计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009827] [CSTR: 32024.14.csa.009827]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向 PyTorch 的 RVV 优化①

王　凡[1,3], 张　飞[2], 宋甫元[1], 于佳耕[2]

[1](南京信息工程大学 计算机学院、网络空间安全学院, 南京 210044)
[2](中国科学院 软件研究所, 北京 100190)
[3](中国科学院大学南京学院, 南京 211135)
通信作者: 于佳耕, E-mail: jiageng08@iscas.ac.cn

摘　要: RISC-V 软件生态正在加速发展, 国际开源社区积极投入 RISC-V 软件生态, 针对 RISC-V 主动适配和优化, 积极推动 RISC-V 软件生态系统向前发展. PyTorch 是一个开源的 Python 机器学习库, 其在性能、开源生态、研究领域都有非常大的优势, 其对 x86、ARM、PowerPC 以及 CUDA 等指令集架构都提供了较好的支持. 但是, 在目前的 RISC-V 架构上, 软件生态移植集中在对 RISC-V 标准指令集的适配, 尚不能充分利用 RISC-V 扩展指令集优化软件生态, 距离 ARM、x86 等成熟软件生态存在较大差距. PyTorch 因缺少 RISC-V V 扩展 (RVV) 的支持, 使得 RISC-V 平台的推理性能与同规格 ARM 平台差距较大. 针对上述问题, 本文提出了一种面向 PyTorch RVV 1.0 的高效开发方案, 并使用 RVV 扩展指令集对 PyTorch 深度卷积算子进行针对性优化, 并在 K230 开发板上进行了对比分析, 实验结果表明, 相比标量实现, 利用 RVV 优化的深度卷积算子性能提升约 1.35–3.8 倍.

关键词: RISC-V; PyTorch; RVV 扩展指令集; 深度卷积

引用格式: 王凡,张飞,宋甫元,于佳耕.面向 PyTorch 的 RVV 优化.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9827.html

RVV Optimization for PyTorch

WANG Fan[1,3], ZHANG Fei[2], SONG Fu-Yuan[1], YU Jia-Geng[2]

[1](School of Computer Science & School of Cyber Science and Engineering, Nanjing University of Information Science & Technology, Nanjing 210044, China)
[2](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[3](University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)

Abstract: The RISC-V software ecosystem is in the stage of accelerated development. International open-source community makes active contributions with focus on adaptation and optimization for RISC-V, driving its software ecosystem forward. PyTorch, an open-source Python machine learning library, has significant advantages in performance, open-source ecosystem, and research areas. It provides strong support for instruction set architectures such as x86, ARM, PowerPC, and CUDA. However, in the current RISC-V architecture, the software ecosystem porting is mainly focused on adapting to the RISC-V standard instruction set and has not yet fully utilized the RISC-V extended instruction sets to optimize the software ecosystem, which leaves a significant gap between the RISC-V software ecosystem and the mature ecosystems like ARM and x86. PyTorch, lacking support of RISC-V V extension (RVV), results in a considerable gap in inference performance between RISC-V platforms and ARM platforms of similar specifications. To address this issue, this study proposes an efficient development scheme for PyTorch RVV1.0 and optimizes deep convolution operators in PyTorch by using the RVV extended instruction set. A comparative analysis is conducted on the K230 development board, with experimental results showing that the performance of deep convolution operators optimized with RVV is

① 收稿时间: 2024-10-14; 修改时间: 2024-10-21; 采用时间: 2024-11-18; csa 在线出版时间: 2025-02-18

improved by approximately 1.35 to 3.8 times compared to scalar implementations.

Key words: RISC-V; PyTorch; RVV extension instruction set; depthwise convolution

RISC-V 是一种新兴的开源精简指令集架构, 由加州大学伯克利分校在 2010 年首次发布[1], 它是在现有的体系结构 (如 x86、ARM、MIPS 等) 经长期发展所暴露出的问题的基础上, 顺应现代信息系统设计需求和体系结构发展趋势而生的. 随着智能互联的到来, RISC-V 架构迸发出新的活力, 尤其是在物联网 (IoT)、人工智能 (AI) 等领域探索出了广阔的发展潜力, 逐渐与 x86、ARM 形成了三足鼎立的局面.

PyTorch 是当前深度学习领域中最主流的开源框架之一, 广泛应用于学术界和工业界的大规模模型训练和推理中. Stanescu 等人[2]将 PyTorch 和 TensorFlow 在对大量乳腺组织病理学图像进行分类时进行比较, PyTorch 在训练和验证中的准确性、精确性、召回率和损失方面都表现更好. Ansel 等人[3]的研究表明 PyTorch 能够通过动态 Python 字节码转换和图形编译加快机器学习速度. Tian 等人[4]对 PyTorch 和 TensorFlow 的推理性能进行了对比, PyTorch 在 EDSR、RDN、RCAN、HAN 和 HAT 模型上表现出比 TensorFlow 更好的推理延迟. Krisilias 等人[5]在同步学习中研究结果表明 PyTorch 比 TensorFlow 平均快 2.65 倍. Chirodea 等人[6]对 TensorFlow 和 PyTorch 在卷积神经网络应用方面进行了比较, PyTorch 在总训练时间上比 TensorFlow 快 25.5%, 平均每轮训练时间快 26.1%, 在网络和优化时间上快 25.1%, 在输出深度图时快 77.7%. PyTorch 已经成为许多深度学习研究和应用的首选工具.

然而, 在 RISC-V 架构上, PyTorch 目前表现出了一些显著劣势, 这些劣势阻碍了其在 RISC-V AI 生态的发展. PyTorch 对主流架构提供了 SIMD 技术的支持. 如 PyTorch 在 x86 架构上提供了 AVX2、AVX512 的支持, 对 ARM 架构提供了 NEON 支持, 对 PowerPC 架构提供了 VSX 的支持, 使得 PyTorch 在使用以上架构的处理器上的训练和推理性能表现出色. 而 RISC-V 作为一种新兴的开放指令集架构, 在物联网 (IoT) 和人工智能 (AI) 领域显示出巨大的潜力, 但 PyTorch 尚未对 RISC-V 向量扩展 (RVV) 提供有效支持, 导致其无法充分利用向量寄存器的优势. 这种缺乏向量化支持的局面使得 PyTorch 在 RISC-V 架构处理器上的性能远落后于 ARM 等其他架构.

同时, 相较于其他深度学习框架, 如 TensorFlow 和 ncnn, PyTorch 在 RISC-V 上的表现也存在显著劣势. TensorFlow 通过社区的积极推动, 已经对 RISC-V 提供了一定程度的支持, 能够在 RISC-V 上进行有效的模型训练和推理. ncnn 作为一个为手机端极致优化的高性能神经网络前向计算框架, 有着腾讯生态的大力支持, 其已经通过 RVV 优化实现了大量的算子, 其中基本数学运算、矩阵乘法等算子均已经适配完成, 使得其在 RISC-V 架构上具备较好的性能表现. 相比之下, PyTorch 由于缺乏对 RVV 的支持, 导致其在 RISC-V 平台上的性能和适配性较弱, 难以满足高性能计算的需求, 所以, 面向 PyTorch 的 RVV 优化就显得至关重要.

此外, PyTorch 本身是一个庞大而复杂的系统, 其代码库规模庞大, 包含了众多运算模块和优化路径, 这使得在 RISC-V 上进行深度优化面临巨大的挑战. PyTorch 的高度灵活性和广泛的功能支持带来了复杂的依赖和构建需求, 尤其是在适配新兴架构时, 涉及大量底层代码的修改和兼容性处理. 相比于其他更为轻量的深度学习框架, PyTorch 在架构适配和性能优化过程中需要克服更多的工程复杂性, 包括对底层运算单元的高效实现以及对不同指令集的支持, 这些都对其在 RISC-V 上的优化提出了更高的要求.

具体而言, 深度卷积算子是神经网络计算中的核心组件之一, 在卷积神经网络的推理过程中被频繁调用, 占用神经网络推理的大部分时间, 因此对深度卷积算子的性能要求较高. 主流架构如 x86 和 ARM, 均使用高度优化的向量指令来提升深度卷积算子的性能, 而 RISC-V 的 RVV 扩展由于缺乏 PyTorch 的原生支持, 导致其在执行这些计算时依然依赖标量实现, 无法充分利用 SIMD 的优势.

因此, 本文选取深度卷积算子进行 RISC-V 向量化优化, 针对深度卷积算子, 使用 Winograd 算法进行进一步优化, 重点解决 Winograd 变换中的矩阵转置问题, 依据 Winograd 算法中的输入数据宽度, 研究 RISC-V 向量寄存器分组、动态向量长度, 并研究通过兼容性方案解决当前 RISC-V PyTorch 环境下可能出现的构

建问题, 极大提高神经网络的推理性能.

针对上述问题, 本文提出了一种面向 RVV 的 PyTorch 优化方法. 本文的主要创新和贡献包括以下几个方面.

(1) 基于 PyTorch 构建系统, 提出了 PyTorch 在 RVV 环境下的兼容性构建方案.

(2) 基于 Winograd 算法和 RVV 特性对 PyTorch 的深度卷积算子进行深度优化.

(3) 在 K230 硬件环境下和神经网络模型上进行实验, 证明该优化后的实现性能相比标量实现和自动向量化实现分别提高了约 1.35–3.80 倍以及约 1.10–3.73 倍. 工作成果贡献 PyTorch 开源社区.

1 相关工作

1.1 RISC-V 软件适配和优化技术

根据 RISC-V 云计算开源软件供应链名录统计, RISC-V 已经适配了 openEuler[7]、Ubuntu 等典型操作系统、Python 等编程语言、GCC、LLVM/Clang 等编译工具链. RISC-V 还支持如虚拟机软件 QEMU[8]等生产工具, 为在低性能硬件上实现 PyTorch RVV 优化提供了条件.

Lee 等人[9]对编译器自动向量化进行了研究, 指出在 RISC-V 上 LLVM 编译器比 GNU 编译器能够自动向量化更多的计算核, 并且在大多数情况下性能提升显著.

RISC-V 在对 RVV 的适配上仍较为缓慢, 目前在对 V 扩展指令集的研究包括基于 RVV 对 OpenBLAS、XNNPACK[10]、OpenCV[11]和 musl libc 库[12]进行优化, 来进一步提高高性能数学计算库的运算速度、图像处理性能和基础 C 库的性能.

在 RVV 算法优化上, Mahale 等人[13]对 VPU 进行了改进, 优化了 RISC-V 架构处理长稀疏向量的性能. Zhao 等人[14]对 RVV 在 FFT 中的应用进行了研究, 使的 FFT 算法库的性能在 RISC-V CPU 上有了显著性能提升. Rodrigues 等人[15]对稀疏矩阵的向量乘法进行了优化, 展示了 2.04 倍的性能提升. Igual 等人[16]使用 RVV 汇编代码对矩阵乘法进行了实现, 达到了相比 OpenBLAS 库 1.3 倍的性能提升. Zou 等人[17]对 Blake3 哈希算法进行了优化, 使用 RVV 实现了约 10 倍的性能提升. Rizi 等人[18]对 AES 加密算法进行了研究, 使用 RVV 实现了约 2 倍的性能提升. Vizcaino 等人[19]使用最大向量长度的 RVV 实现, 在 8-Stockham FFT 算法上相比标量实现看到了近 16 倍的性能提升. van Kempen 等人[20]将 ARM Cortex-M 中的特定内核实现移植到 RVV 后, 在 TinyML 上看到了相比自动向量化约 60% 的性能提升.

Gupta 等人[21]在 RISC-V 向量处理器上对用于卷积层的矢量化 Winograd 算法进行了研究, 研究确定了使用内建指令优化 RVV 上的 Winograd 内核的有效技术, 并展示了某些指令提供更好的性能, 同时得出结论 Winograd 算法受益于长达 2 048 位的矢量长度和高达 64 MB 的缓存大小.

对 RVV 的研究目前还集中在对个别热点算法的优化上, 并且研究的成果很少在实际的开源智能计算框架中进行落地支持, 所以针对 RVV 的研究还有非常多的工作可以进行. 因此, 需要结合 PyTorch 智能计算框架研究 RVV 对神经网络的推理加速.

1.2 AI 智能框架 RISC-V 适配

目前, 在 TensorFlow 中, 已经有了大量补丁来支持 RISC-V 架构编译, 并通过考虑 RISC-V 内存布局对深度卷积算子提供了部分支持, 该深度卷积使用纯 C++ 实现, 没有使用 RVV 内建函数 (intrinsic). 在热门深度神经网络库 oneDNN 中, 通过使用 RVV 1.0 内建函数对神经网络中的池化层进行了优化.在 OpenBLAS 矩阵计算库中, 对矩阵乘法使用 RVV 进行了优化. 在 SLEEF 库中, 对基本数学运算均使用了 RVV 进行了优化. 在 ncnn 库中, 对基本数学运算已经矩阵乘法等算子都进行了 RVV 优化. PyTorch 在性能、开源生态、研究领域都有非常大的优势, 其对 x86、ARM、PowerPC 以及 CUDA 等指令集架构都提供了较好的支持, 但是在 RISC-V 架构上还是一片空白, 只能够进行简单的推理, 相比 ARM 和 x86 架构在性能上有着很大的差异, 无法充分发挥 RISC-V 处理器的性能. 因此为提供 RISC-V 智能计算生态, 需要对 PyTorch 进行 RISC-V 架构的研究.

2 PyTorch RVV 实现

PyTorch 底层算子采用 C++实现, 部分算子在 ARM 等架构中已有对应的向量化实现, 未使用向量化优化的架构如 RISC-V 会调用由 C++实现的标量代码.

在 PyTorch RVV 优化中, 需要重点解决以下问题.

(1) 基于 RVV 1.0 intrinsic 函数的实现无法在不支持 RVV 1.0 的硬件上编译执行, PyTorch 中还没有针对 RVV 硬件的构建系统, 需要研究如何构建 PyTorch RVV 代码实现.

(2) 针对深度卷积算子优化, RVV 硬件下, 256 位

及以上代码实现无法在 128 位硬件环境下运行, 因此需要研究 RVV 硬件兼容性和 Winograd 算法来选用合适的向量长度和寄存器分组方案, 同时研究如何在 Winograd 算法中充分结合 RVV 的特性进一步优化, 针对 RISC-V 矩阵扩展不成熟的问题, 需要使用 RVV 解决矩阵转置操作性能差的问题.

本节对 RISC-V 在 PyTorch 的构建方案和深度卷积算子优化进行说明.

2.1 PyTorch RVV 构建方案

RVV 为 RISC-V 扩展指令集, 因此基于 RVV 1.0 内建函数实现的函数无法在不支持 RVV 1.0 的硬件上编译执行. PyTorch 中还没有针对 RVV 硬件的构建系统, 因此需要研究使用额外的机制来实现 RVV 代码的构建编译, 这里的方案使用用户自主选择是否开启 RVV 指令扩展的方式进行编译构建.

具体而言, 在编译阶段自主决定是否使用编译器参数-march=rv64gcv 使得编译器在构建 PyTorch 过程中能够决定是否将 RVV 代码实现进行编译. RVV 代码通过宏定义的方式进行实现, RVV1.0 的宏定义使用 #defined(__riscv_v_intrinsic) && __riscv_v_intrinsic>=12000. 因此, 为了确保在不开启-march=rv64gcv 编译选项的情况下能够正确编译 PyTorch 标量版本,

PyTorch 代码的实现方式如代码清单 1 所示.

代码清单 1. RVV 优化方案

```
#ifdef ARM_NEON || AVX512
//其他架构下的算子实现
#elif define(__riscv_v_intrinsic) && __riscv_v_intrinsic>=12000)
#include <riscv_vector.h>
//RVV 算子优化实现
#else
//PyTorch 算子标量实现
#endif
```

2.2 深度卷积算子优化

本节基于 Winograd 算法和 RVV, 对 PyTorch 中的深度卷积算子进行实现, 使用 RVV 的特性进行优化, 如寄存器分组, 动态向量长度优化, 同时使用矩阵分块、输入变换、卷积核变换、矩阵转置、输出变换、向量化乘加、向量化乘减等操作进行实现.

基于 Winograd 算法优化的深度卷积流程如图 1 所示.

2.2.1 矩阵分块优化

RVV 定义不同硬件环境下向量寄存器位宽最低为 128, 高于 128 位如 256 位以上的硬件环境兼容 128 位实现, 而 256 位及以上代码实现无法在 128 位硬件环境下运行. 为了兼容不同的寄存器宽度如 256、512 及以上长度, 使用固定 128 位大小的寄存器宽度进行实现是目前最优的选择. 同时, 考虑 Winograd 算法中分块矩阵的大小以及卷积核的尺寸, 通常, 在 Winograd 算法中将输入特征图划分为 4×4 大小的块, 使用 3×3 的卷积核来进行卷积操作, PyTorch 中 Winograd 算法实现的数据类型为 float32, 分块中一行的 4 个数据刚好可以充满 128 位寄存器的位宽, 同时 3×3 的卷积核将每行填充进寄存器也不会产生大量的寄存器空间浪费. 因此使用 RVV 内建函数如__riscv_vfmul_vv_f32m1, 指定分组为 m1 进行实现是 Winograd 算法中的最佳实现.

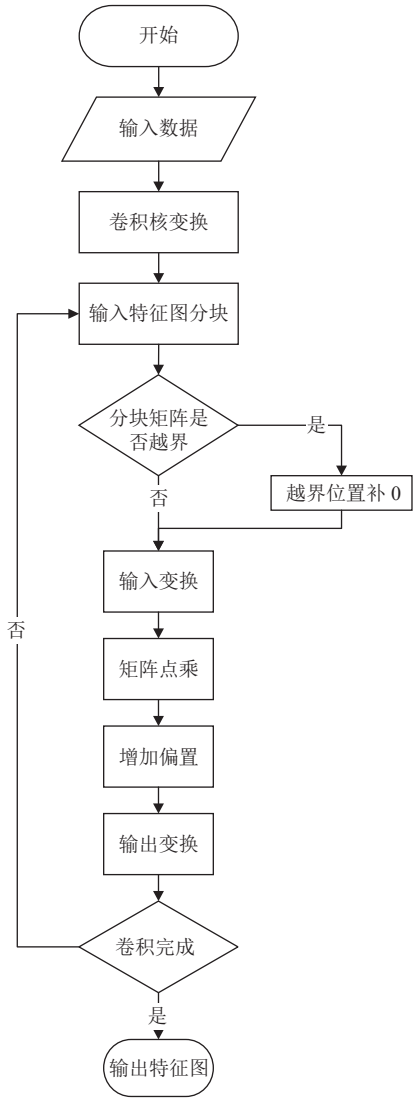

图 1 基于 Winograd 算法优化的深度卷积

2.2.2 RVV 动态向量长度优化

RVV 支持动态向量长度, 可以指定向量寄存器中需要参与运算的数据个数. 利用 RVV 的特性, 可以对 Winograd 算法实现中的细节进行针对性优化, 这里以卷积核上的优化为例.

在 Winograd 算法实现中, 卷积核的尺寸为 3×3, 需要将卷积核按行存储到 3 个寄存器中. 在传统定长向量指令集如 ARM NEON 的实现中, 由于需要充满整个 128 位大小的向量寄存器, 需要额外的操作对最后的寄存器进行填充, 添加脏数据, 并在卷积核变换时需要额外的计算操作, 其实现如代码清单 2 所示.

代码清单 2. ARM NEON 中读取卷积核

```
g0 = vld1q_f32(kernel); //VectorLoad
g1 = vld1q_f32(kernel + 3);
//g2[3] is junk
g2 = vextq_f32(vld1q_f32(kernel+5), vld1q_f32(kernel + 5), 1);
```

在基于 RVV 动态向量长度特性下, 可以通过修改 VL 的参数来指定向量将要处理的元素个数, 可以减少变换时的多余乘加操作来进一步提升性能. RVV 中的实现如代码清单 3 所示.

代码清单 3. RVV 中读取卷积核

```
g0 = __riscv_vle32_v_f32m1(kernel, 3);
g1 = __riscv_vle32_v_f32m1(kernel + 3, 3);
g2 = __riscv_vle32_v_f32m1(kernel + 6, 3);
```

2.2.3 Winograd 变换优化

Winograd 算法通过对输入特征图、卷积核、输出数据进行变换, 来减少矩阵运算过程中的乘法次数.

在一维 Winograd 卷积计算中[22], 以$F(2,3)$为例, 2 为卷积结果的输出长度, 3 为卷积核的尺寸. 这里用$d=[d_0\ d_1\ d_2\ d_3]^{\mathrm{T}}$表示输入向量, $g=[g_0\ g_1\ g_2]^{\mathrm{T}}$表示卷积核, $r=[r_0\ r_1]^{\mathrm{T}}$表示输出向量, 其计算过程可以表示为式 (1):

$$F(2,3)=\begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix}\begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix}=\begin{bmatrix} r_0 \\ r_1 \end{bmatrix} \tag{1}$$

而$F(2,3)$的 Winograd 卷积可写成如式 (2) 所示矩阵乘法形式.

$$F(2,3)=\begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix}\begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}=\begin{bmatrix} m_0+m_1+m_2 \\ m_0-m_1-m_2 \end{bmatrix} \tag{2}$$

其中, m_0、m_1、m_2、m_3 计算如式 (3) 所示:

$$\begin{cases} m_0=(d_0-d_2)g_0 \\ m_1=(d_1+d_2)\dfrac{g_0+g_1+g_2}{2} \\ m_2=(d_2-d_1)\dfrac{g_0-g_1+g_2}{2} \\ m_3=(d_1-d_3)g_2 \end{cases} \tag{3}$$

下面以本文实现的 2 维 Winograd 卷积, 采用 4×4 的区块、3×3 卷积核、步长为 1 为例, 传统的 Winograd 卷积实现可以直接使用式 (4) 来进行计算, 涉及 276 次乘法运算, 相比直接卷积的 36 次乘法所带来的消耗多得多. 不考虑仅需 1 次计算的部分, 经过优化的 Winograd 算法通过嵌套使用两次一维 Winograd 算法实现, 将乘法计算的次数减少到了 16 次, 大幅提升了卷积计算的性能.

$$Y=A^{\mathrm{T}}[[GgG^{\mathrm{T}}]\odot[B^{\mathrm{T}}dB]]A \tag{4}$$

其中, A^{T}、G、B^{T} 均为已知的矩阵, 其中 G 矩阵为:

$$G=\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

在本文的实现中, 我们通过结合一维 Winograd 算法与固定系数的加法来进一步优化. 对于卷积核变换, 如式 (4) 所示我们需要将 G 与 g 进行矩阵乘法运算, 这将带来大量的乘法运算. 但是, G 中包含大量的 0 和 1, 所以通过固定系数代替矩阵乘法运算可以节省大量的无效计算. 实现的代码如代码清单 4 所示.

代码清单 4. 使用固定系数实现卷积核变换

```
1) const_half = vfmv_v_f_f32m1(0.5f, 4);
2) g0_plus_g2 = vfadd_vv_f32m1(g0, g2, 4);
3) half_g0_plus_g2=vfmul_vv_f32m1(const_half, g0_plus_g2, 4);
4) *transform = vset_v_f32m1_f32m1x4(*transform, 0, g0);
5) *transform = vset_v_f32m1_f32m1x4(*transform, 1, vmuladdq_
f32(half_g0_plus_g2, const_half, g1));
6) *transform=vset_v_f32m1_f32m1x4(*transform, 2, vmulsubq_
f32(half_g0_plus_g2, const_half, g1));
7) *transform = vset_v_f32m1_f32m1x4(*transform, 3, g2);
```

首先, 将一个寄存器中的数据全部填充为 0.5, 用于后续矩阵 G 的乘法操作. 第 2、3 行代码中, 将 g_0 与 g_2 相加并除以 2 用于式 (3) 中 m_1 和 m_2 中 g 相关的计算. 第 4–7 行代码用于实现 G 矩阵与 g 的乘法操作. 在

这里, 如第 4 行代码所示, 我们通过将卷积核 g 的第 1 行 g_0 直接赋值给结果矩阵 transform 的第 1 行来实现矩阵 G 的第 1 行[1 0 0]与 g 的乘法运算, 即得到式 (3) 中 m_0 计算中 g_0, 进一步减少了 3 次乘法运算. 同样, 在第 5 行 vmuladdq 函数中, 我们先通过将 g_1 与 *const_half* 相乘, 即 g_1 除以 2, 再与 *half_g0_plus_g2* 相加, 来得到 m_1 中与 g 相关的计算. 同样第 6、7 行分别用于实现 m_2 和 m_3 中 g 相关的计算. 经过固定系数的加法运算, 可以将更多的乘法运算转换为加法运算, 进一步提高了计算性能.

2.2.4 矩阵转置优化

在 Winograd 变换中, 我们需要对输入特征图、卷积核、输出特征图转换后的数据进行矩阵转置操作.

RISC-V 矩阵扩展进展缓慢, 对于高级矩阵操作还没有完善的扩展指令支持. Gupta 等人[21]对矩阵转置操作进行了研究, 其使用额外的临时缓冲区, 并通过创建索引并使用大量的内存操作来实现, 给性能带来了极大负担, 所以在 Winograd 算法中研究高性能矩阵转置操作至关重要.

为了解决上述问题, 我们使用 RVV 的分段存储指令来实现矩阵转置操作, 以 4×4 矩阵为例, 使用分段存储函数__riscv_vsseg4e32_v_f32m1x4((float*)(&ret), *m*, 4), 将源矩阵 m 中的每个元素按照间隔为 4 的偏移将数据存储到目标矩阵 ret 中, 即将 m 的每一列的数据依次存储到 ret 的行中, 实现 4×4 矩阵的转置操作, 减少了大量内存读取和存储操作以及标量指令, 大大提高了矩阵转置的性能.

3 实验结果及分析

3.1 实验环境

本文的实验平台为勘智 K230 开发板, 芯片使用玄铁 C908, 支持 RVV 1.0, 512 MB 内存, 128 GB TF 卡, PyTorch 在 QEMU 内编译构建并迁移至开发板. 编译采用 LLVM 17.0.6 版本编译器, 并通过-O3 级别开启自动向量化. 实验使用 PyTorch 版本为 2.5.0a0+git8f70bf7, torchvision 版本为 0.20.0a0+bf01bab.

实验中, MobileNetV2、MobileNetV3、Shufflenet_v2_x1、EfficientNet_B0 模型使用 torchvision 库中训练的参数进行推理, Xception 使用 pretrainedmodels 库中的模型进行推理, 所有模型测试 5 次取平均值, 来测试经过 RVV intrinsic 优化后的深度卷积算子的性能.

3.2 在 MobileNetV2 模型上评估深度卷积优化实现

图 2 为各个典型尺寸的输入特征图在 MobileNetV2 上进行推理所消耗的时间, 对比了标量实现、RVV 自动向量化和我们经过结合 Winograd 算法以及 RVV intrinsic 优化后的性能.

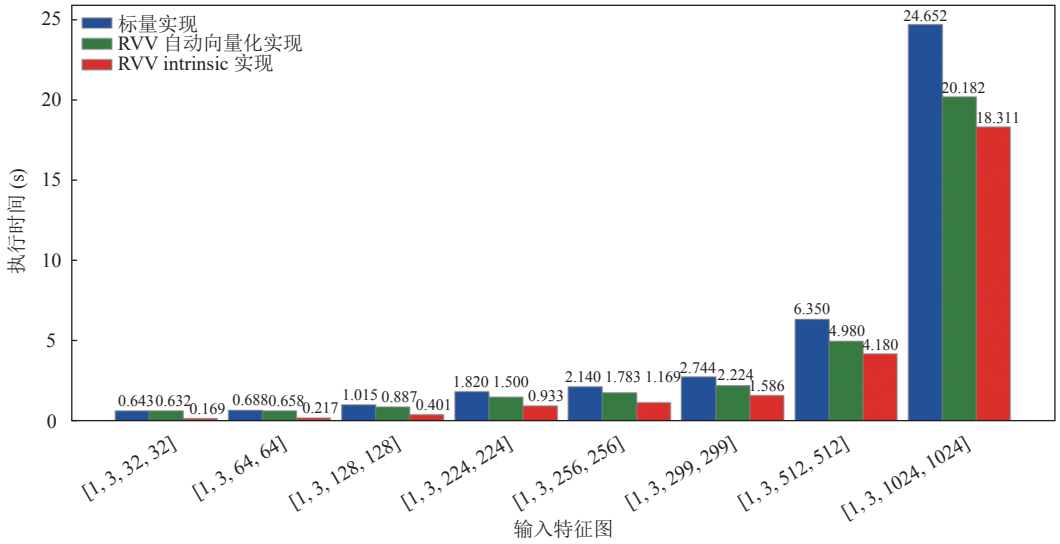

图 2 MobileNetV2 上输入不同尺寸特征图性能对比

对于所有的实现方式, 随着输入特征图尺寸的增大, 执行时间呈现出明显的增长趋势. 在所有输入尺寸下, RVV intrinsic 优化实现的执行时间显著低于标量实现和 RVV 自动向量化. 尤其是在较大的输入尺寸下, 如 [1, 3, 1 024, 1 024], RVV intrinsic 优化实现的执行时间为 18 311 ms, 而标量实现则为 24 652 ms, 有着

巨大的性能差异. RVV 自动向量化的表现介于标量实现和 RVV intrinsic 优化实现之间. 数据表明自动向量化能在一定程度上提升性能, 但与使用 RVV intrinsic 优化后相比仍有差距.

对比较大的输入尺寸 (例如 [1, 3, 512, 512] 和 [1, 3, 1024, 1024]), 可以发现标量实现和 RVV 自动向量化的执行时间增速比 RVV intrinsic 优化实现更快. 这意味着在处理大规模输入时, RVV intrinsic 优化能显著减缓性能下降, 尤其在计算密集型任务中表现出色.

对于较小的输入尺寸 (如 [1, 3, 32, 32]), 深度优化的版本相比自动向量化和标量版本可以带来最高约 3.8 倍的性能提升.

我们将自动向量化和 RVV intrinsic 深度优化的版本测试结果进行了分析, 发现在数据规模较小时, 编译器并不会进行自动向量化, 这也是小尺寸输入下如[1, 3, 32, 32]和[1, 3, 64, 64] , 标量实现与自动向量化没有太大差异的主要原因. 对于大尺寸的输入, 尽管编译器进行了自动向量化, 我们发现, LLVM 仅对基本数学运算如加减乘除运算有着较好的支持, 并能够使用 m2 分组进行向量化, 但是, 对于庞大的深度卷积算子, 编译器并不能自动向量化全部内核. 同时, Winograd 算法与 RVV intrinsic 实现能够更好的结合, 特别是在高级矩阵运算等无法自动向量化的场景, 带来了进一步的性能提升.

我们的实现还受限于开发板的单核性能, 在多核性能表现上有望看到更大的性能提升.

3.3 在多个神经网络上进行测试

图 3 为在模型 MobileNetV2、MobileNetV3、Shufflenet_v2_x1、EfficientNet_B0、Xception 上测试我们的 RVV intrinsic 实现, 通过输入特征图尺寸为[1, 3, 224, 224]时进行推理测试. 可以看到相比标量实现在 MobileNetV2 上有高达 2 倍的性能提升, Xception 上也看到了约 1.35 倍的性能提升, 尽管倍数不及 MobileNetV2, 在时间的消耗上能看到巨大的提升, 在 Xception 提升了约 4 400 ms. 其中提升的效果主要取决于网络模型中 3×3 深度卷积的数量.

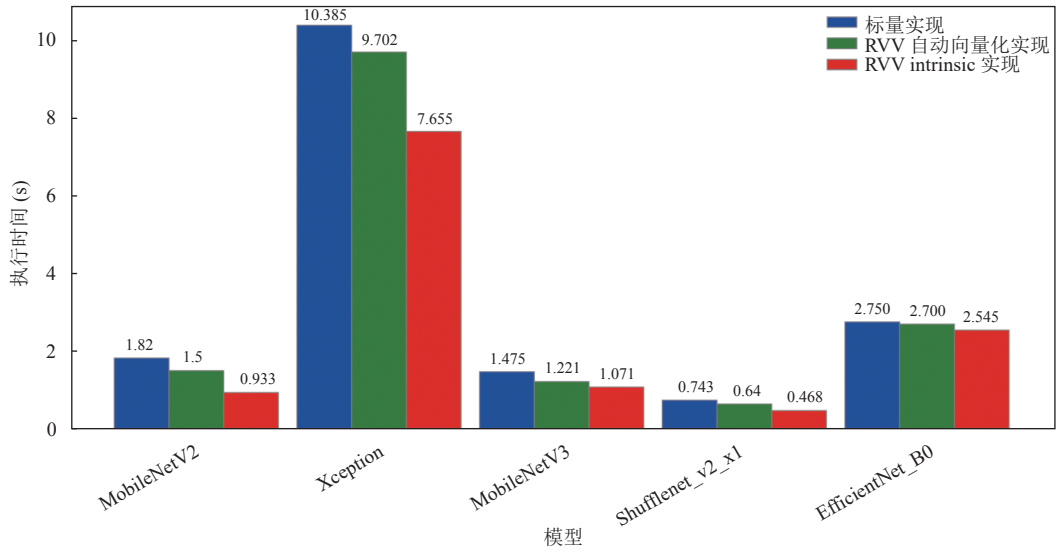

图 3 输入为[1, 3, 224, 224]时不同模型的测试数据对比

从所有模型的执行时间来看, RVV intrinsic 优化实现的性能最优, 其次是自动向量化, 最后是标量实现. 这种趋势与之前的特征图分析一致, 表明 RVV intrinsic 优化显著加速了计算.

Xception 模型的执行时间明显比其他模型长, Xception 是一个计算量较大的模型, 其深度卷积操作复杂度较高, 其他模型的执行时间相对较低, Shufflenet_v2_x1 在 RVV intrinsic 优化实现下的执行时间为 468 ms, 是所有模型中最快的, 其架构设计本身强调高效计算, 特别是在轻量级设备上.

RVV intrinsic 优化带来的加速效果在所有模型上都表现出色, 这种加速效果在计算密集型模型 (如 Xception) 上尤为显著.

尽管自动向量化带来了性能提升, 但与 RVV intrinsic 优化的实现相比, 自动向量化在部分模型上仍有一定差距. 在某些较轻量的模型上 (如 Shufflenet_v2_x1),

自动向量化的表现接近于 RVV intrinsic 优化实现, 但在计算复杂度较高的模型 (如 Xception) 上, 优化的 RVV intrinsic 实现效果更加显著.

标量实现的性能普遍较差, 尤其是在较复杂的模型上, 执行时间远远超过了自动向量化和 RVV intrinsic 优化实现. 这进一步证明, 随着模型计算复杂度的增加, 单纯的标量实现难以充分利用硬件资源.

实验表明, 我们的实现适用于所有包含 3×3 卷积核的深度卷积模型.

3.4 RISC-V RVV 与 ARM_NEON 对比分析

这里将我们的 RVV intrinsic 优化实现与 PyTorch 中的 ARM NEON 实现进行对比分析, 仍然使用[1, 3, 224, 224]的输入特征图, 比较了在不同网络模型上的推理性能差异. ARM NEON 的数据在鲲鹏 920 处理器上进行测试, 通过 taskset 限制使用单核单线程进行推理以与 RISC-V 开发板平行.

我们对 ARM 环境下的实现进行了多项测试, 包括包含完整的第三方库如 NNPACK, 使用 GCC、Clang 多版本编译器以及对 PyTorch 中 ARM NEON 实现的 Winograd 算子的宏定义进行简单的修改, 使得在鲲鹏 920 的 AArch64 架构下能够调用到 PyTorch 本机函数的 Winograd 实现. 测试结果表明 PyTorch 本机函数的 Winograd 实现能够带来最佳的性能, 下面仅使用最佳的测试结果进行分析. 测试结果如图 4 所示.

可以看到, 我们的 RVV intrinsic 优化实现相比 NEON 实现在各个模型上都能看到一定的性能优势. 在 MobileNetV2 、Xception、Shufflenet_v2_x1 上都看到了显著的性能差异, 特别是在 Xception 上最为明显. 这种差距说明我们的 RVV intrinsic 优化实现能够更好地应对大型计算任务.

随后, 我们通过 perf 对推理所花费的指令数进行了对比, 结果如图 5 所示, 可以看到, 在各个模型上 RVV intrinsic 实现相比 NEON 实现总体上使用了更少的指令, 这种减少与执行时间上的差异呈现出一致的趋势. 在 MobileNetV2 模型上, RVV intrinsic 实现的指令数比 NEON 实现低约 35%, 执行时间则缩短了 27%, 表现出显著的指令效率提升, 这个趋势反映了 RVV 对于指令优化的有效性.

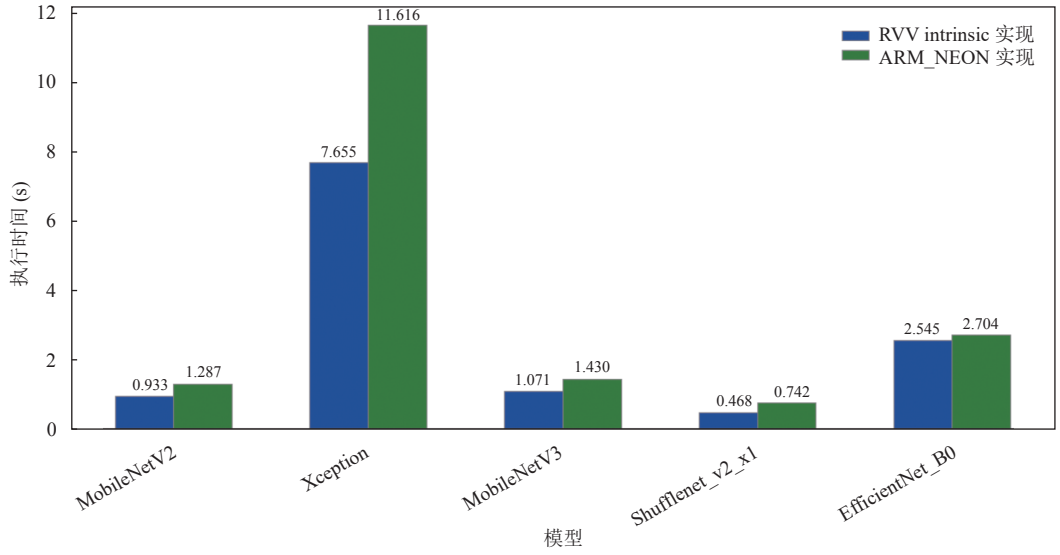

图 4 不同网络模型下 RVV 与 NEON 深度卷积性能对比

此外, 在 Xception 模型上上, RVV 也展示了指令数和执行时间的显著提升, 表明 RVV 对大规模数据计算具备更高的适应性. 这一趋势说明 RVV intrinsic 不仅在复杂模型中具有优势, 在资源受限的环境中亦能发挥其指令效率.

考虑到硬件环境存在差异, 这里的测试数据具有一定的参考价值. 其中 RISC-V 更为精简的指令集设计和模块化设计在其中发挥了重要作用. RVV intrinsic 的优势在于能够利用 RISC-V 架构的灵活性进行向量化扩展, 使得向量处理单元在复杂计算场景下能够更高效地执行批量操作. 而 NEON 实现则受到固定向量宽度的限制, 在处理不同规模的数据时可能需要更多的循环和指令来完成同样的操作, 因此指令数和执行时间都会更高.

以上数据足以说明我们的 RVV intrinsic 优化实现要优于 PyTorch 中的 ARM NEON 实现. 在未来, 随着 RISC-V 硬件的进一步发展, RISC-V 架构上的优化有望带来更大的性能差异.

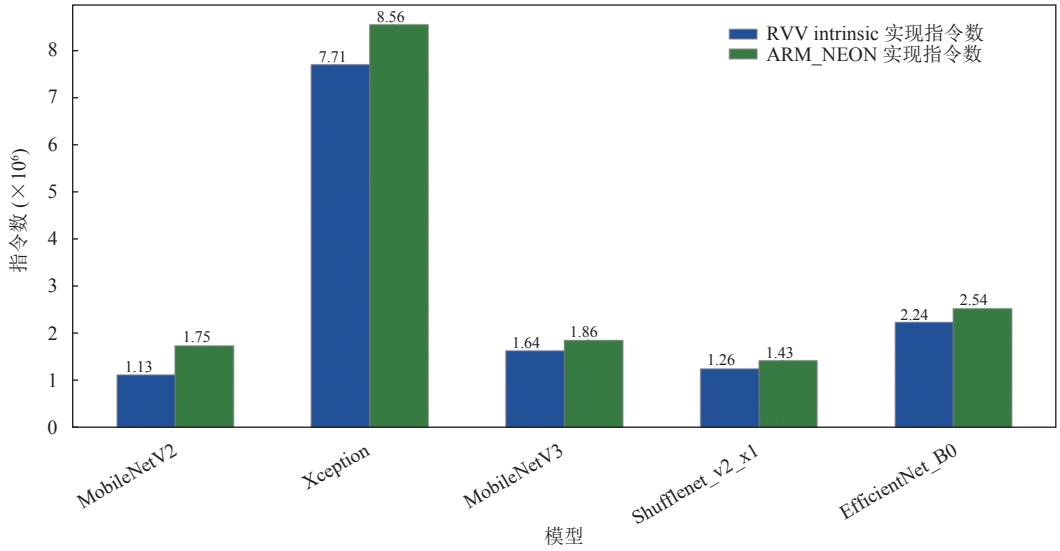

图 5 RVV 与 NEON 实现执行的指令数对比

4 结论与展望

本文提出了一种面向 RVV 1.0 版本的 PyTorch 兼容性开发方案, 同时, 针对深度卷积算子使用 RVV 内建函数进行深度优化, 经过优化的深度卷积算子, 相比 PyTorch 中的标量实现, 可以看到约 1.35–3.8 倍的性能提升. PyTorch 中有大量算子, 还有很多算子可以实现进一步的优化, 在未来的工作中将进一步使用 RVV 对 PyTorch 进行优化, 提升 RISC-V PyTorch 的综合性能. 我们的工作已被 PyTorch 上游主线接收, 可在 https://github.com/pytorch/pytorch/pull/127867 查看相应源码.

参考文献

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

2 Stanescu L, Dinu G. TensorFlow vs. PyTorch in classifying medical images–preliminary results. Proceedings of the 27th International Conference on System Theory, Control and Computing (ICSTCC). Timisoara: IEEE, 2023. 448–453.

3 Ansel J, Yang E, He H, *et al*. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. La Jolla: ACM, 2024. 929–947.

4 Tian JH, Wei Y. An inference performance evaluation of TensorFlow and PyTorch on GPU platform using image super-resolution workloads. Proceedings of the 16th International Conference on Electronics, Computers and Artificial Intelligence (ECAI). Iasi: IEEE, 2024. 1–6.

5 Krisilias A, Provatas N, Koziris N, *et al*. A performance evaluation of distributed deep learning frameworks on CPU clusters using image classification workloads. Proceedings of the 2021 IEEE International Conference on Big Data. Orlando: IEEE, 2021. 3085–3094.

6 Chirodea MC, Novac OC, Novac CM, *et al*. Comparison of TensorFlow and PyTorch in convolutional neural network-based applications. Proceedings of the 13th International Conference on Electronics, Computers and Artificial Intelligence. Pitesti: IEEE, 2021. 1–6.

7 Zhou MH, Hu XW, Xiong W. openEuler: Advancing a hardware and software application ecosystem. IEEE Software, 2022, 39(2): 101–105. [doi: 10.1109/MS.2021.3132138]

8 Bellard F. QEMU, a fast and portable dynamic translator. Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track. Anaheim: USENIX, 2005. 41–46.

9 Lee JKL, Jamieson M, Brown N, *et al*. Test-driving RISC-V vector hardware for HPC. Proceedings of the 38th International Conference on High Performance Computing. Hamburg: Springer, 2023. 419–432.

10 Li JH, Lin JK, Su YC, *et al*. SIMD everywhere optimization from ARM NEON to RISC-V vector extensions. arXiv:2309.16509, 2023.

11 Li RS, Peng P, Shao ZY, *et al*. 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]

12 张飞, 于佳耕, 邢明杰, 等. 基于 musl libc 库的 RVV 优化. 计算机系统应用, 2023, 32(11): 29–35. [doi: 10.15888/j.cnki.csa.009332]

13 Mahale G, Limbasiya T, Aleem MA, *et al*. Optimizations for very long and sparse vector operations on a RISC-V VPU: A work-in-progress. Proceedings of the 38th International Conference on High Performance Computing. Hamburg: Springer, 2023. 472–485.

14 Zhao X, Zhang XY, Zhang YW. Optimization of the FFT algorithm on RISC-V CPUs. Proceedings of the 38th International Conference on High Performance Computing. Hamburg: Springer, 2023. 515–525.

15 Rodrigues A, Sousa L, Ilic A. Performance modelling-driven optimization of RISC-V hardware for efficient SpMV. Proceedings of the 38th International Conference on High Performance Computing. Hamburg: Springer, 2023. 486–499.

16 Igual F, Piñuel L, Catalán S, *et al*. Automatic generation of micro-kernels for performance portability of matrix multiplication on RISC-V vector processors. Proceedings of the 2023 SC Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. Denver: ACM, 2023. 1523–1532.

17 Zou XF, Peng YX, Li T, *et al*. Accelerating Blake3 in RISC-V. Proceedings of the 2nd International Conference on Computing, Communication, Perception and Quantum Technology (CCPQT). Xiamen: IEEE, 2023. 296–301.

18 Rizi MN, Zidaric N, Batina L, *et al*. Optimised AES with RISC-V vector extensions. Proceedings of the 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS). Kielce: IEEE, 2024. 57–60.

19 Vizcaino P, Mantovani F, Ferrer R, *et al*. Acceleration with long vector architectures: Implementation and evaluation of the FFT kernel on NEC SX-Aurora and RISC-V vector extension. Concurrency and Computation: Practice and Experience, 2023, 35(20): e7424. [doi: 10.1002/cpe.7424]

20 van Kempen P, Jones JP, Mueller-Gritschneder D, *et al*. muRISCV-NN: Challenging Zve32x autovectorization with TinyML inference library for RISC-V vector extension. Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions. Ischia: ACM, 2024. 75–78.

21 Gupta SR, Papadopoulou N, Pericàs M. Challenges and opportunities in the co-design of convolutions and RISC-V vector processors. Proceedings of the 2023 SC Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. Denver: ACM, 2023. 1550–1556.

22 牛朝旭, 孙海江. 基于 FPGA 的 Winograd 算法卷积神经网络加速器设计与实现. 液晶与显示, 2023, 38(11): 1521–1530. [doi: 10.37188/CJLCD.2023-0013]

(校对责编: 张重毅)