Sparse Matrix Computations using the Quadtree Storage Format

Ivan Šimeček
xsimecek@fel.cvut.cz

Department of Computer Science and Engineering,
Czech Technical University, Prague

Abstract. Computations with sparse matrices are used in the wide range of science projects. But suitable formats for storing sparse matrices are still under development, because the computation using widely-used formats (like XY or CSR) are slow and specialized and efficient formats (like CARB) have a large transformation overhead.
In this paper, we represent some improvements to the quadtree storage format. We also compare the performance during the execution of some basic routines from the linear algebra using widely-used formats and the quadtree storage format.

1 Introduction

The performance of mathematical operations with sparse matrices depends strongly on the used matrix storage format. In this paper, we represent a variant of the quadtree storage format implementation. We extended ideas represented in [3, ??, ??] and proved that the idea of sparse computations using the quadtree storage format implementation is viable.

2 Common sparse matrix formats

In the following text, we assume that $A$ is a real sparse matrix of order $n$. Let $nZ$ be the total number of nonzero elements in $A$.

2.1 The coordinate (XY) format

In this simplest sparse format, the matrix $A$ is represented by three linear arrays $A, x,$ and $y.$ Array $A[1, \ldots, nZ]$ stores the nonzero elements of $A,$ arrays $x[1, \ldots, nZ]$ and $y[1, \ldots, nZ]$ contain $x$-position and $y$-position, respectively, of the nonzero elements.
2.2 The compressed sparse row (CSR) format

It is the most common format (see [2]) for storing sparse matrices. A matrix \( A \) stored in the CSR format is represented by three linear arrays \( A, \text{adr}, \text{and} \ ci \) (see Figure 1 b)). Array \( A[1, \ldots, n_z] \) stores the nonzero elements of \( A \), array \( \text{adr}[1, \ldots, n] \) contains indexes of initial nonzero elements of rows of \( A \), and array \( \text{ci}[1, \ldots, n_z] \) contains column indexes of nonzero elements of \( A \). Hence, the first nonzero element of row \( j \) is stored at index \( \text{adr}[j] \) in array \( A \).

\[
\begin{align*}
\text{array} & \quad \text{adr} & \quad 1 & 2 & 3 & 4 & 5 \\
1 & \circ & \circ & \circ & \circ & \circ & \circ \\
2 & \circ & \circ & \circ & \circ & \circ & \circ \\
3 & \circ & \circ & \circ & \circ & \circ & \circ \\
4 & \circ & \circ & \circ & \circ & \circ & \circ
\end{align*}
\]

\[
\begin{align*}
\text{array} & \quad A & \quad \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
1 & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
2 & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
3 & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
4 & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ
\end{align*}
\]

\[
\begin{align*}
\text{array} & \quad \text{ci} & \quad 2 & 4 & 6 & 2 & 3 & 4 & 2 & 3 & 5 & 6 & 7 & 3 & 4
\end{align*}
\]

Fig. 1. An example of a sparse matrix \( A \) in the a) dense format, b) CSR format.

2.3 Register blocking formats

Widely-used formats are easy to understand, but sparse operations (like matrix-vector or matrix-matrix multiplication) using these formats is slow (mainly due to indirect addressing). Sparse matrices often contain dense submatrices (blocks in our terminology), so various blocking formats were designed to accelerate matrix operations. Compared to the CSR format, the aim of these formats is to consume less memory and to allow better use of registers. Storing a matrix as a set of small dense blocks can significantly improve the performance. Algorithms for the multiplication of such blocks can be fine tuned for a target architecture. But these specialized and efficient formats (like CARB[1]) suffer from a large transformation overhead.

2.4 Quadtree data format

Definition

In the beginning, this format was used for picture compression (for monochrome pictures). It can be also used for the sparse matrix storage.

Quadtree (for details see [3]) is the recursive data structure (4-ary tree, see Figure 2). This tree represents a partition of the matrix into submatrices (nodes).
One type is assigned to each node of the tree. Inner nodes of the quadtree are divided into "Mixed" or "Empty" nodes. Leafs of the quadtree are divided into "Full" or "Empty" nodes.

Great advantages of quadtree are the following:

- Easy (and fast) conversion from standard sparse matrix storage formats (CSR or XY, see Algorithm 2.4 on the Page 4).
- Modification of the quadtree is relatively easy (in comparison to standard formats). The exact complexity depends on the type of modified node.
- The recursive style of programming and recursive style of storage ("Divide and Conquer" approach) leads to codes with the surprising performance. This is our main motivation in this paper.

![Quadtree Diagram](image)

**Fig. 2.** Tridiagonal matrix in the quad-tree storage format. Locations of nonzero elements and "Empty" nodes are marked.

**Quadtree extensions**

The big drawback of quadtree structures is a larger control and data overhead than the standard formats. The standard quadtree implementation leads to space (and execution) inefficiency. So, we used the additional type of leaves: region-based version of the XY format (it means that we express all coordinates relatively to the beginning of the submatrix). This type of node we call "Sparse". Our second improvement is the elimination of "Empty" nodes, because they do not contain any useful information. They are simply represented by the NULL pointer.
Algorithm \texttt{Transf}(A)

(* The CSR ⇒ the quadtree transformation algorithm (simplified pseudocode) *)

/* Input: A = the matrix for the transformation */
/* Output: the pointer for the root of the quadtree */

\texttt{n}'Z = the number of nonzero elements in matrix A;
\texttt{n}' = the order of matrix A;

if (\texttt{n}'Z == 0) then return NULL;

if (\texttt{n}' > tile size) then
  create \texttt{M} - the leaf of type "Mixed";
  \texttt{M} is parent node of \texttt{M}_1, \texttt{M}_2, \texttt{M}_3, and \texttt{M}_4;
  divide the matrix \texttt{A} into four submatrices \texttt{A}_1, \texttt{A}_2, \texttt{A}_3, \texttt{A}_4;
  \texttt{M}_1 = \texttt{Transf}(\texttt{A}_1);
  \texttt{M}_2 = \texttt{Transf}(\texttt{A}_2);
  \texttt{M}_3 = \texttt{Transf}(\texttt{A}_3);
  \texttt{M}_4 = \texttt{Transf}(\texttt{A}_4);
  return \texttt{M};

else

  density = \texttt{n}'Z / \texttt{n}'^2;
  if (density > \texttt{fill in ratio}) then
    transform the matrix \texttt{A} to the leaf \texttt{F} of type "Full";
    return \texttt{F};
  else
    transform the matrix \texttt{A} to the leaf \texttt{S} of type "Sparse";
    return \texttt{S};

3 Evaluation of the results

All results was measured at Pentium Celeron M420 at 1.6 GHz, 2 GB@ 333 MHz, running OS Windows XP Professional with the following cache parameters:
L1 cache is 32 KB data cache with LRU replacement strategy. L2 cache is 1 MB data cache with LRU strategy.
Microsoft Visual Studio 2003
Intel compiler version 9.0 with switches:
/O3 /Og /Oa /Oy /Ot /Qpc64 /QxB /Qipo /Qsfalign16 /Zp16

3.1 Test applications

We have implemented two very basic routines from the LA:

- the multiplication of sparse matrix by a dense vector,
- the multiplication of sparse matrix by a sparse matrix.

These routines are often used in libraries for the numerical linear algebra. They represent building blocks for more complicated operations.
3.2 Test data
We have used 10 real matrices from various technical areas from MatrixMarket and Harwell sparse matrix test collection.

3.3 Influence of architecture-dependent parameters

![Fill-in ratio influence on the performance.](image)

We have tested the influence of two architecture-dependent parameters (fill_in_ratio and tile_size) on the performance and on the quadtree datasize. Figures 3 and 4 illustrate that the best performance is achieved for fill_in_ratio = 10% and tile_size smaller than 20. Figures 5 and 6 illustrate that the quadtree format is most space-efficient for fill_in_ratio > 16% and tile_size smaller than 20.

3.4 Experimental results

**Results for the matrix-vector multiplication**
The Figure 7 illustrates the performance for the matrix-vector multiplication.

For the XY format, both tested multiplications are relatively slow due to indirect addressing.

For the CSR format, both tested multiplications are faster than in the XY format due to the lower ratio between memory and arithmetic operations.

For the quad-tree format, the multiplication of sparse matrix by a dense vector is slower due to the large control and storage overhead. This operation using quad-tree is slower than with well-known formats due to larger amount of data structure (about 2.5 times larger than the CSR format) and due to the fact that cache locality increases, but data structure is read only once.
Results for the matrix-matrix multiplication

The Figure 8 illustrates the performance for the matrix-matrix multiplication. For the quad-tree format, the multiplication of sparse matrix by a sparse matrix is usually about 2 - 10 times faster than using the CSR format. There are two main reasons for speedup:

- Quadtree format reduces indirect addressing.
- Quadtree data structure is read repeatedly and the cache locality increases due to the recursive memory access pattern.
4 Conclusions

We have implemented some very basic routines from the LA using unusual data structure called quad-tree. Codes with this data format show impressive speedups mainly for more complex matrix operations (like the multiplication of sparse matrix by a sparse matrix).
Fig. 8. Matrix-matrix multiplication.

5 Future works

- We should deeply measure the performance on various platforms.
- We should measure the cache behavior and derive an analytical model of the cache behavior.
- We should implemented another routines from the LA using quad-tree and investigate possibilities of multithreaded version of routines.

Acknowledgement

This research has been supported by MŠMT under research program MSM6840770014.

References