计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(6): 1434−1442, 2025

基于缓存数据重用的稀疏矩阵向量乘序列优化

徐传福　邱昊中　车永刚
（国防科技大学计算机学院　长沙　410073）
（xuchuanfu@nudt.edu.cn）

Optimizing Sequences of Sparse Matrix-Vector Multiplications via Cache Data Reuse

Xu Chuanfu, Qiu Haozhong, and Che Yonggang
（*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073）

Abstract　Many high performance computing (HPC) applications such as solving sparse linear systems involve the calculation of sequences of sparse matrix-vector multiplications (SpMV) like $\boldsymbol{Ax}$, $\boldsymbol{A}^2\boldsymbol{x}$, ⋯, $\boldsymbol{A}^s\boldsymbol{x}$, which is known as the sparse matrix power kernel (MPK). Since MPK calls SpMV using the same sparse matrix, reusing the matrix elements in cache, instead of repeatedly loading them from main memory, can potentially alleviate the memory-constrained problem for SpMV and enhance the performance of MPK. However, reusing the matrix introduces data dependencies between subsequent SpMVs. Prior work mainly either focuses on optimizing an individual SpMV invocation, or introduces significant overheads for cache data reuse in MPK. We propose a cache-aware MPK (Ca-MPK) to optimize MPK via cache data reuse. Based on the dependency graph of a sparse matrix, an architecture-aware recursive partitioning of the graph is designed to obtain subgraphs/submatrices which are fit into cache. Separating subgraphs (i.e., separators) are constructed to decouple data dependencies among subgraphs. Then cache data reuse is achieved by executing sequences of SpMVs on subgraphs with a specified order. Performance evaluation demonstrates that Ca-MPK outperforms the MPK implementations based on the Intel OneMKL library and the state-of-the-art approach, with an average speedup of up to about 1.57 times and 1.40 times, respectively.

Key words　sparse matrix-vector multiplications; matrix power kernel; cache data reuse; data dependencies; solution of sparse linear systems

摘　要　稀疏线性方程组求解等高性能计算应用常常涉及稀疏矩阵向量乘（SpMV）序列 $\boldsymbol{Ax}, \boldsymbol{A}^2\boldsymbol{x}, \cdots, \boldsymbol{A}^s\boldsymbol{x}$ 的计算. 上述 SpMV 序列操作又称为稀疏矩阵幂函数（matrix power kernel, MPK）. 由于 MPK 执行多次 SpMV 且稀疏矩阵保持不变，在缓存（cache）中重用稀疏矩阵，可避免每次执行 SpMV 均从主存加载 $\boldsymbol{A}$，从而缓解 SpMV 访存受限问题，提升 MPK 性能. 但缓存数据重用会导致相邻 SpMV 操作之间的数据依赖，现有 MPK 优化多针对单次 SpMV 调用，或在实现数据重用时引入过多额外开销. 提出了缓存感知的 MPK（cache-aware MPK, Ca-MPK），基于稀疏矩阵的依赖图，设计了体系结构感知的递归划分方法，将依赖图划分为适合缓存大小的子图/子矩阵，通过构建分割子图解耦数据依赖，根据特定顺序在子矩阵上调度执行 SpMV，实现缓存数据重用. 测试结果表明，Ca-MPK 相对于 Intel OneMKL 库和最新 MPK 实现，平均性能提升分别多达约 1.57 倍和 1.40 倍.

关键词　稀疏矩阵向量乘；矩阵幂函数；缓存数据重用；数据依赖；稀疏线性方程组求解

收稿日期：2025-03-01；**修回日期**：2025-04-09
基金项目：国家自然科学基金项目（62272474）
This work was supported by the National Natural Science Foundation of China (62272474).
通信作者：邱昊中（qiuhaozhong16@nudt.edu.cn）

中图法分类号 TP311.13；TP309

DOI: 10.7544/issn1000-1239.202550125 **CSTR:** 32373.14.issn1000-1239.202550125

在稀疏线性方程组求解等高性能计算(high performance computing, HPC)应用中,很多算法需要涉及稀疏矩阵向量乘(sparse matrix-vector multiplications, SpMV)序列 $\boldsymbol{Ax}, \boldsymbol{A}^2\boldsymbol{x}, \cdots, \boldsymbol{A}^s\boldsymbol{x}$ 的计算. 上述 SpMV 序列操作又称为稀疏矩阵幂函数(matrix power kernel, MPK),其中稀疏矩阵 $\boldsymbol{A}$ 保持不变,$\boldsymbol{x}$ 为输入向量,s 为给定常数次幂[1-3]. 例如,通信避免(communication-avoiding) Krylov 子空间法通过调用 MPK,一次迭代计算 s 个 Krylov 基向量[4]. 此外多重网格方法、特征值方法等也涉及对 MPK 的调用. 优化 MPK 性能对于提升大型稀疏线性方程组求解等 HPC 应用效率至关重要.

SpMV 是 HPC、人工智能等很多领域的基础内核. 由于当前 HPC 体系结构上 SpMV 强访存受限特点,很多工作开展了 SpMV 访存性能优化研究,例如设计新的存储格式等[5-8]. 已有工作多针对单次 SpMV 执行开展性能优化,没有考虑 MPK 针对同一稀疏矩阵多次调用 SpMV 的特点. 利用 MPK 特点可开发时间局部性(temporal locality),在缓存中重用稀疏矩阵 $\boldsymbol{A}$,避免每次 SpMV 调用从主存中重复加载 $\boldsymbol{A}$,从而缓解 SpMV 访存受限问题,提升 MPK 性能.

然而,在不同次幂 SpMV 操作之间(例如 $\boldsymbol{A}^i\boldsymbol{x}$ 和 $\boldsymbol{A}^{i-1}\boldsymbol{x}$ 之间)重用矩阵 $\boldsymbol{A}$,会导致数据依赖. 以往 MPK 优化工作主要针对分布式并行平台,通过重用矩阵 $\boldsymbol{A}$ 减少通信和同步的频率[9-12]. 共享内存平台上 MPK 的缓存数据重用工作相对较少,通常借鉴模板计算(stencil computation)里面的经典分块策略[13-16],往往仅适用于结构化稀疏矩阵,推广到普通稀疏矩阵时会导致较大的冗余计算和非规则数据访问开销,影响 MPK 并行效率.

本文提出了缓存感知的 MPK 方法 Ca-MPK,通过缓存数据重用优化共享内存平台 MPK 性能:

1)基于稀疏矩阵的依赖图,设计了体系结构感知的递归划分方法. 根据并行核数和缓存大小,将矩阵图划分为若干适合缓存大小的子图/子矩阵,通过构建分割子图解耦数据依赖,通过子图重排提升 SpMV 数据局部性;

2)根据图划分结果,设计了基于行和基于子矩阵的 2 种 SpMV 并行方法,按照特定顺序在子矩阵上调度执行 SpMV 序列,可实现缓存数据重用;

3)在 x86 和 ARM 处理器上采用 SuiteSparse 矩阵集中的 302 个稀疏矩阵进行了测试分析,相对于 Intel OneMKL 库和最新 MPK 实现[2],平均性能提升多达 1.57 倍和 1.40 倍. 将 Ca-MPK 应用于通信避免(或 s-Step)共轭梯度求解器和稳定双共轭梯度求解器,平均性能提升多达 1.42 倍和 2.00 倍.

1 相关工作

早期 MPK 实现主要针对通信避免算法等应用场景展开. 以稀疏线性方程组求解为例,传统 Krylov 子空间方法通常每次迭代调用一次 SpMV,产生一个 Krylov 子空间基向量,并行性能通常受限于通信或数据访问开销. 通信避免的 Krylov 子空间法[9-12]每次迭代调用 s 次 SpMV,产生 s 个 Krylov 子空间基向量,从而最小化通信开销.

以往 MPK 优化工作主要针对分布式并行平台,通过重用矩阵减少通信和同步的频率. 针对 MPK 的缓存重用多借鉴了模板计算中的经典分块策略,通常需要针对给定的次幂,确定矩阵分块的邻居[13-16]. 由于模版计算对应于结构化稀疏矩阵,这种方法推广到普通稀疏矩阵的 MPK 时通常会导致较大的冗余计算和非规则数据访问开销,影响 MPK 并行效率. 文献 [2] 针对 MPK 提出了一种基于分层的缓存分块方法,可适用于任意稀疏矩阵,是最新的相关工作. 该方法利用递归代数着色引擎(recursive algebraic coloring engine, RACE)[17] 采用距离 k 着色(distance-k coloring)算法将矩阵行划分为若干层. 这种方法能够满足 MPK 中缓存数据重用的数据依赖要求,但限制了并行粒度,每次仅在 1 层内实现并行计算,因而并行性很低且同步开销大. 为了解决上述问题,文献 [2] 引入点对点同步(point-to-point synchronization)及分层粗化(level coarsening)等优化策略,但次幂 s 增加时,MPK 性能仍然下降严重. 文献 [1] 引入了计算流水线,实现了一种重用矩阵行(而非矩阵分块)的细粒度缓存数据重用方法. 该方法实现相对简单,但需要利用图着色实现并行,因而同步开销大,且影响数据局部性[18].

2 预备知识

2.1 SpMV

SpMV 操作将稀疏矩阵 $\boldsymbol{A}$ 乘以稠密输入向量 $\boldsymbol{x}$,

获得稠密输出向量 $\boldsymbol{b}$. 稀疏矩阵通常采用某种压缩格式仅存储其非零元和对应位置, 从而节省计算和存储. 图 1(b)上部给出了采用流行的 CSR(compressed sparse row)格式存储的图 1(a)中的稀疏矩阵 $\boldsymbol{A}$. CSR 格式采用 2 个数组用于存储非零元值(*value*)及其对应的列索引(*colIdx*). 此外, 还采用行指针(*rowPtr*)表示每行的非零元数量. 算法 1 给出了基于 CSR 格式实现的串行版本 SpMV 伪代码.

算法 1. 基于 CSR 的 SpMV 串行实现.

输入: 稀疏矩阵 $\boldsymbol{A}$, 向量 $\boldsymbol{x}$;

输出: 向量 $\boldsymbol{b}$.

① for i←0 to N do
② sum←0;
③ for j←$\boldsymbol{A}$.*rowPtr*[i] to $\boldsymbol{A}$.*rowPtr*[i+1] do
④ sum←sum+$\boldsymbol{A}$.*value*[j]·$\boldsymbol{x}$[$\boldsymbol{A}$.*colIdx*[j]];
⑤ end for
⑥ $b[i]$←$b[i]$+sum;
⑦ end for

2.2 MPK 及其在稀疏迭代求解器中的应用

MPK 针对同一稀疏矩阵 $\boldsymbol{A}$, 反复执行 SpMV, 获得 $\boldsymbol{Ax}$, $\boldsymbol{A}^2\boldsymbol{x}$, …, $\boldsymbol{A}^s\boldsymbol{x}$ 或其线性组合如 $\boldsymbol{Ax}+\boldsymbol{A}^2\boldsymbol{x}$ 等. 算法 2 给出了传统 MPK 实现的伪代码, 即连续调用 s 次 SpMV 函数. 此时, 如缓存容量较小, 无法存储整个稀疏矩阵 $\boldsymbol{A}$, 则 $\boldsymbol{A}$ 可能需要从主存加载 s 次.

算法 2. 传统 MPK 实现.

输入: 稀疏矩阵 $\boldsymbol{A}$, 二维数组 x, 次幂 s;

输出: 二维数组 x.

① for i←0 to s do
② $SpMV(\boldsymbol{A}, x[:,i], x[:,i+1])$;
③ end for

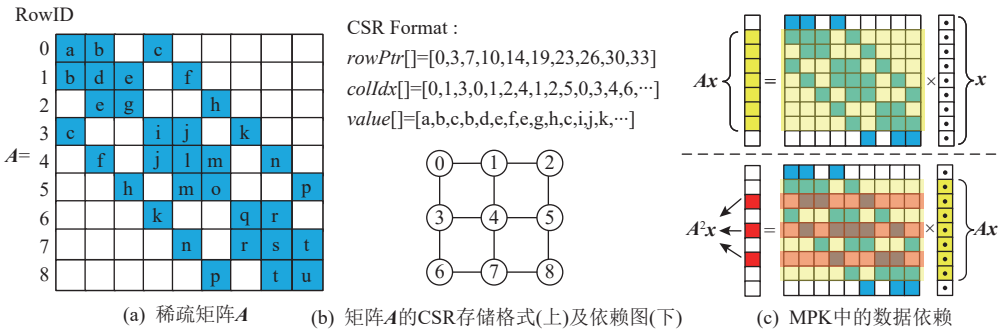
MPK 是 Krylov 子空间方法、多重网格方法、多项式预条件子、特征值问题等很多 HPC 应用的关键步骤. 本文将优化后的 MPK 应用于 s-Step 的共轭梯度(conjugate gradient, CG)求解方法和稳定双共轭梯度(biconjugate gradient stabilized, BiCGStab)求解方法. CG 方法和 BiCGStab 方法是求解对称正定和非对称正定稀疏线性方程组的主流方法, 算法 3 给出了 s-Step CG 求解方法的伪代码[19]. 该方法每次调用 MPK 产生 s 个 Krylov 基向量(行③), 因此在分布式并行时相对于传统 Krylov 求解器可潜在节省 s 倍通信开销. BiCGStab 求解方法的算法流程参见文献 [19].

算法 3. s-Step CG 求解方法.

输入: 对称正定稀疏矩阵 $\boldsymbol{A}$, 初始值 x_0;

输出: 解向量 $\boldsymbol{x}$.

① r_0←$\boldsymbol{b}$−$\boldsymbol{A}x_0$, k=i×s;
② for i←0 to convergence do
③ $\boldsymbol{T}$←$[r_k, \boldsymbol{A}r_k, \cdots, \boldsymbol{A}^s r_k]$; /*调用 MPK*/
④ $\boldsymbol{R}_i$←$[r_k, \boldsymbol{A}r_k, \cdots, \boldsymbol{A}^{s-1} r_k]$; /*从 T 中抽取 R*/
⑤ $\boldsymbol{P}_i$←$[\boldsymbol{A}r_k, \boldsymbol{A}^2 r_k \cdots, \boldsymbol{A}^{s-1} r_k]$; /*从 T 中抽取 P*/
⑥ if i==0 then; /*计算方向 P*/
⑦ P_i←R_i;
⑧ else
⑨ C_i←$-Q_i^{\mathrm{T}} P_{i-1}$; /*块点积计算*/
⑩ 求解 $W_{i-1}B_i = C_i$; /*计算标量 δ*/
⑪ P_i←$R_i + P_i B_i$; /*块点积计算*/
⑫ end if
⑬ W_i←$Q_i^{\mathrm{T}} P_i$;
⑭ g_i←$P_i^T r_i$;
⑮ 求解 $W_i a_i = g_i$; /*计算标量 α*/
⑯ $\boldsymbol{x}_{k+s}$←$\boldsymbol{x}_k + P_i a_i$; /*更新解*/
⑰ r_{k+s}←$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{k+s}$; /*计算残差*/
⑱ if $\|r_{k+s}\|_2 / \|r_0\|_2 \leqslant tol$ then
⑲ stop;
⑳ end if
㉑ end for



(a) 稀疏矩阵$\boldsymbol{A}$ (b) 矩阵$\boldsymbol{A}$的CSR存储格式(上)及依赖图(下) (c) MPK中的数据依赖

Fig. 1 Storage format, data dependencies and data dependency graph for a sparse matrix in MPK

图 1 MPK 中的稀疏矩阵存储格式、数据依赖和数据依赖图

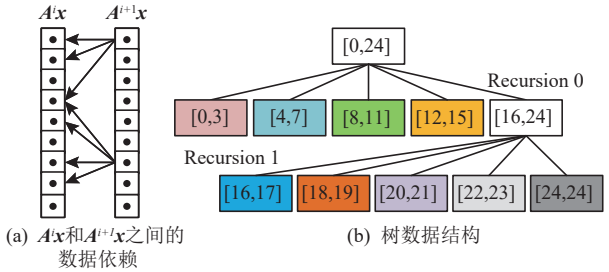
对于共享内存平台，可针对 MPK 特点实现矩阵在缓存中的重用，但缓存重用会导致多次 SpMV 执行之间的数据依赖. 图 1(c)以 s=2 为例描述了上述数据依赖. 假定图中黄色行对应的子矩阵已加载入缓存用于计算 $\boldsymbol{Ax}$，可利用该子矩阵的一部分(红色的行)计算出 $\boldsymbol{A}^2\boldsymbol{x}$ 的一部分，既保持子矩阵在缓存的同时计算出多次 SpMV 结果. 可以观察出为了实现上述数据重用需要满足的数据依赖关系：重用的部分矩阵行(红色的行)包含的列索引必须位于原始子矩阵(黄色的行)包含的行索引的范围内.

3 缓存感知的 MPK 方法 Ca-MPK

本节介绍提出的缓存感知的 MPK(即 Ca-MPK)方法及其实现. Ca-MPK 核心思想是首先根据稀疏矩阵获得其依赖图，然后对依赖图进行体系结构感知的递归划分获得子图和分割子图(即子矩阵)，最后按特定顺序对子图和分割子图进行调度执行，从而实现缓存数据重用.

3.1 稀疏矩阵的依赖图

针对任意稀疏矩阵可构建依赖图(dependency graph, DG)，表示实现缓存数据重用时可能面临的数据依赖. 图 1(b)下部给出了图 1(a)中稀疏矩阵 $\boldsymbol{A}$ 对应的依赖图 $G=(V, E)$. 依赖图采用无向图的形式，图中每个顶点 $v\in V$ 表示矩阵的一行，每条边 $e\in E$ 对应该行的非零元(忽略对角线非零元). 依赖图中的边可表示由缓存数据重用导致的 SpMV 序列之间所有可能的依赖关系. 例如，图 2(a)给出了 2 次 SpMV 调用 $\boldsymbol{A}^{i+1}\boldsymbol{x}[0]$ 和 $\boldsymbol{A}^{i+1}\boldsymbol{x}[6]$ 的计算所依赖的 $\boldsymbol{A}^i\boldsymbol{x}$ 的特定元素，其中 $\boldsymbol{A}^{i+1}\boldsymbol{x}[0]$ 依赖于 $\boldsymbol{A}^i\boldsymbol{x}[0]$, $\boldsymbol{A}^i\boldsymbol{x}[1]$ 和 $\boldsymbol{A}^1\boldsymbol{x}[3]$，而 $\boldsymbol{A}^{i+1}\boldsymbol{x}[6]$ 依赖于 $\boldsymbol{A}^i\boldsymbol{x}[3]$, $\boldsymbol{A}^i\boldsymbol{x}[4]$, $\boldsymbol{A}^i\boldsymbol{x}[6]$ 和 $\boldsymbol{A}^1\boldsymbol{x}[7]$. 对于非对称矩阵 $\boldsymbol{A}$，通过 $\boldsymbol{A}+\boldsymbol{A}^{\mathrm{T}}$ 获得其对称形式对应的无向图，因此依赖图同时适用于所有类型的矩阵.



(a) $\boldsymbol{A}^i\boldsymbol{x}$和$\boldsymbol{A}^{i+1}\boldsymbol{x}$之间的数据依赖　(b) 树数据结构

注：(b)中方括号内数字为子矩阵行索引，[16, 24] 和 [24, 24] 分别是第 1 次和第 2 次递归的分割子图.

Fig. 2 Data dependency between neighbor powers and data structure after recursive partition in Ca-MPK

图 2 相邻幂次间的数据依赖及 Ca-MPK 递归划分后的数据结构

3.2 体系结构感知的递归划分

对于给定矩阵获得依赖图后，本文提出了一种体系结构感知的递归方法对图进行划分，主要考虑的因素是共享内存并行核心数及缓存大小，目的是在开发并行性的同时满足缓存数据重用要求. 首先将依赖图划分为 $numPart$ 个子图(subgraph)，可利用 Metis[20] 等图划分工具实现. 对于基于 CSR 的稀疏矩阵(假定每个非零元采用 8 字节双精度浮点存储，整数索引采用 4 字节整型存储)，每个子图包含不超过 $partSize$ 个非零元.

$$partSize = C\frac{L2cacheSize}{12}, \tag{1}$$

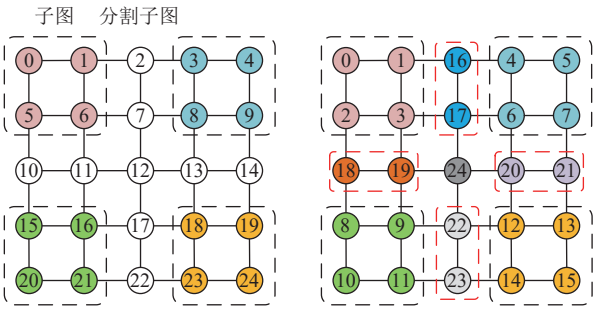
其中 $L2cacheSize$ 是处理器 L2 缓存大小(单位为字节)，C 是系数(取值为 [0.85, 1.05]，用于表示缓存中行偏移指针等其他数据的影响). 上述划分可保证每个子图(对应于子矩阵)全部加载到 L2 缓存中. $numPart$ 根据式(2)确定，其取值为处理器核数 $numCore$ 的整数倍，以确保所有子图在处理器核上均衡分布.

$$numPart = \left(\left\lfloor \frac{NNZ/partSize}{numCore} \right\rfloor + 1\right) \times numCore, \tag{2}$$

其中 NNZ 是总的矩阵非零元数.

完成上述子图划分后，从现有子图中找出所有在其他子图中包含邻居的顶点，从而构建一个特殊子图称为分割子图(separator). 接下来对所有子图(包括分割子图)的顶点进行重编号，确保每个子图顶点索引连续且分割子图在普通子图之后进行编号. 重编号可提升 SpMV 中访问输入/输出向量的数据局部性.

图 3 给出了经过 2 次递归划分($numPart$=4)后的图. 本文采用一个树指针数据结构 $treePtr$ 保存递归



(a) 划分为4个子图及1个分割子图 (未重排序)　(b) 第2次递归后 (含重排序)

Fig. 3 Recursive partition process of dependency graph in Ca-MPK

图 3 Ca-MPK 依赖图递归划分过程

划分的结果(即每个子图对应的起始行编号)，用于后续并行实现. 图 2(b)给出了图 3(b)对应的数据结构.

3.3 并行实现

基于稀疏矩阵的依赖图完成递归划分后，可根据划分结果实现并行计算. 由于通过分割子图消除了所有子图间的边(数据依赖)，每次递归获得的子图/子矩阵是独立的. 这些子矩阵由树数据结构中的起始行索引表示，通过按特定顺序调度执行可实现 MPK 中的缓存数据重用.

为了实现 Ca-MPK，需要对标准 SpMV 实现(参见算法 1)进行调整. 首先需要支持在给定的子图/子矩阵(由给定矩阵的行索引范围确定)上执行 SpMV. 其次，除了传统的按行 SpMV 并行，还需要支持按子图/子矩阵并行.

算法 4. uSpMV 函数.

输入：稀疏矩阵 $\boldsymbol{A}$，向量 $\boldsymbol{x}$，指针 *treePtr*；

输出：向量 $\boldsymbol{b}$.

① for *i*←*treePtr->st* to *treePtr->ed* do

② 算法 1 中行②~⑥；

③ end for

算法 5. puSpMV 函数.

输入：稀疏矩阵 $\boldsymbol{A}$，向量 $\boldsymbol{x}$，树指针 *treePtr*；

输出：向量 $\boldsymbol{b}$.

① for *i*←*treePtr->st* to *treePtr->ed* do in parallel

② 算法 1 中行②~⑥；

③ end for

算法 6. Ca_MPK 函数.

输入：稀疏矩阵 $\boldsymbol{A}$，二维数组 *x*，次幂 *s*，指针 *treeRoot*；

输出：二维数组 *x*.

① if *s*==1 then

② *puSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,0], $\boldsymbol{x}$[:,1], *treeRoot*)；

③ return；

④ end if

⑤ *i*←0；

⑥ *separator*←*treeRoot->separator*；

⑦ *puSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,0], $\boldsymbol{x}$[:,1], *separator*)；

⑧ while *i*< *s* do

⑨ for *j*←0 to *treeRoot->numParts* do in parallel

⑩ *uSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*], $\boldsymbol{x}$[:,*i*+1], *treeRoot-> son*[*j*])；

⑪ if *i*+1< *s* then

⑫ *uSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*+1], $\boldsymbol{x}$[:,*i*+2], *treeRoot-> son*[*j*])；

⑬ else stop

⑭ end if

⑮ end for

⑯ *i*←*i*+2；

⑰ if *i*< *s* then

⑱ *puSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*−1], $\boldsymbol{x}$[:,*i*], *separator-> separator*)；

⑲ for *j*←0 to *separator->numParts* do in parallel

⑳ *uSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*−1], $\boldsymbol{x}$[:,*i*], *separator-> son*[*j*])；

㉑ *uSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*], $\boldsymbol{x}$[:,*i*+1], *separator-> son*[*j*])；

㉒ end for

㉓ *puSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*], $\boldsymbol{x}$[:,*i*+1], *separator-> separator*)；

㉔ else

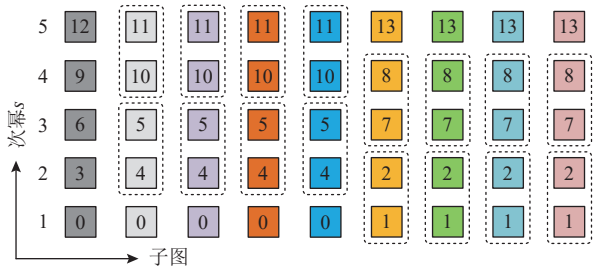
㉕ *puSpMV*($\boldsymbol{A}$, $\boldsymbol{x}$[:,*i*−1], $\boldsymbol{x}$[:,*i*], *separator*)；

㉖ end if

㉗ end while

算法 6 给出了 Ca-MPK 的伪代码. 根据上述需求，将串行版本 SpMV 封装为一个统一的函数 uSpMV(算法 4)，该函数可对给定的子矩阵(行索引位于 *treePtr->st* 和 *treePtr->ed* 之间)执行 SpMV 操作. 对应的按行并行版本封装在另一个函数 puSpMV(算法 5)中. Ca-MPK 基于 uSpMV 和 puSpMV 这 2 个函数实现. 其中，分割子图通过调用 puSpMV 实现按行并行，其他子图通过调用 uSpMV 执行，实现按子矩阵并行.

图 4 给出了运行算法 6 时图 3(b)中不同子图的执行顺序. 由于每个子图/子矩阵均可放入 L2 缓存，且所有数据冲突和数据依赖已消除，执行完当前子矩阵的次幂 *i* 后，可立即调度 SpMV 函数执行次幂(*i*+1)，从而实现该子矩阵在缓存中的重用. 以图 4 为例，标记执行顺序为 1 的子图和执行顺序为 2 的子图均实现了在 L2 缓存中的重用(图中表示为虚点长方



注：每个框代表一个子矩阵，框中数字为执行顺序，同一数字子矩阵可并行执行. 长方形虚框内的子矩阵可在缓存中重用

Fig. 4 Illustration of execution sequence in Ca-MPK with the power *s* equals to 5

图 4 次幂 *s*=5 时 Ca-MPK 执行顺序的示意图

形). 由图 4 可以看出, 除了第 2 次递归产生的分割子图, 大部分子图均可实现缓存重用. 需要指出的是, 图 4 中每个核心分配了 1 个子矩阵, 子矩阵数 *numPart* 可以是核心数的倍数, 按上述执行顺序, 仍然可以实现复用.

4 实验设置

本文实验硬件平台为一个 x86 处理器和一个 ARM 处理器. x86 处理器为 Intel Xeon Gold 6258R, 频率为 2.4 GHz, 包括 28 个核心, 每个核心具有私有的 32 KB 的 L1 数据缓存以及 1 024 KB 的 L2 缓存, L3 缓存大小为 39 MB, 主存大小为 128 GB. ARM 处理器为 Kunpeng 920, 频率为 2.6 GHz, 包括 32 个核心, 每个核心具有私有的 64 KB 的 L1 数据缓存以及 512 KB 的 L2 缓存, L3 缓存大小为 32 MB, 主存大小为 128 GB. x86 上编译器版本为 Intel oneAPI 2022.1[21], ARM 上编译器版本为 GCC8.4.1, 采用“-O3”选项双精度编译. 测试中将每个软件线程绑定到一个处理器物理核心(没有采用超线程), 针对每个矩阵均测试 100 次取平均的 GFLOPS.

测试的稀疏矩阵选自 SuiteSparse 矩阵集[22]. 选取了 302 个相对较大的矩阵(矩阵行数大于 100 000 且非零元数大于 200 000). 在上述矩阵中, 选取 12 个矩阵与现有方法进行详细对比分析, 如表 1 所示. 根据 3.2 节递归划分方法和 Ca-MPK 实现, 只需针对每个矩阵的依赖图执行 2 次划分, 因而引入的前处理开销较小, 通常小于 5 次求解器迭代开销.

Table 1 Detailed Performance Analysis of Twelve Represented Sparse Matrices

表 1 12 个代表性稀疏矩阵的详细性能分析

编号	矩阵	行数	非零元数	对称正定
1	ASIC 320k	321 821	1 931 828	否
2	ASIC 320ks	321 671	1 316 085	否
3	ASIC 680ks	682 712	1 693 767	否
4	Freescale1	3 428 755	17 052 626	否
5	FullChip	2 987 012	26 621 983	否
6	G2 circui	150 102	726 674	是
7	G3 circuit	1 585 478	7 660 826	是
8	memchip	2 707 524	13 343 948	否
9	nxp1	414 604	2 656 177	否
10	pre2	659 033	5 834 044	否
11	rajat30	643 994	6 175 244	否
12	transient	178 866	961 790	否

5 结果分析

本文对比以下版本的 MPK 实现: 1)MKL 表示直接调用 Intel oneMKL 的 SpMV 函数实现的 MPK, 没有开发缓存数据重用; 2)RACE 表示文献 [2] 中基于递归代数着色引擎(RACE)实现的 MPK, 开发了缓存数据重用, 是目前文献公开报道的最新研究成果; 3)Ca-MPK 表示本文实现的版本, 调用标准的基于 CSR 格式的 SpMV, 实现了本文提出的缓存数据重用方法; 4)CSR 表示调用基于 CSR 格式的 SpMV 实现的传统版本 MPK.

5.1 整体性能

图 5 给出了次幂 s=5, 10, 15 时 x86 处理器上不同 MPK 版本在所有 302 个稀疏矩阵上的整体性能. 可以看出, 在 3 个版本 MPK 中, Ca-MPK 性能最好. 次幂 s=5 时, Ca-MPK 相对于 MKL 和 RACE 的平均加速比分别为 1.56 和 1.19; s=10 时, Ca-MPK 相对于 MKL

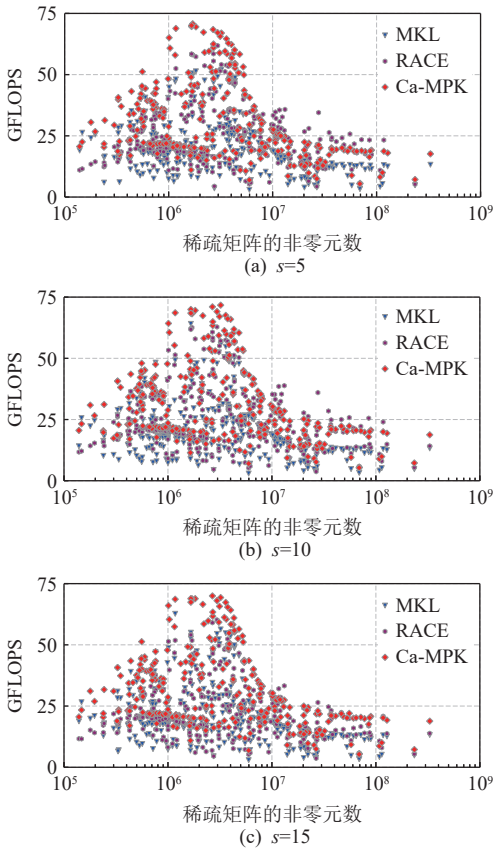


Fig. 5 Performance comparison of 302 sparse matrices on Intel 6258R CPU for different MPK implementations with different powers

图 5 不同 MPK 实现不同幂次时 302 个稀疏矩阵在 Intel 6258R CPU 上时的性能对比

和 RACE 的平均加速比分别为 1.57 和 1.26; s=15 时, Ca-MPK 相对于 MKL 和 RACE 的平均加速比分别为 1.55 和 1.40.

图 6 给出了次幂 s=5, 10, 15 时 ARM 处理器上不同 MPK 版本在所有 302 个稀疏矩阵上的整体性能. 同样在 3 个版本 MPK 中 Ca-MPK 性能最好. s=5 时, Ca-MPK 相对于 CSR 和 RACE 的平均加速比分别为 1.48 和 1.08; s=10 时, Ca-MPK 相对于 CSR 和 RACE 的平均加速比分别为 1.47 和 1.15; s=15 时, Ca-MPK 相对于 CSR 和 RACE 的平均加速比分别为 1.47 和 1.20.

上述结果可以看出, Ca-MPK 相对于 RACE 随着次幂 s 增加, 性能提升更加明显. 而 RACE 方法随着次幂 s 增加而性能下降, 与文献 [2] 中报道的结果基本相符. 其原因在于 RACE 的缓存重用距离为次幂 s, s 增加可能超过缓存容量, 因而无法实现重用. 相应地, Ca-MPK 总是在 2 次 SpMV 计算之间实现缓存数据重用, 不受次幂 s 大小影响.

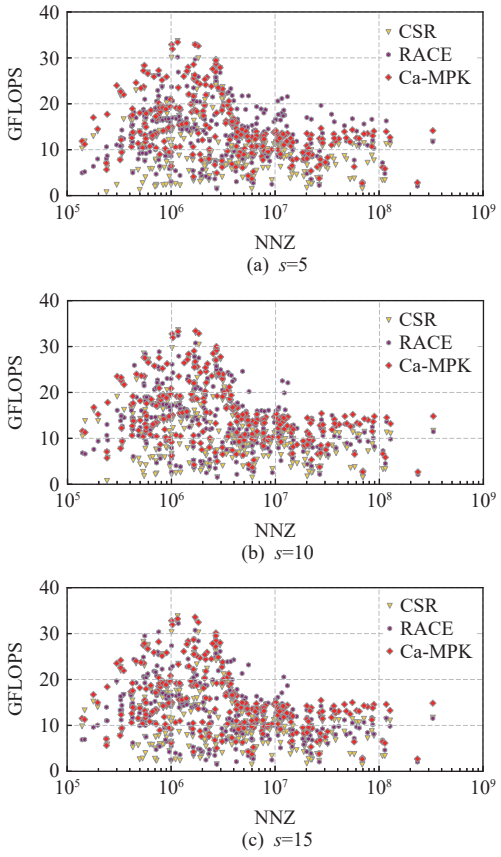


Fig. 6 Performance comparison of 302 sparse matrices on Kunpeng 920 CPU for different MPK implementations with different powers

图 6 不同 MPK 实现不同幂次时 302 个稀疏矩阵在 Kunpeng 920 CPU 上时的性能对比

5.2 典型矩阵性能

图 7 给出了 x86 平台上次幂 s=5, 15 时不同 MPK 版本在 12 个代表性稀疏矩阵上的具体性能. s=5 时, Ca-MPK 相对于 MKL 和 RACE 的平均性能加速比分别为 1.69(最大 2.53, 最小 1.20)和 1.40(最大 2.52, 最小 0.81); s=15 时, Ca-MPK 相对于 MKL 和 RACE 的平均性能加速比分别为 1.67(最大 2.68, 最小 1.10)和 2.26(最大 3.99, 最小 0.84). 图 7 中右侧 y 轴同时给出了 Ca-MPK 应用于 CG 和 BiCGStab 求解器时, 相对于采用 MKL 实现时的性能加速比. 可以看出, s=5 时, 基于 Ca-MPK 的实现, 提升 CG 和 BiCGStab 求解器性能平均达 1.37(最大 1.43, 最小 1.30)和 1.42(最大 1.81, 最小 1.14); s=15 时, 基于 Ca-MPK 的实现, 提升 CG 和 BiCGStab 求解器性能平均达 1.38(最大 1.50, 最小 1.27)和 1.40(最大 1.86, 最小 1.08), 充分说明了 Ca-MPK 在实际应用中的有效性.

图 8 给出了 ARM 平台上 12 个代表性稀疏矩阵上的具体性能. s=5 时, Ca-MPK 相对于 CSR 和 RACE 的平均加速比分别为 2.02(最大 7.63, 最小 0.94)和 1.25(最大 1.92, 最小 0.78); s=15 时, Ca-MPK 相对于 CSR 和 RACE 的平均加速比分别为 2.01(最大 7.28, 最小 0.96)和 1.72(最大 2.94, 最小 0.79). 右侧 y 轴给出了 CG 和 BiCGStab 求解器采用不同 MPK 实现时相对于基于 CSR 实现的加速比. 相对于采用 CSR 实现的求解器, s=5 时基于 Ca-MPK 的实现提升 CG 和 BiCGStab 求解器性能平均达 2.00(最大 2.83, 最小 1.18)和 1.36(最大 1.81, 最小 0.97); s=15 时, 性能平均达

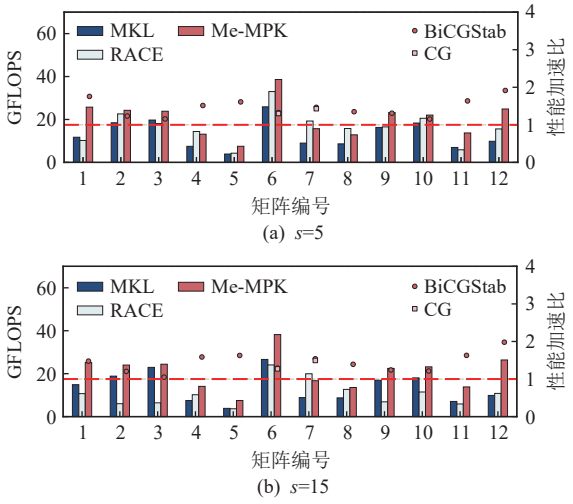


Fig. 7 Performance comparison of 12 matrices on Intel 6258R CPU for different MPK implementations with different powers

图 7 不同 MPK 实现不同幂次时 12 个矩阵在 Intel 6258R CPU 上时的性能对比

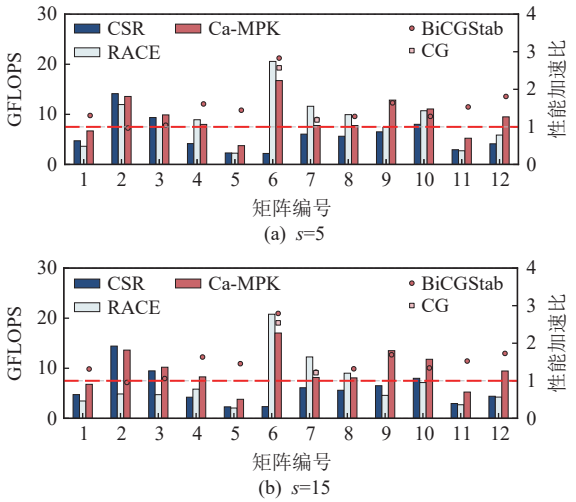


Fig. 8 Performance comparison of 12 matrices on Kunpeng 920 CPU for different MPK implementations with different powers

图 8 不同 MPK 实现不同幂次时 12 个矩阵在 Kunpeng 920 CPU 上时的性能对比

2.00(最大 2.79, 最小 1.21)和 1.37(最大 1.73, 最小 1.00).

图 9 给给出了针对 Freescale1 矩阵, 采用性能工具 LIKWID[23] 在 x86 上测得的主存和 L2/L3 缓存的数据访问量. 可以看出, 相对于 MKL, Ca-MPK 大幅减少了主存和 L3 的数据访问, 充分表明数据在 L2 中被成功复用.

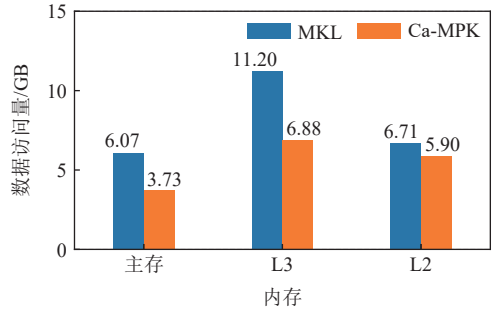


Fig. 9 Data traffic of memory and cache for matrix Freescale1 on x86 with s equals to 15

图 9 Freescale1 矩阵在 x86 上幂次为 15 时主存和缓存数据访问量

6 结 论

本文提出了基于缓存数据重用的 SpMV 序列(MPK)优化方法. 基于稀疏矩阵依赖图, 设计了体系结构感知的递归划分方法, 将依赖图划分为适合缓存大小的子图/子矩阵, 通过构建分割子图解耦数据依赖, 根据特定顺序在子矩阵上调度执行 SpMV, 实现缓存数据重用. 测试表明, 本文方法相对于 Intel OneMKL 库和基于 RACE 的 MPK 实现, 获得了较大的平均性能提升.

后续将在多个体系结构平台上(例如 GPU 平台)对本文方法进行测试分析和优化. 除此之外, 拟结合矩阵对称性质, 引入对称 SpMV 实现 MPK, 减少内存空间占用以提升 MPK 性能. 此外, 本文工作主要关注在多次 SpMV 调用之间重用矩阵. 理论上任何针对单次 SpMV 算子调用的优化, 都与本文工作是互补的, 可以直接与本文方法集成.

作者贡献声明: 徐传福提出问题、设计解决方案和实验方案, 并撰写论文; 邱昊中负责代码实现和优化, 进行测试分析和图表绘制; 车永刚对整体研究思路和论文撰写给出了建议. 徐传福和邱昊中对本文工作有相同贡献, 为共同通讯作者.

参 考 文 献

[1] Zhang Yichen, Li Shengguo, Yuan Fan, et al. Memory-aware optimization for sequences of sparse matrix-vector multiplications[C]//Proc of 2023 IEEE Int Parallel and Distributed Processing Symp (IPDPS). Piscataway, NJ: IEEE, 2023: 379−389

[2] Alappat C, Hager G, Schenk O, et al. Level-based blocking for sparse matrices: Sparse matrix-power-vector multiplication[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 34(2): 581−597

[3] Gurhem J, Vandromme M, Tsuji M, et al. Sequences of sparse matrix-vector multiplication on Fugaku's A64FX processors[C]//Proc of 2021 IEEE Int Conf on Cluster Computing (CLUSTER). Piscataway, NJ: IEEE, 2021: 751−758

[4] Saad Y. Numerical Methods for Large Eigenvalue Problems: Revised Edition[M]. Philadelphia, PA: SIAM, 2011

[5] 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

[6] Barrett R, Berry M, Chan T F, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods[M]. Philadelphia, PA: SIAM, 1994

[7] Vuduc R W. Automatic Performance Tuning of Sparse Matrix Kernels[M]. Berkeley, CA: University of California, 2003

[8] Buluç A, Fineman J T, Frigo M, et al. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks[C]//Proc of the 21st Annual Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2009: 233−244

[9] Demmel J, Hoemmen M, Mohiyuddin M, et al. Avoiding communication in computing Krylov subspaces[R]. Berkeley, CA: University of California, 2010

[10] Hoemmen M. Communication-Avoiding Krylov Subspace Methods[M]. Berkeley, CA: University of California, 2007

[11] Carson E C. Communication-Avoiding Krylov Subspace Methods in Theory and Practice[M]. Berkeley, CA: University of California, 2015

[12] Dongarra J, Tomov S, Luszczek P, et al. With extreme computing, the rules have changed[J]. Computing in Science & Engineering, 2017, 19(3): 52−62

[13] Mohiyuddin M, Hoemmen M, Demmel J, et al. Minimizing communication in sparse matrix solvers[C]//Proc of the Conf on High Performance Computing Networking, Storage and Analysis. New York: ACM, 2009: 1−12

[14] Morlan J, Kamil S, Fox A. Auto-tuning the matrix powers kernel with SEJITS[C]//Proc of 10th Int Conf High Performance Computing for Computational Science-VECPAR 2012. Berlin: Springer, 2013: 391−403

[15] Muranushi T, Makino J. Optimal temporal blocking for stencil computation[J]. Procedia Computer Science, 2015, 51(1): 1303−1312

[16] Huber D, Schreiber M, Schulz M. Graph-based multi-core higher-order time integration of linear autonomous partial differential equations[J]. Journal of Computational Science, 2021, 53(4): 101−139

[17] Alappat C, Basermann A, Bishop A R, et al. A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication[J]. ACM Transactions on Parallel Computing, 2020, 7(3): 1−37

[18] Qiu Haozhong, Xu Chuanfu, Fang Jianbin, et al. Towards scalable unstructured mesh computations on shared memory many-cores[C]//Proc of the 29th ACM SIGPLAN Annual Symp on Principles and Practice of Parallel Programming. New York: ACM, 2024: 109−119

[19] Naumov M. S-step and communication-avoiding iterative methods[R]. Santa Clara, CA: NVIDIA, 2016

[20] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs[J]. SIAM Journal on Scientific Computing, 1998, 20(1): 359−392

[21] Intel Corporation. Intel oneAPI math kernel library (onemkl)[EB/OL]. Santa Clara: Intel Corporation, 2024[2024-03-25]. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[22] Davis T A, Hu Y. The University of Florida sparse matrix collection[J]. ACM Transactions on Mathematical Software, 2011, 38(1): 1−25

[23] Treibig J, Hager G, Wellein G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments[C]//Proc of 2010 39th Int Conf on Parallel Processing Workshops. Piscataway, NJ: IEEE, 2010: 207−216

Xu Chuanfu, born in 1980. PhD, associate professor. His main research interest includes high-performance computing and applications.

徐传福, 1980 年生. 博士, 副研究员. 主要研究方向为高性能计算应用.

Qiu Haozhong, born in 1998. PhD candidate. His main research interest includes high-performance computing and applications.

邱昊中, 1998 年生. 博士研究生. 主要研究方向为高性能计算应用.

Che Yonggang, born in 1973. PhD, professor. His main research interest includes high-performance computing and applications.

车永刚, 1973 年生. 博士, 研究员. 主要研究方向为高性能计算应用.