学　　号　2016011560

有限元法基础
大型线性稀疏系统的求解与储存方法

院 (系) 名 称：　航天航空学院

专 业 名 称：　工程力学

学 生 姓 名：　易泽吉

二〇一八年三月

Contents

1 摘要

向量乘法是许多科学和工业应用中大规模计算的核心。现代高性能计算机系统，例如多核，GPU，使用协处理器和特殊寄存器的单指令多结构计算，可加速针对众多任务的向量乘法计算。稀疏矩阵是非零元占多数的矩阵，因为具有大量重复的零元素，将稀疏矩阵整体完整的存储在内存中是一种巨大的空间开销，同时降低了访问，读写的速度，同时一定程度上降低了运算的速度。为了有效地减少稀疏矩阵的存储空间，研究者们已经提出了许多针对稀疏线性系统的数据结构。这些数据结构通过仅存储矩阵的非零值来减少存储空间，从而大幅减少了存储的代价，同时提升了计算的速度。这对于现在的计算机系统非常重要，因为现有的内存容量时常无法满足大规模的完整矩阵存储。从 1970 年开始，演化出许多格式以减少稀疏矩阵的存储空间，Bell 和 Garland 在 2008 年的论文中指出这一点 [2]。其中，大多数格式是针对特定情况定制的。CSR，COO，ELLPACK 和 Diagonal 是其他格式发展的四种基本格式。所有这些格式在不同的计算平台中表现不同。Greathouse 和 Daga 在 2014 年证明 CSR 格式 [1] 是最适合具有深度缓存内存层次结构的系统，但它在具有合并内存的 GPU 中表现失败并且无法提供并行性。与 CSR 相比，ELLPACK 格式通过提供合并存储器和提供 Bustamam 等人于 2012 年提出的指定的指令级并行性而在 GPU 环境中表现优异。

2　稀疏矩阵存储方法

、

2.1　基本存储方式

2.1.1　Diagonal Format

这种格式适用于非零值集中在矩阵对角线上的矩阵。下方矩阵展示了所考虑的样本稀疏矩阵的对角线格式表示。这种格式既节省空间又节省时间，但不适用于所有类型的矩阵。它用两个数组实现，一个二维数据数组保存非零值，其中对角元素按列放置，另一个是偏置数组，存储子对角线与主对角线的偏移量。对角线格式通过减少数据阵列和 X，Y 向量的存储器访问次数来提升向量操作中的性能，因为它们存储在相邻的存储器位置中。然而，这种格式对存储开销并没有大幅的削减，因为它仍然存储了矩阵对角线中的零值。

$$A_{\text{mxn}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \end{bmatrix} \tag{2.1}$$

$$\text{Dat}\,a = \begin{bmatrix} * & 2 & 4 & 7 \\ * & 5 & 3 & 6 \\ * & 0 & 0 & * \\ 1 & 6 & 2 & * \end{bmatrix} \quad \text{Offset} = \begin{bmatrix} -4 & -1 & 1 & 3 \end{bmatrix} \tag{2.2}$$

2.1.2　ELLPACK Format

这种格式非常适合半结构化和非结构化网格，并且适用于矢量体系结构 [2]。它类似于对角线格式，唯一的区别是列索引是明确指定的。但对于对角线上的元素，列索引隐含在偏移数组中。它由两个二维数组组成，一种一个以行方式存储数据值，另一个存储列索引。如果矩阵大小是 m×n，其中 Nmnzr 表示每行的最大非零值数，则列索引数组的大小为 m×Nmnzr。ELLPACK 适用于矩阵的最大数量与

平均值没有多大差别的矩阵。ELLPACK 格式占用的存储空间在下式中给出。

$$A_{\text{man}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \end{bmatrix} \tag{2.3}$$

$$Data = \begin{bmatrix} 4 & 7 & * \\ 2 & 3 & 6 \\ 5 & * & * \\ 2 & * & * \\ 1 & 6 & * \end{bmatrix} \quad \text{Colum indices } = \begin{bmatrix} 1 & 3 & * \\ 0 & 2 & 4 \\ 1 & * & * \\ 4 & * & * \\ 0 & 3 & * \end{bmatrix} \tag{2.4}$$

$$ELLPACK_{\text{storage}} = 2\left(N_{\text{mnzr}} \times m\right) \tag{2.5}$$

2.1.3 COO Coordinate Format

COO 是用于以非零值的坐标存储稀疏矩阵的最通用格式之一（Dongarraz 等人（1994））。它可以在任何平台上实现，而不必担心丢失任何数据。此格式存储行索引，列索引以及非零值。COO 很简单，对于任何稀疏模式，所需的存储取决于非零值的数量。它由三个 1 维数组实现，一个用于存储非零值，另外两个用于保存相应非零值的行和列索引，如下式所示。为了提高可读性，COO 对行数组或列数组进行排序。同时为了确保连续存储同一行中的元素，对行数组进行排序。记 NZV 为非零元素的数量,COO 存储空间在下式中给出

$$A_{\text{man}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \end{bmatrix} \tag{2.6}$$

$$Data = \begin{bmatrix} 4 & 7 & 2 & 3 & 6 & 5 & 2 & 1 & 6 \end{bmatrix} \tag{2.7}$$

$$Row - indices = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 4 \end{bmatrix} \tag{2.8}$$

$$Column - indices = \begin{bmatrix} 1 & 3 & 0 & 2 & 4 & 1 & 4 & 0 & 3 \end{bmatrix} \tag{2.9}$$

$$COO_{\text{storage}} = 3 \times NZV \tag{2.10}$$

2.1.4 Compressed Row Format

CSR 是最流行和通用的格式，可为高性能架构中的结构化和非结构化稀疏矩阵提供出色的压缩比（Dongarraz 等（1994））。具有 CSR 格式的向量运算在 CPU 上实现时显示出良好的性能改进，并且所有算法（如 BLAS，LAPACK 和 CUSparse）均支持此格式。它使用三个 1 维数组，一个用于保存非零值，第二个用于保存每行非零值的数量，第三个用于保存非零值的列索引。此格式的大小主要取决于矩阵中的非零值的数量。CSR 较好的反映了矩阵的稀疏特性，并且受每行非零值的分布的影响。

$$A_{\text{man}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \end{bmatrix} \tag{2.11}$$

$$Data = \begin{bmatrix} 4 & 7 & 2 & 3 & 6 & 5 & 2 & 1 & 6 \end{bmatrix} \tag{2.12}$$

$$Column - indices = \begin{bmatrix} 1 & 3 & 0 & 2 & 4 & 1 & 4 & 0 & 3 \end{bmatrix} \tag{2.13}$$

$$Ptr = \begin{bmatrix} 0 & 2 & 5 & 6 & 7 & 9 \end{bmatrix} \tag{2.14}$$

CSR 占据的存储空间为

$$CSR_{\text{storage}} = 2 \times NZV + m + 1 \tag{2.15}$$

2.1.5 Compressed Column Format

CSC 格式类似于 CSR 格式，其中列中的非零值连续存储在存储器中（Duff 等（1989））。矩阵的 CSC 格式等同于 CSR 格式的转置。这种格式是由于一些编程语言（例如 FORTRAN）而存在的，这种编程语言以列为单位而不是按行在存储器中存储矩阵。这也是众所周知的 Harwell-Boeing 稀疏矩阵格式。由于 CSC 格式为 CSR 存储格式的转置，所以其存储方式和容量相似。

2.2 基于分块的存储方式

2.2.1 Blocked CSR Format

在某些求解应用中，实时性较强，稀疏矩阵中零的出现表明它们的结构具有规律性，例如在大小为 r 的块中出现零。对于表现出这种子结构的矩阵，这种格式

比 CSR 更好（Im 和 Yelick（2001））。这里，矩阵被分成大小为 r 的块。对于样本矩阵，此格式如图 2.6 所示。它由三个数组组成，例如用于以行方式存储非零块的数据值的数据数组，用于存储块列索引的列索引数组以及用于存储每行中块的开始的行指针。在以下示例中，考虑块大小为 2。

$$A_{\text{mxn}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.16}$$

$$Bcolumn - indices = \begin{bmatrix} 0 & 1 & 0 & 2 & 0 \end{bmatrix} \tag{2.17}$$

$$Row - ptr = \begin{bmatrix} 0 & 2 & 4 & 5 \end{bmatrix} \tag{2.18}$$

如果在所考虑的块大小中没有足够数量的非零值，则此格式的主要缺点是增加了零元素的数量。块大小是确定此格式效率的重要参数，块大小的增加会降低此格式的效率。记矩阵非零元数量为 N_{nzb}, 分块矩阵大小为 r，总的存储量为

$$BCSR_{\text{storage}} = (N_{nab} \times 2r) + N_{ncb} + m/r + 1 \tag{2.19}$$

2.2.2 Row Grouped CSR Format

这是 CSR 格式的扩展，其中矩阵被逐行分解成部分（Oberheuber 等人（2010））。每个部分由一组行组成。这中存储方法需要四个一维阵列。数据数组用于存储每行中第一次出现的非零值，然后是第二次，依此类推..... 在一个组内，然后在第二组中以相同的方式继续。如果每行中的非零值的数量小于最大值，则发生零填充。与 CSR 格式相比，这会产生开销。列数组包含数据值的列索引。行长度数组显示每行中非零值的数量，组指针保持组开头的偏移量。与其他格式相比，这种格式更好地利用了缓存。

$$A_{\text{mxn}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.20}$$

$$Data = \begin{bmatrix} 4 & 2 & 5 & 7 & 3 & 0 & 0 & 6 & 0 & 2 & 1 & 4 & 0 & 6 & 0 \end{bmatrix} \tag{2.21}$$

$$Column-indices=\begin{bmatrix}1 & 0 & 1 & 3 & 2 & * & * & 4 & * & 4 & 0 & 2 & * & 3 & *\end{bmatrix} \tag{2.22}$$

$$Row-length=\begin{bmatrix}2 & 3 & 1 & 1 & 2 & 1\end{bmatrix} \tag{2.23}$$

$$Group-pointer=\begin{bmatrix}0 & 9\end{bmatrix} \tag{2.24}$$

这种方法所需的存储空间为

$$RCSR_{\text{storage}}=2X+m+N_g \tag{2.25}$$

记 $N_{mnzpr}g(i)$ 为每组中最大的非零元数量，则

$$X=\sum_{i=1}^{N_g}(N_{mnzr}g(i))\times G_{size} \tag{2.26}$$

2.2.3 Quad Tree CSR Format

它是 CSR 和 Quad 树格式的组合，其中给定矩阵递归地划分为四个象限，直到节点大小等于密度大小（Zhang et al（2013））。这些节点可以是三种类型，例如整个节点具有零值的空节点，其中存在零和非零值的组合的混合节点以及仅完全由非零值组成的完整节点。依次使用 CSR 格式存储混合节点和完整节点，并忽略空节点。这种格式具有递归编程风格，因此易于从 CSR 等流行格式转换，容易修改数据并拥有更好的缓存利用率。由于这种格式必须存储节点信息和非零值，它经尝不是空间有效的，但使用此格式的向量操作实现比 CSR 格式更快。其中的节点是混合节点，它们使用具有三维一维阵列的常规 CSR 格式存储。这种格式的性能主要取决于密度大小和数据加载到缓存中，这增加了数据的局部性并增加了性能增益。这种存储格式所占据的空间为

$$QCSR_{\text{storage}}=N_q\times CSR_{\text{storage}} \tag{2.27}$$

2.2.4 Minimal Quad Tree Format

这是一种称为 Minimal Quad tree 的新存储格式，可有效地用于 I / O 操作（Simecek 等人（2012））。它是四叉树格式的扩展，克服了四叉树格式的缺点，例如存储指针时的空间开销。此格式侧重于压缩稀疏矩阵的结构，其中非零值是隐式的。诸如存储未加权图的事件矩阵之类的应用程序以这种格式显示空间效率。类似于四叉树格式，给定矩阵被递归地划分为块，并且如果元素全部为零则用比特值 0 表

示块，并且如果块中存在非零值则用值 1 表示。该格式使用比特流来表示矩阵的结构，而不是关于非零值的信息。对于上面的例子，矩阵被分成密度为 2 的块，它表示为比特流。

$$bitstream = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.28}$$

存储效率为

$$\text{Max}(MQT)_{\text{storagec}} = 4 \times N \left(1/3 + \log_4 \left(n^2/N\right)\right) \tag{2.29}$$

2.3 向量化的存储方式

2.3.1 Compressed Multi Row Format

CMRS 是专门为 GPU 设计的新格式（Koza 等人（2012））。它是 CSR 格式的扩展，CSR 格式可以有效地转换为此格式，而无需任何内存开销。与其他格式相比，这种格式的优点在于，它不需要任何零填充和行或列重新排序。这种格式在 GPU 中显示出良好的加速，并且可以很好地适应未来的 GPU 以及其他面向吞吐量的架构。这种格式由四个 1 维数组和一个整数元素组成，用于跟踪高度参数，如下式。数据和列索引数组与 CSR 格式相同。数组字符串指针用于保存每个条带中的非零值的数量，并且条带数组中的行保存每个条带的行号。

$$A_{\text{mxn}} = \begin{bmatrix} 4 & 5 & 0 & 1 & 0 \\ 0 & 7 & 8 & 0 & 0 \\ 0 & 9 & 2 & 3 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 10 & 11 \end{bmatrix} \tag{2.30}$$

$$= \begin{bmatrix} 4 & 5 & 1 & 7 & 8 & 9 & 2 & 3 & 6 & 10 & 11 \end{bmatrix} \tag{2.31}$$

$$Column - indices = \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 1 & 2 & 3 & 3 & 3 & 4 \end{bmatrix} \tag{2.32}$$

$$Row - strip = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.33}$$

$$strip - ptr = \begin{bmatrix} 0 & 5 & 9 & 2 \end{bmatrix} \tag{2.34}$$

记条带数 $N_s =$ 行数/条带高度，则 CMRS 方法的存储空间为

$$CMRS_{\text{storage}} = (3 \times NZV) + N_s + 1 \tag{2.35}$$

2.3.2 Adaptive CSR Format

正常的 CSR 格式在 GPU 环境中表现不佳。这是由于负载不平衡，并行性降低和内存访问模式不规则造成的。为了克服这个问题，已经发展了 GPU 特定格式。这里的缺点主要表现在从 CSR 到这些格式的转换开销，这会产生存储和运行时开销。为了克服这个缺点，由（Great house 和 Daga（2014））提出了一种称为 CSR 自适应的新算法，如下所示。根据行中非零值的数量，矩阵被分类为长行和短行。该算法在面对长短行时在 CSR 流算法和 CSR 矢量算法之间动态切换。

$$Adaptive_{\text{storage}} = 2 \times NZV + m + 1 \tag{2.36}$$

2.3.3 Streamed Storage Format

决定向量乘法内核性能的关键参数是内存带宽。IBM Power Processor 中的预取流组件会增加内存带宽。有研究者提出了一种利用这种硬件组件并提高 SpMV 性能的新流式格式（Guo 和 Gropp（2011））。这些格式在向量乘法内核中显示了相较于 X86 处理器中的 CSR 和 BCSR 格式的性能提升。在流式传输下有两种格式，如流式 CSR 和流式块 CSR。

2.3.3.1 Streamed CSR Format

此格式是适用于矢量化的 CSR 格式的扩展。这里定义了流组的数量，并且通过行号 mod 流的数量找到分配给组流的行。例如，如果假设流的数量是 4，那么第 0 行，第 4 行被分配给流 0，相应地分配其他行元素。将零填充到流中，使得所有流的大小相同。列索引数组包含相应的列 ID。对于添加的零，列索引保持该行中非零值的相同索引。ptr 数组保存每个块行的第一个元素的列索引中的偏移量。与 BCSR 格式相比，此格式增加了更少的非零值，从而提高了计算速度。

$$A_{\text{mxn}} = \begin{bmatrix} 4 & 5 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 7 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 2 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.37}$$

列索引数组：

$$Column - R = \begin{bmatrix} 0 & 1 & 3 & 3 & 4 \end{bmatrix} \tag{2.38}$$

$$Column-G=\begin{bmatrix}1 & 2 & 2 & * & *\end{bmatrix} \tag{2.39}$$

$$Column-B=\begin{bmatrix}1 & 2 & 3 & * & *\end{bmatrix} \tag{2.40}$$

$$Column-P=\begin{bmatrix}3 & 3 & 3 & * & *\end{bmatrix} \tag{2.41}$$

数据数组：

$$R=\begin{bmatrix}4 & 5 & 1 & 10 & 11\end{bmatrix} \tag{2.42}$$

$$G=\begin{bmatrix}7 & 8 & 0 & 0 & 0\end{bmatrix} \tag{2.43}$$

$$B=\begin{bmatrix}9 & 2 & 3 & 0 & 0\end{bmatrix} \tag{2.44}$$

$$P=\begin{bmatrix}6 & 0 & 0 & 0 & 0\end{bmatrix} \tag{2.45}$$

$$ptr=\begin{bmatrix}0 & 3 & 5\end{bmatrix} \tag{2.46}$$

流 CSR 存储格式所占用的空间为

$$StreamedCSR-storage=2\left(N_S\times\max(NZS)\right)+S_r+1 \tag{2.47}$$

2.3.3.2 Streamed BCSR Format

在非零值集中在特定块中的某些应用中，块 CSR 格式是有效的。这种矩阵也可以采用流格式。下方所示给定矩阵被分成 4×4 块。在这种格式中，首先存储所有块的第一行，然后同时存储第二行。列索引跟踪每个块的列索引，ptr 数组保存每个块行中第一个块的列索引数组中的偏移量。块大小为 4 的流式 BCSR 格式对于具有大量非零值的矩阵以及当稀疏矩阵不能适合高速缓存时，显示出比 BCSR 格式更好的加速。流式 BCSR 格式通过当前处理器中提供的单指令多数据模式的指

令展现出较大的性能改进。

$$A_{\text{mxn}} = \begin{bmatrix} 4 & 5 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 7 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 2 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.48}$$

$$R = \begin{bmatrix} 4 & 5 & 0 & 1 & 0 & 0 & 0 & 10 & 11 & 0 & 0 & 0 \end{bmatrix} \tag{2.49}$$

$$G = \begin{bmatrix} 0 & 7 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.50}$$

$$B = \begin{bmatrix} 0 & 9 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.51}$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.52}$$

$$\text{Ind} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \tag{2.53}$$

$$Ptr = \begin{bmatrix} 0 & 1 & 3 \end{bmatrix} \tag{2.54}$$

记 N_{nzb} 为矩阵中非零块的数量，b_{size} 为块的大小

$$StreamedBCSR - storage = Ns \times (N_{nzb} \times b_{size}) + 2\,(N_{nzb}) \tag{2.55}$$

2.3.4 Sliced ELLPACK-C-SIGMA Format

正常的 ELLPACK 格式通过填充零并通过提供合并的存储器访问和指令级并行性来与流式处理器良好地协同工作。基于矩阵特征，该格式优于 CSR 格式。为了克服由于零填充引起的 ELLPACK 的存储开销，切片的 ELLPACK 格式出现（Kreutzer 等人（2014）），如下所示。在此，给定矩阵被分成大小块，并且每个切片使用 ELLPACK 格式存储。它也被称为 SELLPACK-C。这里 C 表示块大小。块大小为 1 导致 CSR 格式，块大小等于矩阵尺寸，导致 ELLPACK 格式。如果块中的所有行具有相等数量的非零值，则该格式克服了 ELLPACK 格式中零填充的缺点。但是在其他情况下，通过添加比 ELLPACK 格式更少的零来以这种格式节省空间

开销。通过按非零值的降序对行进行排序并将它们分组为块，可以克服这个缺点。通过这种方法，具有相等长度的非零值的行将聚集在一起。然而，全局排序将减少向量乘法操作的空间和时间局部性，这将进一步降低内核的性能。

$$A_{\text{mxn}} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \\ 2 & 4 & 0 & 0 & 5 \end{bmatrix} \tag{2.56}$$

$$Slice - ptr = \begin{bmatrix} 0 & 6 & 9 & 15 \end{bmatrix} \tag{2.57}$$

$$Value = \begin{bmatrix} 4 & 2 & 7 & 3 & 0 & 6 & 5 & 2 & 1 & 2 & 6 & 4 & 0 & 5 \end{bmatrix} \tag{2.58}$$

$$Column = \begin{bmatrix} 1 & 0 & 3 & 2 & * & 4 & 1 & 4 & 0 & 0 & 3 & 1 & * & 4 \end{bmatrix} \tag{2.59}$$

存储所需容量为

$$SlicedELLPACK - storage = N_s + 1 + 4 \times \left(\sum_{i=1}^{N_s} N_{nzv}(i) \right) \tag{2.60}$$

其中 Ns 是切片的数量，Nnzv 表示每个切片的行中的非零元素的最大数量。在上面的矩阵中，块大小是 2，因此 σ 大小被认为是块大小的倍数，例如 4,8,12 等。这里我们将 σ 的值视为 4 个连续行，并且元素按降序排序并存储。它最适合具有单指令多数据功能的当前一代处理器，如 Sandy Bridge，Xeon Phi 和 Nvidia Tesla K20。为了在这些体系结构中具有良好的向量乘法性能改进，每行的非零值的平均数应该大于指令所对应的数据的宽度。此格式在各种计算机设备上展示出更好的性能。可以使用编译器或使用 C intrinsic 来对此格式进行矢量化。

Bibliography

[1]

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.