Final Project: Sparse Matrix Transposition

Martina Scheffler Matr. 249609
martina.scheffler@studenti.unitn.it
July 14, 2024

***Abstract*—This work shows three different methods for transposing square non-symmetric sparse matrices on the GPU: (a) using the cuSPARSE library, (b) using a CSR-COO-CSR conversion chain with transposition in COO format and (c) an implementation of CSR-CSC conversion which implicitly transposes the matrix. They are executed on the University of Trento's HPC cluster, using ten randomly selected matrices from the SuiteSparse Matrix Collection. The results show that the implementation of the cuSPARSE library is the most efficient, decidedly outperforming the other two methods, of which the CSR-COO-CSR one is slightly faster.**

I. Git and Instructions for Reproducibility

The code discussed in this paper and instructions to run it can be found on GitHub.

II. Introduction

The goal of this work is to design an efficient algorithm to transpose sparse matrices using the GPU. The matrices are considered to be square ($N \times N$) and highly sparse (sparsity $> 75\%$). Sparse matrices are used in a number of applications like network theory [1], numerical analysis or engineering tasks using partial differential equations, like the Finite Element Method [2]. More recently, they have become a topic of interest for machine learning tasks [2].
In the following sections, the problem will be explained in more detail, focusing also on the storage format used for sparse matrices and state-of-the-art solutions for transposing them. Then, multiple algorithms will be proposed and evaluated regarding their efficiency using the metric of the effective bandwidth in comparison to the state-of-the-art.

III. Problem Statement

This section gives a brief introduction to sparse matrix storage formats and the problem of parallelization.

A. Storage Formats

Using matrices with only few non-zero elements, savings in memory requirements can be made by storing only the non-zero elements instead of the whole matrix. The most common storage formats are Coordinate List (COO) [3], Compressed Sparse Row (CSR) [4] and Compressed Sparse Column (CSC) [5]. All of them use three one-dimensional arrays to store the sparse matrix. In COO, the arrays contain the row and column indices and the values. This format is the most intuitive and also the simplest one for modifying the matrix, however both CSR and CSC save more memory [6].
The idea of CSR is that instead of storing the coordinates directly, the row array contains pointers to the first column index of a row instead, decreasing its size from NNZ (the number of non-zero elements) to $N + 1$. The $+1$ comes from storing NNZ as the last value in it as a convention. Similarly, CSC uses an array of size $N + 1$ to store the offsets of the columns. For the rest of this work, the matrix that should be transposed is assumed to be in CSR format.

B. Parallelization

In order to increase the efficiency of the algorithm, the goal is to parallelize the execution as much as possible. When thinking about the most intuitive way to transpose a matrix - simply swapping row and column indices - parallel execution is not a problem. This, however, becomes more difficult when using CSR format, especially in conversions to COO or CSC, since using pointers depending on the number of elements in a row introduces some degree of concurrency when performing operations on it.

IV. State of the Art

The state of the art for transposing sparse matrices on Nvidia GPUs is the *cuSPARSE* library [7]. It offers support for all matrix formats mentioned above and provides different methods to transpose them on the GPU. When only using the CPU, the standard are libraries like Python's *scipy.sparse* [8] or the C++ *Eigen* library [9], however the focus of this work is transposition on the GPU, so these methods will not be discussed in detail.
Since it seems unrealistic that this work will bring major advances in the field, the focus will be more on trying different methods, identifying how they work and finding out how efficient they are.

V. Contribution and Methodology

This section will be divided into two parts: first, the used method from the cuSPARSE library will be shown, then some self-designed CUDA kernels will be presented in the second part.

A. cuSPARSE

When transposing a CSR matrix using cuSPARSE, the recommendation is to use the *cusparseCsr2cscEx2* function which converts the matrix to CSC format [10]. Doing this conversion implicitly returns the transposed matrix in CSR format. Given the array for row offsets, column indices and values, it returns the row indices, column offsets and reordered values. Saving this as CSR, i.e. the row indices as column indices and the column offsets as row offsets, gives the transposed sparse matrix.

Algorithm 1 CSR-COO Conversion

```
1: procedure CSR2COO(r_off_csr, r_ind_coo)
2:     idx ← bIdx.x · bDim.x + tIdx.x
3:     n_row ← 0                          ▷ number of elements in row
4:     while idx < rows do
5:         n_row ← r_off_csr[idx +1] - r_off_csr[idx]
6:         for i=0 to n_row do
7:             r_ind_coo[r_off_csr[idx] + i] ← idx
8:         end for
9:         idx += gDim.x · bDim.x
10:    end while
11: end procedure
```

Algorithm 2 COO Transposition

```
procedure COO_T(r_ind, c_ind, r_ind_tp, c_ind_tp)
    idx ← bIdx.x · bDim.x + tIdx.x
    while idx < nnz do                  ▷ for all non-zero elements
        r_ind_tp[idx] ← c_ind[idx]
        c_ind_tp[idx] ← r_ind[idx]
    end while
    idx += gDim.x · bDim.x
end procedure
```

B. CUDA Kernels

For the self-designed kernels, two different methods will be shown. Firstly, the CSR matrix can be converted to COO format, where it can easily be transposed by swapping the row and column indices for the non-zero elements. Afterwards, it needs to be converted back to CSR, which is the more complicated part. This method requires three different CUDA kernels, (a) CSR-COO conversion, (b) COO transposition and (c) COO-CSR conversion. Of course, the second kernel can also be used to only transpose a COO matrix itself.
Secondly, an algorithm like the one from the library using CSR-CSC conversion for implicit transposition is shown. This requires only a single CUDA kernel.

The used kernels for the CSR-COO-CSR transposition are shown in Algorithm 1, Algorithm 2 and Algorithm 3. The first kernel (Algorithm 1) takes as input the row offsets of the CSR matrix and an array to store the row indices of the COO matrix. For each row, the number of elements in it is determined by subtracting two consecutive values in the offset array. Then, the row index is inserted the correct amount of times into the array for the COO indices.
The second kernel (Algorithm 2) does the actual transposition which is straight-forward in COO format. It takes as input the arrays for the row and column indices of the matrix and two equally sized arrays to store the transposed version. Then for each non-zero element, the row and column index are swapped and saved to the new arrays.
Finally, conversion back to CSR format is done using Algorithm 3, the most complicated kernel. It requires two synchronization barriers between the threads in order to achieve correctness, limiting parallelization. Inside the kernel, first the number of values per row is counted by looping over the row

Algorithm 3 COO-CSR Conversion

```
1: procedure COO2CSR(r_ind_coo, c_ind_coo, val_coo,
   r_off_csr, c_ind_csr, val_csr)
2:     idx_org ← bIdx.x · bDim.x + tIdx.x    ▷ original idx
3:     idx ← idx_org
4:     offset ← gDim.x · bDim.x              ▷ stride for threads
5:
6:     while idx < rows do                   ▷ count # values in rows
7:         n_rows[idx] ← 0   ▷ number of elements per row
8:         for i=0 to nnz do
9:             if r_ind_coo[i] == idx then
10:                n_rows[idx]++
11:            end if
12:        end for
13:        idx += offset
14:    end while
15:    __syncthreads()                       ▷ wait for other threads
16:
17:    idx ← idx_org
18:    while idx < rows do                   ▷ sum them up
19:        r_off_csr[idx+1] ← 0
20:        for i=0 to idx+1 do
21:            r_off_csr[idx+1] += n_rows[i]
22:        end for
23:        idx += offset
24:    end while
25:    __syncthreads()                       ▷ wait for other threads
26:
27:    idx ← idx_org
28:    while idx < rows do                   ▷ place indices and values
29:        s_rows[idx] ← 0   ▷ already saved values per row
30:        for i=0 to nnz do
31:            if r_ind_coo[i] == idx then
32:                nnz_idx ← r_off_csr[idx] + s_rows[idx]
33:                c_ind_csr[nnz_idx] ← c_ind_coo[i]
34:                val_csr[nnz_idx] ← val_coo[i]
35:                s_rows[idx]++
36:            end if
37:        end for
38:        idx += offset
39:    end while
40: end procedure
```

indices of the COO array. After this is done, the row offset array of the CSR matrix can be determined by summing up the number of elements in all previous rows for each row. In the end, the positions of the column indices and the values in the transposed CSR matrix need to be determined. For each row, a loop over the non-zero elements is executed. If the row index at the current position equals the current row, the position of the column index and values is looked up. To avoid always overwriting the value at the position that the CSR row offset points to, a counter of saved values is increased on saving.

For transposition by CSR-CSC conversion, the used kernel is shown in Algorithm 4. It is similar to the COO-CSR conversion, except for the last block. First, the column offsets

Algorithm 4 CSR-CSC Conversion

```
procedure CSR2CSC(r_off_csr, c_ind_csr, v_csr, r_ind_csc, c_off_csc, v_csc)
    idx_org ← bIdx.x · bDim.x + tIdx.x        ▷ original idx
    idx ← idx_org
    offset ← gDim.x · bDim.x                  ▷ stride for threads

    while idx < columns do  ▷ count # values in columns
        n_cols[idx] ← 0 ▷ number of elements per column
        for i=0 to nnz do
            if c_ind_csr[i] == idx then
                n_cols[idx]++
            end if
        end for
        idx += offset
    end while
    __syncthreads()                           ▷ wait for other threads

    idx ← idx_org
    while idx < columns do                    ▷ find column offsets
        c_off_csc[idx+1] ← 0
        for i=0 to idx+1 do
            c_off_csc[idx+1] += n_cols[i]
        end for
        idx += offset
    end while
    __syncthreads()                           ▷ wait for other threads

    idx ← idx_org
    n_vals ← 0                                ▷ number of elements in row
    while idx < columns do  ▷ place row ind. and values
        s_cols[idx] ← 0 ▷ already saved values per column
        for i=0 to rows do
            n_vals ← r_off_csr[i+1] - r_off_csr[i]
            for j=0 to n_vals do
                if c_ind_csr[r_off_csr[i] + j] == idx then
                    nnz_idx ← c_off_csc[idx] + s_cols[idx]
                    r_ind_csc[nnz_idx] ← i
                    v_csc[nnz_idx] ← v_csr[r_off_csr[i] + j]
                    s_cols[idx]++
                end if
            end for
        end for
        idx += offset
    end while
end procedure
```

are determined by looping over the non-zero elements and counting the numbers in each column and then summing them up in a second step. After this is done, the most complicated part is placing the row indices and values correctly. For this, a loop over all rows is necessary for each column. In it, the number of elements per column is calculated and then looped over. A row index corresponding to the current value in the outer loop is placed at the column offset for this column. To avoid always overwriting this value, an offset is added by counting the number of already saved values per column. The value is placed at the same index as the row index and extracted from the CSR value array using the row offset for the current row plus the iteration variable of the inner loop over the number of elements in a row. During the execution, threads have to be synchronized two times because their next steps depend on previously determined values of the other threads.

TABLE I
HARDWARE AND SOFTWARE DESCRIPTION

Name	Version	Documentation
GPU	Nvida A30	[11]
CPU	Intel Xeon Gold 6238R	[12]
CUDA	12.1	[13]
C++	14	[14]
gcc	8.5.0	[15]

VI. System Description and Experiments

This section gives a description of the used hardware and explains the strategy behind the experiments that were done.

A. Platform Description

All experiments were run on the University of Trento's computing cluster. It is equipped with Nvidia A30 GPUs. The maximum bandwidth of the A30 can be computed as

$$\frac{1215 \cdot \frac{3072}{8} \cdot 2}{10^9} \approx 933\text{GB/s}, \tag{1}$$

and will later be used to analyze the performance of the different transposition methods. The versions of hardware, used compile chain and libraries are shown in Table I. For all experiments, compilation was done using the `-O2` flag of gcc.

B. Experiments

To evaluate the performance of the cuSPARSE CSR-CSC transposition in comparison to the two self-developed methods shown in Section V-B, ten test matrices from the SuiteSparse Matrix Collection [16] were used. They were selected randomly, but are all square and non-symmetric. Before using them for the tests, they were converted to CSR format.
During testing, the execution times of the library function and the kernels were recorded using CUDA events and averaging over 300 runs to compensate for any initial overhead and possible fluctuations. The measured times plus information about matrix size and number of non-zero elements was then saved and evaluated. The results of the experiments can be seen in Section VII.

VII. Results and Discussion

The resulting execution times are shown in Fig. 1. It can be seen that the execution time for the cuSPARSE algorithm stays almost constant no matter which matrix is used, while the self-developed methods depend heavily on the size of the matrices. For this dataset, it can also be seen that the CSR-COO-CSR

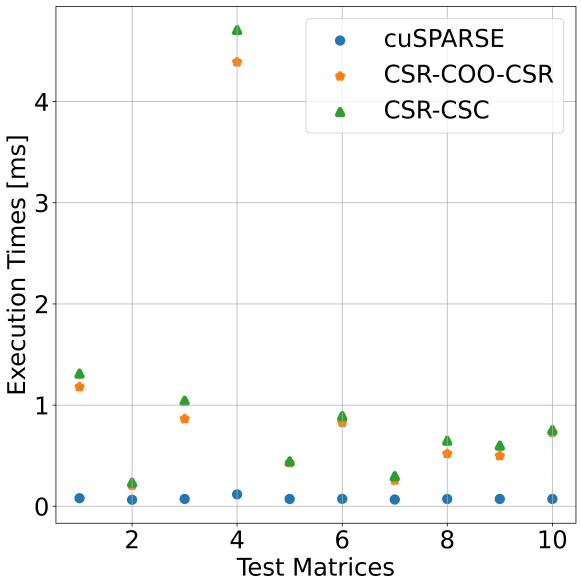

Fig. 1. Execution Times for Test Matrices averaged over 300 Runs

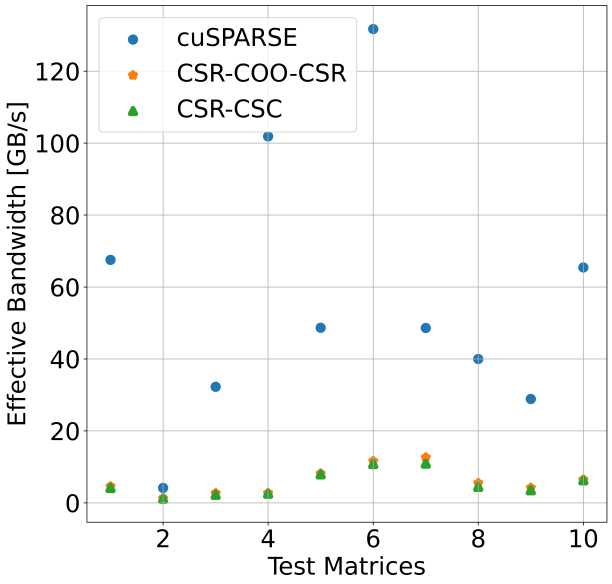

Fig. 2. Effective Bandwidths for Test Matrices averaged over 300 Runs

method slightly outperforms the CSR-CSC method. Looking at the effective bandwidth

$$EBW = \frac{(B_r[Bytes] + B_w[Bytes])/10^9}{t_{\text{routine}}[s]}, \qquad (2)$$

with $B_r + B_w = N \cdot N \cdot 4 \cdot 2$ as the number of bytes, the results are shown in Fig. 2. Here again, the algorithm of the library massively outperforms the other two, with CSR-COO-CSC still being the better one out of the two. This can also be seen in Table II where the maximum effective bandwidths reached on the ten test matrices are shown for the three different strategies. It can be seen that even the cuSPARSE function only reaches 14% of the maximum bandwidth calculated in Eq. (1), however the other two methods perform even worse, having effective bandwidths roughly 10 times lower.

TABLE II
MAX. EFFECTIVE BANDWIDTHS FOR THE TEST MATRICES

Algorithm	Max. EBW	% of Max. GPU BW
cuSPARSE	131.71 GB/s	14.12%
CSR-COO-CSR	12.66 GB/s	1.36%
CSR-CSC	10.92 GB/s	1.17%

VIII. CONCLUSION

As expected, it can be seen that the most efficient way to perform the sparse matrix transposition of a CSR matrix on the GPU is to use the cuSPARSE library. However, the two self-implemented algorithms provide some insight into the complexity of the problems and the CSR-CSC version shows the basic algorithm also used in the library. Another finding to be gained by this is that since the CSR-COO-CSR version outperforms the CSR-CSC algorithm, but the cuSPARSE function using CSR-CSC converion is much faster, the implementation shown here has a lot of potential for improvement.
In a future work, these inefficiencies should be identified and improved in order to increase the level of parallelization. Further, a comparison against a CPU-based approach could be interesting to see at which point using the GPU brings the most benefit.

REFERENCES

[1] S. Kumbha. “Leveraging Sparse Matrices in Social Networks: Optimizing Performance and Efficiency.” (2024), [Online]. Available: https://medium.com/@sanket.kumbhar21/leveraging-sparse-matrices-in-social-networks-optimizing-performance-and-efficiency-52f28d9c5cf3 (visited on 07/13/2024).

[2] F. Lagunas. “Is the future of Neural Networks Sparse? An Introduction.” (2020), [Online]. Available: https://medium.com/huggingface/is-the-future-of-neural-networks-sparse-an-introduction-1-n-d03923ecbd70 (visited on 07/13/2024).

[3] Nvidia. “cuSPARSE. Matrix Formats. Coordinate (COO).” (2023), [Online]. Available: https://docs.nvidia.com/cuda/cusparse/#coordinate-coo (visited on 07/08/2024).

[4] Nvidia. “cuSPARSE. Matrix Formats. Compressed Sparse Row (CSR).” (2023), [Online]. Available: https://docs.nvidia.com/cuda/cusparse/#compressed-sparse-row-csr (visited on 07/08/2024).

[5] Nvidia. “cuSPARSE. Matrix Formats. Compressed Sparse Column (CSC).” (2023), [Online]. Available: https://docs.nvidia.com/cuda/cusparse/#compressed-sparse-column-csc (visited on 07/08/2024).

[6] Y. Kerdcharoen. “Compressed Sparse Row (CSR) — Introduction and Explanation.” (2021), [Online]. Available: https://pnxguide.medium.com/compressed-sparse-row-motivation-and-explanation-cd92c71b7cfa (visited on 07/06/2024).

[7] Nvidia. “cuSPARSE.” (2023), [Online]. Available: https://docs.nvidia.com/cuda/cusparse/ (visited on 07/13/2024).

[8] Scipy. “Sparse Matrices (scipy.sparse).” (2024), [Online]. Available: https://docs.scipy.org/doc/scipy/reference/sparse.html (visited on 07/13/2024).

[9] Eigen. “Sparse Matrix Manipulations.” (2022), [Online]. Available: https://eigen.tuxfamily.org/dox/group__TutorialSparse.html (visited on 07/13/2024).

[10] Nvidia. “cusparseCsr2cscEx2.” (2023), [Online]. Available: https://docs.nvidia.com/cuda/cusparse/#cusparsecsr2cscex2 (visited on 07/13/2024).

[11] Nvidia. “Nvidia A30 Tensor Core GPU.” (2022), [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf (visited on 07/12/2024).

[12] Intel. “Intel Xeon Gold 6238R Processor.” (2024), [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/199345/intel-xeon-gold-6238r-processor-38-5m-cache-2-20-ghz/specifications.html (visited on 07/12/2024).

[13] Nvidia. “CUDA Toolkit Documentation 12.1 Update 1.” (2023), [Online]. Available: https://docs.nvidia.com/cuda/archive/12.1.1/ (visited on 07/12/2024).

[14] cppreference. “C++14.” (2014), [Online]. Available: https://en.cppreference.com/w/cpp/14 (visited on 07/12/2024).

[15] Free Software Foundation Inc. “gcc-8.5.0 Documentation.” (2018), [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-8.5.0/gcc.pdf (visited on 07/12/2024).

[16] Davis, T. and Hu, Y., “The University of Florida Sparse Matrix Collection,” *Transactions on Mathematical Software*, vol. 38, 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663 (visited on 07/12/2024).