Cache Friendly Sparse Matrix Vector Multiplication

Sardar Anisul Haque[†], Shahadat Hossain[⋆], Marc Moreno Maza[†]
[†]Ontario Research Centre of Computer Algebra, University of Western Ontario, London, Canada
[⋆] Department of Computer Science, University of Lethbridge, Lethbridge, Alberta, Canada.

Overview

Sparse matrix-vector multiplication or *SpMxV* is an important kernel in scientific computing. For example, in the conjugate gradient method, where *SpMxV* is the main computational step. Though the total number of arithmetic operations to compute Ax is fixed, reducing the probability of cache misses per operation is still a challenging area of research. This preprocessing is done once and its cost is amortized by repeated multiplications. In this work, we present a new column ordering algorithm for sparse matrices. We analyze the cache complexity of *SpMxV* when A is ordered by our technique. The numerical experiments, with very large test matrices, clearly demonstrate the performance gains rendered by our proposed technique.

Sparsity and Cache Misses

Consider the following *SpMxV* problem:

$$\begin{pmatrix} a_{0,0} & 0 & 0 & 0 & a_{0,4} & 0 \\ 0 & 0 & a_{1,2} & 0 & 0 & a_{1,5} \\ 0 & a_{2,1} & 0 & a_{2,3} & 0 & 0 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

Assume that the cache has 2 lines each of 2 words. Assume also that the cache is dedicated to store the entries from x. During *SpMxV*, the successive states of the cache are:

$$\begin{bmatrix} \emptyset & \emptyset \\ \emptyset & \emptyset \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ \emptyset & \emptyset \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ x_4 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_4 & x_5 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_4 & x_5 \\ x_0 & x_1 \end{bmatrix} \begin{bmatrix} x_2 & x_3 \\ x_0 & x_1 \end{bmatrix}$$

So, the number of cache misses is 6. Reordering of columns or rows might improve data locality. If we permute the columns as below, the number of cache misses becomes 3, which is optimal.

$$\begin{pmatrix} a_{0,0} & a_{0,4} & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{1,2} & a_{1,5} & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{2,1} & a_{2,3} \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_4 \\ x_2 \\ x_5 \\ x_1 \\ x_3 \end{pmatrix}$$

$$\begin{bmatrix} \emptyset & \emptyset \\ \emptyset & \emptyset \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ \emptyset & \emptyset \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ \emptyset & \emptyset \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_2 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_2 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_1 & x_3 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_1 & x_3 \end{bmatrix}$$

Binary Reflected Gray Code Ordering

We develop a new column ordering algorithm based on *binary reflected Gray code* (BRGC for short) for sparse matrices. We will call it BRGC ordering. A p-bit *binary reflected Gray code* is a Gray code denoted by G^p and defined by $G^1 = [0, 1]$ and

$$G^p = [0G_0^{p-1}, \ldots, 0G_{2^{p-1}-1}^{p-1}, 1G_{2^{p-1}-1}^{p-1}, \ldots, 1G_0^{p-1}],$$

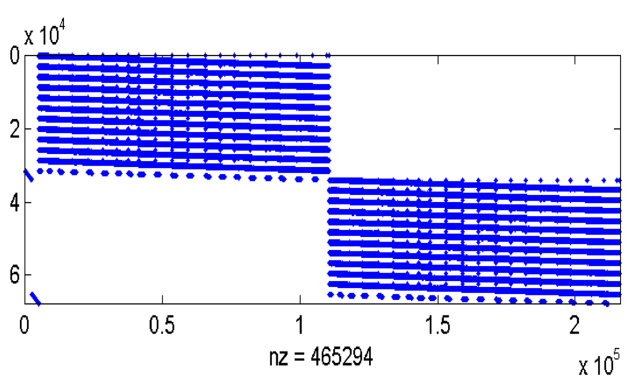
where G_i^p is the i-th string of G^p. We call i the rank of G_i^p in G^p. We consider each column of a $m \times n$ sparse matrix A as a binary string of length m where each nonzero is treated as 1. Hence, we have n binary strings of length m, say $\{b_0, b_1, \ldots, b_{n-1}\}$.
Let Π be the permutation of these strings satisfying the following property. For any pair of indices i, j with $i \neq j$, the rank of $b_{\Pi(j)}$ in G^m is less than that of $b_{\Pi(i)}$ if and only if $\Pi(i) < \Pi(j)$ holds. We refer to A_{brgc} as our sparse matrix A after its columns have been permuted by Π. This procedure is illustrated below.

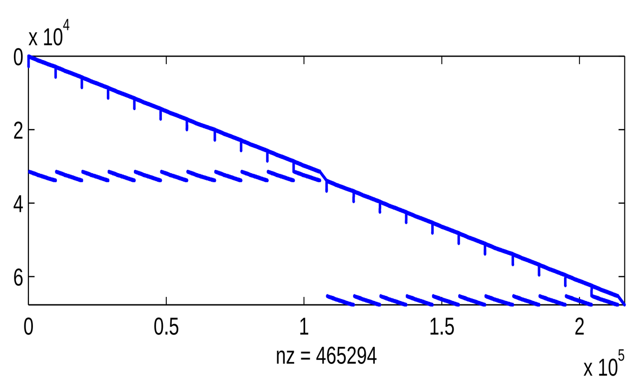
$$\begin{pmatrix} b_{00} & b_{01} & 0 & 0 & b_{04} & b_{05} \\ b_{10} & 0 & 0 & b_{13} & b_{14} & 0 \\ 0 & b_{21} & b_{22} & 0 & b_{24} & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} b_{05} & b_{01} & b_{04} & b_{00} & 0 & 0 \\ 0 & 0 & b_{14} & b_{10} & b_{13} & 0 \\ 0 & b_{21} & b_{24} & 0 & 0 & b_{22} \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Below, the matrix *fome21* from the *University of Florida sparse matrix collection* is shown before (left picture) and after (right picture) our BRGC ordering.





Time and Cache Complexity

The matrix A_{brgc} is obtained from A using $O(\tau)$ integer comparisons (on average) and $O(n+\tau)$ data-structure updates, where τ is the total nonzero entries in A.

For an *ideal cache* of Z words with L cache lines, the total number of expected cache misses in accessing x, where A is not BRGC ordered, is given by:

$$\overline{Q_1} = Z/L + (\tau - Z/L)\tfrac{n - Z/L}{n}.$$

When A is BRGC ordered, the expected number of cache misses in accessing x becomes:

$$\overline{Q_2} = n/L + Z/L + (n - Z/L)\tfrac{n/\rho - Z/L}{n/\rho} + (\tau - 2n)\tfrac{cn/\rho - Z/L}{cn/\rho},$$

where $1 \le c \le \rho$ holds.
For our large test matrices and today's L2 cache sizes, the following conditions hold: $n \in O(Z^2)$ and $Z > 2^{10}$. Using MAPLE, we could prove the following relation: $\overline{Q_1}$ - $\overline{Q_2} \approx n$.

Experimental Results and Conclusion

Matrix name	m	n	τ	SPMxV with BRGC ordering	SPMxV without any ordering
fome21	67748	216350	465294	3.6	3.9
lp_ken_18	105127	154699	358171	2.7	3.1
barrier2-10	115625	115625	3897557	19.0	19.1
rajat23	110355	110355	556938	3.0	3.0
hcircuit	105676	105676	513072	2.6	2.5
GL7d24	21074	105054	593892	3.0	3.2
matrix_9	103430	103430	2121550	8.4	8.0
GL7d17	1548650	955128	25978098	484.6	625.0
GL7d19	1911130	1955309	37322725	784.6	799.0
wiki-20051105	1634989	1634989	19753078	258.9	321.0
wiki-20070206	3566907	3566907	45030389	731.5	859.0

For each test matrix 1000 *SpMxVs* are performed. Our timing results are in seconds. In conclusion, BRGC re-ordering runs in linear time with respect to the number of nonzero entries. Moreover, it improves *SpMxV*, so that its cost can be amortized before $\sqrt{n}$ iterations in conjugate gradient type algorithms.