软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007360] [CSTR: 32375.14.jos.007360]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 RISC-V 向量扩展的高性能算法库优化方法*

韩柳彤[1,2]，张洪滨[1,2]，邢明杰[1]，武延军[1]，赵 琛[1]

[1](中国科学院 软件研究所,北京 100190)

[2](中国科学院大学,北京 100049)

通信作者: 武延军, E-mail: yanjun@iscas.ac.cn

摘 要: 高性能算法库可以通过向量化的方式高效地利用单指令多数据(SIMD)硬件的能力,从而提升其在 CPU 上的执行性能.其中,向量化的实现需要使用目标 SIMD 硬件的特定编程方法,而不同 SIMD 扩展的编程模型和编程方法均存在较大差异.为了避免优化算法在不同平台上的重复实现,提高算法库的可维护性,在高性能算法库的开发过程中通常需要引入硬件抽象层.由于目前主流 SIMD 扩展指令集均被设计为具有固定长度的向量寄存器,多数硬件抽象层也是基于定长向量的硬件特性而设计,无法包含RISC-V向量扩展所引入的可变向量寄存器长度的硬件特性.而若将 RISC-V 向量扩展视作定长向量扩展引入现有硬件抽象层设计中,会产生不必要的开销,造成性能损失.为此,本文提出了一种面向可变长向量扩展平台和固定长度 SIMD 扩展平台的硬件抽象层设计方法.基于此方法,本文重新设计和优化了 OpenCV 算法库中的通用内建函数,使其在兼容现有 SIMD 平台的基础上,更好地支持 RISC-V 向量扩展设备.将采用本文优化方法的 OpenCV 算法库与原版算法库进行性能比较,实验结果表明,运用本方法设计的通用内建函数能够将 RISC-V 向量扩展高效地融入算法库的硬件抽象层优化框架中,并在核心模块中获得 3.93 倍的性能提升,显著优化了高性能算法库在 RISC-V 设备上的执行性能,从而验证了该方法的有效性.此外,本文工作已经开源并被 OpenCV 社区集成到其源代码之中,证明了本文方法的实用性和应用价值.

关键词: RISC-V 向量扩展;数据级并行;算法库优化;开源计算机视觉算法库

中图法分类号: TP311

中文引用格式: 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. http://www.jos.org.cn/1000-9825/7360.htm

英文引用格式: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 (in Chinese). http://www.jos.org.cn/1000-9825/7360.htm

Optimization Method for High-Performance Libraries Targeting RISC-V Vector Extension

HAN Liu-Tong[1,2], ZHANG Hong-Bin[1,2], XING Ming-Jie[1], WU Yan-Jun[1], ZHAO Chen[1]

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

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

Abstract: The performance acceleration of high-performance libraries on CPUs can be achieved by efficiently leveraging SIMD hardware through vectorization. Implementing vectorization depends on programming methods specific to the target SIMD hardware. However, the programming models and methods of different SIMD extensions vary significantly. To avoid redundant implementation of algorithm optimizations across various platforms and improve the maintainability of algorithm libraries, a hardware abstraction layer (HAL) is often introduced in high-performance libraries. Since existing SIMD extension instruction sets are designed with fixed-length vector registers, most hardware abstraction layers only support fixed-length vector types and operations. However, the design of fixed-length vector representations in hardware abstraction layers cannot accommodate the variable vector register lengths introduced by the RISC-V vector extension. Treating RISC-V vector extensions as fixed-length vectors within existing HAL designs would introduce unnecessary overhead and cause performance degradation. To address this problem, the paper proposes a HAL design method compatible with variable-length vector extension platforms and fixed-length SIMD extension platforms. Based on our method, the OpenCV universal intrinsic functions have been redesigned and optimized to support RISC-V vector extension devices better while maintaining compatibility with existing

*基金项目: 中国科学院战略性先导科技专项 (A类) (XDA0320200)

收稿时间:2024-08-26; 修改时间: 2024-10-15, 2024-11-20; 采用时间: 2024-11-26; jos在线出版时间: 2024-12-10

SIMD platforms. Moreover, we designed experiments to compare the performance of the OpenCV library optimized using our approach against the original version. The results demonstrate that the universal intrinsic redesigned by our method efficiently integrates RISC-V vector extensions into the hardware abstraction layer optimization framework. Our method achieved a 3.93x performance improvement in core modules, significantly enhancing the execution performance of the high-performance library on RISC-V devices, thereby validating the effectiveness of this paper. Additionally, our work has been open-sourced and integrated into the OpenCV source code, demonstrating our approach's practicality and application value.

Key words: RISC-V vector extension; data-level parallelism; high-performance library optimization; OpenCV

高性能算法库是专门设计用于执行复杂计算任务并优化其计算效率的软件库,能够在多种硬件平台上高效运行.这种算法库提供高效的算法实现,并通过应用硬件加速和并行计算等方法,减少了开发者从零开始编写复杂算法的需求,显著提高了程序的开发效率和运行性能,在软件生态中的意义重大.高性能算法库在满足计算机视觉应用的高性能计算需求方面也起到了至关重要的作用.如今,计算机视觉应用已经被广泛使用于人类生产生活的诸多领域.这些计算机视觉应用通常涉及多种计算机视觉算法,随着各个应用场景对于多媒体数据处理的实时性需求不断增加,如何提高计算机视觉算法的执行性能已成为该领域的研究热点之一.

计算机视觉算法需要对于输入的多媒体数据进行转换、调整和一系列运算,而这些运算通常需要重复施加于不同的数据上(如视频中的每一帧或图像中的每个通道及像素).基于这样的计算特征,计算机视觉算法的计算负载非常适合使用数据级并行技术来提高计算性能.数据级并行是一种在多个处理单元上同时处理不同数据的技术,随着该技术的发展,图形处理单元(GPU)和单指令多数据(SIMD)体系结构被相继提出,能够极大地提高计算机视觉算法的执行性能.

想要充分利用特定体系结构的硬件能力提高算法的执行性能,通常需要使用与之对应的编程模型.例如,CUDA[1]是 NVIDIA 针对其 GPU 设备提供的并行编程模型,OpenCL[2]则是一个更为开放的标准框架,适用于更多的 GPU 和其他异构设备.而与 GPU 设备不同,SIMD 体系结构更多被设计为通用处理器的多媒体扩展.现今,主流的桌面/服务器处理器和部分嵌入式处理器都拥有 SIMD 扩展,如 Intel x86 架构的 SSE 和 AVX/AVX2/AVX512 指令集扩展[3],ARM 架构中的 Neon 和 SVE/SVE2 指令集扩展[4],以及龙芯 LoongArch 指令集架构的 LSX 和 LASX 指令集扩展[5].RISC-V[6]作为一种新兴的开源精简指令集架构,其向量扩展和压缩 SIMD 扩展也提供了数据级并行处理能力.

不同的指令集扩展具有互不兼容的操作指令和能够发挥各自硬件优势的编程模型.因此,高性能算法库的维护者在面对多种指令集扩展时,通常需要为同一算法编写多个版本的优化实现,以适配不同的硬件特性和优化需求.例如,在图像处理算法的优化实现时,使用 x86 的 AVX512 指令集提供的向量指令编写该算法可以显著提升其处理数据的速度,但这段代码在 ARM 设备、RISC-V 设备、甚至是仅支持 AVX2 的 x86 设备上都无法运行.如果希望该算法能够在不同平台上均获得性能提升,就需要维护同一个算法在多个平台上的不同实现版本.这种方案对于在小型项目中将某几个算法优化到有限的平台上的应用场景而言是可行的,但对于包含大量算法的算法库来说,实现和维护数量庞大的算法到多种平台上会带来组合爆炸的重复实现与代码碎片化的问题,这不仅将消耗大量的资源和时间,还会提高软件的错误率和复杂度.因此,有效组织和优化面向不同平台的代码成为了高性能算法库开发与维护过程中的一个重要挑战.

OpenCV[7]是世界上规模最大的计算机视觉领域高性能算法库,包含诸如滤波、增强、色彩空间转换和卷积等计算密集型算法的高效实现,提供了多种通用的计算机视觉和图像处理功能,被广泛用于对象识别、图像分割、相机校准、运动跟踪等领域.OpenCV 还具备硬件跨平台的能力,力图实现在不同平台设备上的优秀性能.因此,其也面临着上文提到的“多算法-多平台”之间的代码碎片化问题.

为了解决该问题,OpenCV 内部设计了名为通用内建函数(Universal Intrinsic)的硬件抽象层.其通过抽象各个平台的向量类型和向量操作,暴露一套通用内建函数接口,并使用该通用接口编写优化算法,从而解决面向不同处理器平台的代码碎片化问题.使用通用内建函数编写的向量加速代码可以通过编译时调度,在不同平台上调用各平台特定的内建函数,从而实现同一套加速代码的跨平台使用,提高算法库在不同平台上的性能.

目前的 SIMD 指令集扩展大多具有固定长度的向量寄存器,OpenCV 的通用内建函数也基于定长的 SIMD 扩展而设计.然而,RISC-V 的向量扩展[8](RISC-V Vector Extension, RVV)采用了可变长的向量寄存器设计,即对于不同的 RISC-V 向量扩展硬件实现,其向量寄存器长度可以是不同的.这使 RISC-V 向量扩展难以作为 OpenCV 通用内建函数的后端平台,极大地影响了 RISC-V 设备在 OpenCV 算法库中的应用前景.OpenCV 目前将 RISC-V 向量扩展视为固定长度的 SIMD 扩展,从而在兼容现有硬件抽象层设计的情况下添加对 RISC-V 平台的支持,但这反而影响了其在 RISC-V 设备上的性能表现.考虑到 OpenCV 广泛应用于工业和学术界,是许多项目和研究的基础,如果无法在 RISC-V 平台上支持这一重要算法库的关键基础设施,可能会阻碍该架构在研究和开发领域的应用与推广,将在一定程度上限制 RISC-V 软件生态的发展.

针对上述问题,本文提出了一种面向 RISC-V 向量扩展和其他主流 SIMD 扩展的算法库优化方法,并据此为 OpenCV 算法库设计了全新的通用内建函数框架,进而实现了 OpenCV 算法库中通用内建函数的 RISC-V 后端及其他 RISC-V 向量优化.本文将应用了 RISC-V 向量优化的 OpenCV 算法库与原版算法库进行性能比较,实验结果表明本方法可以将 RISC-V 向量扩展引入到现有算法库的优化机制中,从而提升算法库在 RISC-V 设备上的执行性能,验证了方法的有效性.此外,本文所述工作已经开源,并被 OpenCV 社区接受和集成到其源代码中,体现了本方法的实用性和应用价值.

本文第 1 节介绍背景知识和相关工作,分析 RISC-V 向量扩展与其他 SIMD 扩展的核心差异,并讨论 OpenCV 通用内建函数设计的局限性.第 2 节提出面向 RISC-V 向量扩展的算法库优化方法.第 3 节介绍将 RISC-V 向量扩展作为通用内建函数后端的设计挑战,以及适用于可变长向量体系结构的通用内建函数的设计与实现方案.第 4 节通过实验评估上述通用内建函数的执行效率.第 5 节进行总结和展望,并讨论了此方法的局限性.

1 背景知识与相关工作

1.1 算法库的并行优化方法

算法库并行优化方法的设计需要在性能、开发难度和可维护性之间进行权衡.算法库通常专门针对领域特定的计算需求,采用不同的编程策略以实现高性能.编程策略从通用到专用可以分为四类:高级编程语言、特定硬件的通用编程模型、特定硬件的专用编程模型,以及特定硬件的专用内联汇编.

使用高级编程语言进行算法库设计的方法具有较低的开发难度和较强的可维护性.由于这种方法是硬件无关的,其性能表现依赖于编译器的优化能力,例如 GCC 或 LLVM 的自动向量化[9],以及面向领域特定编程语言 Halide[10]的算法实现与编译调度分离策略.尽管其编程方法具有广泛的通用性,但计算负载的实现到硬件指令无法实现最优化映射,通常需要针对特定硬件进行长时间的调优以提高性能.

使用特定硬件的通用编程模型进行算法库设计的方法可以在性能、开发难度和可维护性之间达到相对平衡.这类编程模型在高级编程语言的基础上提供硬件抽象层,使用此抽象层接口来实现领域特定的计算负载.具体实现分为算法库集成的编程接口和独立的硬件编程库.其中,算法库集成的编程接口为特定算法库服务,接口的抽象级别和颗粒度可专门为算法库所面向的领域设计.例如,OpenCV 的通用内建函数(Universal Intrinsic[11])面向图像处理领域设计了统一的高性能编程接口,集成了各种平台特定的 SIMD 及向量计算操作.而如 C++ SIMD 库[12]及 Google Highway[13]等独立的硬件编程库提供了较为全面的硬件抽象,但其面向通用计算需求抽象出的向量操作语义并不完全适合于特定领域的计算需求.此外,在大型开源软件中引入额外的开源第三方库还可能存在许可证不兼容或上游开源软件供应链“投毒”的风险[14].因此,第三方的硬件编程库在高性能算法库的跨平台性能优化中的应用也较为有限.

使用特定硬件专用编程模型进行算法库设计的方法拥有性能优势,被高性能算法库广泛采用.但该方法需要单独对每一种硬件架构进行优化算法的实现与维护,存在重复开发的问题,降低了算法库的开发效率和可维护性.特定硬件平台的专用编程模型针对高性能硬件指令提供了高级语言层面的编程语法和语义,允许开发者为硬件平台实现特定的优化算法.例如图像处理领域的 libjpeg-turbo[15],Intel Performance Primitives (IPP) 库

等,由于其实现方式依赖于平台特定内建函数,在添加 RISC-V 向量扩展支持时需要大量的移植,并且从实践经验来看,增加一种平台的内建函数支持会使整个算法库的维护难度骤增.因此,对可扩展性更强的 RISC-V 指令集生态来说,使用特定硬件的专用编程模型进行算法库的优化并不利于 RISC-V 软硬件生态的健康发展.

使用特定硬件专用内联汇编进行算法库设计的方法可以达到极致的性能表现,但这也带来了极高的开发和维护难度.这种方法在一些算法数量少且迭代升级周期长的优化场景中具备一定的可行性,例如 OpenSSL 中对于加密算法就大量采用内联汇编的方法追求极致的性能表现.此外,一些领域特定的硬件加速器也使用内联汇编作为其软件栈提供软硬件接口,例如 RISC-V 生态中的 Gemmini[16] 加速器通过内联汇编实现深度学习的算子库.这种方法虽在特定场景下有优势,但当算法库的规模较大或有更新与重构需求时,其开发和维护成本可能变得难以承受,因此其应用场景十分有限.

1.2 RISC-V 向量扩展及其特性

RISC-V 是一种新兴的开源精简指令集架构,其模块化的特性使得 RISC-V 架构能够更好地适应不同场景和需求下的处理器设计.RISC-V 向量扩展是指令集的可选模块之一,旨在提供数据级并行能力,从而大幅提升数据处理效率.此扩展定义了一组向量指令及相关寄存器,从而允许程序使用向量运算部件对大量数据进行并行处理.李若时[17]等人评估了 RISC-V 向量扩展在提升计算机视觉算法效率方面的效果,指出性能提升可达 2.98 倍,展现了 RISC-V 向量扩展在高性能计算领域的应用前景.相较于其他支持数据级并行的指令集架构,RISC-V 向量扩展具有如下特性:

RISC-V 向量扩展的一个核心特性是向量长度不可知(Vector Length Agnostic, VLA).这意味着寄存器的长度可以根据不同硬件实现的需要而有所不同.根据 RISC-V 向量扩展的标准规范,寄存器长度(VLEN)可以是 32 至 65536 之间任意 2 的幂.这种灵活的寄存器长度设计允许不同的硬件实现根据其需求和目标应用来选择最适合的向量长度,也使不同长度的向量寄存器(如 128 位和 1024 位)能够共享相同的指令集,并实现完全一致的功能,无需进行代码移植.例如,一个处理 2048 个 32 位元素的程序,在 128 位的向量平台上每次处理 4 个元素,需循环 512 次;而在 256 位平台上,则每次可处理 8 个元素,只需循环 256 次.因此,正确编写的 RISC-V 向量扩展程序可以在不同寄存器长度的硬件设备上都获得最佳性能,而避免产生硬件资源浪费.

另一个特性是寄存器分组,通过设置相关参数将多个向量寄存器组成寄存器组.依据 RISC-V 向量扩展规范,每个处理器核心配备 32 个向量寄存器,每个向量寄存器具有 VLEN 长度.规范还允许将 2/4/8 个寄存器视为一组使用,从而使得指令可以操作相应倍数 VLEN 长度的寄存器组,此时可用的寄存器数量也相应减少.采用适当的寄存器组策略可以使单条向量指令处理更多数据,节约指令条数并在循环结构中减少循环迭代次数,进而在数据量大的可并行程序中获得一定的性能提升.

第三个特性是具有向量长度寄存器,它保存一个无符号整数,指定向量指令操作的元素数量.在编程模型中,此寄存器常用于尾循环处理,即处理数据结尾的少量元素以防止内存越界.与传统 SIMD 运算器单次处理固定数量的数据不同,RISC-V 向量扩展允许通过控制指令设置向量长度寄存器,从而动态调整循环迭代中向量操作处理的数据数量.这样,在处理尾循环时,向量长度可以精确设置为剩余元素数量,确保加载、运算和存储操作的边界安全,而无需使用向量掩码或添加额外的标量循环,从而更高效地实现算法功能.

2 面向 RISC-V 向量扩展的算法库优化设计方法

算法的性能优化可以通过使用特定平台的内建函数来实现,而算法库的性能优化则往往需要引入统一的硬件抽象层,从而提高算法优化的通用性,避免“多算法-多平台”之间的组合爆炸引发的代码碎片化问题.RISC-V 向量扩展与传统 SIMD 扩展的架构差异为硬件抽象层的设计带来了新的挑战,本文基于现有传统 SIMD 扩展架构的硬件抽象,结合 RISC-V 向量扩展架构的特点,提出了面向 RISC-V 向量扩展的高性能算法库优化方法.下面将具体阐述该优化技术的设计方法.

2.1 硬件抽象层设计思路

硬件抽象层的设计目标是为编程人员提供统一的硬件编程接口,帮助其更简便地利用不同硬件的计算能力,而无需关注底层硬件的实现细节和特定编程模型,从而在提高程序性能的同时保证程序的开发效率和代码的可维护性.

为此,硬件抽象层的设计应当从目标硬件的角度出发,尽可能覆盖被抽象的目标硬件集合的共性特征.众所周知,包括 RISC-V 向量扩展在内的所有 SIMD 扩展都提供了对一组数据执行相同操作的计算能力.因此,SIMD 扩展的硬件抽象应当以向量操作和数据类型为核心,抽取目标硬件集合共同支持的向量操作与类型作为抽象目标,从而提高应用程序的并行处理效率.

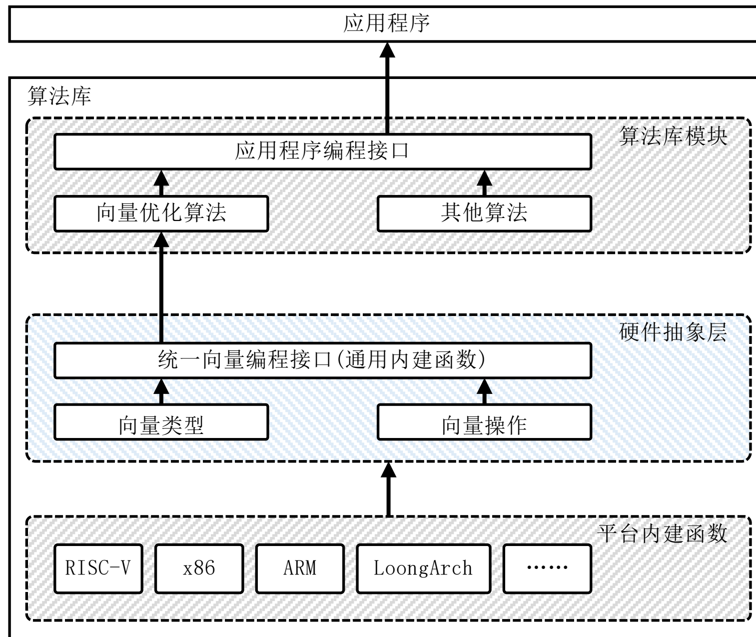


图 1 基于硬件抽象层的算法库向量优化方法示意图

如图 1 所示,在高级语言层面,不同硬件平台都提供了各自的内建函数,从而允许开发人员更好地调用特殊硬件能力;硬件抽象层应当选取适当的向量类型和向量操作,组合形成一套面向算法库计算需求的统一向量编程接口;算法库的各模块应当使用统一向量编程接口实现具有向量优化机会的算法,通过应用程序编程接口(API)对外部应用程序提供调用,从而实现性能提升.

为了能够将通用内建函数分别映射到特定平台的内建函数上,硬件抽象层的设计和实现应当以条件编译机制为基础.条件编译是一种允许在编译阶段根据预定义的条件来包含或排除代码段落的编译技术,这种机制通常在编译器中采用预处理指令或特定的编译构造实现,被广大着眼于性能的编程语言所支持.硬件抽象层为不同目标平台的指令集扩展定义不同的条件属性,从而在不同平台条件下使用特定平台的内建函数实现硬件抽象层的统一向量编程接口.在算法库的构建过程中,应当首先检测目标平台的扩展指令集支持情况,并据此设置条件属性,进而引导编译系统仅包含当前受支持平台扩展指令集的接口实现,从而将硬件抽象层的向量操作和向量类型映射到特定平台的内建函数上.此外,编译器的指令调度过程可以依据不同体系结构对映射后的向量指令进行合理的排布,最终达到通用内建函数能够映射到特定平台,并在各个平台上均获得性能提升的目标.

2.2 向量操作抽象

SIMD 扩展支持的向量操作在语义层面上具有一定共性,是硬件抽象层中抽象向量操作的设计基础.尽管不同的 SIMD 指令集在指令的定义和数量上有很大差异,且在功能上也由于硬件特性的差异而有所不同,但都支持一组常用的基本向量操作,现代主流 SIMD 扩展指令集均支持以下类型的操作:

- 访存操作:包含向量加载和存储操作,实现向量寄存器和内存之间的数据移动;
- 算术与逻辑运算操作:包含算术、逻辑、比较、移位等运算操作;
- 归约操作:包含向量元素求和、求极值等操作,将向量类型归约为标量类型;
- 排列操作:包含向量合并、移动、分散、聚合、压缩等操作,允许以一定规则重新排布向量内元素;
- 类型转换操作:包含向量展宽、缩窄、类型转换等操作,允许向量元素在不同数据类型和向量宽度之间进行转换.

硬件抽象层应当根据目标硬件集合的向量操作支持情况抽象出共性操作.例如,对于单精度浮点数的向量加载和存储操作,RISC-V 向量扩展和 ARM Neon 扩展在高级语言层面以内建函数的形式分别提供了 __riscv_vle32_f32m1(...) / __riscv_vse32(...)函数和 vld1q_f32(...) / vst1q_f32(...)函数,硬件抽象层可以提供形如 v_load(...) / v_store(...) 的统一抽象;类似的,还可以抽象出共性的运算操作:形如 v_add(...)的函数可以用于抽象向量加法,v_mul(...)可以用于抽象向量乘法等.

此外,硬件抽象层还应考虑算法库所涉及的领域需求和计算特征,以领域内高频操作为抽象目标,确保抽象层能够适应领域特定的优化需求.例如,在图像处理或机器学习应用中,乘积累加运算是矩阵乘算法中的高频操作,可以成为抽象的重点,以确保这些计算密集型任务能够高效执行,从而匹配领域需求,便于开发人员使用:v_fma(...)作为描述乘积累加操作通用内建函数供开发人员调用,抽象了 RISC-V 向量扩展的 __riscv_vfmacc(...) 以及其他平台的相关操作.

需要注意的是,领域需求的抽象应当以硬件所能提供的共性计算能力为基础,不应抽象于过高的层级,也不宜抽象仅少数硬件平台支持的特殊操作.例如,卷积算法是机器学习应用中的常用操作,但其算法实现复杂多样,包含多个基本向量操作,具有较高的抽象层级,鲜有硬件设备直接提供卷积操作,因此不宜包含在硬件抽象层中;指数与对数运算的抽象层级较低,但在主流 SIMD 指令集扩展中,仅有 AVX-512 提供了对指数和对数运算的部分支持,而综合考虑其他硬件平台对相关操作的直接支持有限,对此类特殊运算的抽象应当谨慎考量:本文建议仅在应用场景确有相关操作的抽象需求,且其他硬件平台可以通过组合使用基本操作以可接受的开销实现特殊运算时,才考虑将其纳入硬件抽象层中.

总的来说,硬件抽象层所包含的向量操作应当从应用场景计算需求和目标硬件平台集合两个方面综合考虑,以高频计算场景中的热点操作为抽象目标,以所有预期支持的硬件平台所提供的共性操作为基础.剥离高抽象层次的复杂计算,规避个别平台提供的特殊操作,从而定义一套满足算法库需求,且广泛适应于目标硬件平台的硬件抽象层向量操作.

2.3 向量类型抽象

在硬件抽象层中,除了对向量操作的抽象,还包括对于向量数据类型的抽象.向量数据类型通常用于描述一组具有相同类型的标量数据元素.主流 SIMD 扩展指令集都具有较为完善的整数和常用浮点数类型,表 1 列举了部分 SIMD 扩展所支持的标量数据类型.对于近年来在深度学习等领域中关注度较高的半精度浮点数和 BFloat16 类型,RISC-V 和 x86 都在向量扩展之外提供了扩展指令集,其支持情况取决于微架构实现,ARM Neon 则随着处理器架构版本的演进而引入了对新型数据类型的支持.

表 1 部分 SIMD 扩展指令集所支持的数据类型

标量数据类型	RISC-V Vector (RVA23U64)	x86 AVX2/AVX512	ARM Neon
BFloat16	ZVBFMIN 扩展 仅支持类型转换操作	AVX-512-BF16 扩展	ARMv8.6-A 及其以上
16 位半精度浮点数	ZVFH 扩展	AVX512-FP16 扩展	ARMv8.2-A 及其以上
32 位单精度浮点数	√	√	√
64 位双精度浮点数	√	√	ARMv8 及其以上
8 位整数	√	√	√
16 位整数	√	√	√
32 位整数	√	√	√
64 位整数	√	√	ARMv8 及其以上

然而,尽管 SIMD 扩展支持的标量数据类型具有共性,但在各种指令集架构上向量寄存器长度的不同导致其向量数据类型差异较大.以各指令集架构在C语言中的向量类型中为例,对于32位单精度浮点数:在向量寄存器长度为256位的AVX2指令集扩展中,其向量数据类型是“__m256”,表示包含8个单精度浮点数的向量寄存器;在寄存器长度为128位的ARM Neon指令集扩展中,其向量数据类型是“float32x4_t”,表示包含4个单精度浮点数的向量寄存器;RISC-V 向量扩展较为特殊,在指令集架构层面没有明确向量寄存器长度,且支持将多个寄存器视为一组,故其向量数据类型的表达形式是“vfloat32m<LMUL>_t”,表示由“LMUL”个物理向量寄存器组成的一组包含单精度浮点数的向量寄存器,且包含在其中的浮点数元素数量是不确定的.

硬件抽象层需要抽象不同平台上向量类型的差异,才能为开发者提供通用的编程接口.因此,当目标硬件集合的向量寄存器长度不一致时,硬件抽象层应当仅抽象出多种向量类型中的共性部分,即其包含的标量元素类型,而舍去元素个数等有差异的信息.在上述单精度浮点数例子中,抽象出的通用数据类型应当类似于“v_float32”,其仅表示一个包含单精度浮点数元素的向量,而该向量中的元素数量,底层映射的物理寄存器个数等信息均被隐去.

a) 标量实现
```
void saxpy(size_t n, const float a, const float *x, float *y) {
   for (size_t i = 0; i < n; ++i) {
      y[i] = a * x[i] + y[i];                          // y = a * x + y 的标量计算
   }                                                   // 循环展开后具有向量优化机会
}
```
b) 使用 RISC-V Vector 内建函数的向量化实现
```
void saxpy(size_t n, const float a, const float *x, float *y) {
   size_t vl = __riscv_vsetvlmax_e32m1();              // 设置状态寄存器并获取向量寄存器内最大元素个数
   vfloat32m1_t vx, vy;                                // 声明 RVV 内建向量数据类型
   for (size_t i = 0; i < n; i += vl) {                // 循环展开 vl 次,每轮迭代处理 vl 个元素
      vx = __riscv_vle32_f32m1(x+i, vl);               // 向量加载,x
      vy = __riscv_vle32_f32m1(y+i, vl);               // 向量加载,y
      vy = __riscv_vmacc(vy, a, vx, vl);               // 向量乘加运算(RVV 允许向量与标量直接运算,无需广播)
      __riscv_vse32(y+i, vy);                          // 向量存储,y
   }
}
```
c) 使用 ARM NEON 内建函数的向量化实现
```
void saxpy(size_t n, const float a, const float *x, float *y) {
   float32x4_t vx, vy;                                 // 声明 ARM NEON 内建向量数据类型
   float32x4_t va = vdupq_n_f32(a);                    // 广播标量 a 到向量
   for (int i = 0; i < n; i += 4) {                    // 循环展开 4 次,每轮迭代处理 4 个元素
      vx = vld1q_f32(x+i);                             // 向量加载,x
      vy = vld1q_f32(y+i);                             // 向量加载,y
      vy = vmlaq_s32(vy, va, vx));                     // 向量乘加运算
      vst1q_f32(y, vy);                                // 向量存储,y
   }
}
```
d) 使用 x86 AVX2 内建函数的向量化实现
```
void saxpy(size_t n, const float a, const float *x, float *y) {
   __m256 vx, vy;                                      // 声明 x86 AVX2 内建向量数据类型
   __m256 va = _mm256_set1_ps(a);                      // 广播标量 a 到向量
   for (int i = 0; i < n; i += 8) {                    // 循环展开 8 次,每轮迭代处理 8 个元素
      vx = _mm256_loadu_ps(x+i);                       // 向量加载,x
      vy = _mm256_loadu_ps(y+i);                       // 向量加载,y
      vy = _mm256_add_ps(vy, _mm256_mul_ps(va, vx));   // 向量乘加运算
      _mm256_storeu_ps(y, vy);                         // 向量存储,y
   }
}
```
e) 使用通用内建函数的向量化实现
```
void saxpy(size_t n, const float a, const float *x, float *y) {
   v_float32 vx, vy;                                   // 声明通用内建向量数据类型
   v_float32 va = v_setall_f32(a);                     // 广播标量 a 到向量
   size_t N = v_float32::nlanes;                       // 调用接口获取通用类型包含的元素个数
   for (int i = 0; i < n; i += N) {                    // 循环展开 N 次,每轮迭代处理 N 个元素
      vx = v_load(x+i);                                // 向量加载,x
      vy = v_load(y+i);                                // 向量加载,y
      vy = v_add(vy, v_mul(vx, vy));                   // 向量乘加运算
      v_store(y, vy);                                  // 向量存储,y
   }
}
```
图 2 不同平台的 Saxpy 算法向量化实现对比 (不考虑尾循环处理)

值得注意的是,向量数据类型包含的标量元素类型和元素个数是向量编程模型中的重要信息,可以视作向

量类型的元数据.如果无法通过向量数据类型本身直接得到,则应当提供相应接口,允许开发人员获取相关元数据.图 2 以单精度浮点数乘加(Single-Precision A·X Plus Y, saxpy)算法为例,说明向量类型元数据接口的必要性和设计要点.不难看出,对于通过循环展开获得向量化机会的算法而言,向量数据类型包含的标量元素个数即为循环展开次数(或每次迭代处理的元素个数).如图 2 中黄色标记所示,ARM Neon 和 x86 AVX2 的向量类型分别包含 4 个和 8 个单精度浮点数,而 RISC-V 向量扩展中则通过相关内建函数在运行时获得元素个数,上述信息均被用作迭代步长.因此,通用内建函数的向量类型应当根据编程语言特性,以恰当形式提供类型相关的元数据,如图 2 e) 所示,元数据(元素个数)作为类型成员,可以被开发人员获取并使用.

3 面向 RISC-V 向量扩展的硬件抽象层实现方法

为了解决大量算法面向不同平台加速硬件的内建函数编程时面临的重复实现问题,OpenCV 实现了名为通用内建函数的硬件抽象层.在 OpenCV 算法库中,所有具备向量化机会的算法都可以使用通用内建函数的编程接口获得性能提升.然而,现有的通用内建函数仅为固定长度的 SIMD 后端架构设计,与 RISC-V 向量扩展的可变长度向量寄存器设计并不兼容.

本节首先介绍 OpenCV 通用内建函数的设计,之后讨论现有设计与可变长体系结构不兼容的原因,分析将 RISC-V 向量扩展视为定长 SIMD 后端实现通用内建函数面临的问题,最终给出面向 RISC-V 向量扩展的计算机视觉算法库硬件抽象层实现方法.

3.1 OpenCV的通用内建函数实现

OpenCV 的通用内建函数抽象了诸多平台的向量操作和类型,其支持的平台包括 Intel x86 SSE/AVX 系列扩展指令集、ARM Neon 扩展指令集、龙芯 LoongArch LSX 系列扩展指令集、MIPS MSA 扩展指令集和 Power VSX 指令集扩展等.通用内建函数使用上述指令集扩展在高级语言层面提供的内建函数,面向各平台分别实现统一的向量类型及操作.

3.1.1 向量类型

OpenCV 的硬件抽象层的向量类型覆盖了几乎所有常用的基本数据类型,并根据不同平台的向量寄存器长度分别封装,形成向量数据类型.如表 2 所示,在 OpenCV 的发展历史上,硬件抽象层最初被设计为面向 128 位固定长度的向量扩展,从而支持 SSE、MSA 和 VSX 扩展指令集;而随着计算机体系结构的发展,具有更强数据吞吐和处理能力的 AVX2、AVX-512 扩展指令集被提出,OpenCV 硬件抽象层也随之封装了 256 位和 512 位的向量类型,并通过未知长度(size-agnostic)向量类型提供进一步的统一抽象.

表 2 OpenCV 的通用内建函数类型

标量类型	128 位向量类型	256 位向量类型	512 位向量类型	未知长度向量类型
unsigned char	v_uint8x16	v_uint8x32	v_uint8x64	v_uint8
char	v_int8x16	v_int8x32	v_int8x64	v_int8
unsigned short	v_uint16x8	v_uint16x16	v_uint16x32	v_uint16
short	v_int16x8	v_int16x16	v_int16x32	v_int16
unsigned int	v_uint32x4	v_uint32x8	v_uint32x16	v_uint32
int	v_int32x4	v_int32x8	v_int32x16	v_int32
unsigned long int	v_uint64x2	v_uint64x4	v_uint64x8	v_uint64
long int	v_int64x2	v_int64x4	v_int64x8	v_int64
_Float16	v_float16x8	v_float16x16	v_float16x32	v_float16
float	v_float32x4	v_float32x8	v_float32x16	v_float32
double	v_float64x2	v_float64x4	v_float64x8	v_float64

通用向量类型不仅封装了不同平台内建函数的向量类型,还包含向量元数据,如向量所包含的元素类型和元素个数,以类型静态成员的形式对外提供调用接口.编程人员可以使用 <VecType>::lane_type 和 <VecType>::nlanes 获得向量类型对应的元数据.

3.1.2 向量操作

OpenCV 的通用内建函数根据目标硬件平台所提供的向量计算能力和计算机视觉领域的计算特征,提供了

七类向量操作的抽象,以函数调用形式对外提供接口:

- 向量初始化与内存读写操作:v_setall, v_setzero, v_load, v_load_aligned, v_load_low, v_load_halves, v_load_expand, v_load_expand_q, v_store, v_store_aligned, v_store_high, v_store_low 等;
- 算术、逻辑、比较与位运算操作:v_min, v_max, v_mul_expand, v_shl, v_shr 和向量类型的重载运算符(+, -, *, /, <<, >>, &, | ,^, ~, >, >=, <, <=, ==, !=)等;
- 归约与掩码操作:v_reduce_min, v_reduce_max, v_reduce_sum, v_popcount, v_signmask, v_check_all, v_check_any, v_select 等;
- 元素重排序操作:v_load_deinterleave, v_store_interleave, v_expand, v_expand_low, v_expand_high, v_pack, v_zip, v_recombine, v_combine_low, v_combine_high, v_reverse, v_extract 等;
- 类型转换与舍入操作:v_round, v_floor, v_ceil, v_trunc, v_cvt_f32, v_cvt_f64 v_reinterpret_as_* 等
- 矩阵运算操作:v_dotprod, v_dotprod_fast, v_dotprod_expand, v_dotprod_expand_fast, v_matmul, v_transpose4x4 等;
- 其他计算机视觉领域的数学运算操作:v_sqrt, v_invsqrt, v_magnitude, v_sqr_magnitude, v_abs, v_absdiff, v_absdiffs 等.

值得注意的是,由于目标硬件平台集合对于不同操作的支持情况存在差异,并非所有操作对于每种向量类型均有实现,部分向量操作可能仅支持某一个或多个向量类型.

3.1.3 通用内建函数到特定平台内建函数的映射

图 3 展示了通用内建函数映射到各特定平台内建函数的流程.在编译时,构建工具会执行目标平台检查,其分别尝试构建特定平台的测试程序,如果能够成功构建,则在后续编译流程中启用对应的条件宏(如 RISC-V 向量扩展平台上会启用 CV_RVV),并将向量类型和向量操作通过各个后端对于硬件抽象层接口的实现映射到特定平台,完成通用内建函数到平台内建函数的转换过程.

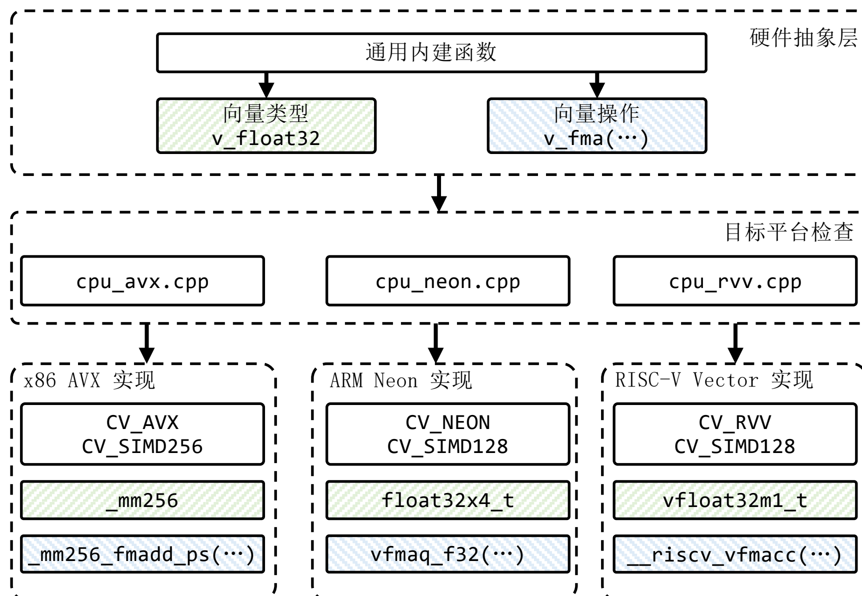


图 3 通用内建函数到平台特定内建函数的映射过程示意图

3.2 RISC-V向量扩展与通用内建函数的兼容性

RISC-V 向量扩展与其他 SIMD 扩展的核心差异之一在于其采用了可变的向量寄存器长度.具体而言,与 SSE 定义了 128 位的向量寄存器、AVX2 定义了 256 位的向量寄存器不同的是,RISC-V 向量扩展中的向量寄存器长度在指令集层面未被确定,不同硬件的微架构被允许根据其应用场景实现不同的寄存器长度.与之类似

的,在 ARM 平台上,SVE(Scalable Vector Extensions)及 SVE2 也采用了类似的可扩展向量机制.

OpenCV 曾试图将 RISC-V 向量扩展视为固定长度的 SIMD 扩展,从而兼容现有硬件抽象层的设计.考虑到对于应用处理器,RISC-V 向量扩展的最小向量寄存器长度为 128 位,OpenCV 将其作为该平台上的向量寄存器标准.显而易见的,以可变长向量扩展的最小实现作为标准将不可避免的浪费硬件寄存器资源:尽管指令能够在拥有不同向量寄存器长度的 RISC-V 设备上正确执行,但在拥有更长向量寄存器长度的硬件设备上,向量寄存器中高于 128 位的部分将不会被使用,无法充分利用硬件资源,影响其性能表现.

更为棘手的是,RISC-V 向量扩展的向量寄存器长度在编译时是未知的,存储其向量类型所需的空间也就无法确定,这种类型被称为“无大小”(Sizeless)类型.此种类型在实际使用中通常受到一定限制:由于实例化对象或确定栈帧布局过程中,编译器需要准确计算出类的大小和类成员的偏移量,而 RISC-V 向量扩展的内建类型大小无法在编译时确定.因此,类或结构体内不能声明这些类型的成员变量,这与 OpenCV 硬件抽象层的向量类型封装机制不兼容.OpenCV 目前采用的替代方案是在类型封装中使用内存中的数组来存放数据,同时增加 RISC-V 向量类型和通用内建函数向量类型之间的转换函数,一定程度上解决了 RISC-V 向量类型与包装类型不兼容的问题.但两种类型之间的转换不可避免的使用访存指令实现数组和寄存器的数据交换,因而产生了冗余的访存指令,如表 3 中“OpenCV 原有通用内建函数实现”部分的灰色阴影部分所示,在使用通用内建函数时,每个 RISC-V 向量类型(数据在向量寄存器中)到通用向量类型(数据在内存数组中)都需要执行一次显式类型转换,进而产生一条冗余的内存写入指令(表中 17、21、27 行);该对象需要再作为向量类型被使用时,又会产生一次隐式的由通用向量类型到 RISC-V 向量类型的类型转换,进而产生一条冗余的内存读取指令(表中 24、25、30 行);此外,如果在使用时将通用向量类型的对象进行了赋值,还会额外产生内存数组的拷贝操作(表中 18、19、22、23、28、29 行).由于访存指令的开销通常较大,这些冗余操作极大影响了执行性能,甚至抵消了向量优化带来的性能提升,以至于通用内建函数中固定长度版本的 RISC-V 向量扩展实现的性能表现还低于未经任何向量优化的标量版本,不具备应用价值.

上述问题均由 RISC-V 向量扩展中向量寄存器可变长的特性而引发,可以预见同样具有可变寄存器长度特性的 ARM SVE/SVE2 扩展也将遇到类似问题.究其根本,是由传统定长 SIMD 扩展发展演化而来的 OpenCV 硬件抽象层设计无法适应新兴的可变长体系结构导致的.

3.3 面向RISC-V向量扩展的通用内建函数优化实现

将 RISC-V 向量扩展视作固定长度 SIMD 扩展将造成寄存器资源浪费,并产生冗余访存指令,没有产生应有的向量优化效果.因此,本文使用所提出的面向 RISC-V 向量扩展的硬件抽象层设计方法,优化了 OpenCV 的通用内建函数的设计与实现.新的通用内建函数可以兼容 RISC-V 向量扩展的可变长特性,并使用 RISC-V 向量扩展的平台内建函数实现了其编程接口,从而使得通用内建函数所编写的优化算法能够在拥有 RISC-V 向量扩展的硬件设备上获得性能提升.

与 OpenCV 现有定长的通用内建函数设计方案相比,本文方法的主要差异如图 4 所示.首先,在通用内建函数的类型封装部分,不再引入包装类,而是直接采用类型别名的方式将 RISC-V 向量扩展的内建向量数据类型映射为通用内建函数所定义的向量数据类型.使用类型别名省去了包装类型到内建类型的转换,能够避免产生冗余访存指令,因此更为高效;其次,为了承载向量类型的元数据,引入了特征类.这是因为在定长的通用内建函数设计中,向量类型元数据作为类成员被包含在包装类中,而本文提出的设计方案不再使用包装类型,引入新的特征类承载向量类型的元数据并提供访问接口是必要的;此外,在函数封装方面,引入了新的函数接口定义,代替现有实现中依赖于包装类的运算符重载函数;最后,为其他平台新引入了兼容层,使新引入的特征类和通用内建函数设计可以顺利映射到其他平台已有的实现上,并面向开发人员提供一致的通用内建函数编程接口.

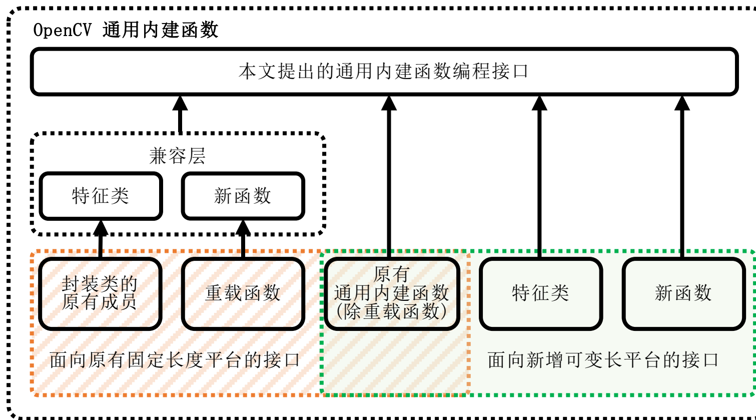


图 4 面向可变长体系结构的通用内建函数编程接口设计

尽管本文的设计方案与现有方案在功能上保持一致,但也存在上述由取消包装类而引入的诸多接口变更,导致新的通用内建函数编程接口无法向前兼容.因此,现有使用通用内建函数所编写的优化代码需要进行部分改写从而迁移到新编程接口,我们开发了针对相关接口变更的自动化迁移工具,并使用新编程接口重构了OpenCV 代码仓库中 90.6%的通用内建函数代码块,从而使大部分现有的向量优化算法实现得以在 RISC-V 设备上启用,且后续使用新编程接口实现的算法优化将能在 RISC-V 平台和其他平台上同步获得性能提升,有效降低了算法库的开发复杂度.

3.3.1 利用类型别名消除冗余指令

为了提供统一的编程接口,OpenCV 通用内建函数使用包装类的方式封装了不同向量后端的内建向量类型.然而,由于 C++语言的限制,可变长的 RISC-V 向量扩展的内建向量类型由于缺少编译时的长度信息而无法被包含在任何类型中.可选的代替方案是使用数组来存放数据,同时增加 RISC-V 向量类型和 OpenCV 通用内建函数向量类型之间的转换函数,该方案在一定程度上解决了 RISC-V 向量类型与包装类型不兼容的问题.但两种类型之间的转换不可避免的使用访存指令实现数组和寄存器的数据交换,在使用通用内建函数编程时,每条指令都会引入两次冗余访存指令,极大影响执行性能.

因此,本文提出了使用类型别名代替包装类的方案,即将 OpenCV 通用内建函数向量类型直接作为 RISC-V 向量类型的别名使用.在使用通用内建函数的向量类型时,本质上是在使用 RISC-V 内建的向量类型,因此省去了类型转换与对象拷贝的开销.如表 3 所示,应用本文方法的 OpenCV 通用内建函数实现所生成的汇编指令中仅包含通用内建函数所述语义对应向量指令(表中以不同颜色高亮)与其他必要指令,解决了原有实现中产生冗余访存指令问题.

3.3.2 特征类、函数封装与兼容性

在 OpenCV 通用内建函数原有的类型封装中,包含了标量元素个数和元素类型等类型元数据.由于本文所提出的新实现方法使用类型别名代替了类型封装,且类型元数据无法作为 RISC-V 内建数据类型的成员存在,因此只能引入新的类型封装元数据,称之为特征类.在特征类中,包含了用于表述向量类型中标量元素类型的成员 lane_type 和表述向量内元素个数的成员函数 vlanes.特别地,与其他 SIMD 后端中对函数 vlanes 的调用会返回固定长度不同,RISC-V 向量后端中 vlanes 通过相应指令在运行时获得向量寄存器所支持的最大元素个数,从而实现对 RISC-V 向量扩展的可变向量寄存器长度特性的支持.

类似的,依托于原有的类型封装,OpenCV 通用内建函数还实现了部分运算符重载,这在使用类型别名的 RISC-V 后端中也由于语言限制无法实现.对于这类函数,本文引入了新的通用内建函数,实现相同的功能.例如其他后端中使用重载运算符“+”实现向量加法,在 RISC-V 后端中则使用新的通用内建函数“v_add”实现,代替运算符重载.

表 3 应用不用方法实现 OpenCV 通用内建函数所生成的 RISC-V 向量汇编指令对比(以 saxpy 算法为例)

行	应用本文方法的 OpenCV 通用内建函数实现	OpenCV 原有通用内建函数实现
1	saxpy(unsigned long, float, float const*, float*):	saxpy(unsigned long, float, float const*, float*):
2	lui a5,%hi(__cv_rvv_e32m2_nlanes)	beq a0,zero,.L10
3	lw a4,%lo(__cv_rvv_e32m2_nlanes)(a5)	addi sp,sp,-48
4	li a3,0	li a6,0
5	li a5,0	li a4,0
6	beq a0,zero,.L7	addi a5,sp,32
7	vsetvli zero,a4,e32,m2,ta,ma	mv t1,sp
8	.L3:	addi a3,sp,16
9	slli a5,a5,2	vsetivli zero,4,e32,m1,ta,ma
10	add a6,a2,a5	.L3:
11	add a7,a1,a5	slli a4,a4,2
12	vle32.v v2,0(a7) # vy = v_load(x+i);	add a7,a1,a4
13	vle32.v v1,0(a6) # vy = v_load(y+i);	vle32.v v1,0(a7) # vx = v_load(x+i);
14	addw a3,a3,a4	add a7,a2,a4
15	mv a5,a3	addiw a6,a6,4
16	vfmacc.vf v1,fa0,v2 # vy = v_fma(a, vx, vy);	mv a4,a6
17	vse32.v v1,0(a6) # v_store(y+i, vy);	vse32.v v1,0(a5) # v_load(x+i)的显式类型转换
18	bltu a3,a0,.L3	vle32.v v1,0(a5) # vx 赋值操作拷贝构造(自动向量化)
19	.L7:	vse32.v v1,0(t1) # vx 赋值操作拷贝构造(自动向量化)
20	ret	vle32.v v1,0(a7) # vy = v_load(y+i);
21		vse32.v v1,0(a5) # v_load(y+i)的显式类型转换
22		vle32.v v1,0(a5) # vy 赋值操作拷贝构造(自动向量化)
23		vse32.v v1,0(a3) # vy 赋值操作拷贝构造(自动向量化)
24		vle32.v v1,0(a3) # v_fma 中 vx 的隐式类型转换
25		vle32.v v2,0(t1) # v_fma 中 vy 的隐式类型转换
26		vfmacc.vf v1,fa0,v2 # vy = v_fma(a, vx, vy);
27		vse32.v v1,0(a5) # vfmacc.vf 的显式类型转换
28		vle32.v v1,0(a5) # vy 赋值操作拷贝构造(自动向量化)
29		vse32.v v1,0(a3) # vy 赋值操作拷贝构造(自动向量化)
30		vle32.v v1,0(a3) # v_store 中 vy 的隐式类型转换
31		vse32.v v1,0(a7) # v_store(y+i, vy);
32		bltu a6,a0,.L3
33		addi sp,sp,48
34		jr ra
35		.L10:
36		ret

4 实验结果与分析

本节以 OpenCV 算法库作为实验对象,在两种具有不同向量寄存器长度的 RISC-V 向量扩展平台上,测试并评估本文所提出的优化方法及其实现方案对高性能算法库的性能优化效果.

4.1 实验环境与测试集

实验平台为 CanMV-K230 开发板和 Banana Pi BPI-F3 开发板,开发板的相关参数如表 4 所示.

表 4 实验平台信息表

实验平台	CanMV-K230 开发板	Banana Pi BPI-F3 开发板
处理器芯片	嘉楠 K230	进迭时空 K1
处理器内核	玄铁 C908	进迭时空 X60
处理器内核数量(向量)	1 个	8 个
向量寄存器长度	128 bits	256 bits
向量扩展标准	RISC-V Vector 1.0	RISC-V Vector 1.0
缓存	32KB L1-I 32KB L1-D 128KB L2	32KB L1-I 32KB L1-D 512KB L2
主频	1.6GHz	2.0GHz
内存	512MB DDR3	4GB DDR3

考虑到编译器对于 RISC-V 指令集的支持日益完善,本文采用撰写时 GCC 编译器的最新发行版(14.1),将 OpenCV 源代码交叉编译到 RISC-V 向量扩展平台上,选取 OpenCV 性能测试套件中的核心(core)模块测试集与图像处理(imgProc)模块测试集,在上述两种开发板上分别运行,获得各个测试用例的执行时间,用于评估本文方法的性能优化效果.

其中,核心模块测试集包含 3363 个测试用例,用于评估 OpenCV 核心组件的性能,包含对向量或矩阵的算术和逻辑运算等操作.本文按照测试用例的操作类型将其分为 38 个类别;图像处理测试集包含 4122 个测试用例,用于评估 OpenCV 图像处理相关操作的性能,包含对向量或矩阵的滤波与模糊、图像金字塔构建、轮廓检测等操作,本文按照测试用例的操作类型将其分为 54 个类别.取各测试类别内所有测试用例的平均执行时间用于性能评估.

4.2 实验设计

为了评估本文所提出的优化方法和实现方案对算法库的性能优化效果,设置如下 5 组实验:

- 标量版本:不启用任何向量优化方法的测试用例,作为测试基准;
- 定长版本:采用 OpenCV 现有固定长度的通用内建函数的测试用例;
- 自动向量化版本:只使用编译器自动向量化,而不采用任何内建函数优化方法的测试用例;
- 可变长版本 (本文方法):采用适用于可变长体系结构的通用内建函数优化方法的测试用例;
- 可变长+自动向量化版本(联用):同时采用自动向量化和可变长通用内建函数优化的测试用例,也是用户构建 OpenCV 时的默认版本.

根据上述 5 组实验的结果,4.3 节通过比较测试用例的执行时间与相对于标量版本的加速比,验证本文方法对于算法库在 RISC-V 平台上性能优化的有效性;4.4 节基于寄存器分组特性进一步讨论本文方法的性能提升原因.

4.3 面向RISC-V向量扩展的硬件抽象层效果分析

为了评估本文方法在 RISC-V 平台上性能优化效果,将 OpenCV 核心模块的测试用例分别编译得到 5 个版本的测试程序,在两种开发板上执行并统计各类别的平均执行时间.

图 5展示了各类测试用例在RISC-V开发板上的平均执行时间,其中纵坐标采用对数轴.从图 5a)中可以发现 DCT、DFT、Eye、LUT、Reduce、Rotate、Round、SorIdx、Sort、TransposeND 和 Zeros 等操作在各自 5 个版本中的执行时间均近似,说明编译器自动向量化无法作用于这些操作,且 OpenCV 源码中也未使用通用内建函数实现这些操作的向量化版本,其原因是这些操作在给定的场景中不具备向量优化机会,不在本文讨论范围内.

通过观察图 5b)中剩余27类操作不难发现,使用本文提出的可变长通用内建函数实现版本(图中黄色,左4)的执行时间都低于使用固定长度内建函数实现版本(图中绿色,左 2),也低于标量实现版本(图中橙色,左 1),说明本文方法改进了 OpenCV 目前的硬件抽象层实现,且具有性能优化效果.

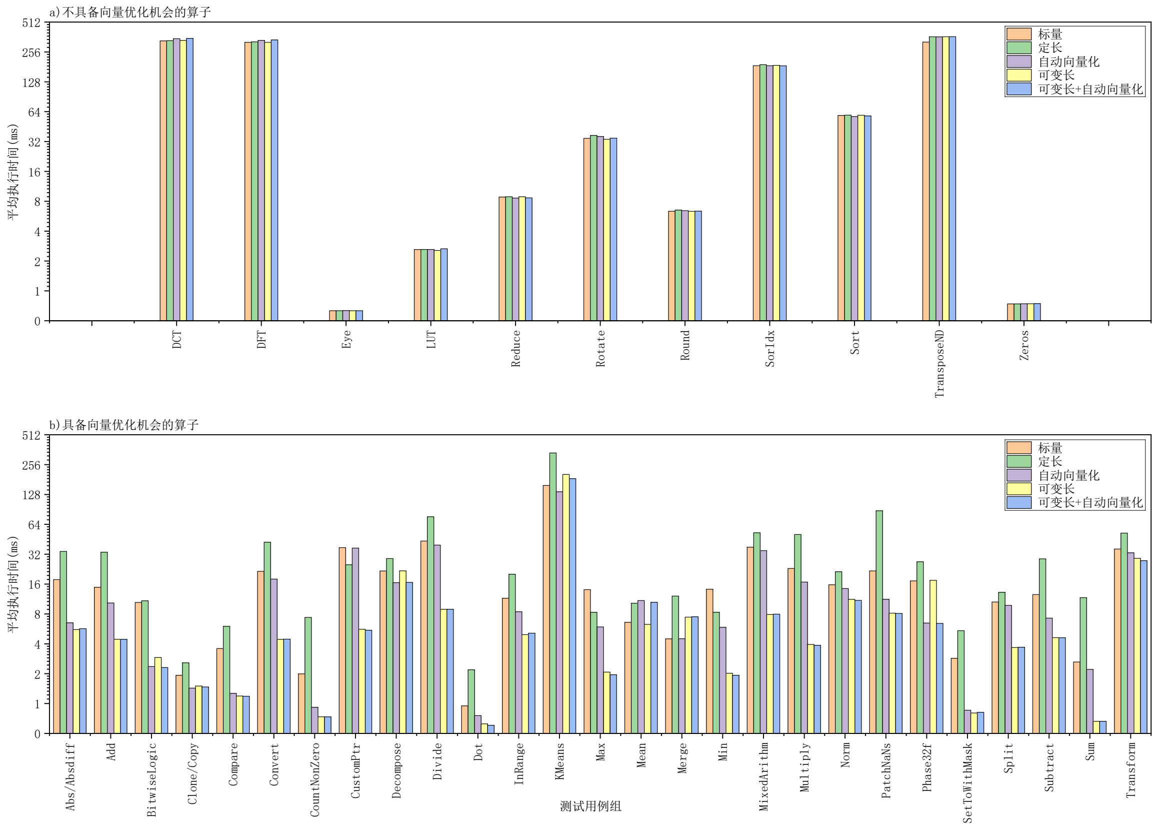


图 5 OpenCV 核心模块测试用例在 RISC-V 开发板上的平均执行时间

表 5 展示了不同优化方法应用于 OpenCV 核心模块中的加速比,加粗值表示测试平台上各测试用例类型的最优加速比.各方法相较于标量实现的加速比可以通过下式计算得到:

$$SpeedUp_i = \frac{AverageCost_i}{AverageCost_{scalar}} \#(1)$$

通过加粗值可以看出,本文方法在 BPI-F3 开发板上对于大部分类别的测试用例产生了最佳的性能提升效果,且本文方法的平均加速比(3.93 倍)高于自动向量化方法加速比(2.16 倍),仅略低于二者联合使用(4.04 倍).说明开发人员使用本文提出的通用内建函数实现优化算法的效果优于自动向量化,而自动向量化在某些场景中可以通过优化其他代码片段的方式在本文方法基础上进一步提高执行效率.CanMK-K230 的结果与 BPI-F3 的结果类似,此处不再赘述.

表 5 不同方法在 OpenCV 核心模块中的加速比

测试用例类别	BPI-F3				CanMV-K230			
	定长	自动向量化	本文方法	联用	定长	自动向量化	本文方法	联用
Abs/Absdiff	0.74	3.51	**4.60**	4.42	0.57	2.06	3.79	**4.41**
Add	0.64	1.76	**4.34**	4.32	0.58	1.16	3.20	**3.37**
BitwiseLogic	1.01	4.52	5.34	**5.48**	0.73	2.66	5.39	**5.81**
Clone/Copy	1.01	1.52	**1.61**	1.56	0.96	1.32	**1.35**	1.32
Compare	0.74	2.55	**2.89**	2.82	0.67	1.61	2.71	**2.72**
Convert	0.59	2.25	**4.91**	4.88	0.55	1.68	**2.51**	2.48

CountNonZero	0.41	2.41	6.22	**6.23**	0.42	1.22	3.46	**3.47**
CustomPtr	1.49	1.01	6.74	**6.90**	1.47	1.01	**6.10**	**6.10**
Decompose	0.92	**1.37**	1.01	**1.37**	0.92	**1.05**	1.01	**1.06**
Divide	0.59	1.50	**4.65**	4.64	0.51	1.09	**2.12**	**2.12**
Dot	0.51	1.55	**5.09**	5.05	0.50	0.89	**3.83**	**3.83**
InRange	0.68	2.04	**3.11**	3.08	0.63	1.33	**2.45**	2.30
KMeans	0.47	**1.17**	0.90	1.04	0.46	**1.11**	1.06	**1.11**
Max	1.76	4.46	8.44	**8.47**	1.48	3.25	7.79	**8.14**
Mean	0.87	0.98	**1.14**	1.11	0.86	0.76	**0.99**	0.76
Merge	0.45	**1.00**	0.93	0.91	0.40	1.02	**1.02**	**1.02**
Min	1.77	4.59	8.52	**8.58**	1.49	3.28	7.79	**8.12**
MixedArithm	0.74	1.48	**4.50**	4.46	0.60	1.06	**2.03**	2.02
Multiply	0.62	2.46	**6.62**	6.59	0.57	1.53	5.38	**5.42**
Norm	0.87	2.11	1.60	**2.76**	0.87	1.37	1.35	**1.68**
PatchNaNs	0.25	1.96	2.72	**2.75**	0.21	0.87	**1.27**	**1.27**
Phase32f	0.63	**2.60**	0.99	**2.60**	0.59	1.39	1.00	**1.40**
SetToWithMask	0.64	3.88	**4.58**	4.46	0.65	**3.15**	3.11	3.09
Split	0.80	1.04	**2.46**	2.44	0.70	1.01	1.39	**1.41**
Subtract	0.68	1.88	**3.68**	3.62	0.61	1.19	3.16	**3.41**
Sum	0.24	1.16	**6.73**	6.59	0.25	1.02	**3.54**	3.50
Transform	0.77	1.60	1.87	**2.08**	0.77	1.36	1.51	**1.59**
平均	0.77	2.16	3.93	**4.04**	0.71	1.50	2.97	**3.07**

类似的,表 6 展示了不同优化方法应用于 OpenCV 图像处理模块中的加速比.结合表 5 和表 6 可以发现,在几乎所有类型的测试用例中,固定长度实现的版本性能都是最差的,甚至其相较于标量实现的加速比也小于1,说明将可变长的 RISC-V 向量扩展整合到固定长度的硬件抽象层的方案对于算法库的性能优化产生了负面作用,其原因在于固定长度的抽象方法在 RISC-V 向量扩展的实现中产生了冗余访存操作,从而引入了大量额外开销,以至于抵消了向量优化带来的性能提升,具体分析已在 3.2 节中给出,此处不再赘述.

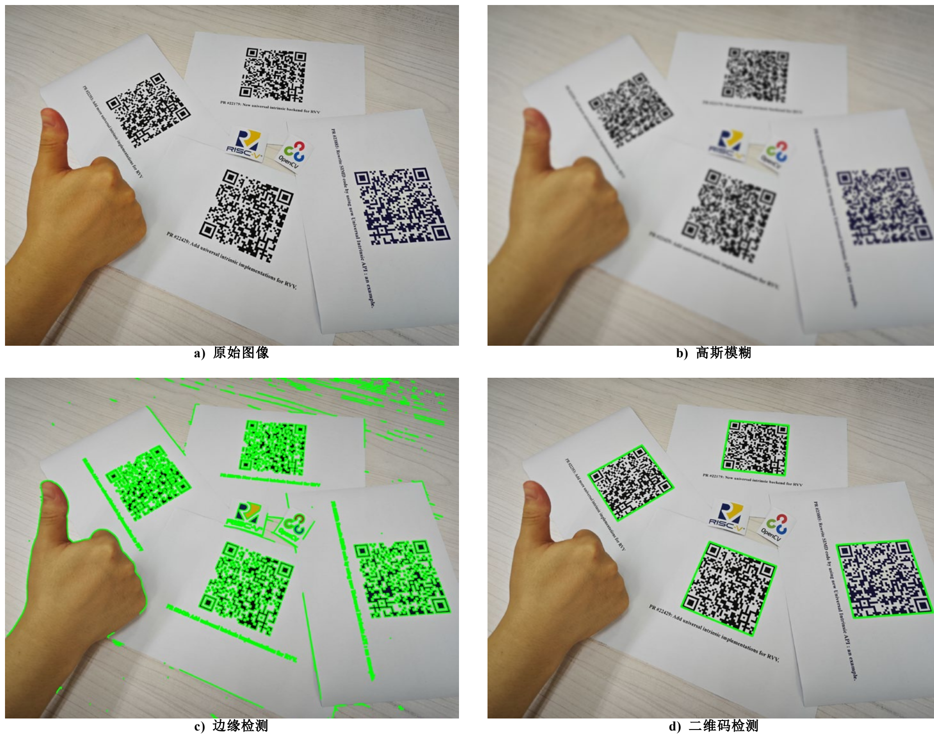
需要说明的是,诸多因素都会共同影响算法的优化效果,其首先取决于是否实现了向量化版本,对于具有向量化实现的算法而言,优化效果还受算法自身向量优化潜力和开发人员实现质量的影响,而各向量指令所能够提供的加速能力也由于指令特性和硬件架构实现的差异而不同,同样会影响算法的优化效果.例如,Decompose 和 Phase32f 等测试用例应用本文方法的加速比介于 0.99~1.01 之间,其原因是当前版本的 OpenCV 算法库中没有使用可变长的通用内建函数接口实现相关算法,因此不会产生性能变动;而 BuildPyramid 和 PyrDown 等测试用例的结果中自动向量化方法的加速比高于本文方法,可能的原因是算法库中相应算法实现未能发掘全部向量化机会,或算法向量优化的实现不佳,未能采用优化效果更好的向量操作或应用更多的硬件资源实现向量化算法,这都为进一步优化算法库性能指出了潜在方向.

此外,在表 5 和表 6 部分测试用例中,本文方法相较于标量版本的加速比也小于 1,其原因有以下两个方面:其一是部分算子虽然是可以向量化的,但 RISC-V 平台上的向量优化的收益不及引入向量化的开销,造成使用通用内建函数编写的向量优化反而产生了负面效果.此类问题常由复杂的访存操作造成,例如向量跨步(Stride)访存指令按照固定的步幅访问内存地址,其空间局部性相较于普通访存指令更差,因而开销较大.在使用此类操作的算子实现中,如果没有更多向量运算操作平摊此类指令的开销,则可能产生性能损失.在本文涉及的测试用例中,Merge 算子中大量使用了向量跨步(Strided Segment)存储指令,且不涉及向量运算操作,而目前的 RISC-V 向量扩展硬件实现中,此类跨步访存操作开销较大,影响了向量优化效果,此类问题可以通过使用条件编译禁用相关优化实现解决;其二是由于输入数据的特殊尺寸无法发挥向量化优势造成的.如 KMeans 算子中向量化了 k 聚类过程对浮点数向量求欧几里得距离平方(L2 norm squared)的算法,向量化后的算法一次迭代可以完成 32 个(当向量寄存器长度为 128 位时)或 64 个(当向量寄存器长度为 256 位时)元素的运算.进一步观察原始测试数据可以发现,在 CanMV-K230 开发板上,当且仅当向量元素个数多于 32 个时,向量化算法才具有优化效果.类似的,在 BPI-F3 开发板上,当且仅当向量元素个数多于 64 个时,向量化算法才具有优化效果.这说明,当输入数据的尺寸过小,以至于无法满足一次向量化迭代的运算量时,其向量化后的向量指令开销及额外控制开销可能超过向量优化的收益,从而引发负面效果.考虑到高性能算法库的输入通常具有较大规模,此类负面效果并不常出现,被认为是可以接受的.

表 6 不同方法在 OpenCV 图像处理模块中的加速比

测试用例类别	BPI-F3				CanMK-K230			
	定长	自动向量化	本文方法	联用	定长	自动向量化	本文方法	联用
Accumulate	0.15	0.99	2.09	**2.10**	0.15	0.79	1.69	**1.71**
AdaptiveThreshold	0.61	2.36	2.05	**3.35**	0.59	1.57	1.63	**2.04**
BilateralFilter	0.15	1.01	3.91	**3.92**	0.16	1.03	**2.83**	**2.83**
BlendLinear	0.57	0.91	**4.04**	3.99	0.56	1.04	**2.35**	**2.35**
Blur	0.78	2.30	2.12	**3.76**	0.79	1.54	1.75	**2.27**
BoundingRect	0.81	**4.77**	0.95	4.72	0.82	2.75	0.96	**2.76**
Box	0.83	2.14	1.24	**2.47**	0.85	1.47	1.17	**1.62**
BuildPyramid	0.37	**1.76**	1.33	1.36	0.33	**1.35**	0.88	0.87
Canny	0.19	1.23	**1.30**	**1.30**	0.19	1.11	**1.28**	1.27
CompareHist	0.66	1.23	3.02	**3.06**	0.67	1.08	**2.20**	2.17
CopyMakeBorder	1.10	**1.11**	1.07	1.10	**1.24**	1.23	1.23	1.23
Corner	0.55	1.35	1.25	**1.97**	0.53	1.09	1.21	**1.37**
CvtColor	0.30	1.06	**1.58**	1.57	0.30	1.01	**1.21**	1.19
Filter2d	0.16	1.00	3.71	**3.73**	0.16	0.87	**2.17**	**2.17**
findContours	0.72	1.03	2.39	**2.51**	0.72	1.04	**2.23**	**2.23**
GaborFilter2d	0.95	0.89	**1.02**	0.84	0.96	0.82	**1.01**	0.86
GaussianBlur	0.31	1.37	3.08	**3.17**	0.32	1.11	**2.10**	2.08
GoodFeaturesToTrack	0.58	1.23	1.20	**1.54**	0.56	1.12	1.19	**1.31**
HoughCircles	0.01	1.06	1.18	**1.20**	0.01	1.03	**1.12**	**1.12**
MatchTemplate	0.94	0.83	**1.06**	0.94	0.93	0.86	**1.01**	0.93
MedianBlur	1.38	4.26	**7.30**	7.22	1.31	2.91	**7.78**	7.75
Moments	0.58	**1.29**	0.95	1.16	0.58	**1.06**	0.99	1.01
PreCornerDetect	0.41	0.75	1.44	**1.46**	0.40	1.15	**1.49**	1.48
Product	0.17	1.10	**2.28**	**2.28**	0.16	0.84	1.78	**1.79**
PyrDown	0.31	**1.67**	1.24	1.22	0.28	**1.30**	0.78	0.76
PyrUp	0.55	1.48	**2.00**	1.96	0.53	1.31	**1.46**	1.43
Remap	0.42	0.90	**1.00**	0.90	0.44	0.78	**1.01**	0.79
Resize	0.86	1.20	1.09	**1.22**	0.83	1.10	1.10	**1.15**
ResizeDownLinear	0.48	1.37	1.29	**1.55**	0.49	1.18	1.25	**1.32**
ResizeUpLinear	0.43	1.36	1.71	**1.86**	0.45	1.15	**1.44**	1.41
ScharrFilter	0.20	1.54	**2.39**	2.26	0.19	1.10	**1.58**	1.51
SobelFilter	0.21	1.04	**2.14**	1.98	0.21	0.92	**1.56**	1.48
SpatialGradient	0.26	1.01	**1.93**	1.85	0.26	1.00	**1.54**	1.49
Square	0.17	1.06	**2.29**	**2.29**	0.17	0.83	**1.94**	**1.94**
Stackblur	0.57	0.80	**3.66**	3.46	0.64	0.84	**2.83**	2.77
Threshold	0.91	3.75	6.24	**6.27**	0.87	2.75	**5.26**	5.21
WarpAffine	0.16	0.92	**1.01**	0.92	0.16	0.86	**1.03**	0.86
WarpPerspective	0.36	0.87	**1.00**	0.87	0.40	0.96	**1.06**	0.96
Weighted	0.55	1.10	**1.88**	1.69	0.55	0.85	**1.42**	1.32
平均	0.50	1.46	2.11	**2.33**	0.51	1.20	1.76	**1.82**

为了在实际应用场景中评估该方法对高性能算法库的性能优化效果,本文选用计算机视觉领域常用的高斯模糊(Gaussian Blur)、边缘检测与二维码检测三个场景,使用 C++语言和 OpenCV 提供的编程接口编写了三个应用程序.在链接时,分别链接采用本文方法优化的 OpenCV 算法库和标量版本的 OpenCV 算法库,并在 BPI-F3 开发板上运行,二者输出的图像一致,但在执行性能上体现出了差异.



a) 原始图像 b) 高斯模糊

c) 边缘检测 d) 二维码检测

图 6 所选计算机视觉算法的效果图

在计算机视觉领域不同场景的应用程序中,都需要对原始图像进行存取和相应操作.在实验评估中,我们分别统计了图像读取、图像操作和图像输出三个部分的执行时间.如图 7 所示,对于尺寸为 4096*3072 的 JPEG 图像,无论是否应用向量优化,图像读取和存储的耗时并无显著差异,这是由于文件 I/O 是外部存储设备和内存之间的数据传输过程,其主要受限于外部存储设备速度和带宽等非向量优化所作用的因素;而在高斯模糊场景中,模糊算子中的计算或访存密集型操作均使用通用内建函数编写,可以通过本文方法在 RISC-V 设备上获得性能提升,当算核尺寸为 63*63 时,所用时间由标量版本的 15291 毫秒下降到 2679 毫秒,图像操作部分的加速比为 5.71 倍,包含图像存取的实际应用场景加速比为 3.73 倍;边缘检测所涉及的算法也可以使用向量优化,能够获得 1.8 倍的加速比,但由于其本身计算复杂度较低,计算量较小,实际应用场景中的主要时间开销是由无法优化的图像存取操作带来的,只能获得 1.06 倍的加速比;二维码检测场景与高斯模糊场景类似,图像操作部分的时间开销占比显著高于图像存取部分,因此在二维码检测算子的加速比为 4.8 倍时,整体应用场景也能够获得较高(4.05 倍)的加速比.

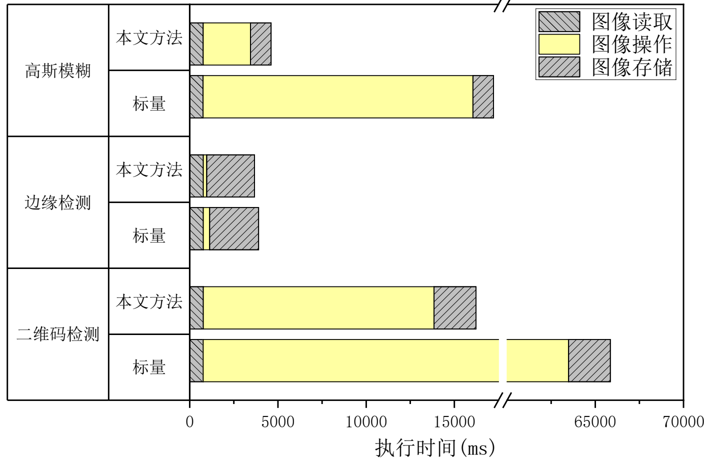


图 7 计算机视觉应用程序在 RISC-V 开发板上的执行时间

4.4 RISC-V向量扩展的寄存器分组特性对于性能优化的影响

RISC-V 向量扩展的寄存器分组特性同样可以用于优化通用内建函数的 RISC-V 向量扩展后端实现.由于部分 SIMD 扩展仅有 16 个向量寄存器,开发者在使用 OpenCV 的通用内建函数编写加速算法时,通常仅使用不超过 16 个向量寄存器,防止在这部分平台上造成寄存器溢出,产生额外的访存指令,影响加速效果.在如今主流 SIMD 扩展普遍提供 32 个向量寄存器的情况下,兼容早期平台的保守寄存器使用策略将导致硬件资源浪费,无法达到最优效率.

而 RISC-V 向量扩展的寄存器分组功能允许将两个向量寄存器视为一组,此时,即便使用通用内建函数编写的算法仅使用了不超过 16 个向量寄存器,在其映射到 RISC-V 向量扩展的后端实现后,此算法将能够使用 RISC-V 向量扩展设备中不超过 32 个物理向量寄存器,从而更加充分地利用硬件资源.在 RISC-V 向量扩展的内建函数中,针对寄存器分组特性提供了相应的向量类型表示和向量操作表示.例如,vfloat32m1_t 表示不分组的浮点数向量类型,而 vfloat32m2_t 表示将两个向量寄存器视为一组的浮点数向量类型,本文提出的硬件抽象层中的通用向量类型 v_float32 就映射到该类型上,从而使用寄存器分组特性进一步提升优化效果.类似的,对 vfloat32m2_t 类型进行加法操作的函数定义为__riscv_vfadd_vv_f32m2,硬件抽象层中的 v_add 函数映射到该函数上,最终由编译器生成相应的状态寄存器配置指令,从而实现分组使用寄存器资源.

表 7 对比了不使用寄存器分组(m1,即向量寄存器各为一组)和将两个向量寄存器视为一组(m2,本文方法采用)时,核心模块的测试用例在不同开发板上的加速比.结果显示,在大部分测试用例中,采用将两个向量寄存器视为一组的设计方案有助于提高性能表现.

表 7 不同寄存器分组在 OpenCV 核心模块中的加速比

测试用例类别	BPI-F3		CanMK-K230	
	m1	m2	m1	m2
Abs/Absdiff	3.95	**4.60**	3.26	**3.79**
Add	3.48	**4.34**	2.75	**3.20**
BitwiseLogic	**5.72**	5.34	4.94	**5.39**
Clone/Copy	1.54	**1.61**	1.27	**1.35**
Compare	2.78	**2.89**	2.24	**2.71**
Convert	3.37	**4.91**	1.74	**2.51**
CountNonZero	4.59	**6.22**	2.33	**3.46**
CustomPtr	6.52	**6.74**	5.24	**6.10**
Decompose	1.00	**1.01**	**1.00**	**1.01**
Divide	4.18	**4.65**	1.81	**2.12**
Dot	4.02	**5.09**	2.62	**3.83**

InRange	2.83	**3.11**	2.20	**2.45**
KMeans	**1.24**	0.90	**1.29**	1.06
Max	7.90	**8.44**	6.91	**7.79**
Mean	**1.17**	1.14	**1.01**	0.99
Merge	**1.12**	0.93	**1.21**	1.02
Min	8.06	**8.52**	6.58	**7.79**
MixedArithm	4.02	**4.50**	1.70	**2.03**
Multiply	5.67	**6.62**	4.22	**5.38**
Norm	1.48	**1.60**	1.25	**1.35**
PatchNaNs	2.02	**2.72**	0.89	**1.27**
Phase32f	**0.99**	**0.99**	**1.00**	**1.00**
SetToWithMask	4.24	**4.58**	2.66	**3.11**
Split	1.04	**2.46**	1.05	**1.39**
Subtract	3.26	**3.68**	2.80	**3.16**
Sum	5.04	**6.73**	2.56	**3.54**
Transform	**2.33**	1.87	1.47	**1.51**
平均	3.46	**3.93**	2.52	**2.97**

5 总结与展望

本文提出了一种面向 RISC-V 向量扩展的高性能算法库优化方法,给出了兼容可变长向量体系结构的硬件抽象层设计与实现方案.该方法不仅可以抽象 RISC-V 向量扩展与其他 SIMD 扩展,还具备兼容其他可变长体系结构硬件的潜力.例如,ARM SVE/SVE2 指令集扩展同样可受益于本文所述基于类型别名和特征类的向量类型抽象方法,从而实现统一向量硬件抽象.

这种可变长的硬件抽象层能够提供统一的向量编程接口,抽象不同硬件的编程模型,从而提高所编写的向量优化算法的通用性,避免高性能算法库中“多算法-多平台”之间组合爆炸而引发的重复实现和代码碎片化等问题,从而降低算法库实现的复杂度,帮助算法库维护者更高效地实现面向各平台的优化代码,更好地支持 RISC-V 软件生态发展.在硬件抽象层设计时,最大的挑战是可变长向量体系结构和固定长度 SIMD 扩展在向量类型上难以抽象,在实现过程中则体现为无法通过统一的向量类型封装二者的元素类型和元素数量等信息.本文采用剥离向量类型和向量信息的方法,以仅抽象出共性部分为目标,使用类型别名的形式提供统一向量类型,再通过额外的特征类提供向量中元素类型与数量等信息,从而实现了统一向量抽象,避免额外性能开销.硬件抽象层设计的另一难题是如何确定所提供的向量操作,在实现过程中体现为如何取舍仅在部分平台中可用的向量指令.本文提出了以覆盖目标硬件平台共有向量操作为基础、以满足算法库计算需求为目标的设计思想,综合考虑某向量操作在应用场景中的使用频次、引入该操作在部分平台上的性能收益和其他平台实现该操作的代价等因素,定义出一套满足算法库需求,且适应于目标硬件平台的硬件抽象层向量操作.

本文还根据所提出的方法,为开源计算机视觉算法库 OpenCV 设计并实现了新的统一向量编程接口,并实现了该接口到 RISC-V 向量扩展的映射,旨在优化其在 RISC-V 平台上的性能.实验结果表明,相较于原有硬件抽象层,本文的设计方案在 OpenCV 核心模块中获得了 2.97~3.93 倍的性能提升,在图像处理模块中获得了 1.76~2.11 倍的性能提升,验证了本文方法的有效性.并通过进一步分析阐述了性能提升来源,为其他高性能算法库的硬件抽象层设计提供参考.本文所述的相关工作已经被 OpenCV 社区接收[①],为丰富 RISC-V 软件生态做出了贡献.

未来的研究将聚焦于通过引入更多向量编程模型进一步优化硬件抽象层的设计,从而继续改进适用于 SIMD 扩展和向量扩展的算法库优化方法.RISC-V 向量扩展中的向量长度寄存器特性有望简化循环向量化的编程模型并减小代码体积,此特性将在接下来的硬件抽象层设计中得到更深入的研究和应用,以进一步提高性能优化效果.此外,本文提出的优化方法也可以被应用于其他高性能算法库的 RISC-V 移植与向量优化工作中,

① 开源地址:https://github.com/opencv/opencv/blob/4.x/modules/core/include/opencv2/core/hal/intrin_rvv_scalable.hpp, 感谢 OpenCV 社区维护者 Vadim Pisarevsky,Alexander Smorkalov 和 Maksim Shabunin 对本工作的建议和帮助.

这有助于提高这些算法库在 RISC-V 平台上的可用性.例如,GGML 是一个可用于 Transformer 架构推理的机器学习开源算子库,被应用于深度学习或大语言模型的 CPU 端推理场景中.其使用内联汇编和内建函数的方式面向 ARM Neon 和 x86 AVX 系列指令集扩展优化了模型权重量化和通用矩阵乘法等少数核心算子,且正逐步添加面向 RISC-V 向量扩展平台的优化实现.随着其所支持的硬件平台数量和向量化算子数量的增加,在 GGML 中引入可变长的硬件抽象层将有助于避免算子面向不同硬件平台的重复实现,降低开发复杂度,提高算法库的可维护性;而深度学习算法库 PyTorch 和线性代数算法库 Eigen 也都计划将 RISC-V 向量扩展引入其现有的硬件抽象层中,应用本文所述方法可以帮助算法库更灵活地设计和实现兼容现有定长平台与可变长平台的硬件抽象层,从而更好地实现 RISC-V 向量扩展后端,提高算法库在 RISC-V 平台上的性能表现.而整合更多的 SIMD 或向量扩展到硬件抽象层中,特别是支持如 RISC-V P 扩展等新兴设备平台,可以增强硬件抽象层的功能性和适用性,对于促进 RISC-V 软件生态发展具有显著意义.

References:

[1] Luebke D. CUDA: Scalable parallel programming for high-performance scientific computing. In: Proc. of the 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro.2008. 836-838.[doi: 10.1109/ISBI.2008.4541126]

[2] Munshi A. The OpenCL specification. In: Proc. of the 2009 IEEE Hot Chips 21 Symposium (HCS).2009. 1-314.[doi: 10.1109/HOTCHIPS.2009.7478342]

[3] Lomont C. Introduction to intel advanced vector extensions. Intel white paper. 2011, 23: 1-21.

[4] Stephens N, Biles S, Boettcher M, et al. The ARM scalable vector extension. IEEE micro. 2017, 37(2): 26-39.[doi: 10.1109/MM.2017.35]

[7] Bradski G, Kaehler A. Learning OpenCV: Computer vision with the OpenCV library. " O'Reilly Media, Inc.", 2008.[doi: 10.1109/MRA.2009.933612]

[8] riscv/riscv-v-spec: Working draft of the proposed RISC-V V vector extension. 2024.https://github.com/riscv/riscv-v-spec

[10] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices. 2013, 48(6): 519-530.[doi: 10.1145/2491956.2462176]

[11] Universal intrinsics. 2024.https://docs.opencv.org/4.x/df/d91/group__core__hal__intrin.html

[12] Kretz M. Extending C++ for explicit data-parallel programming via SIMD vector types [Ph.D. Thesis]. Frankfurt am Main, Johann Wolfgang Goethe-Univ., 2015.

[13] Highway: About Performance-portable, length-agnostic SIMD with runtime dispatch. 2024.https://github.com/google/highway

[15] libjpeg-turbo. A JPEG image codec that uses SIMD instructions to accelerate baseline JPEG compression and decompression.2024.http://sourceforge.net/projects/libjpeg-turbo

[16] Genc H, Kim S, Amid A, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In: Proc. of the 2021 58th ACM/IEEE Design Automation Conference (DAC).IEEE, 2021. 769-774.[doi: 10.1109/DAC18074.2021.9586216]

[17] Li R, Peng P, Shao Z, 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]

附中文参考文献:

[5] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术. 计算机研究与发展. 2023, 60(01): 2-16.[doi: 10.7544/issn1000-1239.202220196]

[6] 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]

[9] 冯竞舸, 贺也平, 陶秋铭. 自动向量化：近期进展与展望. 通信学报. 2022, 43(03): 180-195.[doi: 10.11959/j.issn.1000−436x.2022051]

[14] 纪守领, 王琴应, 陈安莹, 等. 开源软件供应链安全研究综述. 软件学报. 2023, 34(3): 1330-1364.[doi: 10.13328/j.cnki.jos.006717]