计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(6): 1443−1454, 2025

SparseMode: 用于高效 SpMV 向量化代码生成的稀疏编译框架

王昊天　丁　岩　何贤浩　肖国庆　阳王东
（湖南大学信息科学与工程学院　长沙　410000）
（wanghaotian@hnu.edu.cn）

SparseMode: A Sparse Compiler Framework for Efficient SpMV Vectorized Code Generation

Wang Haotian, Ding Yan, He Xianhao, Xiao Guoqing, and Yang Wangdong
（*College of Computer Science and Electronic Engineering*, *Hunan University*, *Changsha* 410000）

Abstract　Sparse matrix-vector multiplication (SpMV) is a core operation in numerical computations, widely used in scientific computing, engineering simulations, and machine learning. The performance optimization of SpMV is mainly constrained by irregular sparse patterns, with traditional optimizations relying on manually designed storage formats, computation strategies, and memory access patterns. Existing tensor compilers such as TACO and TVM use domain-specific languages (DSL) to generate high-performance operators, alleviating developers from the burdensome manual optimization tasks. However, the optimization support for sparse computations remains insufficient, making it challenging to adaptively optimize performance for different sparse patterns. To address these issues, we propose a sparse compiler framework called SparseMode, which generates efficient vectorized code for SpMV computations based on the matrix's sparse pattern and adaptively adjusts optimization strategies according to the characteristics of the hardware platform. The compiler framework first designs a domain-specific language, SpMV-DSL, which efficiently and concisely expresses sparse matrices and computational operations in SpMV. Then, a sparse pattern-aware method is introduced, which dynamically selects computation strategies based on the matrix storage format and non-zero element distribution defined in SpMV-DSL. Finally, efficient parallel SpMV operator code is generated through sparse pattern analysis and scheduling optimizations, fully utilizing SIMD instructions to enhance performance. Experimental results on different hardware platforms show that SpMV operator code generated by SparseMode achieves up to a 2.44 times speedup compared with the existing TACO and TVM tensor compilers.

Key words　SpMV; compilers; sparse patterns; high performance computing; vectorization

摘　要　稀疏矩阵向量乘法（sparse matrix-vector multiplication, SpMV）是数值计算中的核心操作，广泛应用于科学计算、工程模拟以及机器学习中. SpMV 的性能优化主要受限于不规则的稀疏模式，传统的优化通常依赖手动设计存储格式、计算策略和内存访问模式. 现有张量编译器如 TACO 和 TVM 通过领域特定语言（domain specific language, DSL）可实现高性能算子生成，减轻开发人员繁琐的手动优化工作，但对稀疏计算的优化支持尚显不足，难以根据不同的稀疏模式自适应优化性能. 为了解决这些问题，提出了名为 SparseMode 的稀疏编译框架，能够依据矩阵的稀疏模式为 SpMV 计算生成高效的向量化代码，并根据

收稿日期：2025-03-01；**修回日期**：2025-04-11
基金项目：国家重点研发计划项目（2023YFB3001805）；国家自然科学基金项目（62321003, 62227808）；中国博士后科学基金项目（2024M750879, BX20240111）
This work was supported by the National Key Research and Development Program of China (2023YFB3001805), the National Natural Science Foundation of China (62321003, 62227808), and the China Postdoctoral Science Foundation (2024M750879, BX20240111).
通信作者：丁岩（ding@hnu.edu.cn）

硬件平台的特性自适应地调整优化策略. 该编译框架首先设计了领域专属语言 SpMV-DSL, 能够简洁高效地表达 SpMV 的稀疏矩阵和计算操作. 然后提出了基于稀疏模式感知的方法, 根据 SpMV-DSL 定义的矩阵存储格式和非零元素分布动态选择计算策略. 最后通过稀疏模式分析和调度优化生成高效并行的 SpMV 算子代码, 以充分利用 SIMD 指令提升性能. 在不同硬件平台上的 SpMV 实验结果表明, SparseMode 生成的 SpMV 算子代码相较于现有的 TACO 和 TVM 张量编译器实现了最高 2.44 倍的加速比.

关键词 稀疏矩阵向量乘法; 编译器; 稀疏模式; 高性能计算; 向量化

中图法分类号 TP314

DOI: 10.7544/issn1000-1239.202550139 **CSTR:** 32373.14.issn1000-1239.202550139

稀疏矩阵向量乘法(sparse matrix-vector multiplication, SpMV)[1]是数值计算中的核心运算之一, 在众多科学与工程领域中发挥着关键作用. 它不仅是求解线性方程组和优化问题的基础, 而且在多个高性能计算(high performance computing, HPC)应用中具有广泛的应用场景. SpMV 是许多科学计算和工程应用中的基本组成部分, 尤其在求解大型稀疏线性系统[2]时至关重要. 典型的应用包括流体力学、结构分析、电磁场仿真以及地质学中的地震波模拟等. 这些领域中的问题通常会被转化为大规模的稀疏矩阵形式, SpMV 用于快速求算这些问题的解. 此外, SpMV 在机器学习扮演着重要角色, 尤其是在处理大规模稀疏数据和高维特征空间时. 随着数据量的不断增长, 许多机器学习任务涉及到稀疏矩阵的运算, 如协同过滤[3]和支持向量机[4]. 随着硬件架构的不断发展, 多核/众核处理器、GPU, 以及现代向量处理单元(如 Intel AVX512 和 ARM NEON)为 SpMV 提供了加速的潜力, 优化 SpMV 成为提升其性能的关键研究方向.

传统的 SpMV 实现通常需要开发人员手动进行优化, 这包括选择合适的稀疏矩阵存储格式[5]、调整计算策略[6]、优化内存访问模式[7]以及手动编写并行化代码等, 而编译器优化能够消除手动调优的繁琐工作, 同时避免由于开发人员的经验限制而带来的优化不足. 而传统编译器(如 LLVM[8]和 GCC[9])的优化策略通常是针对稠密数据结构设计的, 依赖静态分析技术进行循环展开、指令级并行和数据局部性优化等. 然而, 由于稀疏矩阵的非零元素分布是高度动态的, 传统的编译器优化方法无法在编译期对其进行有效的静态分析, 从而导致编译器缺乏对不同稀疏模式的适配能力.

近年来, 张量编译器和代码生成器(如 TACO[10]和 TVM[11])的出现, 使得开发人员通过几行领域特定语言(domain specific language, DSL)就能实现高性能算子的生成, 然而它们通常只能针对特定模式进行优化, 缺乏自适应调优[12]的机制, 无法根据不同类型的稀疏矩阵动态选择最优的数据存储格式和计算策略. 例如, 某些稀疏矩阵可能更适合使用压缩稀疏列(compressed sparse column, CSC)格式或压缩稀疏行(compressed sparse row, CSR)[13]格式, 而另一些稀疏矩阵则可能需要块压缩稀疏行(block compressed sparse row, BCSR)格式或混合格式, 然而现有的编译器通常无法自动识别并选择合适的存储格式, 从而限制了计算性能的提升. 因此, 要有效提升 SpMV 操作的性能, 必须开发能够感知稀疏矩阵结构, 并根据硬件架构和稀疏模式动态优化计算策略的编译器框架, 这将能够最大化地发挥现代硬件的计算潜力.

针对上述传统张量编译器在处理 SpMV 的挑战, 本文设计了 SparseMode, 用于高效地利用 SpMV 代码生成稀疏编译器. 本文的主要贡献有 3 点:

1)设计了新型的 SpMV 计算领域描述语言 SpMV-DSL, 简洁且高效地表达稀疏矩阵的结构和稀疏计算操作, 能够自适应不同的稀疏模式, 方便与编译器进行集成;

2)提出了基于稀疏模式感知的稀疏计算编译器, 能够根据 SpMV-DSL 设定的存储格式和输入的稀疏矩阵非零元素分布, 从而选择最优的计算策略和内存布局进行优化;

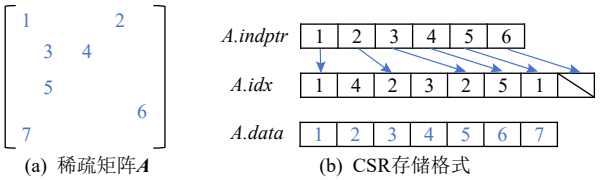
3)提出了生成高性能 SpMV 代码的流程, 通过稀疏模式分析和 SpMV-DSL 指定的并行和向量化策略, 生成高效的并行化 SpMV 算子代码, 充分利用 SIMD 指令加速计算, 同时减少冗余操作和内存访问.

1 研究背景和研究动机

1.1 SpMV

SpMV 的计算表达式为 $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, 其中 $\boldsymbol{A}$ 为稀疏矩阵, $\boldsymbol{x}$ 为输入的稠密向量, 而 $\boldsymbol{y}$ 为输出的结果, 同样为稠密向量. 稀疏矩阵 $\boldsymbol{A}$ 中存在大量零元, 如果采用稠

密矩阵的存储形式会造成大量内存浪费，通常$\boldsymbol{A}$采用压缩存储格式如 CSR，如图 1 所示. CSR 格式采用 3 个数组表示，$A.indptr$表示矩阵每一行的非零元在$A.idx$和$A.data$的起始位置，$A.idx$和$A.data$分别表示非零元列的位置以及存储的值，2 个数组长度相同且一一对应.



(a) 稀疏矩阵$\boldsymbol{A}$ (b) CSR存储格式

Fig. 1 Illustration of CSR format

图 1 CSR 格式示意图

算法 1 所示为基于 CSR 格式的 SpMV 实现. SpMV 优化的难度在于对$\boldsymbol{x}$是间接访存，可能由于访存跨度过大，导致无法高效地利用缓存的局部性特性.

算法 1. CSR 格式的 SpMV 实现.

输入：CSR $\boldsymbol{A}[M,N]$，稠密向量$\boldsymbol{x}$；

输出：稠密向量$\boldsymbol{y}$.

① for $i=1,\cdots,M$ do

② $sum=0$;

③ for $j=A.indptr[i],\cdots,A.indptr[i+1]$ do

④ $sum+=A.data[j]\times\boldsymbol{x}[A.idx[j]]$;

⑤ end for

⑥ $y[i]=sum$;

⑦ end for

1.2 张量编译器

随着深度学习模型的快速发展，张量编译器如 TACO 和 TVM 已经成为提升计算效率、加速深度学习模型部署的重要工具. 这些编译器通过实现自动化的高性能算子代码生成，极大地简化了算子优化过程. 张量编译器通常定义了一系列特定领域语言，用以精确描述算子的计算过程和优化逻辑. 开发人员只需编写几行 DSL 代码，便可生成高效的底层执行代码，从而避免了繁琐且容易出错的手动算子优化.

通过张量编译器，开发者能够在不深入底层硬件实现的情况下，针对特定硬件架构（如 GPU、TPU[14]等）进行优化. 这种方法不仅提升了算子执行效率，也降低了手动优化过程中的开发成本. 张量编译器利用数学模型、编译优化技术及自动调优算法，能够在多种硬件架构之间实现最佳的算子映射和调度策略，从而大幅提升计算性能.

1.3 研究动机

尽管张量编译器表现优异，但它们的设计主要针对的是稠密计算模式. 稠密矩阵的计算模式相对固定，数据访问和计算过程的并行性容易预测和优化. 针对这些特性，张量编译器能够有效地生成高度优化的代码，从而显著提升计算性能. 然而，在面对稀疏计算时，现有的张量编译器面临一定的挑战.

稀疏矩阵的计算模式与稠密矩阵的大不相同. 由于稀疏矩阵中大部分元素为零，非零元素通常分布不均匀，导致其计算模式高度不规则. 对于这种稀疏模式，现有的张量编译器往往无法很好地适应其复杂性. 尤其是在 SpMV 这类问题中，存在着大量的跳跃访问. 因此，传统的稠密计算优化策略无法直接迁移到稀疏计算任务中. 更为重要的是，当前的张量编译器通常缺乏对稀疏模式的深入分析与优化能力. 尽管一些张量编译器能够进行简单的内存优化和调度，但它们在稀疏矩阵的特殊模式下，往往无法自动识别出不同稀疏性带来的计算和存储优化机会. 缺乏对稀疏结构的细致分析使得编译器无法充分发挥稀疏矩阵的潜力，导致计算效率未能达到预期.

针对张量编译器在 SpMV 计算中的挑战，本文提出了高效的 SpMV 代码生成稀疏编译器 SparseMode，其由 SpMV-DSL 和编译优化框架组成，其结构如图 2 所示. SpMV-DSL 负责描述 SpMV 的计算逻辑，并提供包括格式选择、向量化和并行化调度指令等功能. 在编译阶段，编译框架根据 SpMV-DSL 中实现的计算和调度描述，以及稀疏矩阵的非零元素分布，执行稀疏模式分析等操作，并根据稀疏模式分析的结果进一步实现数据分块和调度优化，最终生成高性能的 SpMV 内核代码.

2 SpMV 的领域特定语言设计

2.1 方案设计

SpMV 的计算表达式为$\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}$. 受到 TACO 张量代数表达式的启发[10]，为了准确描述 SpMV 的计算逻辑和稀疏矩阵$\boldsymbol{A}$的存储格式，本文设计了针对 SpMV 的领域特定语言 SpMV-DSL，具体包括计算、格式和调度等描述形式.

1）计算描述

该部分主要描述 SpMV 的计算逻辑，根据计算过程中矩阵和向量的访存关系设计了如下的表达式：

$$y[i]=sum(A[i,j]\cdot\boldsymbol{x}[j],axis=j). \tag{1}$$

2）格式描述

式（1）仅描述了计算中的访存关系，例如在计算$y[i]$时，需要遍历稀疏矩阵$\boldsymbol{A}$的第i行数据与向量$\boldsymbol{x}$，并

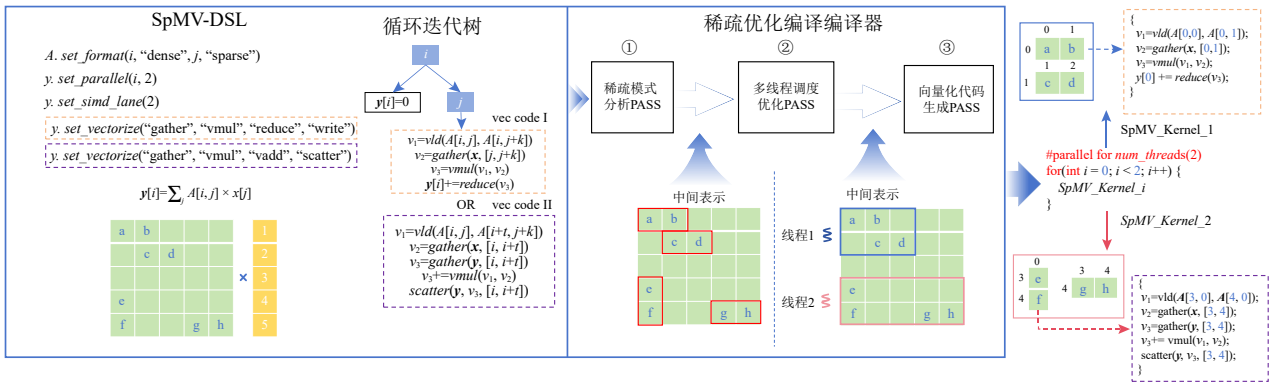


Fig. 2 Compiler optimization framework of SparseMode

图 2 SparseMode 编译优化框架

进行规约求和. 然而, 在实际运算过程中稀疏矩阵 $\boldsymbol{A}$ 通常采用的是压缩存储的格式, 如 CSR 或 CSC 格式, 因此还需要额外的格式描述:

A.set_format(*i*,“dense”,*j*,“sparse”),

A.set_format(*i*,“sparse”,*j*,“dense”).

对于需要设置稀疏格式的矩阵, *set_format* 可以帮助矩阵设定每个维度的稀疏和稠密属性, 例如 (*i*,“dense”,*j*,“sparse”) 表示维度 *i* 是稠密的和维度 *j* 是稀疏的, 能够适配于 CSR 格式, 同理 (*i*,“sparse”,*j*,“dense”) 适配于 CSC 格式.

3) 调度描述

SpMV-DSL 提供了向量化执行和并行优化的调度原语, 用以实现高性能 SpMV 内核的生成. 对于并行执行, 设计的调度为:

y.set_parallel(*i*, 2).

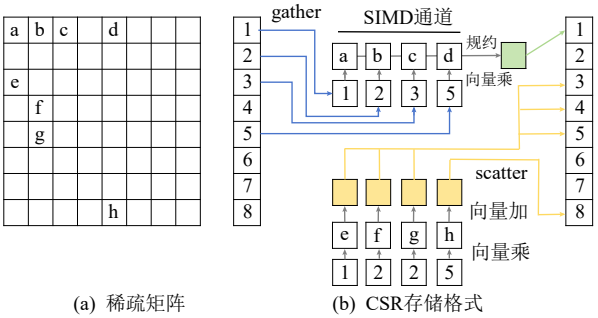
该调度表示在维度 *i* 的方向上进行并行化处理, 且线程数设置为 2. 而对于向量化执行, 考虑到不同的 SIMD 架构所支持的向量化指令不同, 用户可以设置底层向量化执行的操作:

y.set_simd_lane(2);

y.set_vectorize(“gather”,“vmul”,“reduce”,“write”);

y.set_vectorize(“gather”,“vmul”,“vadd”,“scatter”).

其中, *set_simd_lane* 用于设置 SIMD 的数据通道数, 对于 AVX512 指令集, 底层在计算 SpMV 时可以采用 2 种向量化方式: 行向量计算和列向量计算. 如图 3 所示, 行向量计算对稀疏矩阵的某一行进行向量化计算, 采用(“gather”,“vmul”,“reduce”,“write”)的运算模式, 首先通过“gather”指令将指定位置的稠密向量 $\boldsymbol{x}$ 加载到向量寄存器中, 然后通过“vmul”实现向量乘法, 并进行“reduce”规约求和, 最终“write”写回到结果 $\boldsymbol{y}$ 的对应位置. 而列向量计算对稀疏矩阵的多个列进行向量化计算, 采用(“gather”,“vmul”,“vadd”,“scatter”)的运算模式, “gather”, “vmul”与上述行向量计算相同, 列向量采用 Element-wise 的计算, 因此采用“vadd”的向量加操作和“scatter”将结果写回到 $\boldsymbol{y}$.



(a) 稀疏矩阵 (b) CSR存储格式

Fig. 3 Pattern of vectorized computation

图 3 向量化计算模式

2.2 与编译优化框架的关联

在本文设计的 SpMV 编译优化框架中, SpMV-DSL 提供的格式和调度描述(如设置线程数、向量化操作等)用于指导编译框架进行合理的代码生成, 而输入的稀疏数据, 有助于编译框架进行稀疏模式分析, 进而优化稀疏矩阵的存储和 SpMV 的向量化执行.

3 SpMV 的代码生成与优化

3.1 方案设计

SparseMode 提供了针对 SpMV 算子的稀疏数据感知编译优化框架, 其由稀疏模式分析、调度优化和代码生成等模块组成. 框架结构及各模块之间的关系如图 2 所示.

首先, 根据 SpMV-DSL 的表达式描述, 前端解析器会生成一个循环迭代树. 该迭代树的每个节点表示一个循环的变量, 对迭代树进行先序遍历即可得

到循环的顺序. 叶子节点表示相应的计算逻辑, 根据设置的向量化调度原语, 提取适合向量化的计算形式. 在节点 j 的叶子节点形成图中所示的向量化计算逻辑.

其次, 针对多线程并行计算的需求, 调度优化模块会根据 SpMV-DSL 中预定的线程数量以及输入的稀疏数据非零元的分布, 对数据块进行合理划分, 并优化调度策略, 以确保每个线程分配到均衡的计算负载, 从而提升计算资源的利用率和整体执行效率(图 2 中所示的稀疏矩阵分块和并行的线程数).

最后, 代码生成模块基于稀疏模式分析和调度优化的结果, 输出高效的 SpMV 内核代码(图 2 中所示的 2 个内核代码, 每个内核代码根据数据块的排布采用不同的向量化执行), 并针对稀疏矩阵的存储方式进行优化, 使得计算过程更加高效.

3.2 稀疏模式分析

为了生成高性能的 SpMV 代码, 需要根据输入的稀疏矩阵非零元分布, 划分多个可进行向量化运算的数据块. 假设 SIMD 的数据通道数为 d, 对于稀疏矩阵 $\boldsymbol{A}$, 可以将其划分为多个可向量化数据块 $VP=\{[(r_1^1,c_1^1),\cdots,(r_1^d,c_1^d)],[(r_2^1,c_2^1),\cdots,(r_2^d,c_2^d)],\cdots,[(r_V^1,c_V^1),\cdots,(r_V^d,c_V^d)]\}$, 其中 $[(r_i^1,c_i^1),\cdots,(r_i^d,c_i^d)]$ 表示第 i 个向量化数据块中每个元素的行列位置. 如果 $r_i^1=r_i^2=\cdots=r_i^d$, 则表明该数据块采用行向量计算模式, 用 B_i^r 表示; 如果 $\forall j,k$, $r_i^j\neq r_i^k$, 则表明该数据块采用列向量计算模式, 用 B_i^c 表示.

由于 CPU 的缓存具有局部性特征, 即使在向量化执行过程中, 同样要考虑数据读写的局部性. 对于稀疏矩阵 $\boldsymbol{A}$ 中非零元的读取, 因为在编译过程中能获取 $\boldsymbol{A}$ 的非零元分布, 可以提前将需要进行向量化计算的非零元存放在相邻位置. 然而, 对于稠密向量 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的读写是根据 $\boldsymbol{A}$ 中的行列位置确定的, 因此在挖掘 $\boldsymbol{A}$ 中的向量化数据块时, 要考虑对于 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的访存跨度.

定义 1. 稠密向量的访存跨度. 给定一个稠密向量 $\boldsymbol{x}$, 以及其访存位置的顺序为 $P=[p_1,p_2,\cdots,p_T]$, 则该向量 $\boldsymbol{x}$ 的访存跨度:

$$S_P=\sum_{i=2}^{T}|p_i-p_{i-1}|. \tag{2}$$

具体而言, 对于第 i 个向量化数据块 $[(r_i^1,c_i^1),\cdots,(r_i^d,c_i^d)]$, 若该数据块采用行向量计算模式, 则其对于稠密向量 $\boldsymbol{x}$ 的跨度为 $\sum_{t=2}^{d}|c_i^t-c_i^{t-1}|$, 最后计算结果为标量, 写回向量 $\boldsymbol{y}$ 还需要一次访存, 因此 $S_{B_i^r}=1+\sum_{t=2}^{d}|c_i^t-c_i^{t-1}|$; 若该数据块采用列向量计算模式, 除了对稠密向量 $\boldsymbol{x}$ 的读取外, 最后还要将计算结果写回向量 $\boldsymbol{y}$, 因此还需要加上 $\boldsymbol{y}$ 的跨度, 即 $S_{B_i^c}=\sum_{t=2}^{d}|r_i^t-r_i^{t-1}|+|c_i^t-c_i^{t-1}|$. 综上, 稀疏模式分析的优化目标为划分数据块 $VP=\{B_1,\cdots,B_V\}$, $B_i=\{(r_i^1,c_i^1),\cdots,(r_i^d,c_i^d)\}$, 最小化以下目标函数:

$$\arg\min\sum_{i=1}^{V}S_{B_i}. \tag{3}$$

然而, 直接求解最优的向量化数据块的时间复杂度是指数级的, 且由于稀疏矩阵的规模通常较大, 这在编译过程中会导致稀疏分析的开销过大. 稀疏矩阵中的非零元素分布通常是不规则的, 且其稀疏性会因矩阵的规模、非零元素的分布和计算平台的硬件架构不同而有所变化. 为了避免这一瓶颈, 本文提出了近似算法, 旨在尽可能接近最优的向量化数据块选择. 通过对稀疏矩阵存储格式(如压缩稀疏行格式)的特性进行分析, 该算法能够高效地选择适合 SIMD 计算的向量化数据块, 具体如算法 2~4 所示.

算法 2. 行向量块选择函数.

输入: CSR 格式 $\boldsymbol{A}[M,N]$, SIMD 通道数 d, 初始行 t, 访问标记 vis, 计数器 cnt;

输出: 行向量块 $rcand$ 及其访存跨度 S_1.

① $S_1\leftarrow$INF;
② if $cnt[t]\geqslant d$ then
③ for $i=A.indptr[t]$ to $A.indptr[t+1]$ do
④ if $vis.find([t,A.idx[i]])==$ false then
⑤ $rcand.append([t,A.idx[i]])$;
⑥ end if
⑦ if $size(rcand)==d$ then
⑧ $S_1=\sum_{j=2}^{d}|rcand[j,2]-rcand[j-1,2]|$;
⑨ break;
⑩ end if
⑪ end for
⑫ end if

算法 3. 列向量块选择函数.

输入: CSR 格式 $\boldsymbol{A}[M,N]$, SIMD 通道数 d, 初始行 t, 访问标记 vis, 计数器 cnt;

输出: 行向量块 $ccand$, 访存跨度 S_2.

① $S_2\leftarrow INF, L\leftarrow t, R\leftarrow t$;
② while $L\geqslant 0$ and $cnt[L]>0$
③ $L-=1$;
④ end while
⑤ while $R<M$ and $cnt[R]>0$

⑥ $R+=1$;
⑦ end while
⑧ if $R-L\geqslant d$ then
⑨ 从$[L,R]$中搜索最优的范围$[l,r](l\leqslant t\leqslant r, r-l=d)$, 使得每行中第一个符合要求的非零元组成$ccand$的访存跨度$S_2$最小;
⑩ else
⑪ 将$[L,R]$中每行第一个未被访问到的非零元加入到$ccand$;
⑫ 在$[L,R]$之外, 找到与第L和第R行距最近的$d+L-R$个行, 并且每个行的$cnt>0$;
⑬ 将找到的$d+L-R$个行中第一个未被访问到的非零元加入到$ccand$;
⑭ 计算$ccand$的访问跨度S_2;
⑮ end if

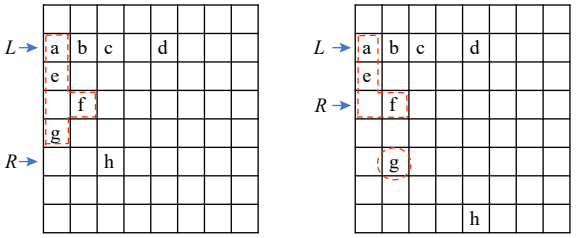
算法 4. 稀疏模式分析函数.

输入: CSR 格式$\boldsymbol{A}[M,N]$, SIMD 数据通道数d;
输出: 向量化数据块VP.

① 初始化 $VP\leftarrow\varnothing$;
② 初始化集合$vis\leftarrow\varnothing$, 用于标记已经访问过的非零元位置;
③ 初始化数组 cnt, 使得 $cnt[i]=A.indptr[i+1]-A.indptr[i]$;
④ do while
⑤ $t\leftarrow\arg\max_i cnt[i], S_1\leftarrow INF, S_2\leftarrow INF$;
⑥ $rcand=[], ccand=[]$;
⑦ $rcand, S_1\leftarrow$算法 2;
⑧ $ccand, S_2\leftarrow$算法 3;
⑨ if $S_1<S_2$ then
⑩ $VP.append(rcand), cnt[rcand[:,1]]-=1$;
⑪ $vis.append(rcand[:])$;
⑫ else if $S_1\geqslant S_2$ && $S_1\neq INF$ then
⑬ $VP.append(ccand), cnt[ccand[:,1]]-=1$;
⑭ $vis.append(ccand[:])$;
⑮ end if
⑯ end while
⑰ 对剩余的非零元按照行/列向量填充零元, 并加入到VP.

算法 2 和算法 3 分别介绍了如何在稀疏矩阵$\boldsymbol{A}$的第t行中分别找到行向量块和列向量块, 而算法 4 为稀疏模式分析的方法, 在执行过程中会调用算法 2 和算法 3, 并判断选择哪种向量化执行方式. 具体而言, 算法 2 的思路比较直接, 如第t行存在大于d数量的非零元, 则将该行中邻近的d个非零元加入到行向量块, 该算法的时间复杂度为 $O(nnz)$, nnz 为第t行的非零元数目.

由于列向量需要考虑读和写的访存, 算法 3 相较于算法 2 更为复杂, 如图 4 所示, 算法 3 首先围绕着第t行搜索范围最大的连续行$[L,R]$, 如果$R-L\geqslant d$, 则直接在该范围内找到能最小化S_2的连续d个行, 并将它们的第一个符合要求的非零元加入到列向量块中(行⑨), 否则先将$[L,R]$行符合要求的非零元添加到列向量块, 剩余的$d+L-R$个非零元则从$d+L-R$个与$[L,R]$行距离最近的行中被找到(行⑪~⑭). 该算法的时间复杂度为 $O(N)$.



(a) $R-L\geqslant 4$, 选择{a, e, f, g}访存跨度最小　(b) $R-L<4$, 先选择{a, e, f}, 再选择行距离最近的g

Fig. 4 Selection of column vector block

图 4 列向量块选择

为了防止非零元被计算多次, 在挖掘向量块时, 算法 2 和算法 3 都采用了辅助数据结构vis, 用于标记已经添加的非零元. 算法 4 根据算法 2 和算法 3 输出的访存跨步S_1和S_2进行比较, 判断当前的d个非零元为行向量块还是列向量块. 在算法 4 的行⑰中, 如果当前找不到向量块, 但还存在未处理的非零元, 则需要将它们进行零元填充, 以适应于向量化处理. 对于未处理的非零元的分布, 最多存在$d-1$个行和$d-1$个列(如果有大于等于d的行或列, 则会被加入到向量块中), 本文以图 5 为例说明零元的填充方法.

该填充方法与算法 2 和算法 3 相似, 只是不需要再考虑能否将 SIMD 的数据通道填满非零元, 该方法首先选取非零元数量最多的行, 根据算法 2 和算法 3 的相同思路, 计算能获取的行向量(图 5 中{a,b,c})和列向量(图 5 中{a,e,f})的访存跨度, 根据它们的大小选择对行向量或者列向量进行零元填充. 此时选择了{a,e,f}填充为列向量块, 但填充零元的位置需要结合向量化运算时“gather”和“scatter”的访存, 选择跨度最小的位置填充, 如图中(4,1)的位置. 接下来, 还有{b,c}要进行填充, 经分析, 在(2,3)和(2,4)的位置进行填充最优, 并将{b,0,0,c}作为行向量块.

3.3 调度优化

在现代 CPU 架构中, 通常采用多核(multi-core)

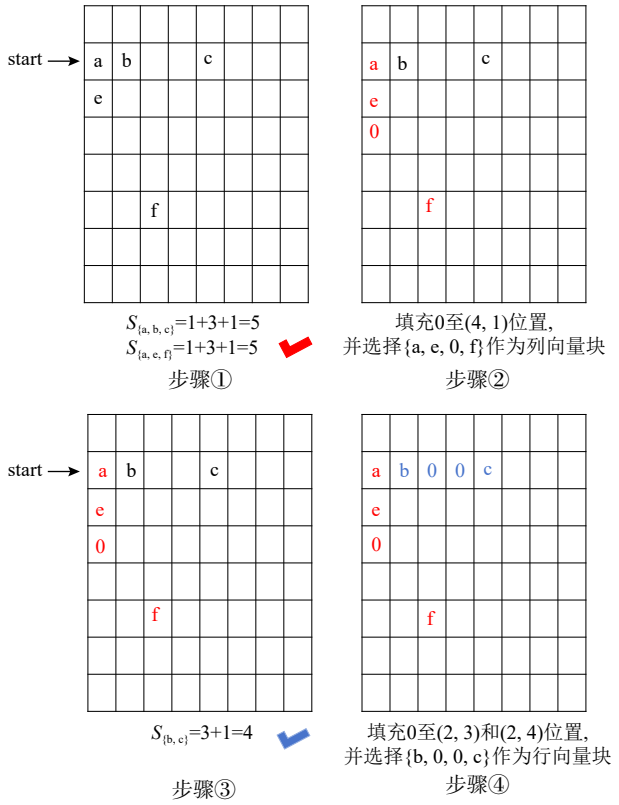


Fig. 5 A method of zero filling for vector block

图 5 向量块零元填充方法

架构，因此在优化过程中还需要考虑如何支持多线程并行处理的需求. 在经过稀疏模式分析后，可以得到多个可以向量化处理的数据块VP. 调度优化目标是将向量数据块VP按照设定的线程数进行划分，使得每个线程尽可能获得均匀的计算负载. 该优化方法如算法 5 所示，首先对VP中的数据块进行升序排列，由于每个数据块的长度为d，对于任意 2 个数据块，可以先比较第 1 个元素的行列位置，如果相同就比较第 2 个，依次类推. 算法 5 先按线程数对VP进行平均划分，然而在多线程运行过程中，考虑到不同线程分配的向量块可能存在相同的行，在写回到结果向量$\boldsymbol{y}$过程中，可能存在线程冲突的问题，如图 6 所示. 为了避免不同的线程同时写向量$\boldsymbol{y}$中相同的列，算法 5 先遍历矩阵的每个行r，再根据每个VP_i是否存在行位置等于r的数据块，将这些VP_i和冲突的数据块记录下来，从记录的VP_i选择最小的数据块数量t，将有冲突的数据块由t进行处理（行④~⑨）.

算法 5. 稀疏模式分析函数.

输入：向量化数据块VP，线程数T，稀疏矩阵行维度M；

输出：T个互不相交的数据块$\{VP_1, VP_2, \cdots, VP_T\}$，且$VP_1 \cup \cdots \cup VP_T = VP$.

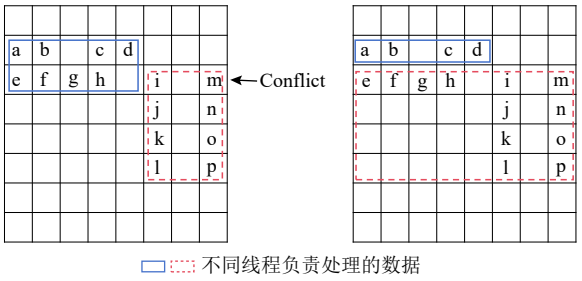


Fig. 6 Avoidation of thread conflict

图 6 避免线程冲突

① 对VP按每个数据块的行列位置升序排列；

② $avg \leftarrow |VP|/T$；

③ 按照每个$VP_i(1 \leqslant i \leqslant T)$的数据块数量为$avg$进行均匀划分；

④ for r = 1, 2, ⋯, M do

⑤ $\omega \leftarrow$存在数据块的行位置等于r的$VP_i(1 \leqslant i \leqslant T)$；

⑥ $t \leftarrow \arg\min|f|, \forall f \in \omega$；

⑦ $\forall f \in \omega, f \neq t$，将$f$中所有行位置等于$r$的数据块去除，并加入到$t$中；

⑧ end for

此外，在向量化执行的过程中，同一线程在计算不同的向量块时可以进行融合. 线程在计算{i, j, k, l}和{m, n, o, p}时，由于它们计算结果写入向量$\boldsymbol{y}$的相同位置，可以使用同一个向量寄存器对它们的结果进行累加，最后一起写入$\boldsymbol{y}$，从而减少访存次数，过程描述如图 6 所示.

3.4 代码生成

根据 3.1~3.3 节分析，最后代码生成模块会产生向量化的多线程 SpMV 内核，以及将稀疏矩阵转化为向量数据块 VP 并存储到磁盘. 在运行时，主线程先将转化后的 VP 加载到内存后创建多个工作线程，每个线程调用相应的 SpMV 代码，并对其负责的VP_i进行独立计算.

4 实　　验

4.1 实验配置

1）实验平台. 基于 Intel Xeon、Intel Core 和鲲鹏 920 开展实验对比. 其中 Intel Xeon 平台包含 16 个处理器核，每个核可开启 2 个线程，最多有 32 个线程，支持 AVX 512 向量指令集，内存为 512 GB；Intel Core 平台包含 14 个处理器核，每个核可开启 2 个线程，最多可开启 28 个线程，支持 AVX 512 向量指令集，内存为 64 GB. 鲲鹏 920 包含 64 个处理器核，每个核有

1个线程, 最多可开启64个线程, 支持ARM NEON向量指令集, 内存为256 GB, 表1列出本文实验平台的详细参数.

Table 1 Configuration Parameters of Experimental Platforms

表1 实验平台配置参数

配置/参数	Intel Xeon	Intel Core	鲲鹏920
处理器	Silver 4110	i5-13600KF	KP920
核心数	16	14	64
每核心线程数	2	2	1
L1 Cache	512 KB	1.2 MB	8 MB
L2 Cache	16 MB	20 MB	64 MB
L3 Cache	22 MB	24 MB	256 MB
内存大小	512 GB	64 GB	256 GB

2)数据集. 本文从稀疏矩阵集合中选取10个有代表性的稀疏矩阵, 覆盖如最优化、流体力学、线性规划等多个领域, 其规模和非零元素分布模式各异. 详细信息如表2所示.

Table 2 Sparse Matrices Datasets

表2 稀疏矩阵数据集

数据集	维度	非零元数量
com-Orkut	3 072 441	234 370 166
dblp-2010	326 186	1 615 400
circuit5M	5 558 326	59 524 291
road_central	14 081 816	33 866 826
soc-Pokec	1 632 803	30 622 564
nv2	1 453 908	37 475 646
StocF-1465	1 465 137	21 005 389
flickr	820 878	9 837 214
mc2depi	525 825	2 100 225
webbase-1M	1 000 005	3 105 536

3)对比方法. 本文将目前最先进的张量编译器TACO和TVM, 以及Intel的MKL数学库作为对比对象, 并对生成的代码统一使用GCC编译器, "-O3 -fopenmp"的编译参数生成可执行程序. 在鲲鹏920的实验中, 由于ARM NEON不支持"gather"和"scatter"指令, 在稀疏模式分析过程中, 通常只能在每一行中寻找连续的非零元素, 或对行进行填充以优化其结构. 考虑到实验过程中的数据波动, 为了更准确地评估SpMV在不同数据集上的性能, 本文采用每个数据集运行100次后的平均时间作为性能指标. 同时, 还对单精度和双精度的SpMV进行了测试, 进一步验证这些方法在不同精度下的表现.

4.2 SpMV性能分析

图7所示为不同方法在多线程场景下(Intel Xeon 64线程, Core 28线程, KP920 64线程)的性能比较. 本文使用SparseMode相对于TACO、TVM和MKL的加速比作为实验的可视化结果. 柱状图第1列表示SparseMode对TACO的加速比, 柱状图第2列表示SparseMode对TVM的加速比, 柱状图第3列表示SparseMode对MKL的加速比.在Intel平台上, SparseMode与TVM的比较中, 表现出较为显著的加速比. 在多数数据集中, SparseMode对TVM的加速比都在1.5以上(如nv2和StocF-1465), 表明SparseMode相较于TVM具有显著的性能提升. SparseMode相较于TACO的加速比也显示出良好的性能提升, 尤其是在某些较大或更稀疏的矩阵数据集上. 例如, 在webbase-1M和StocF-1465数据集上, SparseMode相较于TACO取得了更高的加速比. 而在flikcr数据集上, SparseMode的性能较差, 主要是flickr相较于其他数据集稀疏度较低. 此外, 我们对比了Intel上的MKL函数库, 针对大多数稀疏矩阵, SparseMode对MKL表现出较显著的加速比, 最高可达2.52.

对于单精度和双精度运算, SparseMode的性能表现也有所不同, 主要原因在于SparseMode在计算过程中将非零元素存储在向量寄存器中. 对于AVX 512指令集来说, 每个向量寄存器能够存储的双精度数据的数量少于单精度数据. 因此, 双精度运算的性能略低于单精度运算. SparseMode对TACO, TVM和MKL的加速效果在不同数据集上有所不同, 这可能与数据集的稀疏性、矩阵规模以及计算模式等因素相关. 对于一些稀疏矩阵特别适合的场景(稀疏矩阵维度与非零元数量相近, 如webbase-1M和mc2depi), SparseMode能够显著提高计算效率.

在鲲鹏920平台上, SparseMode的性能要低于其在Intel平台的表现, 主要是因为测试的鲲鹏920平台的ARM NEON指令集没有"gather"和"scatter"指令, 导致在稀疏模式分析过程中只能在每一行中找连续的非零元素或对行进行填充, 因此会造成大量的零元素填充, 增大计算开销. 此外, ARM NEON指令集中每个向量寄存器的长度要小于AVX 512指令集, 因此单条指令能处理的数据量较小.

4.3 扩展性分析

为了探究SparseMode的可扩展能力, 本文在设置的不同线程数量条件下, 分析SparseMode与TVM, TACO和MKL的性能对比. 本文选取了在4.2节中SparseMode性能表现最优的mc2depi和webbase-1M

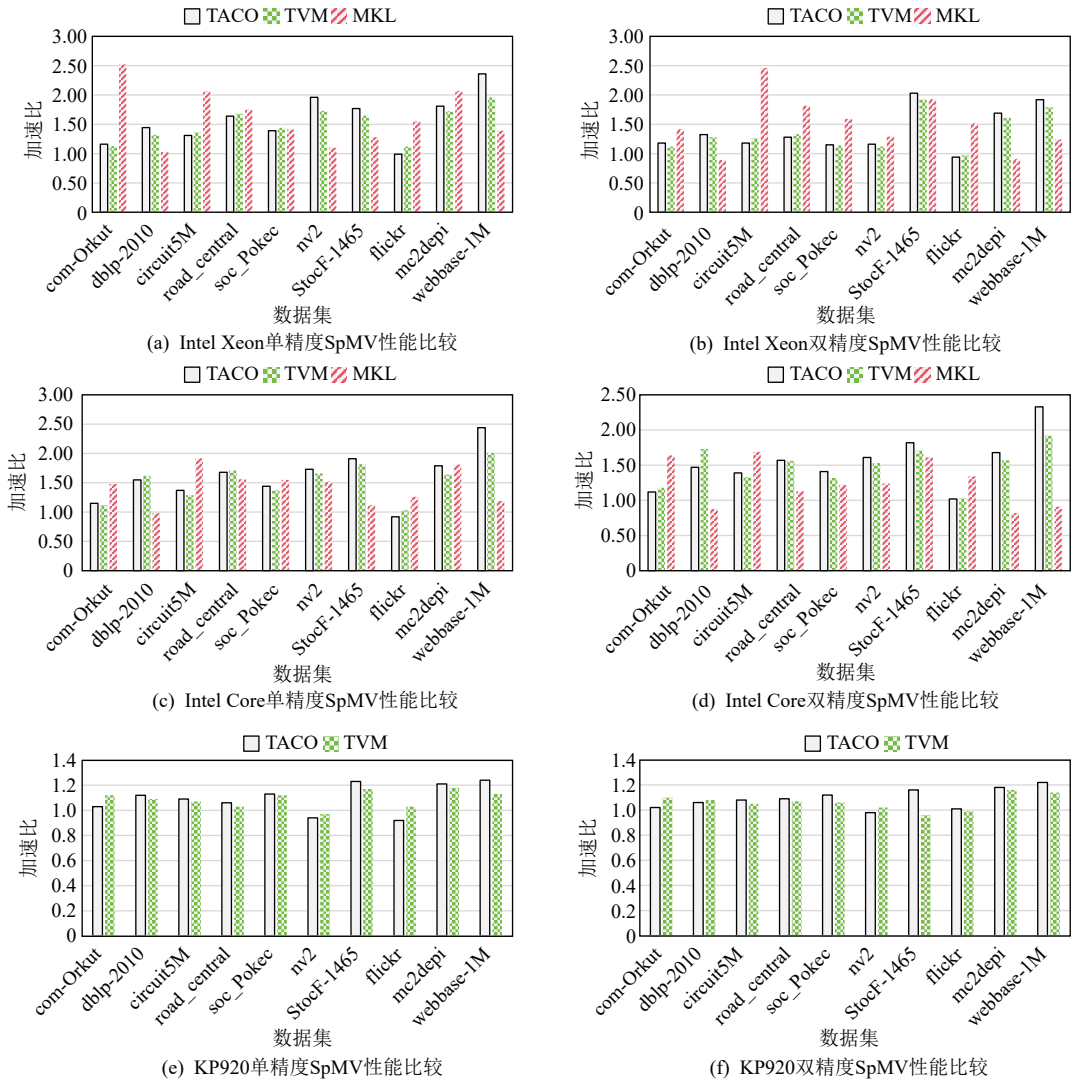


Fig. 7 Performance analysis of SpMV

图 7 SpMV 性能分析

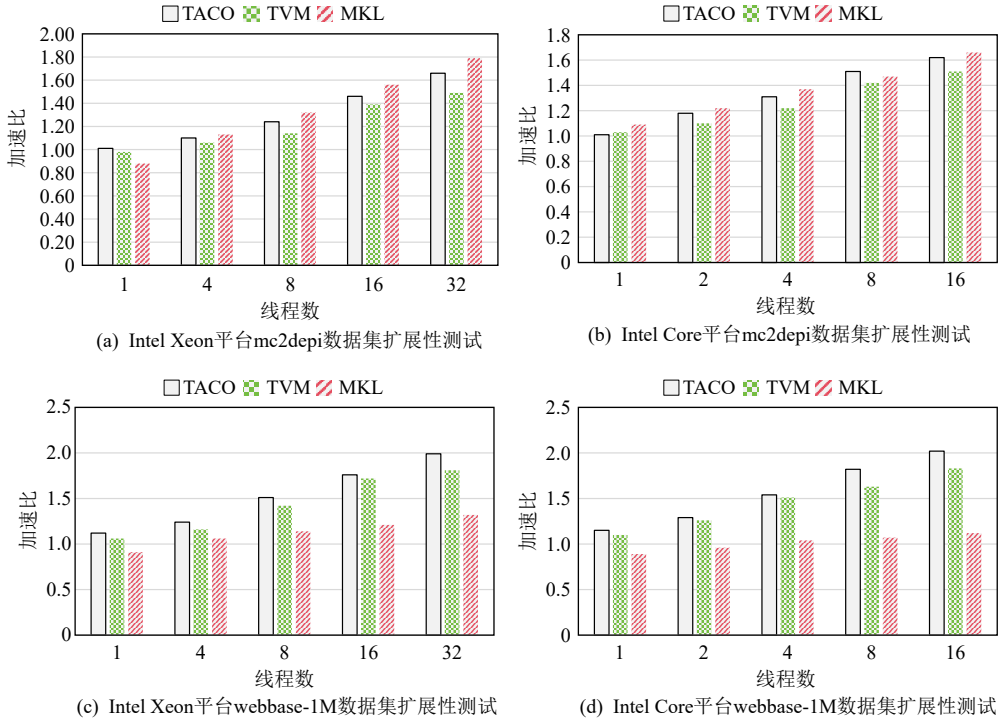
的数据集，并使用单精度运算进行测试实验结果，如图 8 所示.

具体而言，在 Intel Xeon 平台选取了{1, 4, 8, 16, 32}的线程数，而在 Intel Core 平台选取了{1, 2, 4, 8, 16}的线程数. SparseMode 在单线程情况下的性能低于 TVM 和 MKL，然而在大多数情况下的性能还是优于 TVM，TACO 和 MKL，尤其在较高线程数时，SparseMode 表现出了更强的加速性能. 从实验结果上看，SparseMode 表现出了较好的扩展性.

5 相关工作

SpMV[15] 的优化已有多年的研究历史. ALBUS[16] 设计了简单的逐行矢量计算机制，将单行内的非零元素依次加载到矢量寄存器中进行计算；VHCC[17] 通过连接多行来提高向量计算效率，同时将矩阵划分为 2 个维度，以增强数据的局部性；CVR[18] 将非零行顺序加载到 SIMD 通道中进行计算，同时对 SIMD 通道宽度的行进行处理，可以大大提高矢量计算效率；BCSR[19] 通过划分密集子块，将稀疏矩阵计算转化为整洁的数据计算模式；BCCOO[20] 采用分区策略，利用位压缩行索引存储，缓解内存压力；LAV[21] 为遵循幂律分布的矩阵设计了行/列分区机制，提高了数据访问局部性；有部分研究者探索在 GPU 上实现高效的 SpMV[22] 以及不同稀疏格式的性能分析[23-24].

现代编译器框架，如 MLIR[25] 等在稀疏矩阵运算优化方面已经取得了一定进展，它们通常采用符号化表示、自动调度、存储优化和向量化等方法来优化稀疏计算. 然而，这些框架在处理复杂和不规则的稀疏模式时仍然存在局限性，难以自动感知不同存



(a) Intel Xeon平台mc2depi数据集扩展性测试 (b) Intel Core平台mc2depi数据集扩展性测试

(c) Intel Xeon平台webbase-1M数据集扩展性测试 (d) Intel Core平台webbase-1M数据集扩展性测试

Fig. 8 Scalability analysis of SpMV

图 8 SpMV 扩展性分析

储模式, 并为特定硬件架构生成最优代码.

TACO[10] 是专门为稀疏张量计算设计的编译器框架, 主要用于自动生成优化的稀疏矩阵运算代码. 其核心方法包括: 稀疏迭代表示(sparse iteration representation)、自动格式转换、代码生成优化、自动向量化. 但它缺乏对不规则稀疏模式的自适应优化, 对硬件优化的适应性较差, 其 SIMD 向量化能力不足. TVM[11] 是用于深度学习编译优化的框架, 但其计算调度(scheduling)机制也可以用于稀疏计算优化. 但缺乏直接的稀疏模式感知能力、不具备对复杂稀疏模式的自动识别能力以及对不规则稀疏模式支持有限. MLIR[25] 在编译优化领域广泛应用, 也开始支持稀疏计算的优化. 但其对硬件优化的适配能力不足, MLIR 的 Sparse Tensor Dialects 仍需要用户手动指定稀疏模式, 而非自动分析数据分布情况.

近年来, 学术界和工业界针对不同稀疏模式的优化提出了模式分析、存储优化、并行计算和 SIMD 加速等多种方法. 稀疏感知集合归属于归一化最小均方算法 SM-NLMS[26], 用对数和 log-sum 函数取代 ZASM-NLMS 中的ℓ_1范数惩罚项, 应用于稀疏信道估计和回声消除. Kumar[27] 等人设计了自适应信号处理方案来利用系统脉冲响应的稀疏性质的自适应滤波方案. SPARQ[28] 在不同的表示粒度下利用非结构化和动态激活稀疏性, 实现了较小的精度降低和实用硬件. DZiG[29] 以稀疏感知的增量处理技术进行递归方式表达计算, 能够安全地识别和修剪更新, 它在保证 BSP 语义的同时, 保持了稀疏计算的效率. WACO[30] 通过考虑稀疏性模式、格式和调度, 设计基于神经网络的成本模型, 来预测稀疏张量程序的运行, 从而选择合适的稀疏格式和调度策略.

6 结 论

本文深入分析了 SpMV 计算过程中稀疏模式带来的性能优化瓶颈, 并探索了矩阵稀疏导致的不规则内存访问及难以应用向量化加速等挑战. 基于此, 本文构建了基于稀疏模式感知的编译优化框架 SparseMode, 用于高效的 SpMV 代码自动生成. 该方案通过设计 SpMV 领域描述语言 SpMV-DSL 来简洁地表达稀疏矩阵结构及计算操作, 并能够自适应不同稀疏模式. SparseMode 结合稀疏模式分析与编译优化, 动态选择最优的计算策略和内存布局, 最大化硬件性能. 通过并行化和向量化策略生成高效的 SpMV 算子代码, 充分利用 SIMD 指令加速计算. 经实验分析论证, 该方案相较于张量编译器如 TACO 和 TVM, 在提升 SpMV 计算性能方面具有显著优势.

作者贡献声明: 王昊天提出方案和撰写论文; 丁

岩构建实验并提供论文思路；何贤浩实现代码并完成测试；肖国庆改进实验；阳王东指导论文写作并修改论文.

参 考 文 献

[1] Yan Zhiyuan, Xie Biwei, Bao Yungang. HVMS: A hybrid vectorization-based optimization mechanism for SpMV[J]. Journal of Computer Research and Development, 2024, 61(12): 2969−2984 (in Chinese)
(颜志远, 解壁伟, 包云岗. HVMS: 基于混合向量化的 SpMV 优化机制[J]. 计算机研究与发展, 2024, 61(12): 2969−2984)

[2] Chen Jie. Graph neural preconditioners for iterative solutions of sparse linear systems[J]. arXiv preprint, arXiv: 2406.00809, 2024

[3] Shi Xiangting, Zhang Yakang, Pujahari A, et al. When latent features meet side information: A preference relation based graph neural network for collaborative filtering[J]. Expert Systems with Applications, 2025, 260: 125423

[4] Tanveer M, Rajani T, Rastogi R, et al. Comprehensive review on twin support vector machines[J]. Annals of Operations Research, 2024, 339(3): 1223−1268

[5] Li Jiajia, Zhang Xiuxia, Tan Guangming, et al. Research on selecting optimal storage formats for sparse matrix multiplication[J]. Journal of Computer Research and Development, 2014, 51(4): 882−894 (in Chinese)
(李佳佳, 张秀霞, 谭光明, 等. 选择稀疏矩阵乘法最优存储格式的研究[J]. 计算机研究与发展, 2014, 51(4): 882−894)

[6] Yuan E, Zhang Yunquan, Liu Fangfang, et al. Research on automatic performance optimization implementation techniques for SpMV and its applications[J]. Journal of Computer Research and Development, 2009, 46(7): 1117−1126 (in Chinese)
(袁 娥, 张云泉, 刘芳芳, 等. SpMV 的自动性能优化实现技术及其应用研究[J]. 计算机研究与发展, 2009, 46(7): 1117−1126)

[7] Li Kenli, Yang Wangdong, Li Keqin. Performance analysis and optimization for SpMV on GPU using probabilistic modeling[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 26(1): 196−205

[8] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proc of the Int Symp on Code Generation and Optimization (CGO 2004). Piscataway, NJ: IEEE, 2004: 75−86

[9] Gough B J, Stallman R. An Introduction to GCC[M]. Bristol, UK: Network Theory Limited, 2004

[10] Kjolstad F, Kamil S, Chou S, et al. The tensor algebra compiler[C]// Proc of the ACM on Programming Languages (OOPSLA). New York: ACM, 2017: 1−29

[11] Chen Tianqi, Moreau T, Jiang Ziheng, et al. TVM: An automated end-to-end optimizing compiler for deep learning[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI 18). Berkeley, CA: USENIX Association, 2018: 578−594

[12] Sun Xiangzheng, Zhang Yunquan, Wang Ting, et al. Adaptive performance optimization for SpMV on diagonal sparse matrices[J]. Journal of Computer Research and Development, 2013, 50(3): 648−656 (in Chinese)
(孙相征, 张云泉, 王婷, 等. 对角线稀疏矩阵的 SpMV 自适应性能优化[J]. 计算机研究与发展, 2013, 50(3): 648−656)

[13] Gan Xinbiao, Tan Wen, Liu Jie. CSR-based large-scale graph storage optimization using bidirectional bitmaps[J]. Journal of Computer Research and Development, 2021, 58(3): 458−466 (in Chinese)
(甘新标, 谭雯, 刘杰. 基于双向位图的 CSR 大规模图存储优化[J]. 计算机研究与发展, 2021, 58(3): 458−466)

[14] Wang Y E, Wei G Y, Brooks D. Benchmarking TPU, GPU, and CPU platforms for deep learning[J]. arXiv preprint, arXiv: 1907.10701, 2019

[15] Gao Jianhua, Liu Bingjie, Ji Weixing, et al. A systematic literature survey of sparse matrix-vector multiplication[J]. arXiv preprint, arXiv: 2404.06047, 2024

[16] Bian Haodong, Huang Jianqiang, Liu Lingbin, et al. ALBUS: A method for efficiently processing SpMV using SIMD and load balancing[J]. Future Generation Computer Systems, 2021, 116: 371−392

[17] Tang Waiteng, Zhao Ruizhe, Lu Mian, et al. Optimizing and auto tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi[C]//Proc of the 13th Annual IEEE/ACM Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2015: 136−145

[18] Xie Biwei, Zhan Jianfeng, Liu Xu, et al. CVR: Efficient vectorization of SpMV on x86 processors[C]//Proc of the 16th Annual IEEE/ACM Int Symp on Code Generation and Optimization. New York: ACM, 2018: 149−162

[19] Eberhardt R, Hoemmen M. Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures[C]//Proc of the 2016 IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW). Piscataway, NJ: IEEE, 2016: 663−672

[20] Yan Shengen, Li Chao, Zhang Yunquan, et al. yaSpMV: Yet another SpMV framework on GPUS[J]. ACM SIGPLAN Notices, 2014, 49(8): 107−118

[21] Yesil S, Heidarshenas A, Morrison A, et al. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization[C/OL]//Proc of the 2020 Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020[2023 07−31]. https://ieeexplore.ieee.org/ document/9355205

[22] Filippone S, Cardellini V, Barbieri D, et al. Sparse matrix-vector multiplication on GPGPUs[J]. ACM Transactions on Mathematical Software, 2017, 43(4): 1−49

[23] Mohammed T, Mehmood R. Performance enhancement strategies for sparse matrix-vector multiplication (SPMV) and iterative linear solvers[J]. arXiv preprint, arXiv: 2212.07490, 2022

[24] Langr D, Tvrdik P. Evaluation criteria for sparse matrix storage formats[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(2): 428−440

[25] Lattner C, Amini M, Bondhugula U, et al. MLIR: Scaling compiler infrastructure for domain specific computation[C]//Proc of 2021

IEEE/ACM Int Symp on Code Generation and Optimization (CGO). Piscataway, NJ: IEEE, 2021: 2−14

[26] Li Yingsong, Wang Yanyan, Jiang Tao. Sparse-aware set-membership NLMS algorithms and their application for sparse channel estimation and echo cancelation[J]. AEU-International Journal of Electronics and Communications, 2016, 70(7): 895−902

[27] Kumar K, Pandey R, Karthik M L N S, et al. Robust and sparsity-aware adaptive filters: A review[J]. Signal Processing, 2021, 189: 108276

[28] Shomron G, Gabbay F, Kurzum S, et al. Post-training sparsity-aware quantization[J]. Advances in Neural Information Processing Systems, 2021, 34: 17737−17748

[29] Mariappan M, Che J, Vora K. DZiG: Sparsity-aware incremental processing of streaming graphs[C]//Proc of the 16th European Conf on Computer Systems. New York: ACM, 2021: 83−98

[30] Won J, Mendis C, Emer J S, et al. WACO: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2023: 920−934



Wang Haotian, born in 1997. PhD, assistant researcher, postdoctor. Member of CCF. His main research interests include high performance computing and compiler optimization.

王昊天, 1997 年生. 博士，助理研究员，博士后，CCF 会员. 主要研究方向为高性能计算、编译优化.



Ding Yan, born in 1992. PhD, assistant professor. Member of CCF. His main research interests include parallel computing, mobile edge computing, and artificial intelligence.

丁　岩, 1992 年生. 博士，助理教授. CCF 会员. 主要研究方向为并行计算、移动边缘计算、人工智能.



He Xianhao, born in 1995. PhD candidate. His main research interests include sparse compiler and artificial intelligence.

何贤浩, 1995 年生. 博士研究生. 主要研究方向为稀疏编译、人工智能.



Xiao Guoqing, born in 1988. PhD, professor. His main research interests include high performance computing and intelligent computing.

肖国庆, 1988 年生. 博士，教授. 主要研究方向为高性能计算、智能计算.



Yang Wangdong, born in 1974. PhD, professor. His main research interests include high performance computing and parallel numerical algorithms.

阳王东, 1974 年生. 博士，教授. 主要研究方向为高性能计算、并行数值算法.