

本科生毕业设计

|  |
| --- |
| 稀疏矩阵乘优化机制研究 |

|  |  |
| --- | --- |
| 院 系 | 计算机科学与技术 |
| 专业班级 | 计算机1907 |
| 姓 名 | 程潇鹏 |
| 学 号 | U201915139 |
| 指导教师 | 张宇 |

2023年06月01日

**学位论文原创性声明**

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名： 2023年06月01日

**学位论文版权使用授权书**

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保 密□，在 年解密后适用本授权书

2、不保密☑。

（请在以上相应方框内打“√”）

作者签名： 2023年06月01日

导师签名： 2023年06月01日

摘 要

稀疏矩阵乘法是许多科学计算和工程计算中常见的核心计算操作之一。由于稀疏矩阵的特殊性质，传统的稠密矩阵乘法算法在稀疏矩阵上的效率很低，因此需要专门的稀疏矩阵乘法算法来提高计算效率。稀疏矩阵乘法优化研究的主要挑战在于如何提高计算效率同时保持较低的存储空间占用。对稀疏矩阵存储格式的研究是稀疏矩阵乘优化领域的一项重点。

复现了基于压缩稀疏行-五元组CSR5(Compressed Sparse Row - Quintuple）存储结构和对角线DIA（Diagonal）存储结构的稀疏矩阵向量乘法SpMV（Sparse Matrix-Vector Multiplication）算法，并将两种算法的运行效率进行了对比，提出了根据对角线横纵坐标差的最大值与矩阵规模的比值（即对角度）作为矩阵特征，根据这一特征选择矩阵使用何种存储结构。并将采取了选择存储格式的算法与基于CSR5和DIA的单一存储格式SpMV算法进行了比较。

实验结果表明，采用单一存储结构时，在对角度很小时，基于DIA存储格式的SpMV算法效率很高，优于CSR5，而其它情况下远低于CSR5。引入存储结构预测后，在理想条件下，算法性能最高比CSR5提高了22.6%。

**关键词**：稀疏矩阵乘法；稀疏矩阵向量乘；压缩稀疏行-五元组；对角线存储；

Abstract

Sparse matrix multiplication is one of the common core computation operations in many scientific and engineering calculations. Due to the special properties of sparse matrices, traditional dense matrix multiplication algorithms are inefficient on sparse matrices, so special sparse matrix multiplication algorithms are needed to improve computational efficiency. The main challenge in optimizing sparse matrix multiplication is to improve computational efficiency while maintaining low storage space usage. Research on sparse matrix storage formats is a key focus in the field of sparse matrix multiplication optimization.

This study reproduced sparse matrix-vector multiplication (SpMV) algorithms based on the compressed sparse row-quintuple (CSR5) storage structure and the diagonal (DIA) storage structure, and compared the efficiency of the two algorithms. The study proposed using the ratio of the maximum difference between diagonal coordinates to matrix size (i.e., diagonal degree) as a matrix feature to select the storage structure. The algorithm that selects the storage format was compared with the single-storage-format SpMV algorithm based on CSR5 and DIA.

Experimental results showed that when using a single storage structure, the SpMV algorithm based on the DIA storage format was highly efficient for small diagonal degrees, better than CSR5, but much lower than CSR5 in other cases. After introducing storage structure prediction, under ideal conditions, the algorithm performance was improved by 22.6% compared to CSR5.

**Keywords:** SpMM;SpMV;CSR5;DIA

目 录

[摘 要 I](#_Toc9895)

[Abstract II](#_Toc2915)

[1 绪 论 1](#_Toc20811)

[1.1 课题背景 1](#_Toc938)

[1.2 国内外研究现状 3](#_Toc32502)

[1.3 研究目的和主要内容 7](#_Toc14793)

[1.4 论文结构 8](#_Toc5980)

[2 相关技术基础 9](#_Toc22516)

[2.1 背景知识概述 9](#_Toc16357)

[2.2 方案可行性分析 11](#_Toc10707)

[2.3 开发环境以及开发工具 12](#_Toc27132)

[2.4 关键技术分析 12](#_Toc17943)

[2.5 本章小结 12](#_Toc22704)

[3 基于CSR5和DIA的SpMV算法 13](#_Toc30231)

[3.1 SpMV算法 13](#_Toc121)

[3.2 基于CSR5的SpMV算法 15](#_Toc21244)

[3.3 基于DIA的SpMV算法 18](#_Toc31272)

[3.4 两种存储格式比较 20](#_Toc3447)

[3.5 对角线偏移量与DIA存储结构 24](#_Toc19012)

[3.6 选择存储格式 26](#_Toc15394)

[3.7 本章小结 27](#_Toc2148)

[4 实验与结果分析 28](#_Toc11773)

[4.1 对角度的取值测算 28](#_Toc3653)

[4.2 引入存储格式选择后算法的效率 29](#_Toc27038)

[4.3 本章小结 30](#_Toc21683)

[5 总结与展望 31](#_Toc1802)

[致 谢 32](#_Toc5234)

[参考文献 33](#_Toc16736)

# 绪 论

本章首先介绍了当前稀疏矩阵乘法研究的现状，然后分析了稀疏矩阵乘法优化在存储格式方向的技术产生以及发展现状，介绍了国内外稀疏矩阵存储优化相关的研究工作，并对本文研究内容的工作意义做出了说明。

## 课题背景

### 研究背景和趋势

稀疏矩阵乘法（Sparse Matrix Multiplication，简称SpMM）是许多科学计算和工程计算中常见的核心计算操作之一。由于稀疏矩阵的特殊性质，传统的稠密矩阵乘法算法在稀疏矩阵上的效率很低，因此需要专门的稀疏矩阵乘法算法来提高计算效率。

稀疏矩阵乘法优化研究的意义在于可以提高大规模稀疏矩阵计算的效率和可扩展性。随着科学计算和工程计算的不断发展，越来越多的应用需要处理大规模稀疏矩阵，如图像处理、自然语言处理、社交网络分析等。这些应用中的大规模稀疏矩阵计算需要高效的算法和系统来支持。稀疏矩阵运算效率受稀疏矩阵存储格式，稀疏矩阵向量乘算法等影响。2019年Torsten Hoefler发表的COSMA算法[1]，在绝大多数条件下可以接近并行矩阵乘法的通信复杂度的理论下界。因此对稀疏矩阵存储格式优化的研究逐渐受到重视。其中，单一存储格式在近些年有大量研究，混合存储格式的研究也开始出现。近几年的研究重点集中在混合存储格式上。稀疏矩阵乘法优化研究的主要挑战在于如何提高计算效率同时保持较低的存储空间占用。

### 稀疏矩阵存储优化的途径

稀疏矩阵存储格式的优化思路主要是压缩存储空间，除了忽略为零的元素外，还包括用指针和索引代替非零元素的位置等。基于这种思想产生的压缩存储格式包括COO、CSR、ELL、DIA、CSR5等单一存储格式。

### 面临的问题和挑战

单一存储格式的研究逐渐遇到瓶颈，对于不同的单一存储格式，在处理具有不同特征的稀疏矩阵时，存储效率各不相同。如DIA存储格式在处理对角矩阵时存储效率很高，但是在处理分布均匀的稀疏矩阵时效率大幅下降[2]，如图 1‑1和图 1‑2所示：

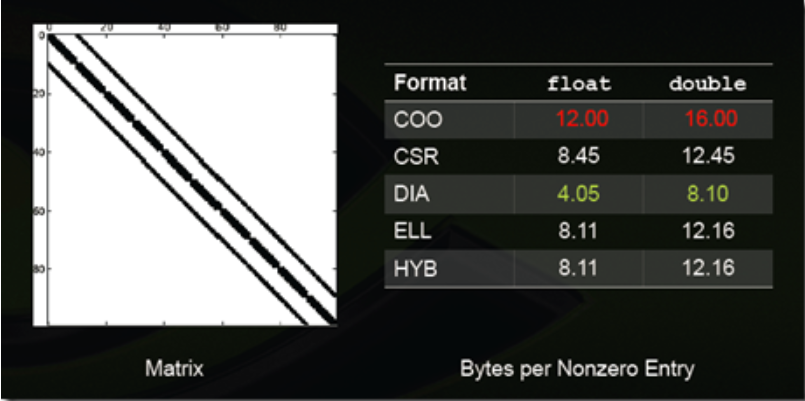


图 1‑1几种存储格式处理对角矩阵时的压缩效率

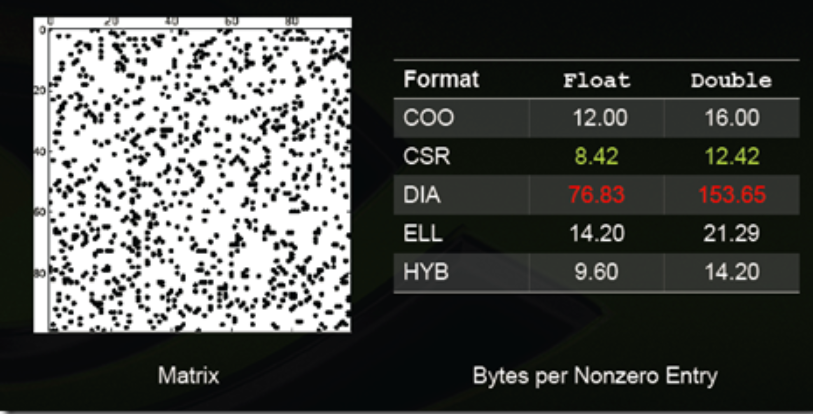


图 1‑2 几种存储格式处理随机矩阵时的压缩效率

压缩效率指每个非零元素平均占用的字节数，数值越小表示压缩效率越高。可见DIA存储格式在处理非对角矩阵时效率极低，而COO、CSR等格式非常稳定。

因此，如果采用单一存储格式，就不能在所有情况下都取得最优的压缩效率。因此，研究混合存储格式具有必要性。

## 国内外研究现状

### 单一存储格式

稀疏矩阵中非零元素个数远远小于矩阵元素总个数，因此可以采用压缩二维数组的格式来提高存储效率。

COO[4]：这是一种非常直观的存储格式，它的基本思想就是压缩存储。COO存储格式包含非零元素值和该元素值的行索引和列索引，并且非常容易由二维数组转换得到，然而这种格式非常不利于计算。COO存储格式常用于从文件中读取稀疏矩阵，martix market[3]采用的就是COO存储格式。如图 1‑3所示，其中COL表示列索引，ROW表示行索引，DATA存储非零元素。

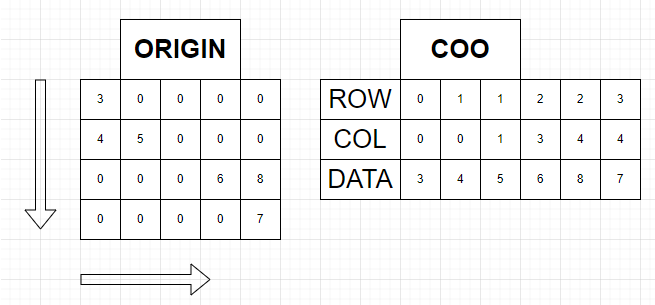


图 1‑3 COO存储格式

CSR：CSR存储格式是广泛流行的稀疏矩阵存储结构。这种格式要求元素按行存储，但是每一行元素可以乱序存储。CSR通过三个数组来存储稀疏矩阵，数组col\_ptr[i]存储data[i]的列索引，数组data存储非零元素，数组index\_ptr用于存储第i行的非零元素在数组data中的起始位置以及数组data中的结束位置，其中左边取闭区间，右边取开区间，长度为非零元素个数加一。CSR格式实际上是COO格式的压缩，它节省了存储空间，并且引入了row\_ptr以便于查找，方便计算。缺点是不直观，不易转换到稀疏结构。CSC存储格式是CSR存储格式的一种变种，它们的原理相同。

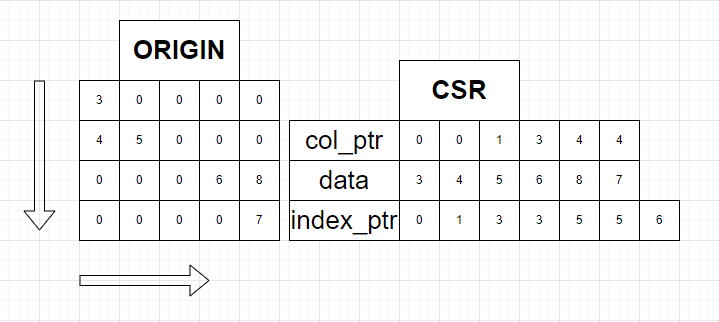


图 1‑4 CSR存储格式

ELL[5]：它需要用两个与原始矩阵行数相等的矩阵来存储，第一个矩阵存放列号，第二个矩阵存放非零元素，行号就是数据所在行。这种格式的缺点在于如果有一行过长，那么存储效率会大幅降低。因此ELL存储格式的压缩效率不稳定。

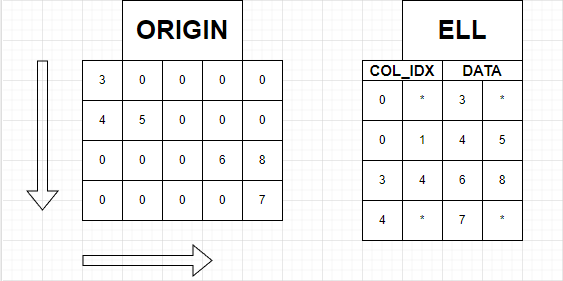


图 1‑5 ELL存储格式

DIA：即对角线存储法，设置一个新矩阵，用行表示行，用列表示对角线，忽略全为零的对角线，用\*补齐没有元素的位置。DIA存储格式在处理具有对角特性的矩阵时，具有很高的效率，但是处理随机矩阵时效率极低。因此DIA存储格式的压缩效率不稳定。

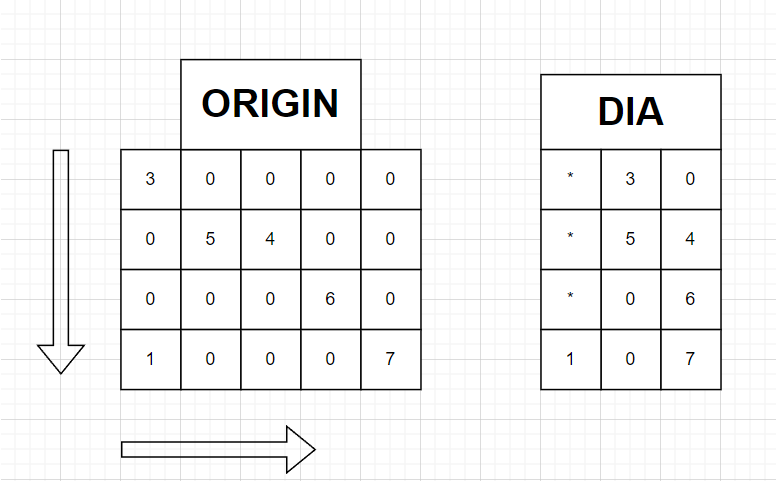


图 1-6 DIA存储格式

SELL[6]：SELL是为了解决ELL存储格式存在的问题的另一种存储格式，将ELL切割成每块有C行的矩阵快，再分别用ELL格式进行存储，从而减少零元素的扩充。下图 1-7是C取2时的SELL存储格式，前两行一组用ELL格式存储，长度为2，后两行长度为3.

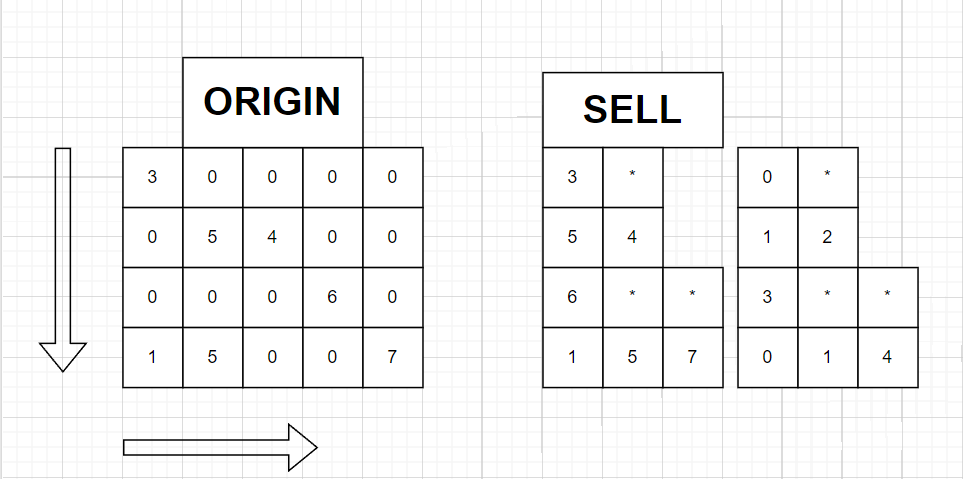


图 1-7 SELL存储格式

SELL-C-σ[7]：SELL-C-是SELL的变体，它在 个连续行内排序，通常选择为C的倍数。排序策略是按行长度由大到小排序，通过排序可以将长度相近的行放在一起，这样能进一步减少零元素填充占据的空间。SELL-C-σ存储格式在宽SIMD下拥有较好的性能，但是它同样有ELL类存储格式的通病：对于某一行非常长的特殊矩阵，它的存储效率并不好。

CSRL[10]：在CSR基础上，对稀疏矩阵中列下标连续的非零元素，存储首个非零元素的列下表和段长度。CSR存储格式在SpMV算法中表现不佳，是由于数据的空间局部性和时间局部性不好，造成大量访存导致的。然而，实际应用中的稀疏矩阵在一定程度上具有局部性特征，即下标相邻。CSRL存储格式可以有效利用这些局部性特征来提高其在SpMV算法中的表现。

CSR5[8]：将所有非零元素都划分成相同大小的2D块(除了最后一块)，记行数为，列数为。CSR5引入了块指针和块描述符作为辅助，并对CSR存储格式中的col\_idx数组和val数组都进行转置，当每个处理器处理一个2D块时，处理器的每个SIMD(单指令多数据流)进行合并内存访问。

块指针存储每个2D块中第一个元素在原始矩阵中的行索引。

CSR5的块描述符包含以下四个部分：

1. bit\_flag：该矩阵与col\_idx矩阵大小一致，指示对应位置的元素是否是一行的第一个非零元素。
2. y\_offset：长度与col\_idx矩阵列数一致，记录每一列的行索引偏移。第一列的行偏移固定为0，而后续偏移可以通过计算bit\_flag矩阵中该列之前所有列的T数量来获得。
3. seg\_offset：作为生成分段和的辅助阵列，可以在对应bit\_flag矩阵中向右搜索，计算全部为F的连续列的数量来获得。
4. empty\_offset：考虑这种情况，矩阵中有一行空行，那么在计算y\_offset时，行偏移就会发生错误。因此引入了empty\_offset，记录正确的偏移量。

CSR5是目前该领域效率最高的存储结构之一，并且对于任何稀疏矩阵结构都有很好的效果。

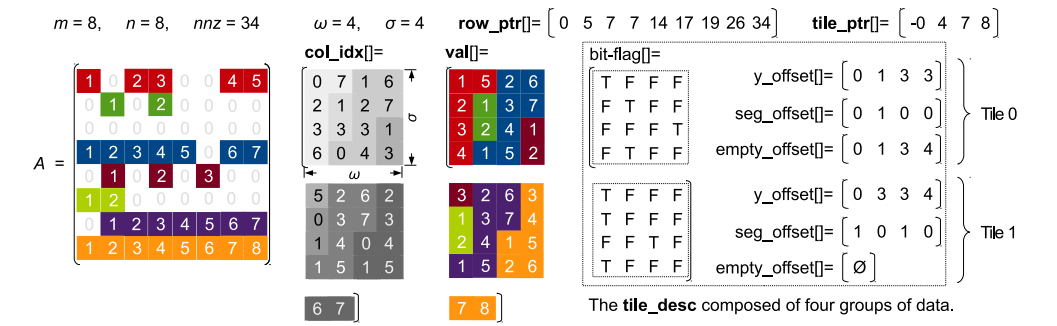


图 1-8 CSR5存储结构

### 混合存储格式

HYB[5]：为了解决ELL存储格式在某一行过长造成的臃肿问题，引入了COO+ ELL存储格式，即HYB存储格式，当ELL某一行过长时，用COO存储格式单独存储，提高存储效率。

HYB5[11]：HYB5是基于SELL-C-σ 格式和 CSR5 格式相结合设计出的一种混合稀疏矩阵存储格式。HYB5 按列将矩阵切割为两个部分，较规则的一部分由SELL-C-σ 来存储，相对不规则的一部分存储为 CSR5 格式。[11]切割矩阵的位置由性能参数K决定。然而，论文作者并未给出K的确定方法，无法预先确定最优切割点的K值。

## 研究目的和主要内容

### 课题研究的意义

围绕稀疏矩阵存储格式开展的性能优化工作有助于提高SpMV(稀疏矩阵乘向量)的运算效率，在高性能计算领域有着重要的理论意义，在工业领域、生物信息技术领域、计算化学领域以及游戏领域有实用价值。

### 课题研究的内容

1. 对常用单一存储结构以及混合存储结构的综述；
2. 从CSR5存储结构出发，结合DIA等存储格式，设计一种新的混合存储结构；
3. 在CSR5源码基础上进行修改，编写混合存储结构代码；
4. 将混合存储格式与CSR5存储格式在不同测试集下的表现进行比较。

### 课题研究的目标

在本次课题的研究中，首先我们调研了多种存储格式，随后从CSR5存储结构出发，设计一种新的混合存储结构，编写代码并进行测试，最终目的是实现课题要求，预期新的存储结构在面对多种矩阵时综合存储效率高于CSR5。

## 论文结构

本文的主要内容如下：

第一章我们首先介绍了稀疏矩阵存储结构的研究背景和趋势，然后分析了稀疏矩阵存储结构当前存在的问题，介绍了国内外在稀疏矩阵存储机构优化方向的相关研究工作，并对本文的主要研究内容及工作意义作了具体说明。

第二章着重介绍了混合存储格式HYB和HYB5，分析了它们如何通过混合结构取得性能提升、以及它们的优缺点。并分析如何改进这些缺点，给我们在混合存储结构的设计上能够带来什么启发。并提出了新混合存储结构的设计思路以及可能存在的问题，并对问题的解决提供了初步方法。

第三章着重研究了SpMV算法以及它在CSR5和DIA存储格式上的具体实现，并给出了伪代码。对基于两种存储格式的SpMV算法进行了性能测试，发现在面对元素集中在对角线附近的矩阵时，DIA表现更好，反之CSR5表现更好。通过进一步研究，设计了存储格式预测算法。

第四章通过测试得出了预测算法参数，并将引入存储格式预测的算法与单一存储格式算法进行比较，实验结果表面，引入存储格式预测的算法能够取得两种单一存储格式算法的最优解。

第五章总结了所做的工作，并指出了研究过程中的问题，对进一步研究提出了方向。

# 相关技术基础

在绪论中，稀疏矩阵混合存储结构的重要性已经得到了论证，然而，什么样的混合存储结构能够提升存储效率，乃至提升SpMV性能，这是需要讨论的问题。在这一章节，混合存储格式HYB和HYB5将得到详细的分析。同时，方案的可行性将得到论证。改进的混合存储结构将会面临两个问题，分别是采用何种存储结构的判断问题和效率能否提升的问题，对于前者，已经提出了解决方案，对于后者需要在实验中验证。

## 背景知识概述

稀疏矩阵混合存储结构是一种将不同存储格式组合使用的方法，用于存储稀疏矩阵。由于不同的存储格式适用于不同的稀疏矩阵特征，混合存储结构可以充分利用不同格式的优点，提高存储和计算的效率。

常见的混合存储格式有HYB和HYB5。

HYB存储格式的引入是为了解决ELL存储格式在某一行过长造成的臃肿问题，HYB格式将矩阵划分为两个部分：密集部分和稀疏部分。它的具体思路是当ELL某一行过长时，用COO存储格式单独存储，从而提高存储效率。如下图图 2-1 HYB存储格式，原始矩阵8个元素占用了20个单位的存储空间，然而，采用了ELL存储格式后，占用了32个单位的存储空间，存储效率反而下降，这是因为第二行过长。而采用HYB存储格式后，只占用了15个单位的存储空间，压缩效率得到有效提升。

HYB5存储格式将CSR5和SELL-C-σ存储格式结合起来[11]，它将矩阵分为两个部分，较规则的一部分采用SELL-C-σ存储格式，其他部分用CSR5存储格式。HYB5引入了参数K，将矩阵划分为左右两部分，左侧采用SELL-C-σ，右侧采用CSR5存储结构。一般来说，SELL-C-σ处理的部分非零元更少，在这种情况下它的处理效率高于CSR5存储格式。而CSR5存储格式处理空行较多的稀疏矩阵时，可以采取压缩空行的方法提升存储效率。

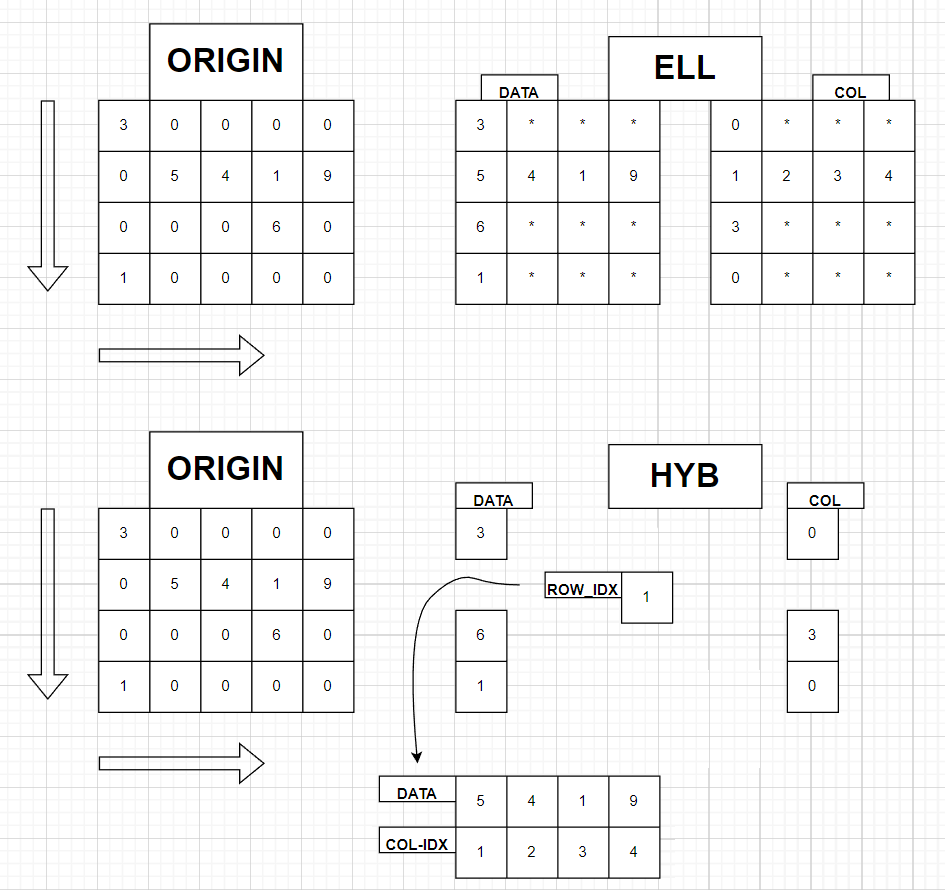


图 2-1 HYB存储格式

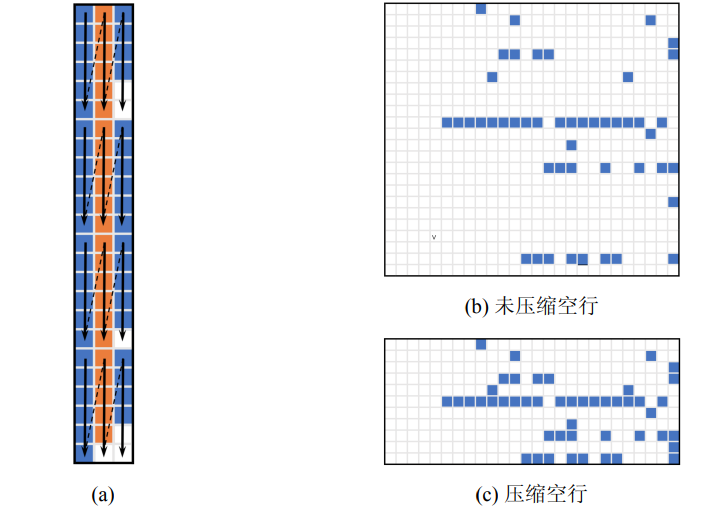


图 2-2 HYB5存储格式

## 方案可行性分析

没有一种存储格式可以在面对所有稀疏矩阵时都取得最优存储效率[15]，因此，采用混合存储格式是提升稀疏矩阵存储效率的突破点。在前人研究中，陈世钊等人提出的HYB5[11]存储结构能在选取最优的 K 值时，取得比CSR5存储结构更优的存储效率。这证明混合存储结构技术上是可行的。

Putnam模型估算成本如下：

CK值取3000，源码行数L预估500行，开发时间td取0.25年，按以下公式估算工作量：

(1)

计算得出的工作量约0.296年/人。

本文属于高性能计算领域，不存在伦理、社会等问题，实验所用工具均为开源或正版工具，未侵犯他人知识产权，不存在学术诚信问题。

## 开发环境以及开发工具

开发环境：ubuntu16.0.4 操作系统

开发工具：G++编译器，CMake构建工具

## 关键技术分析

改进的混合存储结构主要采用如下思想：由于DIA存储格式在处理具有类对角性质的稀疏矩阵时能取得比CSR5存储格式更高的存储效率，而CSR5存储格式在面对不同稀疏矩阵时表现十分稳定，因此，可以设计一种判断机制，当稀疏矩阵具有类对角性质时，采用DIA存储格式，否则采用CSR5存储格式。

混合存储结构的设计主要要考虑以下两个问题：

1. 什么时候采用CSR5存储结构，什么时候采用DIA存储结构。
2. 优化节省的开销能够抵消分支判断增加的开销。

对于第一个问题，我们采用如下解决方案：引入一个参数p，对于稀疏矩阵，当所有非零元集中在以主对角线为中心的个对角线上时，认为该稀疏矩阵具有类对角性质。当一个稀疏矩阵满足类对角性质时，采用DIA存储格式。

对于第二个问题，我们将在后续实验中验证。

## 本章小结

本章主要分析了HYB和HYB5两种混合存储结构的原理，并由此得出了新的混合存储结构的设计方案。论证了设计方案的可行性，并分析了如何解决新混合存储结构方案潜在的两个问题。

# 基于CSR5和DIA的SpMV算法

SpMV算法，即稀疏矩阵的向量乘法。研究高效的稀疏矩阵存储格式目的之一，就是提升SpMV算法的效率。本章介绍了SpMV算法，通过基于CSR的SpMV算法对其进行了分析，并给出分别了基于CSR5存储格式和DIA存储格式的SpMV算法，并给出了在混合存储结构中使用SpMV算法的策略。

## SpMV算法

SpMV算法指稀疏矩阵与一个向量相乘，对于不同的稀疏矩阵，由于稀疏特性差异较大，SpMV算法在实现细节上也各有不同。SpMV算法可以代替稀疏矩阵和稠密矩阵的运算，从而节省开销和内存。

由于本文用于实验的稀疏矩阵格式为Matrix Market，存储格式为COO，在进行SpMV算法算法前，需要先由COO存储格式转换为CSR5存储格式或DIA存储格式，然后再参与SpMV算法，因此，格式转换步骤的开销也应该计入SpMV算法开销。

通过使用广泛的CSR存储格式来简单分析SpMV算法。

下面是基于CSR格式的SpMV算法伪代码：

|  |
| --- |
| **Algorithm 1 基于 CSR 的 SpMV算法** |
| **INPUT** mtx格式矩阵A，向量  **OUTPUT** 向量y  1: **procedure** SpMV(A,x,y)  2: COO格式←读取mtx文件  3: CSR格式 A←COO格式  4: data\_m m ← A.m;  5: data\_r \*row\_idx ← A.row\_idx;  6: data\_c \*col\_idx ← A.col\_idx;  7: data\_v \*value ← A.value;  8: **for** i←1 to m **do**  9: data\_s sum ← 0;  10: **for** p←row\_idx[i]...row\_idx[i+1] **do**  11: data\_k k ← col\_idx[p];  12: sum += value[p]\*x[k];  13: y[i] ← sum;  14: return y;  15:**end procedure** |

在上述代码中，矩阵的数据结构可以是类或者结构体，它有五个成员，分别是行数m、列数n、存储数据的数组value，存储非零元素对应列数的数组col\_idx，以及可以指明一行在value数组中起始和终止位置的数组row\_idx。在执行SpMV算法时，本质上是执行，因此只需要知道行数即可，列数隐含在数组col\_idx的长度中。在8-13行的双层循环中，执行的就是SpMV算法的核心部分，索引i表示行数，矩阵A的每一行通过数组row\_idx寻找自己在数组value中的起始和终止位置，向量x通过数组col\_idx确定要与数组value中非零元相乘的元素，随后计算乘积之和。算法结果存储在向量y中。

上述算法还涉及了从COO存储格式到CSR存储格式的转换。转换过程的伪代码如下：

|  |
| --- |
| **Algorithm 2 COO到CSR格式转换** |
| **INPUT** mtx(COO)格式矩阵A  **OUTPUT** CSR格式矩阵B  1:**procedure** TransCOOToCSR(A)  2: **for** i←1 to A.value.length **do**  3: B.value[i] ← A.value[i];  4: **for** i←1 to A.col\_ptr.length **do**  5: B.col\_idx[i] ← A.col\_idx[i];  6: **for** i←1 to A.m+1 **do**  7: B.row\_idx[A.row\_idx[i]]++;  8: **return** B;  9:**end** **procedure** |

COO存储结构的数据结构较为简单，包含行数、列数、非零元数组value、非零元对应的行号数组row\_idx和列号数组row\_idx。从COO存储格式转换到CSR存储格式，value数组和col\_idx数组保持不变，只将COO存储格式中的行在矩阵中的索引转换为在value数组中的索引即可。

*struct COOMatrix {*

*int n; // 矩阵行数*

*int m; // 矩阵列数*

*int nnz; // 非零元素个数*

*vector<int> row\_idx; // 行索引*

*vector<int> col\_idx; // 列索引*

*vector<double> values; // 非零元素值*

*};*

基于CSR存储结构的SpMV算法存在以下问题：对数组value和数组row\_idx显然是连续访问的，但是这两个数组地址不一定对齐，因此无法进行合并访问提高访存效率，同时对数组x的访问是随机的，因此cache命中率不会太高。在不进行其它优化的情况下，上述算法时间复杂度为。

## 基于CSR5的SpMV算法

在讨论基于CSR5的SpMV算法之前，需要先进行CSR存储格式到CSR5存储格式的转换，这一转换过程比COO到CSR的转换更为复杂，首先要确定两个参数和，然后再进行转换。

在进行转换时，CSR5存储格式会将稀疏矩阵中的非零元划分为长为，宽为的矩阵块，而这两个参数的确定取决于硬件。通常取决于处理器的SIMD执行单元大小，例如对于double类型的数据(64位)，在拥有256位SIMD单元的CPU上，取,即取4。CSR5希望让一个SIMD处理一个分块，通过并行处理提高SpMV算法效率。而参数的选取就较为复杂，可以总结为下列公式：

(1)

上述公式中nnz意为一行中非零元数量，row即为一行长度，r、s、t和u

是与硬件相关的参数，它们是一些边界值，用于防止过大或者过小[8]。CSR5的数据结构第一章已经介绍过，这里不再赘述。

CSR转换为CSR5存储格式的算法主要分为三部分，分别是生成每个块的首行指针tile\_ptr，转置排列成块的col\_idx和value以及生成每个分块的信息，包括bit\_flag、y\_offset、seg\_offset和empty\_offset。

生成tile\_ptr的思路比较简单，因为此时和的大小已经确定，第i个矩阵块的首行索引在数组row\_idx中的索引即为(i从0开始计数)，需要注意的一点是如果一个矩阵块中含有空行，需要将这个行索引取负值以便于后续计算。数组tile\_ptr元素类型应使用无符号数，防止第一个分块存在空行时无法取负值。

生成bit\_flag只需遍历一遍value矩阵块，判断元素是否是一行的第一个非零元即可，是则为TRUE，否则为FALSE。y\_offset记录value矩阵块每一列中相对该矩阵块首元素所在行的相对行数，只需要遍历前一列对应的bit\_flag中TRUE的数量即可。生成seg\_offset只需要计算该列右侧全FALSE的连续列的数量即可。empty\_offset只在矩阵块索引为负值时生成，它纠正错误的y\_offset，纠正方法是计算出bit\_flag中为TRUE的对于元素在原始矩阵中的索引，计算公式为：

(2)

通过该索引找出在数组row\_idx它在value数组中的索引即可。需要注意的是，对于一般的矩阵，在分块后总会剩余一些非零元素，对于这些元素，不做分块处理，以CSR格式保留。

*class CSR5 {*

*public:*

*// 非零元素的值*

*std::vector<double> values;*

*// 非零元素的列索引*

*std::vector<int> columns;*

*// 每一行在values和columns数组中的起始索引*

*std::vector<int> row\_offsets;*

*// CSR5的压缩等级*

*int r;*

*// CSR5的块大小*

*int c;*

*// 构造函数*

*CSR5(const std::vector<double>& values,*

*const std::vector<int>& columns,*

*const std::vector<int>& row\_offsets,*

*int r,*

*int c)*

*: values(values),*

*columns(columns),*

*row\_offsets(row\_offsets),*

*r(r),*

*c(c)*

*{}*

*};*

完成格式转换后，即可开始基于CSR5的SpMV算法。

|  |
| --- |
| **Algorithm 3 基于CSR5的某一矩阵块的SpMV算法** |
| **INPUT** CSR5格式矩阵块A，向量x  **OUTPUT** 向量y  1:**procedure** SpMV\_CSR5(A,x)  2: **for** i ← 0 to -1 **do**  3: sum ← 0;  4: **for** j ← 0 to -1 **do**  5: ptr ← ;  6: sum += value[ptr] \* x[col\_idx[ptr]];  7: **if** bit\_flag[i][j] && !bit\_flag[0][j] && j>0 **then**  8: tmp[i-1] ← sum;  9: sum ← 0;  10: **else if** bit\_flag[i][j] && 存在k<i使bit\_flag[k][j]==TRUE **then**  11: **if** empty\_offset == nullptr **then**  12: y[tile\_ptr[tid]+y\_offset[i]] ← sum;  13: y\_offset[i]++;  14: **else then**  15: y[tile\_ptr[tid]+empty\_offset[i]] ← sum;  16: sum ← 0;  17: last\_tmp[i] ← sum;  18: **for** i ← 0-1 **do**  19: last\_tmp[i] += tmp[i];  20: y[tile\_ptr[tid]+y\_offset[i]] ← last\_tmp[i];  21: return y;  22:**end** **procedure** |

对于一个矩阵块的SpMV算法，最重要的部分在于求分段和。分段和分为三种情况，原作者提出将矩阵块的不同段着不同颜色：

1. 在矩阵块bit\_flag的一列中（非第一列），如果存在TRUE，那么第一个TRUE之前的着红色；如果不存在TRUE，那么整个列着红色。
2. 在矩阵块bit\_flag的一列中（非最后一列），如果存在TRUE，那么最后一个TRUE之后的段着蓝色。
3. 其他情况着绿色。

在计算分段和时，绿色段的分段和就是最终分段和，连续的蓝色和红色段分段和要相加。

在进行基于CSR5的SpMV算法时，可以为每个矩阵块的SpMV算法分配一个线程，通过并行大大提高算法效率。

## 基于DIA的SpMV算法

在进行基于DIA的SpMV算法之前，同样要考虑格式转换问题。为了便于表述，规定横坐标减去纵坐标为对角线偏移量，后续会大量使用这个概念。

从COO存储格式到DIA存储格式的转换可以归纳为三个步骤，首先是计算每个非零元素的对角线偏移量，并存储在数组中，同时统计每个对角线偏移量出现的次数；第二步是根据对角线偏移量的数量确定需要多少个数组来储存对角线；第三步是根据每个对角线偏移量出现的次数来确定每个数组的长度。

DIA存储结构的数据结构包含一个对角线数量num\_diags，存储对角线偏移量的数组diag\_offset，以及存储每个对角线上元素的二维数组diag，通过上述三个信息可以通过DIA还原一个矩阵。

*struct DIAMatrix {*

*int n; // 矩阵行数*

*int m; // 矩阵列数*

*int nd; // 对角线个数*

*int ndnz; // 对角线上非零元素个数*

*vector<int> offsets; // 对角线偏移量*

*vector<double> values; // 非零元素值*

*};*

格式转换的伪代码如下：

|  |
| --- |
| **Algorithm 4 COO到DIA格式转换** |
| **INPUT** mtx(COO)格式矩阵A  **OUTPUT** DIA格式矩阵B  1:**procedure** TransCOOToDIA(A)  2: **for** i ← 0 to A.value.length-1 **do**  3: row ← A.row\_idx[i];  4: col ← A.col\_idx[i];  5: offset ← col - row;  6: **if** /\*not find offset in diag\_offset\*/ **then**  7: B.num\_diags++;  8: B.diag\_offset.push\_back(offset);  9: num\_elems = array[B.num\_diags];  10: memset(num\_elems,0);  11: **for** i ← 0 to A.value.length-1 **do**  12: row ← A.row\_idx[i];  13: col ← A.col\_idx[i];  14: val ← A.value[i];  15: offset ← col - row;  16: diag\_index ← find(B.diag\_offset, B.num\_diags, offset);  17: num\_elems[diag\_index]++  18: **for** i ← 1 to B.num\_diags **do**  19: B.diag[i] = resize(B.diag[i], num\_elems[i])  20: **for** i ← 0 to A.value.length-1 **do**  21: row ← A.row\_idx[i];  22: col ← A.col\_idx[i];  23: val ← A.value[i];  24: offset ← abs(row-col);  25: diag\_index ← find(B.diag\_offset, B.num\_diags, offset);  26: B.diag[diag\_index][row - B.diag\_offset[diag\_index]] ← val;  27: return B;  28:**end** **procedure** |

代码的2-8行遍历COO格式矩阵中的每个非零元素，统计每个非零行的对角线偏移量，并将其存储到DIA格式矩阵的对角线偏移量数组中。9-19行分配DIA格式需要的存储空间，统计了对角线数量和对角线数组中每个对角线所需存储的元素数量。20-26行再次遍历COO格式矩阵中的每个非零元素，获取其所在的对角线偏移量和对应的对角线下标，将该元素存储到对应的对角线数组中。可以注意到11-17和20-26两段代码有所重复，第一段是分配空间，第二段才是数组赋值，如果合并会产生数组越界问题。

基于DIA的SpMV算法较为简单，伪代码如下：

|  |
| --- |
| **Algorithm 5 基于DIA的SpMV算法** |
| **INPUT** DIA格式矩阵块A，向量x  **OUTPUT** 向量y  1:**procedure** SpMV\_DIA(A,x)  2: **for** i ← 0 to A.m - 1 **do**  3: **for** j ← 0 to A.num\_diags **do**  4: k ← i + A.diag\_offset[j];  5: **if** k > 0 && k < A.m **then**  6: y[i] += A.diag[j][k] \* x[k];  7: return y;  8:**end** **procedure** |

已知对角线下标和偏移量可以快速得到该非零元在diag矩阵中的下标，注意该坐标合法时才能访问diag数组，否则有地址越界风险。

基于DIA的SPMV算法的时间复杂度为O(num\_diags \* n)，其中num\_diags是对角线的数量，n是矩阵的列数。该算法的空间复杂度也比较低，因为只需要存储对角线上的元素和偏移量，而不需要存储矩阵中的所有元素。因此，DIA格式非常适合存储具有对角线结构的稀疏矩阵，并且可以有效地加速SPMV运算。

## 两种存储格式比较

随机选取了七个稀疏矩阵测试集，在Intel CPU平台上进行了测试，分别执行了基于CSR5存储格式的SpMV算法和基于DIA存储格式的SpMV算法。测试集全部来自于佛罗里达大学稀疏矩阵集合。[2]

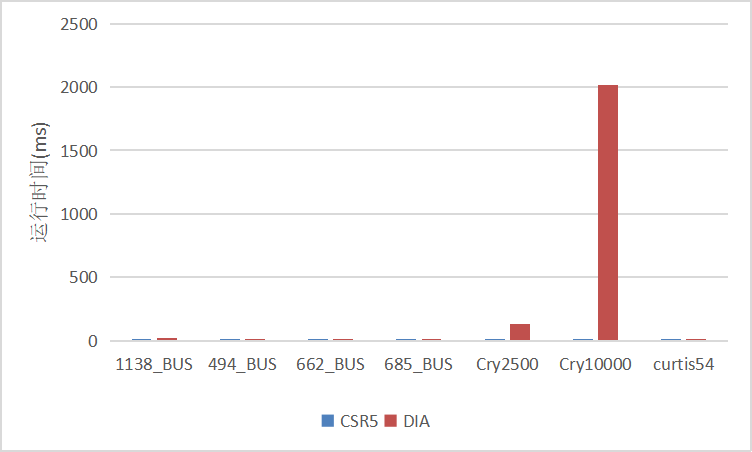
测试集名称和参数如下表 3-1：

表 3-1 测试集参数

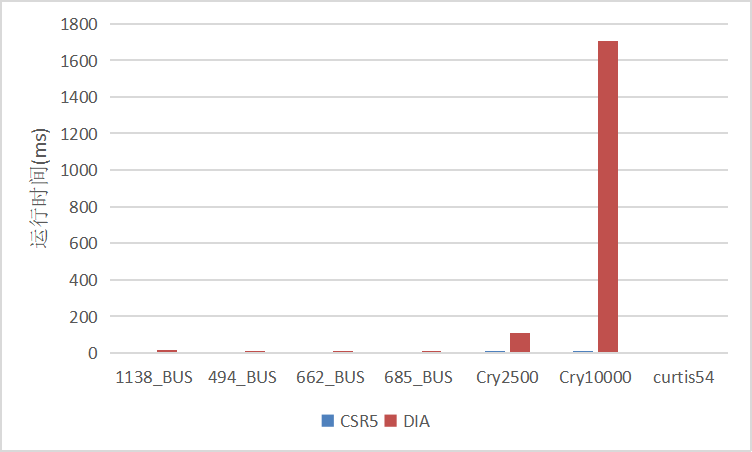
|  |  |  |
| --- | --- | --- |
| 矩阵名称 | 规模 | 非零元个数 |
| 1138\_BUS | 1138 \* 1138 | 4054 |
| 494\_BUS | 494 \* 494 | 1666 |
| 662\_BUS | 662 \* 662 | 2474 |
| 685\_BUS | 685 \* 685 | 3249 |
| Cry10000 | 10000 \* 10000 | 49699 |
| Cry2500 | 2500 \* 2500 | 4899 |
| curtis54 | 54 \* 54 | 291 |

矩阵的共性是对角线附近集中了大量元素，用这些测试集可以很好地鉴定优化效果。测试包括两个部分，分别是格式转换消耗的时间，和执行SpMV算法消耗的时间，参与运算的向量全部通过stdlib.h头文件下的rand()函数取随机值，运算时间取执行100次程序的平均值。CSR5的参数全部取4，全部取16。原始数据经过处理后，形成了如下图表：**图 3-1**，**图 3-2**，**图 3-3**。

从**图 3-1**可以看出，对于1138\_BUS,494\_BUS,662\_BUS，685\_BUS和curtis54这五个矩阵，COO转换为CSR5的开销略高于COO转换为DIA的开销，而对于cry2500矩阵，COO转化为DIA的开销反超了COO转换为CSR5的开销，而在cry10000矩阵中，COO转换为DIA的开销猛增，超过了2000ms。同时，转换为CSR5的过程在面对不同规模的矩阵时，表现相对平稳，而转换为DIA存储格式这一过程在面对不同矩阵时，运行时间波动相当大。



**图 3-1 COO转换为CSR5和DIA运行时间比较**



**图 3-2 分别基于CSR5和DIA的SpMV算法运行时间**

由于cry10000和cry2500数据过于极端，影响图表直观性，其它五个矩阵的运行时间图如下：

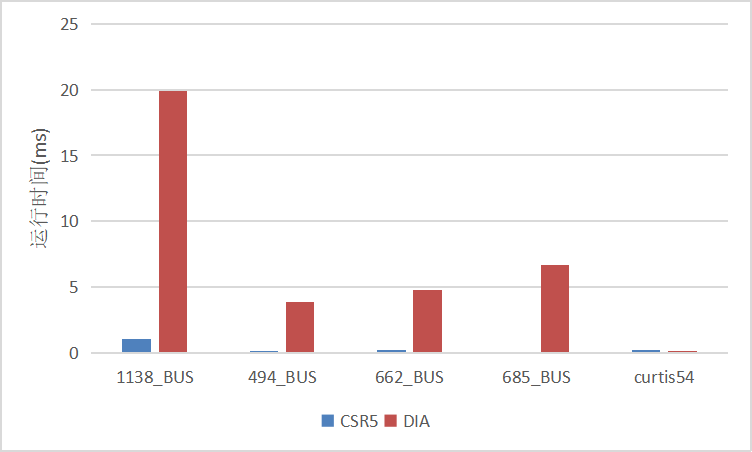


图 3-3 格式转换时间对比

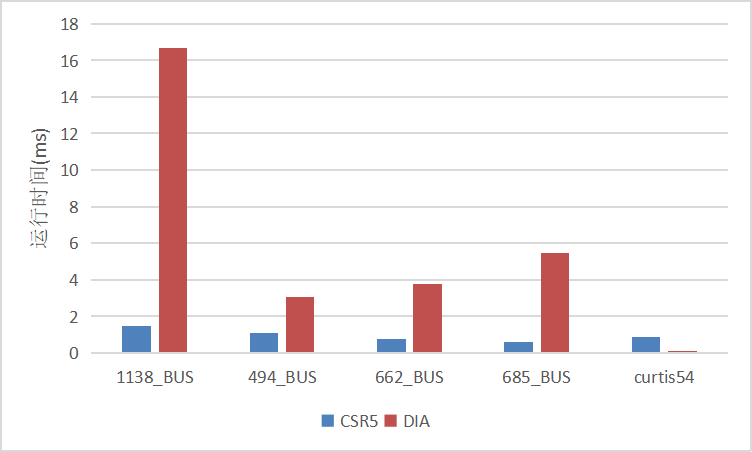


图 3-4 SPMV算法时间对比

从**图 3-2**可以看出，基于CSR5存储格式的SpMV算法运行时间全面少于基于DIA存储格式的SpMV算法。对于1138\_BUS, 494\_BUS, 662\_BUS， 685\_BUS和curtis54这五个矩阵，两者表现相差并不大，差距都在20ms以内，尤其对于curtis54矩阵，前者耗时0.879ms，后者0.042ms，基于DIA的SpMV算法实现了反超。同时，基于CSR5存储格式的SpMV算法面对不同矩阵时表现一如既往的稳定，而基于DIA存储格式的SpMV算法波动极大。

如图 3-3，图 3-4由于DIA存储格式在处理cry10000矩阵时开销过大，去掉了这个矩阵的数据，使得柱状图更加直观。整体来说，CSR5在SpMV算法中表现稳定且高效，然而，对于某些矩阵，DIA存储格式能够实现性能上的反超。

下面，对上述用到的矩阵结构进行分析。

下列七个矩阵大多数元素都集中在对角线附近，并有少量元素在其他位置分布，其中除了cry2500和cry10000，其它矩阵元素分布都较为随机。对于cry2500和cry10000，在这两个矩阵上DIA存储结构的表现远差于CSR5存储结构，咋一看非常反直觉，因为它们的元素几乎都集中在对角线附近，而这种结构下往往DIA存储格式表现更佳。实际上，这两个矩阵都有少量元素分布在左下角和右上角，导致对角线偏移量极大，使得COO格式向DIA格式转移时，开辟的数组极大，以至于接近数组原规模，因此算法用时极长。

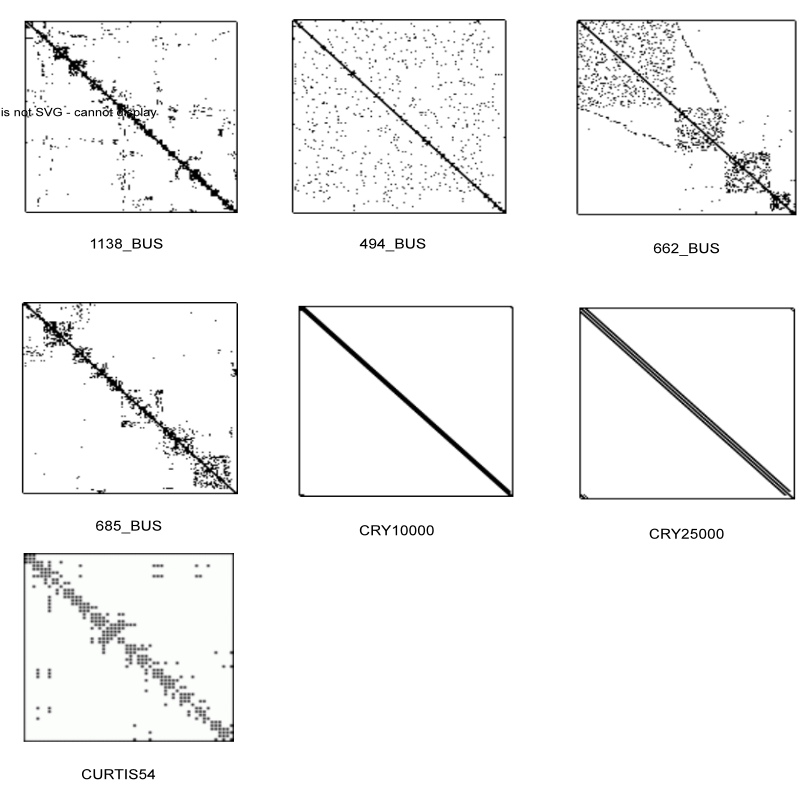


图 3-5矩阵结构图

只有在处理662\_BUS和curtis54这两个矩阵时，基于DIA存储格式的SpMV算法能够接近甚至超越基于CSR5存储格式的SpMV算法。观察矩阵形状，可以看出这两个矩阵的对角线偏移量较小，而对于对角线偏移量最大的cry10000和cry2500矩阵，DIA表现最差，因此，对角线偏移量与基于DIA存储格式的SpMV算法效率有关，接下来将研究对角线偏移量与基于DIA存储格式的SpMV算法效率的关系。

## 对角线偏移量与DIA存储结构

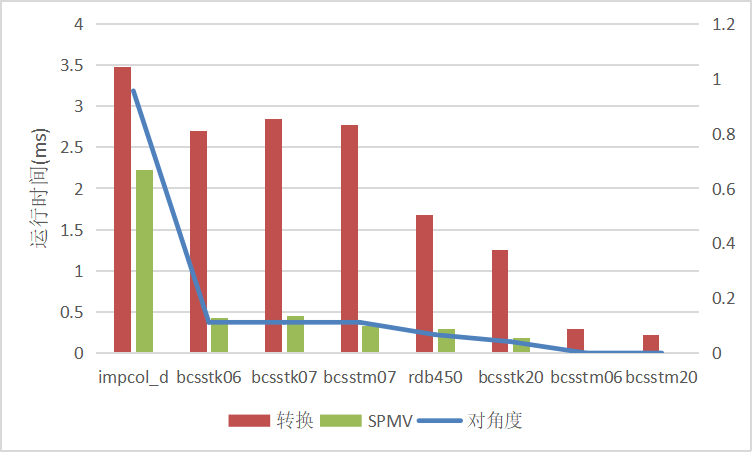
为了排除随机生成向量对SpMV算法的影响，下面的实验全部采用单位向量。设置一个参数p，表示对角线偏移量与矩阵行数的比值，称之为对角度。为了排除矩阵规模和非零元个数产生的影响，挑选了规模和非零元个数相近的矩阵进行实验。

选取的矩阵参数如下：

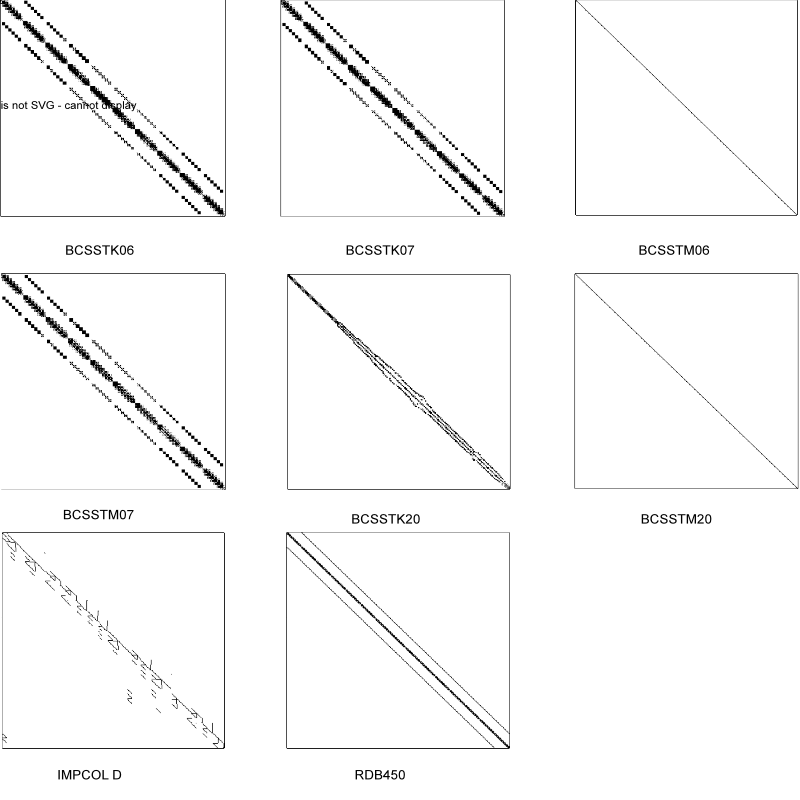
表 3-2选取的矩阵参数

|  |  |  |
| --- | --- | --- |
| 矩阵名称 | 规模 | 非零元个数 |
| bcsstk06 | 420 \* 420 | 4140 |
| bcsstk07 | 420 \* 420 | 4140 |
| bcsstk20 | 485 \* 485 | 1810 |
| bcsstm06 | 420 \* 420 | 420 |
| bcsstm07 | 420 \* 420 | 3836 |
| bcsstm20 | 485 \* 485 | 485 |
| impcol\_d | 425 \* 425 | 1339 |
| rdb450 | 450 \* 450 | 2580 |

矩阵结构图如图 3-7所示。实验结果如图 3-6所示，转换即为格式转换开销，单位为ms。将矩阵按照对角度从大到小排序，可以看到这样的趋势：对角度越大，DIA格式转换和SpMV算法的开销越大。这也解释了之前的实验结果，即对于对角线偏移量最大的cry10000和cry2500矩阵，DIA表现最差。同时，本文在DIA代码中使用的数据结构，也可能是矩阵规模过大时算法效率骤降的因素之一，但是并不影响结论。



**图 3-6 对角度与算法性能**



**图 3-7矩阵结构示意图**

## 选择存储格式

在前面研究的基础上，提出这样一种机制，在读取mtx文件后，先计算矩阵的对角线偏移量以及对角度，根据对角度不同来选择存储格式，基于之前的研究，存在一个对角度的值p，当对角度大于p时，采用CSR5效率更高，而对角度小于p时，采用DIA效率更高。

该算法较为简单，伪代码如下：

|  |
| --- |
| **Algorithm 5 计算对角度并选择矩阵格式** |
| **INPUT** COO格式矩阵块A，p  **OUTPUT**  1:**procedure** Choose\_fomart(A,p)  2: max\_offset = 0;  3: for i ← 1 to A.nnz  4: offset ← abs(A.col\_idx[i] - A.row\_idx[i]);  5: max\_offset ← max(max\_offset,offset);  6: x = max\_offset / A.n;  7: if x > p then  8: //choose CSR5  9: else  10: //choose DIA |

其中计算对角度的算法时间复杂度仅为O(N)，经过测算，这段算法在代码中占用的时间仅为格式转换和SpMV算法执行时间的不到百分之一，额外的开销非常小，可以忽略不计。

## 本章小结

本章介绍了基于DIA存储格式和CSR5存储格式的SpMV算法，并对两种算法在不同矩阵下的效率进行了对比，发现CSR5的表现较为稳定，而DIA的表现在面对不同矩阵时差距较大。进一步将对规模相近的矩阵进行基于DIA的 SpMV算法，得出的结论是算法效率与矩阵对角度（对角线偏移量与矩阵行数的比值）相关，对角度越小，DIA效率越高。最后，提出了一种选择存储格式的算法。

# 实验与结果分析

在上一章，已经发现了在矩阵规模不变时基于DIA的SpMV算法效率与矩阵的对角度有关，在这一章，将深入研究这一关系，并找出当基于DIA的SpMV算法效率超过基于CSR5的SpMV算法效率时，矩阵的对角度取值或者取值范围。

## 对角度的取值测算

选取一系列规模相近且对角度各不相同的矩阵如下：

表 4-1 矩阵名称及参数

|  |  |  |
| --- | --- | --- |
| 矩阵名称 | 规模 | 非零元个数 |
| bcsstk06 | 420 \* 420 | 4140 |
| bcsstk20 | 485 \* 485 | 1810 |
| bcsstm06 | 420 \* 420 | 420 |
| impcol\_d | 425 \* 425 | 1339 |
| rdb450 | 450 \* 450 | 2580 |
| BFW398A | 398 \* 398 | 3768 |
| WEST0381 | 381 \* 381 | 2157 |
| CK400 | 400 \* 400 | 2860 |

新增加的三个矩阵结构如图：

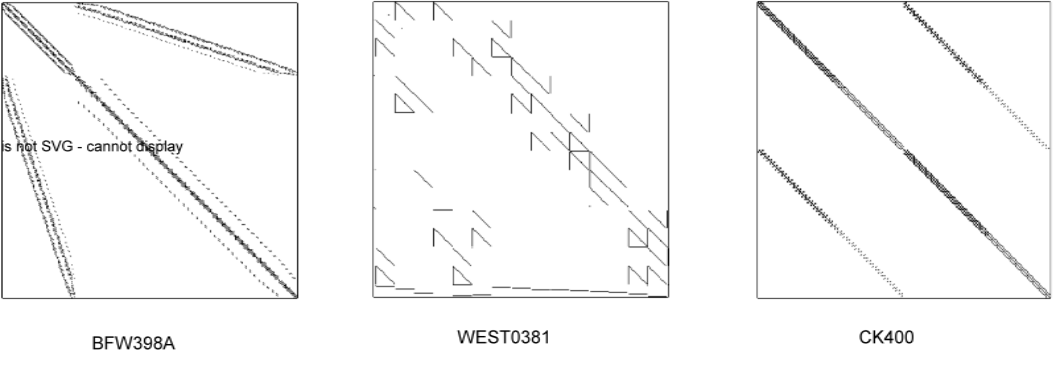


图 4-1矩阵结构图

对于这八个矩阵，重新在同一环境下进行实验，实验结果重复十次取平均值。

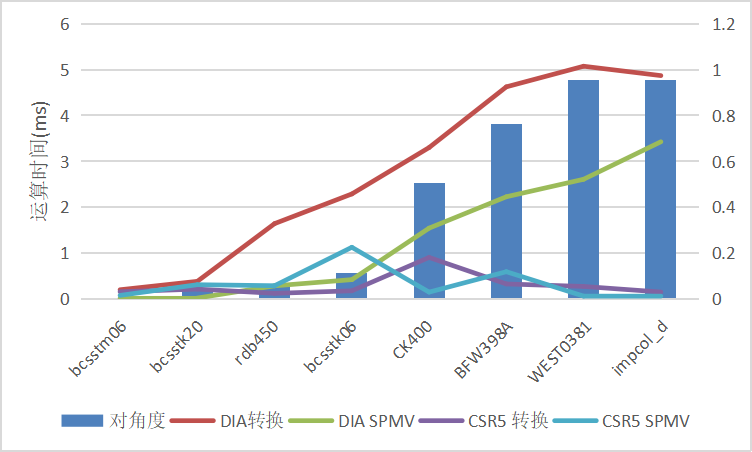


图 4-2不同对角度矩阵下DIA和CSR5对比

如图 4-2所示，DIA在格式转换开销和SpMV算法开销上明显随矩阵对比度增大而提高，CSR5格式相对稳定。同时，当矩阵对比度低于0.05时，基于DIA的SpMV算法效率高于基于CSR5的SpMV算法。

## 引入存储格式选择后算法的效率

将存储结构的选择，以及基于CSR5和DIA的SpMV算法结合起来，其中CSR5算法参数取。程序执行时间取存储格式选择，存储格式转换以及SpMV算法之和，而不是统计程序运行时间，防止其它语句影响测量结果。使用的矩阵如表 4-1所示。运行时间的测算仍取执行100次的平均值。

实验结果如图 4-3所示，可以看到引入了存储格式选择后，算法效率兼具CSR5存储格式的稳定性，以及特殊情况下DIA存储格式的高效性。

经过计算，在处理bcsstm06时，选择算法的效率比CSR5提高了8.3%，在处理bcsstk20时比CSR5提高了22.6%，而在处理其它矩阵时，效率会有下滑，平均下滑了5%。

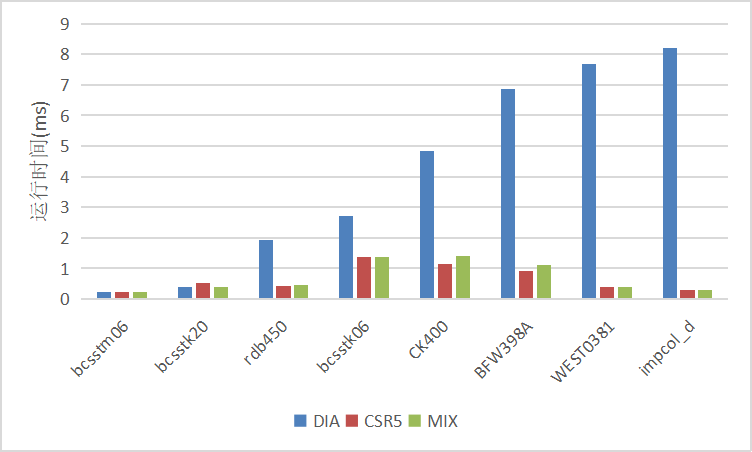


图 4-3 三种算法运行时间对比

## 本章小结

本章通过实验确定了对角度的大致取值，当对角度低于0.05时采用DIA存储格式，高于0.05时采用CSR5存储格式，能够取得最好的算法效率，并通过实验验证了引入存储格式选择后，算法效率能在大部分情况下与基于CSR5的SpMV算法效率持平，并在特殊情况下，即矩阵对角度小于0.05时超过基于CSR5的SpMV算法。

# 总结与展望

对稀疏矩阵乘法优化机制，尤其是稀疏矩阵存储格式进行研究，是为了克服当前单一稀疏矩阵存储格式在效率上遇到的瓶颈。存储格式优化主要有两种思路，一种是混合存储格式，另一种是进行存储格式选择，取得不同情况下的最优解。基于第二种思路，做了以下工作：

1. 复现了基于CSR5存储格式的SpMV算法，在Intel CPU平台上测试了性能，包括存储格式转换时间和SpMV算法执行时间。
2. 复现了基于DIA存储格式的SpMV算法，在相同环境下测试了性能，并与CSR5进行比较。
3. 研究了基于DIA存储格式的算法效率与矩阵对角线偏移量的关系，得出了对角线偏移量占矩阵长度0.05时，DIA能够取得比CSR5更高的算法效率。
4. 设计了选择存储格式的算法，并实验验证了引入选择后，算法效率在特殊情况下能够取得不错的提升。

本文对稀疏矩阵乘法优化机制研究进行了探索，但是研究仍有缺陷，主要有以下几点：

1. 只在CPU平台进行了实验，没有在其它平台进行实验。
2. 测试集规模有限，在现有测试集下得出的结论不一定准确，需要大量实验数据支持。在面对其它测试集时，以对角度0.05作为判断条件不一定成立。
3. 只对方阵进行了研究，没有对非方阵进行研究。
4. 可以引入更多存储格式，采用更先进的技术进行选择，如采用机器学习模型预测何种存储格式最优。

随着高性能计算领域不断发展，对高效存储格式的需求也进一步增大，相信新型存储格式研究的发展能够满足人类需

致 谢

在本论文完成之际，我要向所有支持我、帮助我的人表示最诚挚的感谢。首先，我要感谢我的指导老师张宇教授。张老师在整个研究过程中给予了我莫大的支持和帮助，包括对我的论文选题、研究方法和论文框架等方面的指导和帮助。他的严谨治学和勤奋工作的态度也深深地影响了我，让我懂得了做事情要认真、要有耐心和恒心，这对我的未来发展将产生重要影响。

同时，我还要感谢我的家人和朋友们。感谢他们在我学业和生活中的支持和鼓励，让我能够在学业上取得这样的成绩。感谢我的同学们，他们在研究过程中给予了我很多帮助和建议，与他们的交流让我更深入地了解了自己的研究方向。

最后，再次感谢所有为我提供帮助和支持的人们，是你们的支持让我能够完成这篇论文。我相信，在未来的学习和工作中，我将不断努力，发扬“勤奋、创新、诚信、责任”的精神，为实现自己的人生价值做出更大的贡献。

参考文献

1. Grzegorz Kwasniewski et al. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication.[J]. CoRR, 2019, abs/1908.09606
2. Sparse Matrix Representations & Iterative Solvers, Lesson 1 by Nathan Bell. <http://www.bu.edu/pasi/files/2011/01/NathanBell1-10-1000.pdf>
3. Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. ACM Trans. Math. Softw. 38, 1, Article 1 (November 2011), 25 pages. <https://doi.org/10.1145/2049662.2049663>
4. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors [C]. In SC. 2009.
5. Kincaid D, et al. ITPACKV 2D user’s guide [R]. 1989.
6. Monakov A, Lokhmotov A, Avetisyan A. Automatically Tuning Sparse MatrixVector Multiplication for GPU Architectures [C/OL]. In High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010,Pisa, Italy, January 25-27, 2010. Proceedings. 2010: 111–125. <https://doi.org/10.1007/978-3-642-11515-8_10>.
7. Kreutzer M, et al. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units [J]. SIAM J. Scientific Computing. 2014
8. Weifeng Liu 0002;Brian Vinter CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. [J] IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,2015
9. DavidR.Kincaid and ThomasC.Oppe. Recent vectorization and parallelization of ITPACKV[J]. Lecture Notes in Mathematics, 1990, 1457(1) : 58-78.
10. 刘芳芳,杨超.一种提高SpMV向量化性能的新型稀疏矩阵存储格式[J].数值计算与计算机应用,2014,35(04):269-276.
11. 陈世钊. 新型众核并行体系结构高效稀疏矩阵向量乘研究[D].国防科技大学,2018.DOI:10.27052/d.cnki.gzjgu.2018.000507.
12. 谢佩珍. 适合向量化的稀疏矩阵存储格式研究[D].国防科学技术大学,2016.
13. 杨世伟,蒋国平,宋玉蓉等.基于GPU的稀疏矩阵存储格式优化研究[J].计算机工程,2019,45(09):23-31+39.DOI:10.19678/j.issn.1000-3428.0053513.
14. 王志奇. GPU上稀疏矩阵向量乘积优化及最优存储格式预测方法[D].东北师范大学,2020.DOI:10.27011/d.cnki.gdbsu.2020.000347.
15. Bhuyan L N, Chong F, Sarkar V. Proceedings of the 29th ACM on International Conference on Supercomputing, ICS’15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015 [C/OL]. ACM, 2015
16. Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14). IEEE Press, 781–792. https://doi.org/10.1109/SC.2014.69