C++ Sparse Matrix Container

Zenkar S.
PES1201701532
Department of Computer Science and Engineering
PES University
Bangalore
zenkarsrinivas@gmail.com

Ravendra Singh.
PES1201700706
Department of Computer Science and Engineering
PES University
Bangalore
ravendras2@gmail.com

Kumar N S
Professor
Department of Computer Science and Engineering
PES University
Bangalore
kumaradhara@gmail.com

***Abstract*—Sparse matrix is a matrix with more zero values than non-zero values. When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. This is what the project's main focus is on, to provide generic and efficient C++ Standard Template Library(STL) compatible container to implement a new representation for sparse matrices, which can exploit their nature.**

***Index Terms*—Containers, Iterators, Standard Template Library, Algorithms**

I. Introduction

Matrices have significant applications in fields of study like image processing, computer graphics and machine learning. Efficient processing of these matrices is vital for algorithms used in these fields. When it comes to sparse matrices it becomes even more necessary that unnecessary processing of the indexes with null values is avoided. Keeping this in mind a container and related efficient operations are proposed through this paper. The project is an attempt to implement C++ Standard Template Library compatible container to support efficient storage and operations for 2 dimensional sparse matrices. The container makes use of C++ generics to support matrix storage using an array of map or unordered map containers based on user's preference and tries to support sufficient number of operations required for a sparse matrix. Although there exists many implementations of sparse matrix such as list of lists, coordinate lists, compressed sparse row, etc. ,they may have shortcomings like insertion time complexity, deletion time complexity or non-constant access time. The proposed container provides constant insertion and deletion time with unordered map and logarithmic time with map and better search complexity than that of 2D array. The generic nature of the container allows it to store data of any primitive or user defined type and using search efficient containers map or unordered map. It also provides improved matrix operations like addition, multiplication and transpose about which we discuss in the upcoming sections.
The project also implements a forward iterator class for the corresponding container for efficient traversal of only the non-zero elements and supporting necessary operators for the same.

II. Implementation

The project implemented using C++ tries to make the best use of the STL, template classes and template functions to make the container generic without much compromising on the efficiency of matrix operations. The container is defined as a template class with storage data type and underlying storage class for matrix row as the template arguments.

```
template<typename V,
typename container>
class sparse_matrix
```

The main container as a member contains an array of rows. Indexing on this array gives the corresponding matrix row. The storage class for the row is decided by the user as second argument to the template class which defaults to map. By just storing the non-zero elements we reduce the space complexity of storing to theta(number of nun-zero elements). Other class members include dimensions of the matrix rows and columns of integer type, integer variable to store number of non-zero elements and a variable to store zero(null) value of the storage data type, for example 0 for integer and 0.0 for float data type. The container provides 7 overloads for the constructor, some of which include the basic types such as the default constructor, copy constructor, constructor to create container from a 2D matrix,etc. Defined within the scope of the container is an Iterator about which we will discuss in the Iterator section.

A. Matrix Row Storage Class

The storage class for row of the matrix is constrained only to map and unordered map keeping in mind the time complexity to access elements of the matrix. Using sequential containers for rows pose efficiency issues in operations like random access or insertion and deletion of elements. Maps and Unordered Maps were the best choices available from remaining container classes in STL. Both these containers can be looked upon as collection of key value pairs where key corresponds to column index of the sparse matrix and value corresponding to data at the specifies row and column of the matrix. Maps provide element access with logarithmic time complexity while unordered maps provide element access in constant time complexity, so for huge matrix size one is suggested to go with unordered map for better performance. Since elements in the map are ordered internally based on keys, iterating a row gives elements in the order of column index, while order of elements on iterating an unordered map row is undetermined.

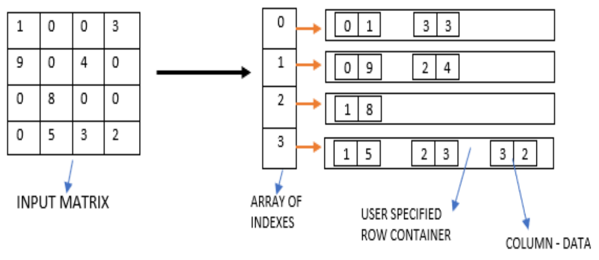


Fig. 1. Structure of Sparse Matrix Container

B. Operations

The implemented container supports the following operations:

- Copy Assignment : Performs a deep copy and returns reference to the assigned container
- begin : returns an iterator pointing to first non-zero element of matrix, if empty returns end
- end : Returns an Iterator to end of last row.
- Indexing operator[][] : Returns the data the specified index. Cannot be used to change the data.
- Element Insertion: Inserting non-zero element at specified index.
- Delete Index: Make the element at specified index as zero, internally remove it from the container.
- Matrix compression: Identify all the zero elements and call delete index function on it.
- Matrix Addition: A template friend function that accepts reference to the input and output matrices, performs matrix addition in O(nnz) time complexity and stores the result in output matrix. Where nnz is number of non-zero elements.
- Matrix Multiplication : Accepts reference to first matrix A(M*N),second matrix B(N*P) and Result C(M*P), performs multiplication in O(P*nnz(A) + M*nnz(B)) time complexity and write the result to reference of output matrix.This operation is constrained only to Map type row container. Where A is the first matrix with dimensions(M*N). Requires additional space complexity of nnz(B).
- Transpose : Returns a transpose of the matrix performed in O(N+nnz) time complexity, where N is number of rows in the matrix.

C. Iterator

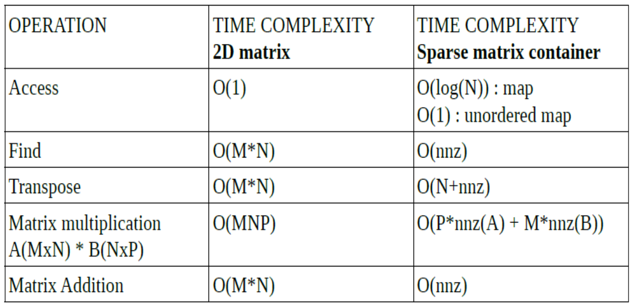
The Container supports Iterator for effective traversal through non-zero elements of the matrix. The Iterator belongs to forward Iterator Category. The Iterator is compliant with standard algorithms find and accumulate which are applicable for a matrix. The Iterator class is implemented as wrapper around the Iterator of the row storage class. It contains as its members integer variable storing row index, a pointer to the sparse matrix it iterates upon and an Iterator belonging to the matrix row storage class.The Iterator performs a row-major traversal over the non zero elements of the matrix. The Iterator supports two constructor overloads.
The Iterator supports the following operations:

- Equality: Accepts an Iterator and returns true if both point to same element in the matrix.
- Inequality: Accepts an Iterator and returns false if both point to same element in the matrix.
- Dereferencing: Returns a reference to the data at current position of the iterator.
- Pre-increment: Returns reference an Iterator to next column index in same row or to first element in next row if it has reached end of row. Returns end Iterator on reaching end of matrix.
- Post-increment: Same as pre-increment, except it returns r-value instead of l-value.

III. Results

Looking with respect to generics point of view the proposed container supports primitive and user-defined types and operations like transpose, addition and find can be performed between any matrices irrespective of container type used to store matrix rows, but when it comes to multiplication only map data type can be supported to store rows, since efficient multiplication requires ordering of elements in the row major form, which cannot be achieved using unordered map.

Very sparse matrices(15% dense) hog up a lot of memory(about 85% is wasted) and take the same amount of time for operations, despite being mostly populated by element that does not contribute to the result. For such matrices, our container takes up only memory for the non-zero elements, and performs faster operations. The container is also suitable for representing graphs in their adjacency list representation. Adjacency matrices are really sparse, and hence to iterating through connected nodes requires checking with every node if an edge exists. This can be overcome with adjacency list representation in the sparse matrix container, as it only iterates



OPERATION	TIME COMPLEXITY **2D matrix**	TIME COMPLEXITY **Sparse matrix container**
Access	O(1)	O(log(N)) : map O(1) : unordered map
Find	O(M*N)	O(nnz)
Transpose	O(M*N)	O(N+nnz)
Matrix multiplication A(MxN) * B(NxP)	O(MNP)	O(P*nnz(A) + M*nnz(B))
Matrix Addition	O(M*N)	O(nnz)

Fig. 2. Time complexity of matrix operations compared to 2-D array

through nodes with edges(non zero weight)which boosts the graph algorithms and needless to say, saves a lot of space.

In terms of performance of matrix operations, the results are not as one would expect in theory. The container is highly variant with the density of the matrix. For a very sparse matrix (for matrix density less than 5%) the container outperforms 2-D array while further increase in density the matrix performance takes a big hit because although the time complexity of this container is better than 2-D array,time taken by each operation is more compared to 2D array and also because the result of product of such sparse matrices isn't a sparse matrix. Additionally the container 2-D array takes advantage of fast indexing and elemental operations and beats the container in denser matrices. Usage of proposed container is best suited for extremely sparse matrices and where there is a need for storage and time efficiency.

IV. Conclusions

Storing sparse matrix as a 2-D array although waste of space in terms of storing null values has an advantage of constant time access and fast elemental operations and is cache-friendly because of continuous memory allocation. The sparse matrix container works well with very sparse matrices because product such matrices remain sparse. If only the non-zero elements of the sparse matrix are stored and a proper storage structure is designed with sufficient support for matrix operation great level of efficiencies both in terms of storage and time can be achieved. The proposed container although processes lesser elements than 2-D matrix, in matrix operations it works better only when matrix is very sparse , for denser matrix time required for individual operations on the elements outweighs the time gained in processing lesser elements.

The container is an effective representation way for adjacency matrix, saving memory besides boosting the algorithms. The sparse matrix can be supported to store data of various types by making use of STL and generic features of C++ such as template classes, template functions, friend functions, etc.

Additional Work

After observing dissatisfactory results using map and unordered map for storing matrix rows, we extended the generics of the container to support a more efficient underlying storage container for row in terms of matrix operations. An array based representation was adopted which was faster than the previous container, yet not faster than a 2D array for higher density. Similar to a coordinate list for storing sparse matrix we store row index, column index and data of non-zero elements separately using vectors. To choose this storage mechanism void data type is to specified in class template during object creation. This new container although being inefficient at insertion, deletion and ordered storage of elements owing to its sequential structure beats the previously proposed container in terms of matrix operations, since constant time element access through indexing adds on as an advantage to the container. It outperforms 2-D till 10% of matrix density, exceeding which the resultant matrix no longer remains sparse and faces same issues as that of previous container.
Similar to previous container an Iterator class is made to support sequential access of the elements of the matrix. Traversal of matrix using the iterator is in the order of insertion of elements.

Acknowledgment

The members of this project would like to thank Prof. Kumar NS for providing us an opportunity to work on this interesting project which helped us in exploring various features of C++ and STL. Through this project we were able to experiment different designs for the Sparse Matrix container by thinking not only in terms of its generic usage but also in terms of efficiency of the related operations in terms of space and time.

References

∗ https://www.cplusplus.com/
∗ https://en.cppreference.com/w/
∗ https://www.wikipedia.org/
∗ https://www.geeksforgeeks.org/