Sparse Matrix Transposition - GPU Computing Final

*https://github.com/Sminnitex/FinalProject_GPUComputing

Michele Minniti
Student ID : 247168
michele.minniti@studenti.unitn.it

***Abstract*—A Sparse Matrix is a Matrix where most elements are zeros, in my previous work I analyzed some basic techniques of matrix transposition using the CPU and the GPU; but this scenario opens to new possible horizons of efficiency. Knowing our matrices will be composed almost everywhere by zeros we can change our way of thinking the transposition trying to implement a method that takes advantage of this information. To perform a transposition everything we need to do is to change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. In fact we don't need to go through every value in memory, but we can just change the coordinates of that values that aren't zero, and by knowing the dimension of the matrix we implicitly know that every other value is indeed zero. In this paper we are going to explore some state of the art methods found in other researches, then we are going to discuss my method and make a comparison with the cuSPARSE library by Nvidia and with other simpler method to perform a normal matrix transpose with the GPU developed during the homeworks of this semester; then we are going to plot the results obtained and make some conclusions about it.**

I. Introduction

In this project, we are going to analyze the problem of sparse matrix transposition. A sparse matrix is a type of matrix where most elements are zeros, and only some of them are non zero values [1].

Considering the work developed in [4], where I explored the efficiency of a classic matrix transposition algorithm, the problem of sparse matrix transposition introduces unique optimization opportunities. In a standard matrix, every element must be processed and stored, leading to significant computational and memory overhead. However, in a sparse matrix, the majority of elements do not need to be considered at all, as they are zeros. So we can build specialized algorithms for handling the non zero elements.

In fact traditional matrix transposition involves swapping the rows and columns of a matrix. For a normal matrix, this operation requires iterating over all elements, which can be computationally expensive for large matrices.

This project aims to compare the performance of different transposition algorithms, including the classic matrix transposition algorithm developed on [4] and the optimized library designed for sparse matrices by NVIDIA, cuSPARSE [2].

II. State of the art

One of the most efficient ways of overcoming this task is to store our sparse matrix in a Compressed Row Storage (CRS), and then perform a transposition passing to a Compressed Column Storage (CCS) [3]; or as they are called in [2], Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). CSR and CSC are two storage techniques to represent sparse matrices, used also by the cuSPARSE library to perform the transposition: basically it consist in dividing our matrix in three vectors;

- **Value Vector**: of type dtype (int or float considering the case of study), where we contain only the non zero values of our sparse matrix;
- **Pointer Vector**: This array stores the pointer in the value vector where each row or column, considering the format chosen, starts. The length of this array is the number of rows or columns in the matrix plus one;
- **Index Vector**: This array stores the index of the other dimension, column or row, in order to triangulate the position of the non zero value contained in the value vector.

As we can understand we know that every element is zero but the non zero values, for which we keep only a pointer and an index that can be exchanged in order to perform the transpose. This is how the cuSPARSE library perform the transposes [2]; therefore our work, considering the state of the art, will be to start from a CSR compression format, and pass to a CSC compression format to handle the transpose in a comparable way to cuSPARSE, but also in what [3] calls one of the most efficient ways possible.

III. My methodology

Considering what I pointed out in the last paragraph, I choose to try and replicate a method to perform a transpose stepping from a CSE format to a CSC format. To handle this I built my algorithm in the following way:

The function at Algorithm 1 calls three different kernels to perform three basic steps:

- **Count Non zeros per column**: The first kernel how many Non zero values there are for column index, as we can see on Algorithm 2.

Algorithm 1 Sparse Matrix Transpose

Require: $rows, cols, nonzeros, csr\_vectors, csc\_vectors$

$NNZPerCol \leftarrow countNNZ(nonzeros, csr\_vectors.colIndex)$
$CSCPointer(cols, csc\_vectors.colPointer, NNZPerCol)$
$CSCVal\&Ind(rows, cols, nonzeros, csc\_vectors, csr\_vectors)$

Algorithm 2 Count NNZ

Require: $nonzeros, csr\_vectors.colInd, nnzPerCol$

$colInd \leftarrow csr\_vectors.colInd$
if $address < nonzeros$ **then**
  $add(nnzPerCol[colInd[address]], 1)$
end if

The chose of an atomic sum for safely increment the value even when multiple threads might attempt to update it simultaneously.

- **Fill CSC Column Pointers**: The second Kernel fills the column pointers array for the CSC format as we can see on algorithm 3.

Algorithm 3 CSC Pointer

Require: $cols, csc\_vectors.colPointer, NNZPerCol$

$colPtr \leftarrow csc\_vectors.colPtr$
if $address == 0$ **then**
  $colPtr[0] \leftarrow 0$
  while $i < cols$ **do**
    $colPtr[i+1] \leftarrow colPtr[i] + NNZPerCol[i]$
  end while
end if

The first element of the Column Pointer array is set to 0, which is the starting index for the first column in the CSC format. Then a loop iterates over each column index from 0 to $n-1$. For each column i, the next element of the pointer is set to the sum of the current element and the count of non zero elements in the current column. This builds the prefix sum (exclusive scan) which determines the starting indices of each column in the CSC format.

- **Fill CSC Values and Row Indexes**: The third Kernel serves the purpose to populate the CSC matrix with the CSR values and indexes, as we can see on algorithm 4. Each thread processes the non zero elements of the row given. The range for the loop is representing the start and end of the non zero elements in the row. Then we perform the transpose passing from the CSR format to the CSC format.

Algorithm 4 CSC Val & Ind

Require: $rows, cols, nonzeros, csc\_vectors, csr\_vectors$

$rowPtr \leftarrow csr\_vectors.rowPtr$
$colInd \leftarrow csr\_vectors.colInd$
$valCsr \leftarrow csr\_vectors.val$
$colPtr \leftarrow csc\_vectors.colPtr$
$rowInd \leftarrow csc\_vectors.rowInd$
$valCsc \leftarrow csc\_vectors.val$
if $address < rows$ **then**
  while $i < rowPtr[end]$ **do**
    $col \leftarrow colInd[i]$
    $dest \leftarrow add(colPtr[col], 1)$
    $valCsc[dest] \leftarrow valCsr[i]$
    $rowInd[dest] \leftarrow address$
  end while
end if

IV. Experiments

Now after a brief description of the system where I run the experiments, and a couple of recommendations to replicate my results (more can be read at the README in my GitHub Repository linked at the beginning of the paper), then we are going to see some results of my simulations.

A. System description

The experiments will take place on the Unitrento Cluster, equipped with an A30 with maximum bandwidth of 870.22GB/s. In particular we will run our jobs using a slurm system with the following characteristics: the partition i used is edu-20h with one node and one task per node 1, I used one GPU A30 and one CPU per task, loading the module cuda 11.8.

B. To reproduce the experiments

On the top of my report I linked my Github repository, and as I say there to run the code is sufficient to follow the instructions of the Readme file. Since the matrices used to benchmark my code are present in the "Dataset" directory, is enough to run the code to obtain the results I'll discuss in this report. The system has of course to be equipped gcc, cmake and the cuda toolkit to run the code in C, and to produce the plots with python3 equipped with matplotlib, pandas and numpy. To have more insight on how the transposition works is possible to use the debug print functions I made on the library.h file, example of usage is already present in the sparse.cu file, commented, all the matrices seems to be transposed correctly with each technique used.

Talking about the benchmark matrices, I took them from https://sparse.tamu.edu/ focusing on matrices filled with float values and with less then 15% of the structure composed by non zero elements. Other matrices can be used to test my code but the results will be discussed on the basis of the ten you can also find on the GIthub repository on the Dataset directory.

C. Plots

Now let's see some results considering the bandwidth as our main performance measure. To calculate the bandwidth

I considered as the only data moved the Non zero elements, therefore I didn't multiplicate the numerator for the whole rows and columns dimension as I did in [4], since the zero elements in the process of transposition are ignored both in My Kernel for Sparse Transposition and in the one of the library cuSPARSE [2]. On top of that, consider the zero elements in the bandwidth calculation could lead to some misleading results since in optimized algorithms as the one produced by Nvidia it inflates the amount of data considered to be moving, even thought the zeros aren't really considered by the technique proposed. Even in normal matrix transpose could lead to some strange results given the possible errors in the use of cache, but even thought I decided to consider only the non zero elements to give a more fair comparison between the techniques and compare the actual workload of all kernels.

The first plot at fig. 1 is divided in two parts, above there is the bandwidth value calculated in respect to the Non Zero values of each sparse matrix, and below there is the plot of Non zero values for each benchmark matrix. For the plot above I decided to put a confrontation between the cuSPARSE library of NVIDIA [2], the global and shared memory kernels used on [4] in order to confront simple transposition techniques with kernel ad hoc for sparse matrices, and with the label "MySparse" my function to manage sparse matrix transposition.

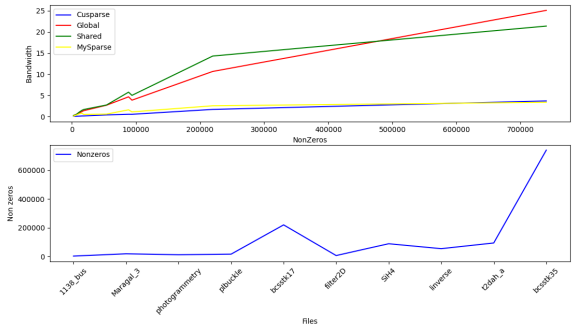

Fig. 1. Plot Of Bandwidth per Non zero values in Matrices

As we could expect, even considering the way I calculated the bandwidth, the Bandwidth calculated in GB/s grows almost linearly with the grows of non zero values. We can also see on the plot below how "bcsstk35", the last benchmark matrix, has way more non zero values than every other matrix.

Now in the second plot fig. 2 we will see the bandwidth value in respect with the dimension of the matrices, so considering also the zero values in the transposition, and below another plot to confront the dimension of each matrix.

As we can see here, considering the bandwidth value in GB/s as before, and the dimensions that has to be multiplied by $1e8$, the bandwidth grows for each technique with dimensions, but not quite as before. In fact we have a significant drop in performance for matrices of dimensions between $1 * 1e8$ and $2 * 1e8$ such as "bcsstk17", "linverse" and "t2dah_a" as can be seen in the plot of reference below. That happens because those are the sparse matrices that in respect with dimensions

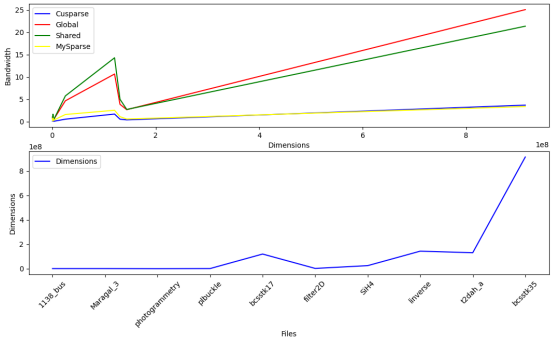

Fig. 2. Plot Of Bandwidth per Dimensions in Matrices

have an higher concentration of non zero values, as we can see in fig. 1. In the more dense matrices we have a significant improve in performance in the classic transpose techniques, and a less important improve also for the sparse techniques.

Then we step to the last plot, where we analyze the bandwidth in respect to each Matrix.

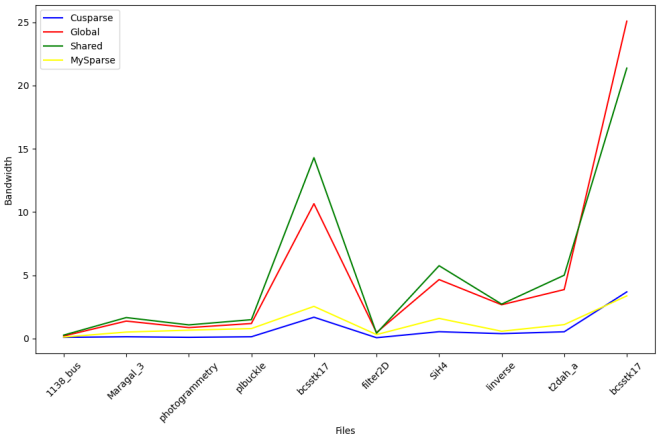

Fig. 3. Plot Of Bandwidth per Matrix

Here we can see clearly how all four techniques follow a general trend in performance in respect to the amount of data they can move, with the two normal transposition techniques that are comparable in performance in the same way the sparse kernels are comparable. One thing that should also be noted, is that if we try, for example, to consider in the calculation of the bandwidth also the zero values we would have a significant improvement in performances. In fact the effective bandwidth value would easily overcame the theoretical maximum of bandwidth of the A30, but it wouldn't be a trustful measure; since of course the sparse kernels aren't even actually dealing with those values but only moving the pointers and indexes [5].

V. Conclusion

All considered we can say that bandwidth grows almost linearly with the number of non-zero elements in the matrix.

Including zero elements in bandwidth calculations shows less consistent performance growth, highlighting the efficiency of sparse-specific techniques. Specialized sparse transposition techniques (cuSPARSE and ”MySparse”) are more efficient for sparse matrices, focusing on non-zero elements and outperforming general transposition techniques in these cases.

The plots showed how the classic techniques have higher bandwidth value but the reason for this difference is about the complexity of the two functions, where the classic only perform a transpose the two methods perform multiple operations, and in case of my function opening multiple kernels with different grid and block size. In fact performing the timer measure only on the third kernel the one which actually perform the transpose the results in bandwidth would be higher even of the cuSPARSE library, but since even the function perform multiple operations it wouldn't make a fair comparison to count only the last step of my function for the plot results. However the two functions are perfectly comparable in each step therefore I found the results acceptable.

REFERENCES

[1] Sudarshan Khasnis, “Operations on Sparse Matrices,” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/operations-sparse-matrices/

[2] NVIDIA, “cuSPARSE library.” [Online]. Available: https://docs.nvidia.com/cuda/cusparse/

[3] “Parallelizing the Sparse Matrix Transposition: Reducing the Programmer Effort Using Transactional Memory,” Procedia Computer Science, vol. 18, pp. 501-510, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050913003578

[4] “Homeworks GPU Computing.” [Online]. Available: https://github.com/Sminnitex/Homework2_GPUComputing

[5] Class Lectures, “GPU Computing.” [Online]. Available: https://didatticaonline.unitn.it/dol/course/view.php?id=38172