISC-V 生态系统的相对不成熟, 这一过程经常遇到编译器优化不足、运行时库缺失和依赖项不兼容等问题[34]. 其次, 编译生成的可执行文件需要进行静态链接, 将所有依赖库打包到单一的可执行文件中, 以避免运行时的动态链接问题. 这种静态编译方式虽然简化了部署复杂度, 但也带来了代码体积膨胀、内存占用增加和优化机会减少等副作用[35]. 算子部署过程中还需要考虑 RISC-V 架构的多种配置变体对部署策略的影响. RISC-V 指令集采用模块化设计, 基础指令集 RV32I/RV64I 可以通过添加扩展模块 (如 M 扩展用于乘除法、F 扩展用于单精度浮点、D 扩展用于双精度浮点、V 扩展用于向量操作等) 来增强功能[36]. 不同的 RISC-V 实现可能支持不同的扩展组合, 这要求算子部署时必须根据目标硬件的具体配置选择合适的编译选项和优化策略. 特别是对于数值密集型的深度学习算子, 浮点扩展和向量扩展的支持情况直接影响算子的性能表现和实现方式.

RISC-V 架构的精简指令集设计理念对深度学习算子的功能实现和性能表现产生了深远影响. 在功能层面, RISC-V 采用固定长度的简单指令, 复杂的数学运算需要通过多条基础指令的组合来实现. 深度学习中常见的矩阵乘法、卷积运算等计算密集型操作需要执行更多的指令才能完成相同的计算任务[37], 这不仅影响执行效率, 也增加了指令缓存的压力. 浮点运算支持的差异进一步影响算子功能, RISC-V 的浮点扩展在处理特殊数值和舍入模式时的行为可能与其他架构不完全一致, 影响深度学习算子的数值精度和结果一致性[38].

在性能层面, RISC-V 架构的内存层次结构对算子性能具有决定性影响. 典型的 RISC-V 处理器采用相对简单的缓存层次结构, L1 缓存容量通常在数十 KB 级别, L2 缓存容量在数百 KB 到数 MB 范围内, 远小于现代 x86 和 ARM 处理器的多级缓存体系[39]. 深度学习算子中的大规模张量操作对内存带宽和缓存效率要求极高, 有限的缓存容量可能导致频繁的缓存未命中, 严重影响算子执行性能[40]. 同时, RISC-V 处理器通常采用相对简单的流水线设计, 分支预测和乱序执行能力有限[41,42], 在处理具有复杂控制流的算子时可能成为性能瓶颈.

总体而言, RISC-V 架构的深度学习算子部署是一个涉及指令集特性、编译优化、资源管理和性能调优等多个方面的复杂系统工程. 成功的算子部署不仅需要解决基本的功能移植问题, 更需要针对 RISC-V 架构的独特特征进行深度优化和适配. 这些挑战使得传统的深度学习算子测试方法在 RISC-V 架构上暴露出明显的局限性, 迫

切需要发展专门化的测试方法来确保算子在新兴指令集架构上的可靠运行.

2 RIVdoo: 面向 RISC-V 架构的深度学习算子测试方法

2.1 方法概述

图 2 展示了 RIVdoo 的整体技术架构和测试流程. 该方法从深度学习算子的计算参数出发, 通过系统性的复杂度分层策略生成涵盖高、中、低这 3 个复杂度层级的测试用例, 每个层级对应不同的 RISC-V 架构资源挑战模式. 在测试执行阶段, RIVdoo 采用差分测试机制, 同时在 TVM[43]和 TFLite[44]两个主流算子库环境下运行相同的测试用例, 并从精度、效率、内存和系统这 4 个层面全面采集算子在 RISC-V 架构上的性能表现数据. 基于上述技术架构, 本文在第 2.2 节详细阐述基于输入空间复杂度分组的测试生成策略, 在第 2.3 节深入分析面向 RISC-V 架构的多维度测试分析方法, 在第 2.4 节介绍基于复杂度放大系数的差分测试机制.

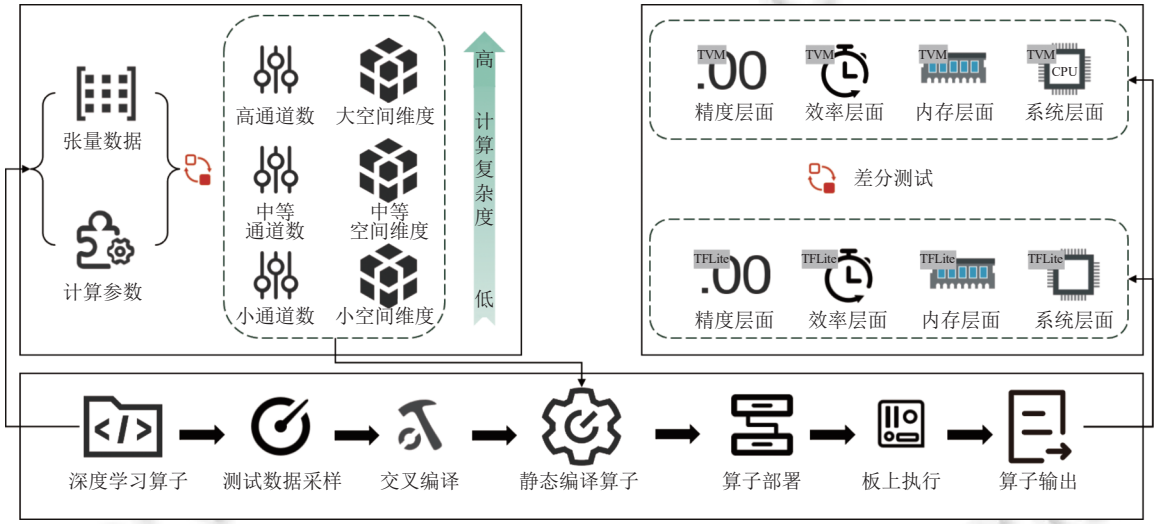

图 2 RIVdoo 的测试流程

2.2 基于输入空间复杂度分组的测试生成

传统深度学习算子测试方法通常采用固定参数配置或随机采样策略, 无法有效识别不同计算负载对资源受限 RISC-V 架构的差异化影响. 仅评估轻负载场景的性能表现难以预测重负载场景的资源瓶颈, 而仅评估重负载测试可能掩盖轻负载场景的效率问题. 为全面反映深度学习算子在不同计算负载下的性能表现, RIVdoo 设计了一种基于输入参数复杂度的分组策略. 该策略的核心思想是: 算子的计算量、内存占用与访存开销由关键输入参数共同决定, 而这些参数在特定硬件资源约束下会引起显著的性能差异. 因此, RIVdoo 从理论层面对参数规模与硬件资源消耗之间的关系进行分析, 确定参数的高、中、低复杂度区间, 并据此生成不同计算负载的测试配置. 这种设计既能在有限测试成本下覆盖多层次计算场景, 又能保持在不同算子与平台上的可扩展性和可复现性.

2.2.1 分组依据

深度学习算子的计算复杂度源于多个关键参数的交互组合效应, 这些参数在 RISC-V 平台的资源约束环境下产生显著不同的性能挑战模式. 设算子的参数配置 P 涉及 k 个关键参数 $p_1, p_2, \ldots, p_k$, 其计算复杂度可表示为 $C(P) = \alpha \cdot \prod_{i=1}^{k} p_i$, 其中 α 为与算子类型相关的常数. 对于两个参数配置 P 和 P', 若 P' 的每个参数均不小于 P 对应参数, 即 $p'_i \geqslant p_i$ 对所有 i 成立, 则有:

$$\frac{C(P')}{C(P)} = \prod_{i=1}^{k} \frac{p'_i}{p_i} \geqslant 1.$$

因此 $C(P') \geqslant C(P)$, 这表明参数增长与计算复杂度增长之间存在严格的单调递增关系. 这种累乘效应使得即使单个参数适度增长, 多个参数协同变化也能产生指数级的复杂度提升.

在 RISC-V 平台的资源约束环境下, 这些参数的不同取值区间会产生显著差异的性能挑战模式. 因此, 在每个参数维度上可以独立地定义高、中、低复杂度区间, 且该划分在理论上与算子复杂度增长趋势一致.

算子在实际硬件平台上的性能变化主要受 3 类资源限制因素共同决定: 计算能力、缓存容量与内存带宽. 当输入参数逐步增大时, 算子的执行时间会经历 3 个阶段: 计算受限、缓存溢出与带宽受限, 不同阶段之间的跃迁点, 即构成高、中、低复杂度区间的临界边界. 首先, 计算饱和点反映了算子乘加操作量接近 CPU 峰值算力的区间, 当参数规模较小, CPU 能够在缓存命中率高的情况下维持线性加速, 此时性能主要由计算复杂度决定, 当参数进一步增大, 乘加次数接近处理器吞吐极限, 计算速率开始下降, 进入计算饱和阶段. 其次, 缓存驻留边界由 L2 缓存容量决定, 设单次前向传播的特征张量大小为 S_T, 若 $S_T \leqslant C_{L2}$ (其中 C_{L2} 为 L2 缓存容量), 所有数据均可缓存, 访存延迟最小; 当 $S_T > C_{L2}$ 时, 部分数据需要反复在主存与缓存之间调度, 访存延迟显著上升, 性能出现“阶跃”式变化. 此外, 带宽受限区对应数据吞吐量超过内存带宽上限的位置, 当算子需读写的数据量 S_T 超过带宽可支持范围时, 内存访问成为主要瓶颈, 算子执行时间呈非线性增长. 因此, 性能变化在“完全缓存可驻留→部分缓存溢出→频繁带宽竞争”这 3 个阶段间形成明显的拐点.

基于上述分析, 我们在每个参数维度上定义 3 档复杂度区间, 使其临界点与性能跃迁位置一一对应: 低复杂度区对应参数取值使得算子数据完全驻留于缓存内, 计算量远低于 CPU 峰值, 性能近似线性; 中复杂度区对应部分数据溢出缓存, 但尚未达到带宽瓶颈, 访存开销开始主导, 性能随参数缓慢下降; 高复杂度区对应参数进一步增长导致缓存频繁置换、主存带宽饱和, 执行时间曲线出现明显拐点.

以本文实验平台 (8 GB 内存、四核 1.5 GHz RISC-V CPU、2 MB L2 缓存) 为例进行具体说明. 当 *batch* 从 2 增至 4 时, 计算量虽翻倍但仍处缓存可控范围; 当超过 8 时, 中间张量规模超过 L2 容量的两倍, 需频繁访问主存, 性能急剧下降, 因而 [1, 2)、[2, 4)、[4, 8) 可合理定义为低、中、高复杂度区. 对于通道数 (in_C/out_C), 当值小于 64 时, 单层特征图仅占用约 256 KB 内存, 可完全缓存; 当增加至 128–256 时, 占用约 0.5–1 MB, 已接近 L2 缓存上限的一半, 访存延迟显著上升; 超过 256 后溢出频繁、带宽占用迅速增大, 因此定义区间 [8, 64)、[64, 256)、[256, 512). 对于空间维度 (H/W), 在 32×32 以下时计算稳定, 扩展至 56×56 时单张特征图近 1 MB, 达到缓存临界; 至 112×112 时超出缓存两倍, 主存访问成为主要瓶颈, 因而采用 [8, 28)、[28, 56)、[56, 112) 划分.

这些临界点均由硬件资源约束推导而来, 并在实测中对应算子性能曲线的突变位置. 因此, 不同参数的高、中、低分组既符合计算复杂度的单调递增规律, 又准确反映了硬件资源利用的分层边界. 当多参数共同作用于算子时, 其整体复杂度划分自然继承单参数划分的合理性, 实现了理论与实验的一致. 对于不同硬件平台, 只需根据其缓存容量和内存带宽重新计算相应的临界点位置, 即可获得适配该平台的分组方案.

2.2.2 分组设计

RIVdoo 采用基于区间约束的参数空间分组方法, 将关键参数维度抽象为可复用的参数池并对参数空间进行分组, 每个参数池包含低、中、高这 3 个复杂度区间. 不同算子往往涉及相同类型的参数维度, 例如批量维度、通道维度和空间维度在卷积算子、池化算子以及大多数逐元素算子中普遍存在. 通过将这些共性参数抽象为统一的参数池, 不同算子可以直接复用相同的参数区间定义, 避免了为每个算子单独设计参数空间的重复工作, 提升了本方法的可复用性.

表 1 展示了 RIVdoo 定义的核心参数池及其复杂度区间划分. 部分参数维度 (如批量、通道、空间维度) 定义了多个版本的参数池, 这是由不同算子的计算特性差异决定的. 例如, 卷积算子由于涉及输入输出通道的乘积计算, 采用标准批量 [1, 2)、[2, 4)、[4, 8) 即可触发 3 个资源压力阶段; 而计算简单的逐元素运算算子 (如 ReLU、Add) 数据规模较小, 需要采用扩展批量 [1, 2)、[2, 8)、[8, 16) 才能达到相同的缓存溢出和带宽饱和效果. 这种差异化设计确保不同算子在各自的低、中、高复杂度配置下, 实际触发的硬件资源压力保持在相同的量级范围内.

表 2 展示了各算子如何从参数池中选择并组合参数, 以及不同参数组合对 RISC-V 平台资源的压力特征. 以

Conv2D 算子为例, 其使用标准批量、标准通道 (输入输出)、标准空间以及 Conv2D 卷积核的组合. 通过这种参数选择和组合策略, 每个算子都能够获得与其计算特性相匹配的复杂度配置, 同时不同算子之间又保持了统一的复杂度层级体系.

表 1　主要参数的区复杂度区间分组配置

参数类别	参数名称	低复杂度	中复杂度	高复杂度
批量维度 (*batch*)	标准批量	[1, 2)	[2, 4)	[4, 8)
	扩展批量	[1, 2)	[2, 8)	[8, 16)
	大批量	[1, 128)	[128, 512)	[512, 2048)
通道维度	标准输入通道 (in_C)	[8, 64)	[64, 128)	[128, 256)
	扩展输出通道 (out_C)	[8, 64)	[64, 256)	[256, 512)
	宽范围通道 (C)	[1, 32)	[32, 256)	[256, 512)
空间维度 (H, W)	标准空间	[8, 28)	[28, 56)	[56, 112)
	扩展空间	[8, 32)	[32, 112)	[112, 224)
核尺寸 (*kernel* size)	卷积核	[1, 3)	[1, 3)	[3, 5)
	池化核	[2, 3)	[3, 5)	[5, 7)
步长 (*stride*)	步长	[1, 2)	[2, 3)	[3, 4)
特征维度	标准特征维度 (*in_dim*, *out_dim*)	[64, 512)	[512, 2048)	[2048, 4096)
	扩展特征维度 (*feature_dim*)	[64, 512)	[512, 4096)	[4096, 8192)
矩阵维度	矩阵行列维度 (M, N)	[32, 256)	[256, 1024)	[1024, 2048)
	矩阵内积维度 (K)	[32, 128)	[128, 1024)	[1024, 2048)

表 2　算子和参数映射关系

算子类别	算子列表	涉及参数
卷积类	Conv2D	*batch*, in_C, out_C, H, W, *kernel*
	DepthwiseConv2D	*batch*, C, H, W, *kernel*, *depth_multiplier*
池化类	AvgPool2D, MaxPool2D	*batch*, C, H, W, *kernel*, *stride*
全连接类	Dense, FullyConnected	*batch*, *in_dim*, *out_dim*
矩阵运算类	MatMul	M, K, N
归一化类	BatchNorm, L2Normalization	*batch*, C, H, W
归约类	Softmax, LogSoftmax	*batch*, *feature_dim*
	Mean, Sum	*batch*, C, H, W
激活类	ReLU, ReLU6, ELU, LeakyReLU, HardSwish, Logistic, Tanh, PReLU	*batch*, C, H, W
逐元素二元运算类	Add, Sub, Mul, Div, Maximum, Minimum, SquaredDifference, BiasAdd	*batch*, C, H, W
逐元素一元运算类	Abs, Ceil, Floor, Round, Neg, Cos, Sin, Exp, Log, Sqrt, Rsqrt, Square	*batch*, C, H, W
逻辑运算类	Equal, NotEqual, Greater, GreaterEqual, Less, LessEqual, LogicalAnd, LogicalOr, LogicalNot	*batch*, C, H, W
张量操作类	Concatenation, ExpandDims, Pad, Cast	*batch*, C, H, W
	Reshape	*batch*, *total_elements*

基于参数池的设计使得 RIVdoo 具有良好的算子扩展能力. 当需要测试新的算子时, 首先识别该算子涉及的参数维度类型, 然后从表 1 的参数池中选择对应的参数区间, 最后组合生成低、中、高这 3 个复杂度配置. 由于参数池的设计基于硬件约束的理论分析, 任何使用这些参数的算子组合都能够保证明显的复杂度梯度, 无需针对每个新算子重新验证分组有效性.

考虑到 RISC-V 平台的性能表现对参数配置具有高度敏感性, 即使在同一复杂度区间内, 不同的具体参数值也可能触发截然不同的性能模式和资源利用模式. 因此, 在确定各复杂度层级的参数区间后, RIVdoo 在每个区间内采用随机采样策略生成具体的测试参数配置. 通过随机采样策略, RIVdoo 能够有效避免固定参数配置可能引入

的测试偏差, 确保在每个复杂度层级内获得具有统计代表性的性能评估结果. 同时, 该策略能够发现区间边界附近的性能突变现象, 识别可能导致 RISC-V 平台性能急剧变化的临界参数配置, 为算子优化和平台调优提供精确的参数敏感性分析数据. 此外, RIVdoo 还能够模拟真实深度学习应用中算子参数的多样性分布特征, 使得测试结果具备更好的泛化性和实用指导价值, 避免因参数配置过于规整而产生的性能评估失真问题.

2.3 面向 RISC-V 架构的多维度测试分析

RIVdoo 将影响深度学习算子在基于 RISC-V 架构的平台上运行质量的因素映射到软硬结合的多个核心层面. 现有的深度学习算子测试方法主要通过验证算子输出与预期结果的一致性来判断质量, 这种功能正确性测试仅从纯软件视角考虑算子实现的逻辑正确性, 无法识别算子与特定硬件架构交互过程中的适配问题. RISC-V 作为新兴的精简指令集架构, 在编译工具链成熟度、指令集优化程度、内存管理机制和系统调用处理等方面都与成熟的 x86 和 ARM 架构存在显著差异, 这些差异会在软硬件交互的多个层面影响算子运行质量, 而传统的单一功能测试无法全面评估这些潜在问题.

为全面评估深度学习算子在 RISC-V 平台上的表现, RIVdoo 构建了涵盖精度、效率、内存和系统这 4 个层面的度量指标体系: 精度层面度量对应 RISC-V 在编译工具链和运行环境方面的特殊性, 通过评估算子输出正确性和执行稳定性来保障评估基础的可靠性; 效率层面度量对应 RISC-V 精简指令集特征, 从指令级微观角度评估算子在基础指令层面的适配质量; 内存层面度量对应 RISC-V 的内存容量限制和简化内存管理机制, 从存储子系统角度评估算子的资源利用效率; 系统层面度量对应 RISC-V 在多核协调和系统调用处理方面的简化特征, 从操作系统级宏观角度评估算子的系统协调能力. 这 4 个层面具有高度互补性和完整性, 覆盖了从纯软件逻辑到软硬件交互的完整评估链条, 相互独立又相互补充, 避免了传统单一功能测试的评估盲区, 这些指标从不同维度刻画算子在 RISC-V 架构下的适配质量, 为后续的差分测试分析提供多元化的量化基础.

2.3.1 精度层面

精度层面度量主要评估算子在 RISC-V 架构上的数值计算正确性表现, 重点关注交叉编译、静态链接等 RISC-V 特有部署方式对计算准确性的潜在影响. RIVdoo 采用算子精度损失来衡量深度学习算子部署到 RISC-V 架构时在精度层面的表现, 通过比较算子在 RISC-V 环境下的输出结果与开发环境基准的一致性进行评估, 采用均方根误差作为精度偏差的度量标准:

$$Loss_i = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\left(O_{(j)}^{\text{RISC-V}} - O_{(j)}^{\text{x86}}\right)^2},$$

其中, $Loss_i$ 表示复杂度层级 i 的部署损失, n 为算子输出张量的元素总数, $O_{(j)}^{\text{RISC-V}}$ 和 $O_{(j)}^{\text{x86}}$ 分别为第 j 个输出元素在 RISC-V 部署平台和开发平台的计算结果, 能够有效捕获算子在跨架构部署过程中的数值精度变化. 在 RISC-V 的交叉编译环境中, 编译器优化过程可能引入计算逻辑偏差, 静态链接过程中数学库函数实现存在细微差异, 而 RISC-V 浮点单元的精度特性也可能导致数值计算偏差. 当算子精度损失超出预期范围时, 通常反映算子实现在 RISC-V 编译或运行环境下存在功能层面的适配问题.

2.3.2 效率层面

效率层面采用 CPU 时间作为核心度量指标, 重点评估算子在 RISC-V 架构上的执行效率表现, 主要关注算子实现的时间性能和稳定性特征. CPU 时间直接反映算子在 RISC-V 平台上的计算效率水平, 该指标以毫秒为单位记录算子执行过程中消耗的 CPU 时间. 算子实现对 RISC-V 指令集特性利用不充分、编译优化策略不当导致的指令级效率损失, 以及算法实现复杂度过高而不适应 RISC-V 资源约束环境都会导致 CPU 时间延长.

2.3.3 内存层面

内存层面采用常驻内存和缺页率作为核心度量指标, 主要评估算子的内存资源管理表现, 重点关注 RISC-V 有限内存环境下的适配质量.

常驻内存用于刻画算子执行过程中的常驻内存峰值特征, 通过监控 RSS 的最大值进行统计, 该指标以 MB 为单位记录算子执行过程中常驻内存的峰值大小. 内存分配策略设计不当导致的资源浪费、临时数据生命周期管理

不合理以及内存碎片化程度严重, 都会增加常驻内存占用. 当常驻内存超过理论内存需求的合理范围时, 通常表明算子在 RISC-V 内存管理方面存在显著的优化需求.

缺页率用于刻画算子内存访问模式与 RISC-V 内存管理机制的匹配程度, 本文具体采用主缺页频率进行度量, 其计算公式为:

$$R^{\text{major}} = \frac{MajorFaults}{Time},$$

其中, *MajorFaults* 表示算子执行过程中的主缺页次数, *Time* 表示算子执行时间 (ms). 数据访问模式的时空局部性不足、缓存友好性优化程度不够、内存预取策略与 RISC-V 内存层级结构匹配程度较低, 都会提高缺页率.

2.3.4 系统层面

系统层面采用用户态 CPU 占比作为核心度量指标, 主要评估算子的 CPU 资源利用和系统协调能力表现, 重点关注 RISC-V 系统资源管理机制的协调效率.

用户态 CPU 占比反映算子计算过程中用户态计算的纯净程度, 计算公式为:

$$P^{\text{user}} = \frac{CPU^{\text{user}}}{CPU^{\text{Total}}},$$

其中, CPU^{user} 表示算子执行过程中的用户态 CPU 时间, CPU^{Total} 表示用户态和内核态 CPU 时间的总和. 算子实现中包含的过量系统调用操作、I/O 操作阻塞导致的 CPU 资源空闲, 以及内存访问模式不当引起的处理器等待时间过长都会降低用户态占比, 导致系统开销增大.

2.4 基于复杂度放大系数的差分测试

基于前述多层面度量指标体系, RIVdoo 提出基于复杂度放大系数的差分测试策略, 通过分析算子在不同复杂度层级间各项度量指标的放大系数模式来识别实现问题. 在面向基于 RISC-V 架构的平台进行部署时, 不同算子库实现的同个算子在面临复杂度增加时, 各项度量指标应表现出相对一致的变化模式, 其复杂度放大系数分布应呈现相似的统计特征. 当某算子实现的放大系数模式显著偏离其他实现的统计分布时, 表明该实现可能存在 RISC-V 架构适配缺陷或质量问题.

2.4.1 放大系数计算

复杂度放大系数的核心思想在于量化算子实现在不同工作负载下的性能变化特征. 由于 RISC-V 平台的资源约束特性, 算子在复杂度增加时往往表现出非线性的性能退化模式, 这种退化程度的差异正是识别实现问题的关键信号. 通过建立标准化的放大系数计算框架, 可以将不同算子库在复杂度敏感性方面的差异转化为可比较的量化指标. 对于每个待测算子实现, 分别在高、中、低这 3 个复杂度层级下执行测试, 收集各项度量指标在不同复杂度层级的数值. 针对任一度量指标 *Metric*, 构建多层级复杂度放大系数函数:

$$R_{h/l} = \frac{Metric_{\text{high}}}{Metric_{\text{low}}},\ R_{m/l} = \frac{Metric_{\text{medium}}}{Metric_{\text{low}}},\ R_{h/m} = \frac{Metric_{\text{high}}}{Metric_{\text{medium}}}.$$

这 3 个比值函数从不同维度揭示算子实现在 RISC-V 平台上的性能特征. $R_{h/l}$ 反映算子从基础负载跃升至极限负载时的性能退化倍数, 该放大系数异常增大通常表明算子实现在面临 RISC-V 平台的综合资源压力时缺乏有效的负载适应机制. $R_{m/l}$ 体现算子在不对称参数配置下相对于基础配置的性能变化, 该放大系数的异常波动往往指向算子在特定资源维度上的适配问题, 如内存访问模式优化不当或向量化计算效率低下. $R_{h/m}$ 揭示算子从中等负载向极限负载过渡时的性能变化梯度, 该放大系数的异常特征可以帮助识别算子实现在高负载区间是否存在性能突变或稳定性问题. 通过联合分析这 3 个放大系数的数值特征和变化模式, 可以构建算子实现在 RISC-V 平台上的完整复杂度敏感性轮廓, 为异常检测和问题定位提供丰富的分析维度.

2.4.2 异常损失检测

为客观刻画算子在不同复杂度配置下的性能行为差异, 并识别可能需要进一步关注的实现情况, RIVdoo 采用基于四分位距 (*IQR*) 的离群点检测方法. 收集所有算子在不同库实现上的放大系数 R (包括 $R_{h/l}$、$R_{m/l}$ 和 $R_{h/m}$), 计

算该分布的第 1 四分位数 Q_1 (25% 分位点)、第 3 四分位数 Q_3 (75% 分位点) 和四分位距 $IQR = Q_3 - Q_1$, 根据四分位距法, 定义离群点判定阈值为:

$$\begin{cases} \text{下界}: L = Q_1 - 1.5\times IQR \\ \text{上界}: U = Q_3 + 1.5\times IQR \end{cases}.$$

若某算子实现的放大系数 R 满足 $R < L$ 或 $R > U$, 则标记为离群点, 作为潜在异常实现进一步分析. 这一方法的优势在于不依赖于放大系数分布的具体形态 (如是否服从正态分布), 由于放大系数为比值变量, 其分布往往呈现右偏态且可能包含极端离群值, 传统的基于正态假设的统计方法 (如 Z-score) 在此场景下可能适用性有限. 相比之下, 基于四分位距的离群点检测对分布偏态和极端值具有良好的鲁棒性, 更适合作为一种用于初步筛选异常性能行为的统计工具, 而 1.5 倍 IQR 的选择基于统计学经验法则, 能够在不过度依赖参数假设的前提下, 对显著偏离整体行为模式的实现进行标记, 从而为后续结合源码分析和体系结构特征的定性分析提供线索支持.

3 研究问题和实验设置

3.1 研究问题

本文围绕 RIVdoo 方法在 RISC-V 架构深度学习算子测试中的有效性展开研究, 设计 3 个递进式研究问题以系统性验证 RIVdoo 各核心组件的技术有效性和实用价值.

研究问题 1: RIVdoo 的复杂度分组策略能否有效识别算子在 RISC-V 架构上的性能分化特征?

该研究问题旨在验证 RIVdoo 提出的基于参数空间区间约束的复杂度分组方法是否能够有效揭示深度学习算子在 RISC-V 平台上的性能分化模式. RISC-V 架构的资源约束特性使得算子性能对参数配置变化极其敏感, 传统固定参数测试难以捕获这种敏感性的全貌. RIVdoo 通过将算子输入参数空间划分为高、中、低这 3 个复杂度层级, 预期能够识别 RISC-V 平台在不同计算强度下的性能瓶颈和资源利用特征. 本研究问题通过在各复杂度层级下收集算子的精度损失等关键指标, 分析复杂度增长对这些指标的影响模式, 验证分组策略在性能分化识别方面的有效性. 此外, 本研究还将 RIVdoo 与现有的算子测试方法进行对比, 评估不同方法在 RISC-V 静态编译环境下生成测试用例的参数空间覆盖度和多样性.

研究问题 2: RIVdoo 采用的基于复杂度放大系数的差分测试方法能否有效识别算子在 RISC-V 架构上的性能适配问题?

本研究问题聚焦于验证 RIVdoo 提出的基于复杂度放大系数的差分测试机制在 RISC-V 算子性能适配评估中的有效性. 现有差分测试方法通常依赖绝对指标对比, 难以揭示算子在不同工作负载下的复杂度敏感性差异, 而 RIVdoo 提出的复杂度放大系数方法通过分析算子在不同复杂度层级间多维度指标的变化模式, 为识别异常实现模式提供了新的分析视角. 具体而言, 本研究问题将同一类型算子的多个不同库实现部署到 RISC-V 平台, 在高、中、低不同复杂度层级下收集 CPU 时间、常驻内存、缺页率和用户态 CPU 占比指标, 计算各实现在复杂度层级之间的放大系数, 并通过统计分析比较不同实现的分布特征与离群情况, 从而验证该方法在揭示复杂度适应性差异、发现潜在异常实现以及识别系统层面隐蔽问题方面的有效性.

研究问题 3: RIVdoo 能够发现哪些类型的深度学习算子在 RISC-V 架构上存在部署问题?

该研究问题在前两个研究问题的基础之上, 进一步聚焦于评估 RIVdoo 的实际应用效果和问题发现能力. 通过运用经过验证的 RIVdoo 完整方法论, 对主流深度学习算子进行系统性的 RISC-V 适配质量评估, 深入识别哪些算子类型在该架构上表现良好, 哪些存在显著的适配问题, 以及这些问题的具体表现形式和根本原因. 该研究问题不仅验证 RIVdoo 作为完整方法论的综合应用价值, 还为 RISC-V 深度学习生态的算子优化策略和技术选择决策提供有价值的实证指导.

3.2 实验设置

为了全面验证 RIVdoo 方法的有效性, 特别是其基于损失比值的差分测试机制, 本研究构建了 3 层次的实验环境体系, 分别承担不同的技术验证功能.

为了回答上述研究问题, 本研究在开发环境中完成算子的交叉编译等预处理工作. 该环境采用配置 AMD Ryzen™ 5 5600H 处理器的 Ubuntu 24.04.2 LTS 系统, 内核版本为 6.14.0-29-generic. 开发环境安装 Python 3.11 作为主要开发语言, TensorFlow Lite 2.18.0 用于模型导出和 TFLite 算子库的构建, Apache TVM 0.19.0 用于 TVM 算子库的代码生成和优化. 该环境覆盖了 RISC-V 部署环境中的主流算子库[42], 确保了所有测试算子在部署前的标准化处理, 为后续的跨平台对比提供一致的起始状态.

本研究选用 VisionFive 2 开发板作为 RISC-V 部署环境和目标测试平台. 该开发板搭载 JH7110 处理器 (四核 64 位 RISC-V CPU, 主频 1.5 GHz), 运行 Linux 5.15.0-starfive 操作系统. 该配置代表了当前 RISC-V 生态中的主流硬件水平, 具备完整的深度学习算子部署和执行能力. 在该环境中收集的性能数据反映了算子在真实 RISC-V 硬件约束下的实际表现, 是 RIVdoo 方法评估的核心数据源.

本研究选择了 47 个主流的深度学习算子进行系统性评估, 具体为: (1) Abs, (2) Add, (3) AvgPool2D (TVM 兼容表示), (4) Cast, (5) Ceil, (6) Concatenation, (7) Conv2D (TVM 兼容表示), (8) Cos, (9) DepthwiseConv2D, (10) Div, (11) ELU, (12) ExpandDims, (13) Exp, (14) Floor, (15) FullyConnected, (16) HardSwish, (17) L2Normalization, (18) LeakyReLU, (19) Logistic, (20) LogSoftmax, (21) Log, (22) Maximum, (23) MaxPool2D, (24) Mean, (25) Minimum, (26) Mul, (27) Neg, (28) ReLU6, (29) ReLU, (30) Round, (31) Rsqrt, (32) Sin, (33) Softmax, (34) Sqrt, (35) Squared-Difference, (36) Square, (37) Sub, (38) Sum, (39) Tanh, (40) AVERAGE_POOL_2D (TFLite 兼容表示), (41) BatchNorm, (42) BiasAdd, (43) Conv2DLite, (44) Dense, (45) MatMul, (46) SoftmaxLite, (47) CONV_2D (TFLite 兼容表示). 这些算子覆盖了深度学习模型的核心计算组件, 具备不同的计算特征和资源需求模式, 能够全面验证 RIVdoo 方法在多样化算子类型上的适用性. 实验覆盖了计算密集型 (Conv2D 等)、内存密集型 (BatchNorm 等) 和轻量型 (ReLU 等) 这 3 种主要计算模式. 这种分类设计能够全面评估 RISC-V 架构在面对不同类型计算负载时的适配表现, 识别平台在各种计算模式下的性能特征和约束限制. 每个算子都同时测试 TVM 和 TFLite 两个主流深度学习算子库的实现, 并在 RISC-V 向量扩展指令 (RVV) 启用/禁用两种配置下分别进行评估, 以覆盖 RISC-V 架构的特殊场景, 为 RIVdoo 的差分测试方法提供充足的库间对比和架构特性对比数据基础. 通过比较不同算子库在相同算子上的实现质量以及 RVV 对算子性能的影响, 能够验证损失比值差分测试在识别异常实现方面的准确性, 证明 RIVdoo 在 RISC-V 复杂应用场景中的适用性.

此外, 我们选择两个典型的算子测试方法作为对比基线: (1) Predoo[8], 一种基于变异和运行时反馈的测试生成方法; (2) DPM[21], 一种基于动态程序监控和反馈引导的参数生成方法. 由于这两个方法都依赖运行时反馈进行测试输入的迭代搜索, 而 RISC-V 平台采用静态编译模式, 即每次参数变化都需要重新编译整个算子库并进行静态链接, 导致无法获得 RISC-V 开发板的即时反馈. 因此, 我们使用 Predoo 和 DPM 在其原有平台 (x86) 上搜索生成的测试输入, 将这些输入迁移至 RISC-V 平台进行测试, 以评估不同方法生成的测试用例在 RISC-V 平台上的有效性.

4 实验结果

4.1 研究问题 1

图 3–图 5 展示了 47 个深度学习算子在 RISC-V 平台上使用 TFLite 和 TVM 两个算子库实现时的多维度性能表现, 分别对应高、中、低这 3 个复杂度层级. 热力图横轴为算子编号 (1–47), 纵轴为 5 个关键性能指标: 常驻内存、缺页率、用户态 CPU 占比、CPU 时间和精度损失, 颜色深浅表示 TFLite 与 TVM 的性能比值 (TFLite/TVM), 蓝色表示 TFLite 性能更优 (比值<1), 红色表示 TVM 性能更优 (比值>1). 实验结果揭示了两个算子库的显著性能差异, CPU 时间维度显示 TFLite 具有明显的速度优势, 在 3 个复杂度层级下, 几乎所有算子的 CPU 时间比值均呈现深蓝色, 表明 TFLite 执行时间比 TVM 短 30%–50%. 算子 42 (BiasAdd) 的比值在高、中、低这 3 个层级分别约为 0.6、0.6、0.7, 算子 7 (Conv2D) 的比值约为 0.5、0.6、0.7, 显示出该趋势在不同数据规模下具有一致性. 值得注意的是, 轻量级算子如算子 29 (ReLU)、算子 1 (Abs) 等在 CPU 时间维度的比值更接近 0.4–0.5, 表明在简单计算场景下 TFLite 的实现能够获得更高的执行效率.

用户态 CPU 占比和缺页率维度揭示了 TFLite 的“空间换时间”优化策略. 用户态 CPU 占比呈现显著红色, 比

值普遍在 1.2–2.0 之间, 表明 TFLite 在算子执行过程中更多依赖用户态计算而较少触发系统调用. 特别是算子 15 (FullyConnected)、算子 33 (Softmax) 在高复杂度下的用户态 CPU 占比接近 100%, 而 TVM 在这些算子上出现明显的系统调用开销. 缺页率几乎全部呈现深蓝色, 算子 7 (Conv2D)、算子 42 (BiasAdd) 等在高复杂度下的缺页率比值接近 0.5, 算子 40 (AVERAGE_POOL_2D) 和算子 41 (BatchNorm) 的比值甚至低于 0.4, 表明在这些场景中 TFLite 的内存访问行为具有更好的局部性特征, 而 TVM 的动态内存分配策略在数据规模超过缓存容量后产生严重的缓存失效. 这归因于 TFLite 针对端侧小资源场景的专门优化: 预先将优化后的算子内核和中间结果加载到内存中, 避免运行时的动态编译和频繁的内存页调度, 从而维持接近 100% 的用户态执行并大幅减少缺页中断. 值得注意的是, 这类复杂度相关的差异仅在跨层级测试中才能被充分观测, 若仅在低复杂度场景下进行测试, 则难以体现其影响.

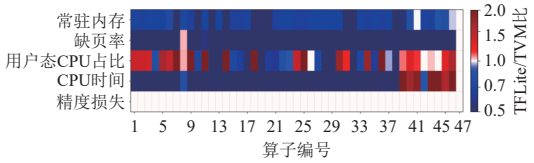

图 3 深度学习算子在 RISC-V 平台上的多维度性能表现 (高复杂度)

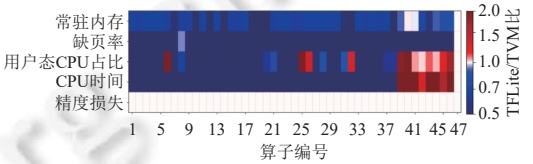

图 4 深度学习算子在 RISC-V 平台上的多维度性能表现 (中复杂度)

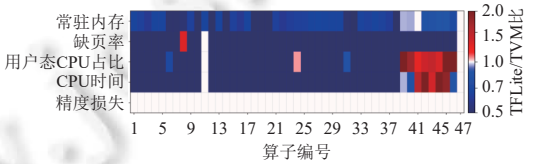

图 5 深度学习算子在 RISC-V 平台上的多维度性能表现 (低复杂度)

系统层面的用户态 CPU 占比数据揭示了库间实现机制的根本差异: TFLite 在绝大多数情况下维持接近 100% 的用户态占比, 而 TVM 在低复杂度场景下出现显著的系统开销 (如 Conv2D 低复杂度时系统开销高达 26.7%). 然而, TFLite 的性能优势以更高的内存占用为代价. 常驻内存维度显示, TFLite 在几乎所有算子和所有复杂度层级下的内存占用比值均小于 1; 部分算子 (如算子 7 (Conv2D)) 在高复杂度下的比值几乎只有一半, 表明 TFLite 在计算密集型算子上需要额外 60%–100% 的内存开销. 相比之下, 轻量级算子 (如算子 29 (ReLU) 、算子 11 (ELU)) 的常驻内存比值则略高, 内存代价相对较小.

综合上述结果可以看出, RIVdoo 通过多维度指标和复杂度分组分析, 系统性刻画了 TFLite 和 TVM 在 RISC-V 平台上的性能权衡关系: TFLite 在计算密集型算子 (Conv2D、MatMul、FullyConnected) 上以 60%–100% 内存开销换取 30%–50% 速度提升, 在轻量级算子 (ReLU、Abs) 上以 10%–30% 内存开销换取 50%–60% 速度提升, 为开发者在性能与资源约束之间的权衡提供了全面依据. 其次, RIVdoo 的复杂度分组验证了优化策略在不同数据规模下的一致性, 表明 TFLite 的端侧优化在 RISC-V 平台上具有良好的可扩展性. 第三, RIVdoo 有助于明确两种算子库在不同资源约束条件下的适用范围: 内存充足时选择 TFLite, 内存受限时选择 TVM, 这种差异化的性能洞察为 RISC-V 深度学习应用的算子库选择提供了精确指导. 因此, RIVdoo 提供的分析结果可作为 RISC-V 平台上算子库选择和优化决策的参考依据.

图 6–图 8 展示了 47 个深度学习算子在 RISC-V 向量扩展指令 (RVV) 支持下的多维度性能表现, 分别对应高、中、低这 3 个复杂度层级. 热力图横轴为算子编号 (1–47), 纵轴为 5 个关键性能指标: 常驻内存、缺页率、用户态 CPU 占比、CPU 时间和精度损失, 颜色深浅表示 RVV 启用与禁用配置下的性能比值 (rw/rv), 蓝色表示 RVV 带来性能提升 (比值<1), 红色表示 RVV 导致性能下降 (比值>1).

实验结果揭示了 RVV 在不同算子类型和复杂度配置下呈现出显著差异化的性能行为. 计算密集型算子在高

复杂度下显著受益于 RVV: 算子 45 (MatMul) 的 CPU 时间比值在高复杂度下明显低于中、低复杂度, 表明其在大规模计算场景中能够更充分地受益于向量化执行. 同样, 算子 7 (Conv2D) 和算子 41 (BatchNorm) 在高复杂度配置下的 CPU 时间、用户态 CPU 占比及缺页率比值整体低于中、低复杂度, 显示出在较大数据规模下更稳定的性能表现. 相比之下, 部分算子在启用 RVV 后未表现出一致的性能收益. 例如, 算子 24 (Mean) 和算子 33 (Softmax) 在中复杂度下的缺页率比值明显升高, 表明归约类算子在该配置下, 其内存访问模式可能对缓存局部性产生一定影响, 从而增加内存访问开销.

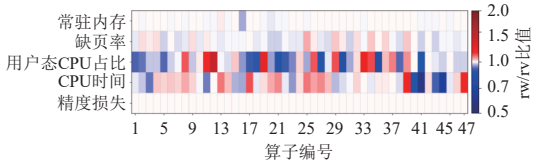

图 6 深度学习算子在向量扩展指令支持下的多维度性能表现 (高复杂度)

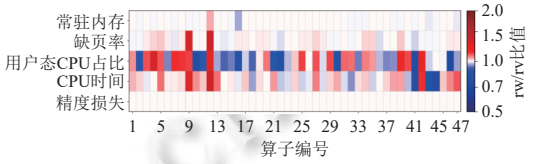

图 7 深度学习算子在向量扩展指令支持下的多维度性能表现 (中复杂度)

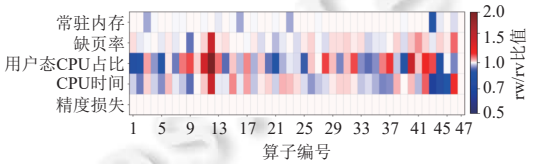

图 8 深度学习算子在向量扩展指令支持下的多维度性能表现 (低复杂度)

进一步分析发现, RVV 带来的性能变化与数据规模具有显著相关性. 算子 4 (Cast) 的 CPU 时间比值从高复杂度到中、低复杂度呈现逐步回升趋势, 表明在小规模数据场景中, 向量化的启动与配置开销在总执行时间中占据更高比例, 从而削弱了向量化带来的收益. 在低复杂度下, 算子 29 (ReLU) 等激活函数的 CPU 时间比值接近 1.0, 但在高复杂度下有明显下降, 同样验证了这一规律. 此外, 算子 6 (Concatenation) 和算子 12 (ExpandDims) 等张量操作在所有层级的 CPU 时间比值变化都不大, 但缺页率比值从低复杂度到高复杂度有明显上升, 表明这类算子的 RVV 优化主要体现在内存布局效率而非计算加速.

基于上述实验观察, 可以看出 RIVdoo 在刻画 RVV 适用范围方面的分析能力. 第一, RIVdoo 的复杂度分组策略使得 RVV 在不同负载区间下的性能行为得以系统呈现. 通过高、中、低这 3 个层级的系统测试, RIVdoo 发现计算密集型算子 (MatMul、Conv2D) 在高复杂度下 RVV 收益显著, 而部分算子 (Div、Mean) 反而性能劣化, 且这种差异在不同复杂度层级呈现不同模式. 这些结果为 RISC-V 平台上 RVV 的选择性启用和参数配置提供了实验依据, 而非简单的“启用或禁用”结论. 第二, RIVdoo 的多维度监控揭示了 RVV 优化的系统级副作用. 除 CPU 时间外, 缺页率和用户态 CPU 占比等指标暴露了向量化可能带来的缓存失效和系统开销增加问题, 帮助开发者避免“看似加速实则引入新瓶颈”的优化陷阱. 第三, 47 个算子的大规模覆盖确保了结论的可靠性, 成功识别出 Div、Mean 等 RVV 适配薄弱的特殊案例, 为后续 RVV 优化策略的改进和验证提供了参考样本.

现有的深度学习算子测试方法主要采用固定参数配置或单一场景验证, 无法发现这些复杂度相关的性能分化问题. 值得注意的是, 图 3–图 8 中的精度损失维度在所有算子和所有复杂度层级下均呈现白色 (比值≈1),表明 TFLite 和 TVM 的精度误差保持在相同的数量级 (10E–5 到 10E–6), 复杂度变化对算子输出精度的影响微乎其微. 这一结果说明, 上述性能差异主要体现为实现层面的架构适配行为, 而非数值计算正确性问题. 更重要的是, 现有方法聚焦于算子的输出正确性而忽略效率、内存和系统层面的适配质量, 无法全面反映算子在资源受限的 RISC-V 环境下的真实部署表现. 本研究发现的性能分化问题 (如 TVM 的缺页率异常、TFLite 的内存占用翻倍) 均未伴随精度损失, 说明这些问题属于实现层面的架构适配行为而非算法层面的正确性错误, 传统基于输出验证的测试方法难以识别此类隐蔽的质量问题.

表 3 展示了 3 种方法生成的测试输入在复杂度区间的分布情况. 实验结果显示, Predoo 和 DPM 生成的测试输入 100% 集中在低复杂度区间, 完全未覆盖中高复杂度场景. 这是因为两种方法都依赖运行时反馈引导测试生成, 而它们在原始 x86 平台上收敛至低复杂度区间, 即 x86 平台资源充足, 算子在常规负载下已充分优化, 问题主要集中在边界条件等低复杂度场景. 然而, RISC-V 平台的资源约束特性 (如 2 MB L2 缓存、12.8 GB/s 内存带宽) 使得算子在中高复杂度配置下更容易呈现出与平台资源相关的行为差异. 在本研究的实验结果中, 中高复杂度区间暴露出多种与资源使用模式相关的行为特征, 包括 TVM 的碎片化内存布局、TFLite 的过度内存占用等问题, 这些问题在 x86 平台可能因硬件资源规模较大而不易被触发或观察到. Predoo 和 DPM 因完全未覆盖中高复杂度场景, 其生成的测试输入在识别此类与资源约束相关的行为特征方面存在局限性. 相比之下, RIVdoo 基于硬件资源阈值推导复杂度区间, 实现了 3 个场景的均衡覆盖 (33.3%、33.3%、33.3%), 从而能够系统性地覆盖不同计算负载条件下的算子执行行为. 上述结果表明, 跨复杂度层级的测试策略在资源受限架构环境下, 对于全面刻画算子行为特征具有重要意义.

表 3 不同方法生成的测试输入的分布 (%)

方法	低复杂度	中复杂度	高复杂度
RIVdoo	33.3	33.3	33.3
Predoo	100	0	0
DPM	100	0	0

4.2 研究问题 2

基于第 4.1 节的实验结果, 我们识别出 RISC-V 平台上算子的显著性能分化现象和库间实现差异. 在此基础上, 我们进一步分析算子的复杂度敏感性特征, 通过构建复杂度放大系数来量化算子在不同工作负载间的性能变化模式. 这种分析能够获得传统绝对性能对比无法提供的深层次洞察: 区分算子的本质计算特性与实现质量问题, 识别复杂度适应性问题, 以及发现隐蔽的系统层面异常.

图 9 展示了 4 个关键性能指标 (CPU 时间、常驻内存、缺页率、用户态 CPU 占比) 在 3 种复杂度放大系数 ($R_{h/l}$、$R_{m/l}$ 和 $R_{h/m}$) 下的箱形图分布. 通过箱形图可以清晰观察到各指标的中位数、四分位距以及离群点分布情况, 为基于 Tukey 法则的异常检测提供直观的统计依据.

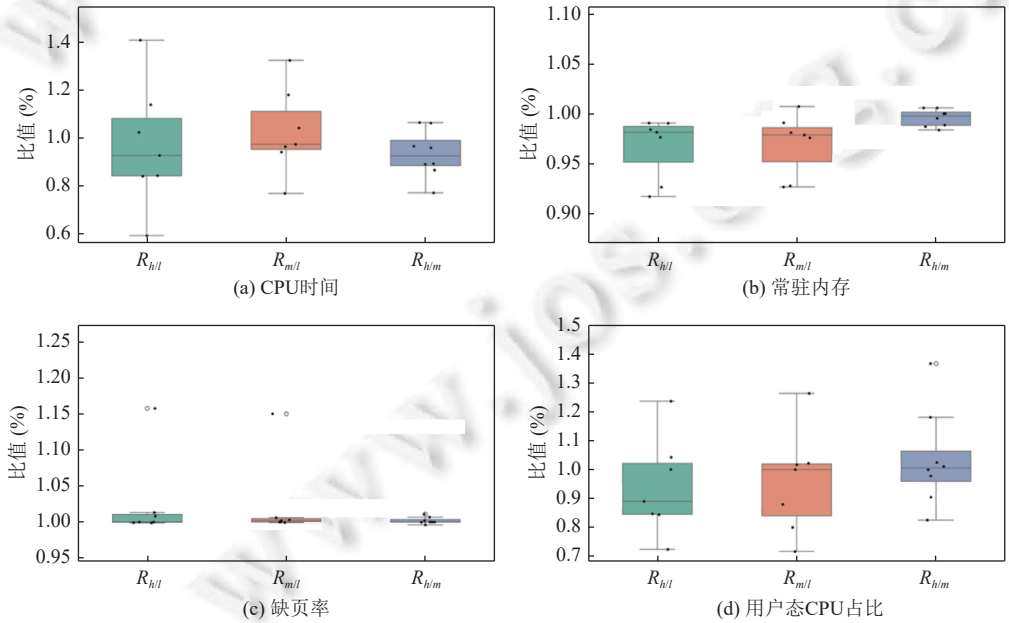

图 9 各评估维度的复杂度放大系数

CPU 时间层面 (图 9(a)) 呈现明显的右偏分布特征. $R_{h/l}$ 的中位数约为 0.95, 箱体范围为 [0.85, 1.08], 但存在多个上侧离群点 (最大值约 1.4), 基于 1.5 倍 *IQR* 法则 (上界 $U=Q_3+1.5\times IQR\approx1.43$), 这些离群点被标记为统计意义上的异常放大系数样本, 表明部分算子在高复杂度下的 CPU 时间增长显著偏离正常模式. $R_{m/l}$ 和 $R_{h/m}$ 的分布相对紧凑, 中位数分别约 1.0 和 0.92, 离群点数量较少, 表明多数算子在复杂度逐级提升过程中保持了相对稳定的性能变化特征. 为进一步验证不同复杂度层级间性能差异的统计显著性, 我们对 CPU 时间指标进行了单因素方差分析 (One-way ANOVA). 结果表明, 3 个复杂度层级 (低、中、高) 间的性能差异具有极高的统计显著性 (F=127.34, p<0.001), 说明基于复杂度分组的测试能够在 RISC-V 平台上有效触发并区分不同的算子执行模式.

常驻内存层面 (图 9(b)) 展现出极高的一致性, 3 种放大系数的中位数均接近 1.0, 四分位距极小 (约 0.03–0.05), 表明无论复杂度如何, 不同算子实现在内存占用的敏感性方面高度一致. 这种集中分布使得常驻内存的离群点检测阈值范围极窄 (上界 $U\approx1.02$), 即使微小的偏离也能被识别为异常. 方差分析同样显示复杂度层级对常驻内存有显著影响 (F=89.21, p<0.001) 表明该指标在刻画复杂度敏感行为方面具有统计有效性.

缺页率指标 (图 9(c)) 同样呈现高度集中的分布, 3 种放大系数的中位数均为 1.0, 箱体几乎退化为一条线. 尽管分布集中, $R_{h/l}$、$R_{m/l}$ 仍各存在一个上侧离群点 (约 1.15), 基于 *IQR* 法则 (上界 $U\approx1.03$) 被识别为显著偏离整体分布的异常样本, 对应的算子在内存访问模式上存在内存访问行为的异常放大趋势, 导致缓存失效率随复杂度增加而异常上升. 综合 4 个指标的箱形图分析可以看出: CPU 时间和用户态 CPU 占比是识别异常实现的最有效指标, 其离群点数量最多且分布范围最广; 常驻内存和缺页率的分布高度集中, 说明 RISC-V 平台上不同算子库在内存管理策略方面具有较强的一致性.

用户态 CPU 占比指标 (图 9(d)) 的箱体范围较大 (约 0.2–0.3), 表明不同算子实现在系统开销方面存在一定差异. $R_{h/m}$ 存在多个上侧离群点 (最大值约 1.37, 上界 $U\approx1.28$), 对应的算子从中到高复杂度时系统开销异常增加, 这与 TVM 在高负载场景下引入的额外运行时管理与调度开销具有相关性.

我们进一步对比 TVM 和 TFLite 的源码, 来对检测到的异常的产生原因进行分析. TVM 采用基于 LLVM 的动态代码生成策略, 其 *tir*::*transform*::*StorageRewrite* 模块在处理复杂度增加时产生碎片化内存布局, TVM 的 *ScheduleOps*::*ComputeStorageScope*() 函数生成的内存访问跨度随复杂度呈指数级增长, 在 RISC-V 的简化缓存控制器上表现对存储层次结构更加敏感. 相反, TFLite 采用静态预分配, 实现连续内存布局, *tflite*::*graph_utils*::*CalculateOptimalAllocation*() 在图优化阶段确定内存排布, 确保复杂度增加时仍保持线性扩展的内存访问模式. 汇编代码分析表明, TVM 生成的内存访问指令存在非对齐访问, 产生跨 cache line 的 ld 指令序列; 而 TFLite 保持规律的步长访问模式, 与 RISC-V cache line 大小良好对齐. 上述源码级差异为统计分析中识别出的异常放大系数提供了合理的实现层解释, 表明 RIVdoo 基于 *IQR* 的异常检测结果能够与底层实现行为建立一致关联, 从而支持其在揭示 RISC-V 平台相关行为差异方面的有效性.

4.3 研究问题 3

第 4.1 和 4.2 节揭露了深度学习算子执行时间、常驻内存峰值、主缺页率和 CPU 占比等方面可能呈现出与平台特性相关的性能差异与资源敏感行为, 为了进一步聚焦于研究问题 3, 我们在上述关键性能指标的基础上, 对 47 个算子的实验结果进行了系统化分析, 以识别其在 RISC-V 架构上的具体适配表现, 以刻画其在 RISC-V 架构下的具体行为表现与适配特征. 通过该分析, 我们能够区分算子在资源受限环境下的不同表现模式, 并进一步揭示这些行为差异在效率、内存与系统层面的具体体现形式, 从而评估 RIVdoo 在识别平台相关行为特征方面的能力.

在效率层面, RIVdoo 的结果揭示了 TVM 与 TFLite 的不同算子在 RISC-V 平台上的 CPU 时间存在显著差异, 热力图分析显示 TFLite 在几乎所有 47 个算子上的 CPU 时间均比 TVM 短 30%–50%, 这一普遍性优势源于 TFLite 针对端侧部署的预编译策略. 这一差异与两者的源码实现紧密相关. TFLite 在 fully_connected.cc 和 kernel_utils.h 中采用固定的循环嵌套结构, 在编译期完成了所有算子内核的优化和代码生成, 避免了运行时的编译开销和动态调度成本. 这种策略在 RISC-V 平台上尤为有效, 因为 RISC-V 的静态编译特性使得运行时优化的收益有限, 而

TFLite 的预优化内核能够直接在目标平台上高效执行. 而 TVM 的算子内核由张量表达式自动生成, 并通过 schedule_auto_inline 与 loop_tiling 等优化 pass 实现循环分块和寄存器缓存, 但这种依赖运行时调度与反馈的动态优化机制呈现出一定的局限性: 第一, TVM 的 AutoTVM 调优过程依赖于运行时反馈, 而 RISC-V 的静态编译模式无法提供即时反馈, 导致生成的调度策略往往基于默认启发式规则而非平台特性; 第二, TVM 的 Graph Executor 在每次算子调用时需要进行上下文切换和内存重分配, 这在算子粒度较细或低复杂度场景下产生的开销占比可达总执行时间的 20%–30%, 从而抵消了编译优化的收益.

然而, TFLite 的预编译策略在内存管理上呈现出明显的资源权衡特征. 热力图显示 TFLite 的常驻内存占用是 TVM 的 1.5–2 倍, 这是因为 TFLite 为实现预编译优化, 在图优化阶段 (*tflite*::*graph_utils*::*CalculateOptimalAllocation*()) 将所有可能的中间结果和临时缓冲区提前分配并常驻内存. 这种“空间换时间”策略在算子执行时避免了动态分配开销, 但在高复杂度场景下会导致内存峰值激增. 相比之下, TVM 的 Relay 到 TIR lowering 过程为每个计算阶段生成独立的中间张量, 并通过 StorageRewrite pass 进行按需分配与回收, 在单算子测试中虽然增加了动态分配的系统调用开销, 但有效控制了内存峰值. 这一差异揭示了在 RISC-V 这种内存受限平台上, 预编译优化与内存效率之间呈现出明显的权衡关系: TFLite 牺牲内存效率以获取执行速度, 适合内存充足的场景; TVM 牺牲部分执行效率以节省内存, 更适合极端资源受限的嵌入式场景.

此外, RIVdoo 还发现 TVM 的缺页率显著高于 TFLite (比值约 0.4–0.5). 这一差异源于两者内存访问模式的不同. TFLite 的实现基于连续内存块, 张量排布规则固定, 从而具备良好的空间局部性和页表友好性. 而 TVM 通过 StorageRewrite 与 DoubleBuffer 等 pass 引入动态张量切分和缓存调度, 这种动态调度策略在 x86 等拥有大容量 TLB (通常 512–1 024 项) 和多级缓存 (L3 可达数十 MB) 的平台上能够有效提升数据复用率, 但在 RISC-V 平台上对存储层次结构更为敏感. 具体而言, 当前主流 RISC-V 实现的 TLB 容量通常仅为 32–64 项, L2 缓存不超过 2 MB, 这使得 TVM 的 StorageRewrite pass 生成的碎片化内存布局无法被有效缓存. 箱形图中缺页率的离群点 (比值约 1.15) 恰好对应于 TVM 在 Softmax、Mean 等归约类算子上的实现, 这些算子需要遍历大规模特征维度, TVM 的 *ScheduleOps*::*ComputeStorageScope*() 函数为每个归约维度分配独立的临时缓冲区, 导致内存访问跨度随复杂度呈指数级增长. 在 RISC-V 的简化缓存控制器上, 这种访问模式频繁触发 TLB miss 和 cache miss, 导致缺页率异常上升. 这一发现表明, TVM 的存储调度策略在当前 RISC-V 平台条件下对资源约束较为敏感, 其在此类平台上的行为特征有待进一步分析与优化, 例如通过合并临时缓冲区、限制张量切分粒度等方式减少页表压力.

在系统层面, TVM 的 Conv2D 算子的用户态占比普遍高于 TFLite, 其在高复杂度配置下达到 73.3%, 而 TFLite 为 59.2%. 这一现象反映出 TVM 内核在算子执行过程中更多依赖用户态计算逻辑, 例如在 conv2d_nchw_spatial_pack.cc 中通过循环展开与张量重排实现卷积操作, 从而增加了用户态 CPU 时间. 而 TFLite 在 optimized_ops.h 中直接调用 NEON/Reference 内核, 并通过轻量化的系统调用完成内存访问与调度, 因而表现为更高的系统开销比例. 相对地, Dense 算子则呈现相反趋势: TVM 的系统调用开销在低复杂度下达到 30.2%, 显著高于 TFLite 的 15.6%. 这一问题可追溯至 TVM 的动态内存分配逻辑, 特别是在 runtime/module.cc 中频繁触发的小块分配与释放操作, 造成系统调用占比较高. 而 TFLite 的 ArenaPlanner 则通过统一的内存池管理机制减少了分配次数, 在一定程度上减少了系统调用次数.

此外, RVV 实验揭示了向量化优化在不同算子类型和复杂度层级下的差异化效果. 计算密集型算子 (Conv2D、MatMul) 在高复杂度下从 RVV 获得 30%–50% 的性能提升, 表明向量化指令在大规模并行计算中能够有效利用 RISC-V 的 SIMD 能力. 然而, 箱形图中 CPU 时间的离群点揭示了 RVV 优化的局限性: 部分算子 (如 Div、取模运算) 在启用 RVV 后性能反而下降, 用户态 CPU 占比异常增加 (比值>1.2). 源码分析表明, 这些算子在当前 RISC-V 向量扩展规范 (RVV 1.0) 中缺乏专门的硬件加速指令, 编译器被迫通过软件模拟实现向量化除法和取模操作, 引入了额外的循环展开和条件分支, 导致指令数量激增和分支预测失效. 更关键的是, RVV 向量化在低复杂度场景下的收益被启动开销完全抵消: 向量寄存器的初始化、向量长度配置以及标量-向量数据传输在小规模数据下占据了总执行时间的 30%–40%, 使得向量化反而不如标量实现高效. 此外, 归约类算子 (Mean、Softmax) 在中复杂度下因 RVV 导致缓存失效率上升 (缺页率比值约 1.5–1.8), 这是因为向量化加载指令 (vle) 的内存访问粒度 (通常

为 128 位或 256 位) 破坏了原有的缓存行对齐, 在中等数据规模下反而增加了跨缓存行访问次数. 上述结果表明, RVV 的性能收益与算子类型和数据规模高度相关, 更适合在特定条件下进行选择性应用.

综上所述, RIVdoo 能够系统性地揭示并量化 RISC-V 平台上算子在效率、呈现出的平台相关行为差异与资源敏感特征. 具体而言, RIVdoo 在实验中观察到并量化了不同实现的 MatMul 与 Dense 算子在 CPU 时间上存在的显著差异, Softmax 与 BatchNorm 算子在常驻内存与缺页率指标上呈现出特定配置下、明显偏离常规模式的行为特征, 以及 TVM 在低复杂度场景中出现系统开销占比显著上升的现象. TFLite 的预编译策略在 RISC-V 静态编译环境下相比 TVM 的动态优化具有系统性优势, 但以 1.5–2 倍的内存开销为代价; TVM 的动态存储调度策略在 RISC-V 有限的 TLB 和缓存容量下导致页表抖动, 需要针对平台约束重新设计; RVV 向量化优化在计算密集型算子和高复杂度场景下有效, 但在特定算子类型 (除法、取模) 和低复杂度场景下反而引入性能劣化, 需要进行选择性应用. 这些行为差异与潜在适配风险通过 RIVdoo 的多维度评估得以被捕捉和量化, 而传统仅依赖精度正确性验证的方法则通常难以覆盖上述与资源约束和体系结构特性相关的行为模式. 需要指出的是, 本文所识别的结果均基于多维度性能指标与复杂度放大系数的系统性实验分析, 其目的在于刻画算子在 RISC-V 架构下的运行行为特征及潜在适配风险, 而非对算子实现正确性作出最终裁决. 相关现象已整理为可复现的问题描述并提交至上游社区 (详见 https://github.com/yanyanyyy720/OperatorTestForRiscv/blob/main/issue.md), 供后续验证与讨论.

5 总结与展望

本文提出了一种面向 RISC-V 架构的深度学习算子质量评估方法 RIVdoo, 针对现有算子测试方法难以适应 RISC-V 特有的资源受限与高度可定制的问题, 提供了系统化的测试与分析框架. 该方法通过基于算子输入空间复杂度的分组策略覆盖不同计算负载, 结合多维度指标评估算子的精度、执行效率、内存占用及系统开销, 并引入复杂度放大系数的差分测试机制, 用于刻画算子在不同复杂度层级下的行为变化模式, 有效识别潜在的性能异常与平台相关的适配风险. 在覆盖 47 个包括计算密集型、内存密集型及轻量型在内算子的实验中, RIVdoo 发现 TFLite 的预编译策略在 RISC-V 平台上相比 TVM 具有 30%–50% 的速度优势但以 1.5–2 倍的内存开销为代价; TVM 的动态存储调度策略因 RISC-V 有限的 TLB 和缓存容量导致缺页率比 TFLite 高 60%–150%; RVV 向量化优化在计算密集型算子和高复杂度下有效, 但在部分算子和低复杂度场景下反而导致性能下降. 上述结果表明, RIVdoo 能够系统性地揭示算子在 RISC-V 架构下的复杂度敏感行为与资源相关特征, 相关现象已整理为可复现的问题描述, 并提交至 TVM、TFLite 等上游开源社区, 目前仍处于社区进一步验证与讨论阶段.

未来工作将重点考虑扩展复杂度分组理论至 RISC-V 异构计算场景, 研究 RVV 参数配置 (向量长度、寄存器分组) 对算子性能的影响规律, 以及将复杂度分组策略扩展到子图和完整模型层级的端到端测试, 这将有助于构建更完善的 RISC-V 深度学习算子测试体系, 为算子优化、平台调优及高质量 RISC-V 软件生态的发展提供坚实基础.

References

[1] Liu C, Wu YJ, Wu JZ, Zhao C. Survey on RISC-V system architecture research. Ruan Jian Xue Bao/Journal of Software, 2021, 32(12): 3992–4024 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6490.htm [doi: 10.13328/j.cnki.jos.006490]

[2] Li YW, Yin LZ, Dong W, Liu JX, Hu YF, Li SS. Hetrify: Efficient verification of heterogeneous programs on RISC-V. In: Proc. of the 47th IEEE/ACMInt'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2025. 618–618. [doi: 10.1109/ICSE55347.2025.00081]

[3] Han JC, Wang ZD, Ma H, Song W. Spike-FlexiCAS: RISC-V processor simulator supporting flexible cache architecture configuration. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 3954–3969 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7358.htm [doi: 10.13328/j.cnki.jos.007358]

[4] Han LT, Zhang HB, Xing MJ, Wu YJ, Zhao C. Optimization method for high-performance libraries targeting RISC-V vector extension. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 3985–4005 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7360.htm [doi: 10.13328/j.cnki.jos.007360]

[5] Agosta G, Galimberti A, Zoni D. Deep learning on RISC-V platforms at the edge: A perspective on the hardware and software support. ACM Computing Surveys, 2025, 58(5): 121. [doi: 10.1145/3772277]

[6] Li RS, Peng P, Shao ZY, Jin H, Zheng R. Evaluating RISC-V vector instruction set architecture extension with computer vision

workloads. Journal of Computer Science and Technology, 2023, 38(4): 807–820. [doi: 10.1007/s11390-023-1266-6]

[7] Gu DD, Shi YN, Liu XZ, Wu G, Jiang HO, Zhao YS, Ma Y. Defect detection for deep learning frameworks based on meta operators. Chinese Journal of Computers, 2022, 45(2): 240–255 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2022.00240]

[8] Zhang XF, Sun N, Fang CR, Liu JW, Liu J, Chai D, Wang J, Chen ZY. Predoo: Precision testing of deep learning operators. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2021. 400–412. [doi: 10.1145/3460319.3464843]

[9] Liu JW, Huang YH, Wang ZJ, Ma L, Fang CR, Gu MZ, Zhang XF, Chen ZY. Generation-based differential fuzzing for deep learning libraries. ACM Trans. on Software Engineering and Methodology, 2023, 33(2): 50. [doi: 10.1145/3628159]

[10] Ou SW, Li YW, Yu L, Wei CK, Wen TK, Chen QP, Chen Y, Tang HZ, Pan ZL. MirrorFuzz: Leveraging LLM and shared bugs for deep learning framework APIs fuzzing. IEEE Trans. on Software Engineering, 2025, 52(1): 360–375. [doi: 10.1109/TSE.2025.3619966]

[11] Lee JKL, Jamieson M, Brown N, Jesus R. Test-driving RISC-V vector hardware for HPC. In: Proc. of the 2023 ISC High Performance Int'l Workshops on High Performance Computing. Hamburg: Springer, 2023. 419–432. [doi: 10.1007/978-3-031-40843-4_31]

[12] Xu XZ, Yang DH, Wang L, Wang T, Huang AW, Li Q. Sequential consistency per location theorem proving in RISC-V memory consistency model. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 3919–3936 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7292.htm [doi: 10.13328/j.cnki.jos.007292]

[13] Li CD, Yi R, Luo YW, Wang XL, Wang ZL. Lazy shadow paging under the RISC-V architecture. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 3970–3984 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7359.htm [doi: 10.13328/j.cnki.jos.007359]

[14] Poorhosseini M, Nebel W, Grüttner K. A compiler comparison in the RISC-V ecosystem. In: Proc. of the 2020 Int'l Conf. on Omni-layer Intelligent Systems (COINS). Barcelona: IEEE, 2020. 1–6. [doi: 10.1109/COINS49042.2020.9191411]

[15] Adit N, Sampson A. Performance left on the table: An evaluation of compiler autovectorization for RISC-V. IEEE Micro, 2022, 42(5): 41–48. [doi: 10.1109/MM.2022.3184867]

[16] Bjäreholt J. RISC-V compiler performance: A comparison between GCC and LLVM/Clang [BS. Thesis]. Karlskrona: Blekinge Institute of Technology, 2017.

[17] Zhang XF, Liu JW, Sun N, Fang CR, Liu J, Wang J, Chai D, Chen ZY. Duo: Differential fuzzing for deep learning operators. IEEE Trans. on Reliability, 2021, 70(4): 1671–1685. [doi: 10.1109/TR.2021.3107165]

[18] Wen ZZ, Liu HY, Zhu TW, Pan MX, Wang SH, Lin YY, Liu KR, Zhang T, Li XD. A study of floating-point precision tuning in deep learning operators implementations. ACM Trans. on Software Engineering and Methodology, 2025. [doi: 10.1145/3773992]

[19] Ma XY, Du XT, Cai Q, Zheng Y, Hu Z, Zheng Z. Survey on testing of deep learning frameworks. Ruan Jian Xue Bao/Journal of Software, 2024, 35(8): 3752–3784 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7059.htm [doi: 10.13328/j.cnki.jos.007059]

[20] Zhang XY, Jiang WP, Shen C, Li Q, Wang Q, Lin CH, Guan XH. Deep learning library testing: Definition, methods and challenges. ACM Computing Surveys, 2025, 57(7): 187. [doi: 10.1145/3716497]

[21] Liu JW, Zhang XF, Xu LR, Fang CR, Gu MZ, Luo WS, Chai D, Wang J, Zhao ZH, Chen ZY. Automated detection and repair of floating-point precision problems in convolutional neural network operators. ACM Trans. on Software Engineering and Methodology, 2025, 34(7): 203. [doi: 10.1145/3715104]

[22] Shi JY, Xiao Y, Li YK, Li YT, Yu DS, Yu CD, Su H, Chen YF, Huo W. ACETest: Automated constraint extraction for testing deep learning operators. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 690–702. [doi: 10.1145/3597926.3598088]

[23] Chen JY, Jia CY, Yan YJ, Ge J, Zheng HB, Cheng Y. A miss is as good as a mile: Metamorphic testing for deep learning operators. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 89. [doi: 10.1145/3660796]

[24] Chen JJ, Liang YH, Shen QC, Jiang JJ, Li SC. Toward understanding deep learning framework bugs. ACM Trans. on Software Engineering and Methodology, 2023, 32(6): 135. [doi: 10.1145/3587155]

[25] Ge J, Yu HQ, Fan GS, Tang JH, Huang ZJ. Just-in-time defect prediction for intelligent computing frameworks. Ruan Jian Xue Bao/Journal of Software, 2023, 34(9): 3966–3980 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6874.htm [doi: 10.13328/j.cnki.jos.006874]

[26] Wei AJ, Deng YL, Yang CY, Zhang LM. Free lunch for testing: Fuzzing deep-learning libraries from open source. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 995–1007. [doi: 10.1145/3510003.3510041]

[27] Pham HV, Lutellier T, Qi WZ, Tan L. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 1027–1038. [doi: 10.1109/ICSE.2019.00107]

[28] Guo QY, Xie XF, Li Y, Zhang XY, Liu Y, Li XH. Audee: Automated testing for deep learning frameworks. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 486–498. [doi: 10.1145/3324884.3416571]

[29] Li ZY, Wu JZ, Ling X, Luo TY, Rui ZQ, Wu YJ. The seeds of the FUTURE sprout from history: Fuzzing for unveiling vulnerabilities in prospective deep-learning libraries. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2020. 1616–1627. [doi: 10.1109/ICSE55347.2025.00132]

[30] Harzevili NS, Mohajer MM, Wei MS, Pham HV, Wang S. History-driven fuzzing for deep learning libraries. ACM Trans. on Software Engineering and Methodology, 2025, 34(1): 19. [doi: 10.1145/3688838]

[31] Deng YL, Xia CS, Peng HR, Yang CY, Zhang LM. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 423–435. [doi: 10.1145/3597926.3598067]

[32] Deng ZZ, Meng GZ, Chen K, Liu K, Liu T, Xiang L, Chen CY. Differential testing of cross deep learning framework APIs: Revealing inconsistencies and vulnerabilities. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 414.

[33] Waterman A, Lee Y, Patterson DA, Asanovic K. The RISC-V instruction set manual, volume I: Base user-level ISA. Technical Report, UCB/EECS-2011-62, University of California at Berkeley, 2011.

[34] Celio C, Chiu PF, Nikolic B, Patterson DA, Asanović K. BOOMv2: An open-source out-of-order RISC-V core. Technical Report, UCB/EECS-2017-157, University of California at Berkeley, 2017.

[35] Lopoukhine A, Ficarelli F, Vasiladiotis C, Lydike A, van Delm J, Dutilleul A, Benini L, Verhelst M, Grosser T. A multi-level compiler backend for accelerated micro-kernels targeting RISC-V ISA extensions. In: Proc. of the 23rd ACM/IEEE Int'l Symp. on Code Generation and Optimization. Las Vegas: ACM, 2025. 163–178. [doi: 10.1145/3696443.3708952]

[36] Hussain T, Tahir MW, Mushtaq M, Khalid S. Design and benchmarking of a low-cost RISC-V-based high-performance computing cluster for edge computing. In: Proc. of the 2025 Int'l Conf. on Energy, Power, Environment, Control and Computing (ICEPECC 2025). Gujrat: IET, 2025. 579–586. [doi: 10.1049/icp.2025.1168]

[37] Titopoulos V, Alexandridis K, Peltekis C, Nicopoulos C, Dimitrakopoulos G. Optimizing structured-sparse matrix multiplication in RISC-V vector processors. IEEE Trans. on Computers, 2025, 74(4): 1446–1460. [doi: 10.1109/TC.2025.3533083]

[38] Kovač M, Dragić L, Malnar B, Minervini F, Palomar O, Rojas C, Olivieri M, Knezović J, Kovač M. FAUST: Design and implementation of a pipelined RISC-V vector floating-point unit. Microprocessors and Microsystems, 2023, 97: 104762. [doi: 10.1016/j.micpro.2023.104762]

[39] Lee Y, Waterman A, Cook H, Zimmer B, Keller B, Puggelli A, Kwak J, Jevtic R, Bailey S, Blagojevic M, Chiu PF, Avizienis R, Richards B, Bachrach J, Patterson D, Alon E, Nikolic B, Asanovic K. An agile approach to building RISC-V microprocessors. IEEE Micro, 2016, 36(2): 8–20. [doi: 10.1109/MM.2016.11]

[40] Xie XF, Liang Z, Gu P, Basak A, Deng L, Liang L, Hu X, Xie Y. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In: Proc. of the 2021 IEEE Int'l Symp. on High-performance Computer Architecture (HPCA). Seoul: IEEE, 2021. 570–583. [doi: 10.1109/HPCA51647.2021.00055]

[41] Cavalcante M, Schuiki F, Zaruba F, Schaffner M, Benini L. Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2020, 28(2): 530–543. [doi: 10.1109/TVLSI.2019.2950087]

[42] Gautschi M, Schiavone PD, Traber A, Loi I, Pullini A, Rossi D, Flamand E, Gürkaynak FK, Benini L. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2017, 25(10): 2700–2713. [doi: 10.1109/TVLSI.2017.2654506]

[43] Chen TQ, Moreau T, Jiang ZH, Zheng LM, Yan D, 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: USENIX Association, 2018. 578–594.

[44] David R, Duke J, Jain A, Reddi Vj, Jeffries N, Li J, Kreeger N, Nappier I, Natraj M, Wang TZ, Warden P, Rhodes R. TensorFlow lite micro: Embedded machine learning for tinyML systems. Proc. of Machine Learning and Systems, 2021, 3: 800–811.

附中文参考文献

[1] 刘畅, 武延军, 吴敬征, 赵琛. RISC-V 指令集架构研究综述. 软件学报, 2021, 32(12): 3992–4024. http://www.jos.org.cn/1000-9825/6490.htm [doi: 10.13328/j.cnki.jos.006490]

[3] 韩金池, 王智栋, 马浩, 宋威. Spike-FlexiCAS: 支持缓存架构灵活配置的 RISC-V 处理器模拟器. 软件学报, 2025, 36(9): 3954–3969.

http://www.jos.org.cn/1000-9825/7358.htm [doi: 10.13328/j.cnki.jos.007358]
[4] 韩柳彤, 张洪滨, 邢明杰, 武延军, 赵琛. 面向 RISC-V 向量扩展的高性能算法库优化方法. 软件学报, 2025, 36(9): 3985–4005. http://www.jos.org.cn/1000-9825/7360.htm [doi: 10.13328/j.cnki.jos.007360]
[7] 谷典典, 石屹宁, 刘譞哲, 吴格, 姜海鸥, 赵耀帅, 马郓. 基于元算子的深度学习框架缺陷检测方法. 计算机学报, 2022, 45(2): 240–255. [doi: 10.11897/SP.J.1016.2022.00240]
[12] 徐学政, 杨德亨, 王璐, 王涛, 黄安文, 李琼. RISC-V 内存一致性模型的同地址顺序一致性定理证明. 软件学报, 2025, 36(9): 3919–3936. http://www.jos.org.cn/1000-9825/7292.htm [doi: 10.13328/j.cnki.jos.007292]
[13] 李传东, 衣然, 罗英伟, 汪小林, 王振林. RISC-V 架构下的懒惰影子页表模型. 软件学报, 2025, 36(9): 3970–3984. http://www.jos.org.cn/1000-9825/7359.htm [doi: 10.13328/j.cnki.jos.007359]
[19] 马祥跃, 杜晓婷, 采青, 郑阳, 胡峭, 郑征. 深度学习框架测试研究综述. 软件学报, 2024, 35(8): 3752–3784. http://www.jos.org.cn/1000-9825/7059.htm [doi: 10.13328/j.cnki.jos.007059]
[25] 葛建, 虞慧群, 范贵生, 唐锏浩, 黄子杰. 面向智能计算框架的即时缺陷预测. 软件学报, 2023, 34(9): 3966–3980. http://www.jos.org.cn/1000-9825/6874.htm [doi: 10.13328/j.cnki.jos.006874]

作者简介

刘佳玮, 博士, CCF 专业会员, 主要研究领域为智能软件测试, 算子测试.
高岩, 硕士生, 主要研究领域为深度学习算子测试.
张犬俊, 博士, 教授, CCF 专业会员, 主要研究领域为软件测试, 程序修复, 代码智能体.
尚也, 博士生, CCF 学生会员, 主要研究领域为程序修复, 代码智能体.
房春荣, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为智能软件工程, 大代码, AI 测试.
陈振宇, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为群体智能, 深度学习测试和优化, 大数据质量, 移动应用测试.