FINAL REPORT

SIKAI CHENG & PRISCILLA ZHANG

April 16, 2023

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

General Matrix-Matrix Multiplication (GEMM), the most fundamental and crucial linear operation, is widely used in all aspects of computational science. In most cases, GEMM is trivial and straightforward. It calculates the resulting matrix C ($\in \mathbb{Q}^{n\times k}$) based on two input matrices, A ($\in \mathbb{Q}^{n\times m}$) and B ($\in \mathbb{Q}^{m\times k}$), where $c_{ij} = \sum_l a_{il} b_{lj}$. However, in several particular scientific computing scenarios and combinatorial, such as multi-source breadth-first searching (Then et al., 2014), algebraic multigrid solvers (Bell et al., 2012), triangle counting (Davis, 2018), the shortest path finding (Chan, 2007), and colored intersecting (Kaplan et al., 2006), matrices A and B will contain ample zeros. These zeros and corresponding numerous computations of zero entries during the multiplication result in a high cost of both storage and operations. Therefore, SpGEMM (General Sparse Matrix-Matrix Multiplication) was formally proposed, quickly attracted the attention of many researchers, and became a standard computational block in these fields. In contrast to GEMM, SpGEMM requires exploiting the sparsity pattern in the matrix and obtaining a computational performance enhancement.

As opposed to traditional CPUs, which are tailored for optimizing the performance of sequential tasks in a single thread and minimizing the latency, modern GPUs contain massive parallelism, with dozens of multiprocessors each is capable of executing hundreds of hardware-scheduled threads and emphasizes total task throughput (Garland & Kirk, 2010). These properties equip GPUs with higher peak floating-point operations and memory bandwidth than CPUs, allowing them to impart tremendous performance in many high-performance computing applications, including SpGEMM definitely. However, realizing the full potential of GPUs for high-performance computing is not trivial. For multiple high-performance computing concerns, Dalton et al. (2015) underscore that the pervasive problem is that the existing methods, even those proven suitable for multicore CPUs, are not immediately applicable to the GPU. Meanwhile, they accentuate that effective use of GPUs requires substantial fine-grained parallelism at all stages of the computation. More specifically, for SpGEMM, critics have also argued that the superiority of current methods cannot escape from the dependence on the particular sparsity pattern of the matrix. Liu and Vinter (2014) claim that the methods either only work best for fairly regular sparse matrices (with most of the nonzero entries on the diagonal) or bring extra high memory overhead for matrices with some specific sparsity structures. Gremse et al. (2015) obtained a similar conclusion.

2 LITERATURE REVIEW

Therefore, the desire for a fast and general SpGEMM implementation has driven researchers to explore various matrix formats and computing algorithms. Some of the sections associated with this paper will be reviewed in the following.

2.1 Different Matrix Format

A natural idea, which is intuitively considered to have a direct effect on reducing the storage of sparse matrices, is to change the form of the matrices, in particular, to omit a large number of zeros from the matrix. Meanwhile, almost all previous explorations prove that it significantly impacts the algorithms' performance. Therefore, researchers have identified various operative formats for storing sparse matrices. One such format is the Coordinate list (COO), which involves storing the data in a tuple consisting of row index, column index, and entry value. The COO format is intuitive and does not depend on the number of zero values in a row. However, it generally performs worse in the SpGEMM problem compared to other formats (Li et al., 2015). The Compressed Sparse Row (CSR) format is another commonly used format, Fig. 1, which consists of three arrays. The first array stores the entry values, the second array stores the corresponding column indices, and the third array records the index where a new row begins.

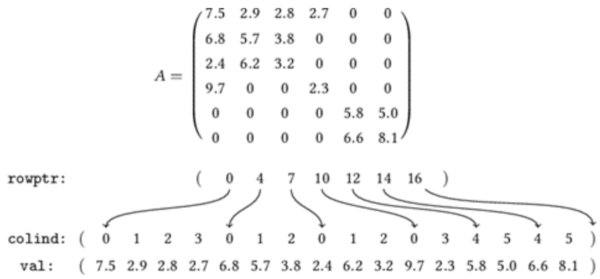


Figure 1: Example: The Compressed Sparse Row (CSR) format.

One significant observation is that the *rowptr* array is typically much smaller in size than the *colind* and *val* arrays, indicating that CSR format requires less storage space for matrix representation, particularly when dealing with large sparse matrices. Nonetheless, it should be noted that the CSR format's performance is highly dependent on the frequency of zero entries in a row. In addition to the commonly used COO and CSR formats, there exist other matrix formats, such as the ELLPACK (ELL) and hybrid (HYB) formats. Although these formats are not utilized in our study, they can be efficient options when the number of non-zero and zero elements are comparable.

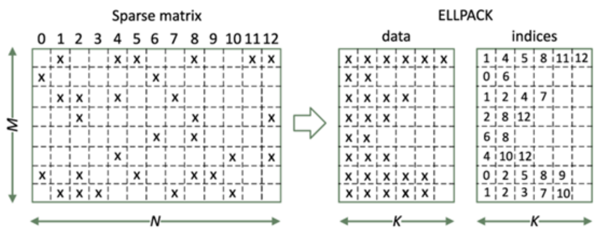


Figure 2: Example: The ELLPACK (ELL) format.

2.2 Row Merging

In addition to the ESC algorithm, alternative methodologies have been investigated in our literature review, among which is the iterative row merging algorithm using the CSR format (Gremse et al., 2015). In the context of matrix multiplication, namely $C = AB$, a possible approach involves translating the problem into $c = aB$, where c and a represent the corresponding rows of matrices C and A, respectively. Subsequently, c can be expressed as a linear combination of columns in B, with the weights determined by the entries in a. As a result, a row merging operation can be employed to compute each row in matrix C, as depicted in the Fig. 3. For the sake of simplicity, we assume the values of B and a to be 2 and 0.5, respectively. During each iteration of the row merging operation, the values with column indices are added with respect to the values of a, leading to the merging of two rows into a single row with a sorted column index. This process reduces the number of rows while potentially increasing the row size, depending on the frequency of identical indices present in the original rows.

In the case of computing $C = AA$, the authors assert that the row merging algorithm exhibits superior performance compared to other SpGEMM algorithms, such as MLK, Cusp, and Cusparse, in the Fig. 4, especially when working with large sparse matrices (Gremse et al., 2015). It is worth noting that the authors of the

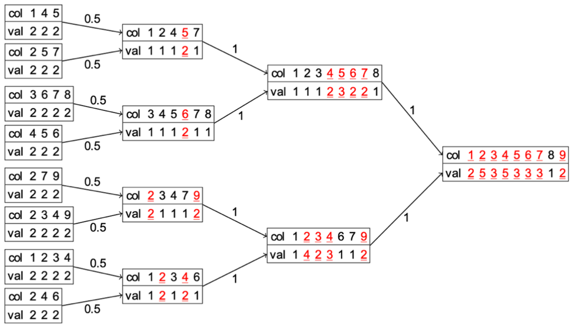


Figure 3: Example: The Row Merging Operation.

paper we are reproducing developed Cusp, a SpGEMM library. Subsequently, they made improvements to Cusp, which were published around the same time as the row merging algorithm. However, no additional comparison or analysis has been conducted to compare the performance of these two algorithms.

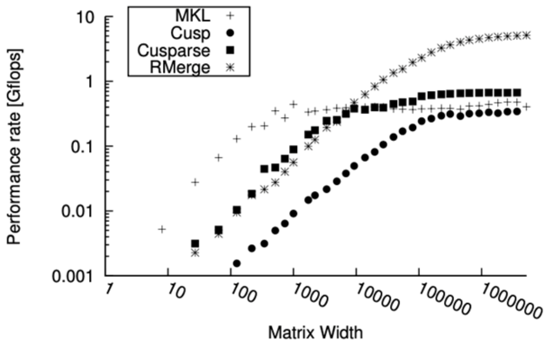


Figure 4: Comparison: The performance of MKL, Cusp, Cusparse, and RMerge.

2.3 Layered Graph Model

In the Expansion phase of the ESC algorithm, the authors utilize a geometric formulation of SpGEMM known as the Layered Graph Model (Cohen, 1996). This model employs a bipartite graph structure, where the vertices represent the rows and columns of the matrix. In the case of SpGEMM, where $C = AB$, the Layered Graph Model considers the rows of A as the base level vertices and the columns of A as the second level vertices. Notably, the number of columns in A is identical to the number of rows in B, which is crucial for leveraging this model in SpGEMM computations. The model then introduces a third layer of vertices representing the columns of B. In the Layered Graph Model, for each non-zero entry of matrix A with row index i, column index j, and entry value v, an edge is created between the i-th vertex of the base layer and the j-th vertex of the second layer with a weight of v. Similarly, for every nonzero entry of matrix B, a weighted edge is established between a second-layer vertex and a third-layer vertex. A visualization of such a model is illustrated in the Fig. 5 below as an example.

One of the primary benefits of the Layered Graph Model for SpGEMM problems is that it facilitates the formulation of each non-zero element in the resulting matrix C as a set of paths from a base-layer vertex to a top-layer vertex. More specifically, a non-zero entry of C with row index i, column index j, and entry value v, can be represented by the ensemble of paths from the i-th vertex of the base layer to the j-th vertex of the top layer, where the sum of the edge weights along all feasible paths is equivalent to v, Fig. 6.

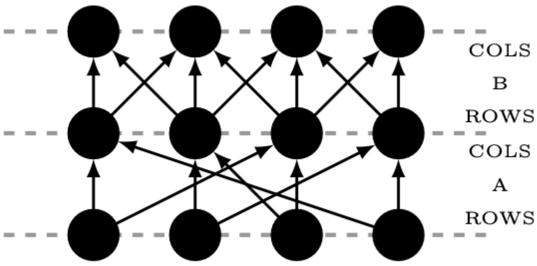


Figure 5: Example: A visualization of the Layered Graph Model.

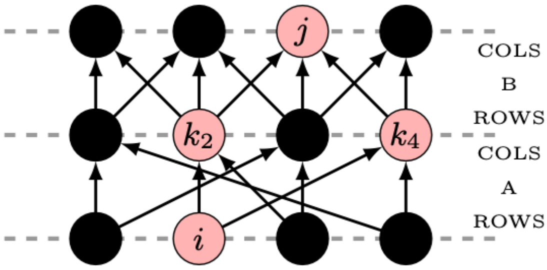


Figure 6: Example: Represent a non-zero entry of C by the ensemble of paths.

2.4 Sparse Matrix-Vector Multiplication

It is plausible that Sparse Matrix-Vector Multiplication (SpMV) is substantially similar to SpGEMM. Thus, the near-optimal algorithm for SpMV (Choi et al., 2010) gives the impression that a broad-spectrum and speedy algorithm for SpGEMM is close to being found. However, the study of Williams et al. (2009) points out that SpMV and SpGEMM are qualitatively different problems. Dalton et al. (2015) support the idea and explain it in detail. A common strategy for improving SpMV performance is to exploit a priori knowledge of the sparsity structure of the matrix in order to minimize expensive off-chip memory operations. This approach is considerably profitable when computing specific iterative algorithms that use the same sparse matrix multiple times or when the number of SpMV operations is sizeable. However, this approach is not entirely appropriate for SpGEMM since the interaction of sparsity from two matrices complicates the situation, and the SpGEMM is generally a fleeting operation, meaning that they are called at most once for a given set of matrices in most applications.

2.5 ESC Algorithm

The method proposed by Dalton et al., which our team wants to implement in the project, is strictly based on the Expansion, Sorting, and Compression (ESC) algorithms and consists of many finely designed bandwidths optimizing operations. The high-level structure of the ESC algorithm includes three steps:

- *Expansion* of AB into a intermediate coordinate format T.
- *Sorting* of T by row and column indices to form $\hat{T}$.
- *Compression* of $\hat{T}$ by summing duplicate values for each matrix entry.

Dalton et al. (2015) analyzed the algorithm's performance, as shown in Fig. 10, and accordingly optimized each part, especially the sorting phase. In the Setup phase, the researchers recognized that the static assignment of computational units

$$\mathtt{A} = \begin{bmatrix} 5 & 10 & 0 \\ 15 & 0 & 20 \end{bmatrix} \quad \text{and} \quad \mathtt{B} = \begin{bmatrix} 25 & 0 & 30 \\ 0 & 35 & 40 \\ 45 & 0 & 50 \end{bmatrix}$$

(a) Matrix Multiplication

$$\mathtt{T} = \begin{bmatrix} (0,0, & 125) \\ (0,2, & 150) \\ (0,1, & 350) \\ (0,2, & 400) \\ (1,0, & 375) \\ (1,2, & 450) \\ (1,0, & 900) \\ (1,2, & 1000) \end{bmatrix}$$

(b) Expansion

Figure 7: Example: Expansion

$$\hat{\mathtt{T}} = \begin{bmatrix} (0,0, & 125) \\ (0,1, & 350) \\ (0,2, & 150) \\ (0,2, & 400) \\ (1,0, & 375) \\ (1,0, & 900) \\ (1,2, & 450) \\ (1,2, & 1000) \end{bmatrix}$$

Figure 8: Example: Sorting

to process a fixed number of rows of matrix A would cause an acute load imbalance, closely related to poor performance. So, they permute the rows of matrix A to guarantee that the adjacent threads can handle rows with similar workloads. In the expansion phase, they adopt a formulation of SpGEMM as a layered graph model and employ the parallel bread-first-search to compute it. In the sorting and compression phase, they conclude that the parallel primitives used in the previous ESC algorithm, although general and avoiding excessive imbalance, force many data operations in global memory, inevitably leading to serious inefficiencies. Therefore, they focus on sorting and compressing within the GPU's higher-bandwidth shared memory for increased efficiency.

3 ALGORITHM & IMPLEMENTATION

In this section, we will explain in detail the four critical parts of the algorithm, attach their implementation, and explain the code. Before starting the explanation, I want to declare the following three names:

- **The ESC algorithm**: The sparse matrix-matrix multiplication algorithm proposed in *EXPOSING FINE-GRAINED PARALLELISM IN ALGEBRAIC MULTIGRID METHODS*. It is implemented by combining multiple fine-grained parallel primitives in the Thrust library, most of which are optimized at the global memory level.

- **The Modified ESC algorithm**: The modified version of the ESC algorithm proposed in *Optimizing Sparse Matrix-Matrix Multiplication for the GPU*. In addition, it makes full use of shared memory to improve the efficiency of sorting and contraction. This is the algorithm we want to reproduce.

- **The Referenced paper**: *Optimizing Sparse Matrix-Matrix Multiplication for the GPU*, the article we want to reproduce.

Two other points to note are that in the description of the referenced article, the authors repeatedly use functions from the open-source libraries Thrust and CUB. In our implementation, we avoid using existing functions except where explicitly

$$
\mathsf{C} = \begin{bmatrix} (0,0,\ \ 125) \\ (0,1,\ \ 350) \\ (0,2,\ \ 550) \\ (1,0,1275) \\ (1,2,1450) \end{bmatrix}
$$

Figure 9: Example: Compression

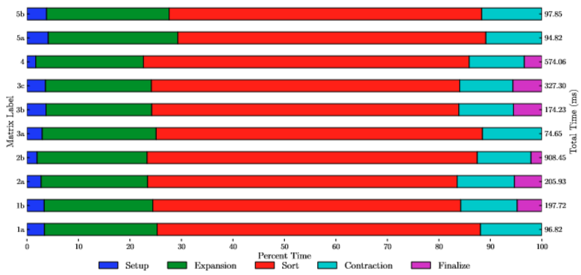


Figure 10: Performance Analysis: The ESC Algorithm

stated in the article. In addition, the referenced article considers that each part of expansion, sorting, and contraction may be applied separately to other algorithms, and therefore their implementations are independent of each other. In our implementation, I have integrated them into the **expansionSortingContractionKernel**, Fig. 20.

To plainly demonstrate the algorithm, we show the full intermediate process of multiplying two 7×7 sparse matrices as an example, Fig. 11.

```
 --- PRINT THE MATRIX ----
  0  0  0  0  0  0  0...
  8  0  0  0  0  0  0...
  0  0  6  0  1  6  0...
  2  0  0  0  0  0  0...
  0  0  0  0  0  0  0...
  0  0 10  0  0  0  0...
  0  0  3  0  0  9  0...
...
 --- PRINT THE MATRIX ----
  0  0  0  0  7  9  0...
  0  0  5  0  0  0  9...
  0  3  0  0  0  0  0...
  5  0  0  0  0  0  0...
  7  0  0  0  2  4  0...
  8  3  0  0  0  0 10...
  6  7  9  0  9  0  0...
...
 --- PRINT THE CSR FORMAT MATRIX ----
ptr -    +0   +0   +1   +4   +5   +5   +6   +8
indices -    +0   +2   +4   +5   +0   +2   +2   +5
data - +7.682296+6.357117+1.416026+6.069689+2.428868+9.727750+2.833148+9.190265
 --- PRINT THE CSR FORMAT MATRIX ----
ptr -    +0   +2   +4   +5   +6   +9  +12  +16
indices -    +4   +5   +2   +6   +1   +0   +0   +4   +5   +0   +1   +6   +0   +1   +2   +4
data - +6.632269+8.902326+4.577017+9.022080+2.666657+5.316064+7.385343+1.659742+4.401045+8.292011+3.303371+9.564683+5.886401+6.573040+8.586763+9.239697
 --- SET UP ---
original counting: 0 2 7 2 0 1 4
 --- AFTER REORDER - THRUST VECTOR
d_counting_vec:  7 4 2 2 1 0 0
d_order_vec:  2 6 1 3 5 0 4
d_operations_vec:  0 7 11 13 15 16 16 16
 --- AFTER REORDER - CONVERT TO ptr
order: 2 6 1 3 5 0 4
counting: 7 4 2 2 1 0 0
operations: 0 7 11 13 15 16 16 16
 --- AFTER EXPANSION and BEFORE SORTING
phase1_row: 2 2 2 2 2 2 2 6 6 6 6 1 1 3 3 5
phase1_col: 1 0 0 4 1 5 6 1 0 1 6 4 5 4 5 1
phase1_val: 12 7 48 1 18 4 54 4 72 27 81 42 56 12 16 18
 --- AFTER SORTING and BEFORE CONTRACTION
phase2_row: 2 2 2 2 2 2 2 6 6 6 6 1 1 3 3 5
phase2_col: 0 0 1 1 4 5 6 0 1 1 6 4 5 4 5 1
phase2_val: 7 48 12 18 1 4 54 72 4 27 81 42 56 12 16 18
 --- FINALLY ---
resulting_row: 2 0 2 0 2 2 2 6 6 0 6 1 1 3 3 5
resulting_col: 0 0 1 0 4 5 6 0 1 0 6 4 5 4 5 1
resulting_val: 55 0 30 0 1 4 54 72 31 0 81 42 56 12 16 18
--- COO FORMAT ---
coo_row:  2 2 2 2 2 6 6 6 1 1 3 3 5
coo_col:  0 1 4 5 6 0 1 6 4 5 4 5 1
coo_val:  55.0 30.0 1.0 4.0 54.0 72.0 31.0 81.0 42.0 56.0 12.0 16.0 18.0
```

Figure 11: Example: Sparse matrix-matrix multiplication by using the modified ESC algorithm with all intermediate results.

3.1 Setup

The primary objective during the setup phase of the modified ESC algorithm is to determine the optimal allocation of threads and memory per row for the output matrix C. The most naïve approach is to assign a uniform number of threads and memory per row. Nevertheless, this method is not suitable for sparse matrices since the number of operations needed in each row may vary significantly. Therefore, if a fixed number of threads and memory is allocated to all rows, many rows with high sparsity will waste a significant amount of resources, which is referred to as load imbalance.

Prior studies have examined various methods to mitigate load imbalances, but these methods have generally required considerable data movement between stages or some specific shape of nonzero entries in the sparse matrix. Therefore, in the referenced paper, the authors have proposed a novel approach based on reordering the A's rows with the C's rows based on the amount of computational work in the model. It is noteworthy that the minimum number of floating-point operations (FLOPs) associated with forming C_{row_i} is directly proportional to

$$\sum_{j \in NNZ(A_{row_i})} \text{nnz}\left(B_{row_j}\right) \tag{1}$$

By sorting the amount of computational work required per row in a non-decreasing order, a permutation matrix P can be derived for C, such that $PC = PAB$. This implies that the rows of matrix A can be processed in a permuted order. Utilizing this permutation technique makes it possible to group together rows that necessitate comparable computational work and allocate the appropriate amount of resources to each group. Nonetheless, the authors acknowledged that this approach has a disadvantage: the requirement of sorting the final output matrix back to its original order. However, the reassembly cost associated with this sorting process is typically low and can be deemed acceptable in most cases.

parameters: A, B $\{A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}\}$
return: P {reordering vector}

$F_i = 0$ for $i = 1$ to m
foreach row i in C **do**
  for $j \in NNZ(A_{row_i})$
    $F_i \leftarrow F_i + \texttt{nnz}(B_{\texttt{row}_j})$ {gather B row lengths based on A column indices}
$P \leftarrow \texttt{sort}(F)$ {set P to permutation of F in non-decreasing order}

Figure 12: Algorithm: Setup Phase

The codes used in the setup phase are shown in Fig. 13 & 14. First, I implemented a **coutingKernel** to calculate the number of operations needed for each row of C, where one thread corresponds to one row. Then, **thrust::stable_sort_by_key** (line 15) is used to sort the number of operations obtained in the previous step. And **thrust::exclusive_scan** (line 17) is used to calculate the prefix sum of the sorted array, which will help to allocate a reasonable amount of memory for each row in the subsequent operations. The results of the setup phase are rendered in Fig. 11 line 27-36.

3.2 Expansion

The essential aim of the expansion phase is to perform computations on the rows of matrix B to produce an intermediate buffer, denoted as $\hat{C}$, which is an expanded and scaled version of B. A representation of such an expansion can be seen in the accompanying figure, Fig. 15, where B_{row_k} is scaled by A_{ik}. This scaled expansion enables the computation of $\hat{C}$, which is similar to the COO format and consists of row indices $\hat{I}$, column indices $\hat{J}$, and values $\hat{V} = A_{i,k} * B_{k,j}$. In the literature review section, we introduced the layered-graph model, which was utilized by the authors during the expansion phase. An example of this model is presented in Fig. 5. As discussed previously, this model effectively demonstrates the formation of every non-zero entry of the final output matrix C as a group of paths from the initial

```
__global__
void countingKernel(int n, int* d_counting, int* A_ptr, int* A_ind, int* B_ptr) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    __syncthreads();
    if (idx < n) {
        int start = A_ptr[idx];
        int end = A_ptr[idx + 1];
        for (int i = start; i < end; i++) {
            int j = A_ind[i];
            d_counting[idx] = d_counting[idx] + B_ptr[j + 1] - B_ptr[j];
        }
    }
    __syncthreads();
}
```

Figure 13: Code: Counting Kernel

```
int* counting = (int*)malloc(A_row * sizeof(int));
int* d_counting;
cudaMalloc(&d_counting, A_row * sizeof(int));
cudaMemset(d_counting, 0, A_row * sizeof(int));
int set_up_n_blocks = (A_row + N_THREADS_PER_BLOCK - 1) / N_THREADS_PER_BLOCK;

countingKernel <<< set_up_n_blocks, N_THREADS_PER_BLOCK >>> (A_row, d_counting, d_A_ptr, d_A_ind, d_B_ptr);
cudaDeviceSynchronize();
cudaMemcpy(counting, d_counting, A_row * sizeof(int), cudaMemcpyDeviceToHost);

thrust::device_ptr<int> d_counting_ptr(d_counting);
thrust::device_vector<int> d_counting_vec(d_counting_ptr, d_counting_ptr + A_row);
thrust::device_vector<int> d_order_vec(A_row);
thrust::sequence(d_order_vec.begin(), d_order_vec.end());
thrust::stable_sort_by_key(d_counting_vec.begin(), d_counting_vec.end(), d_order_vec.begin(), thrust::greater<int>());
thrust::device_vector<int> d_operations_vec(A_row + 1);
thrust::exclusive_scan(d_counting_vec.begin(), d_counting_vec.end(), d_operations_vec.begin());
int tot_operations = d_counting_vec.back() + d_operations_vec[A_row - 1];
d_operations_vec[A_row] = tot_operations;
```

Figure 14: Code: Setup Phase

vertex to the final vertex. The vector $\hat{V}$ denotes the value of each path, with each path representing a scaled expansion of an entry of B.

The code implements the expansion phase is shown in Fig. 20 line 10-35. We use one block to expand one row of A and one thread in the block to compute one non-zero entry in the row. The obtained arrays are stored in the shared memory. Then, the results of the expansion phase are presented in Fig. 11 line 37-40.

3.3 Sorting

As the outcome of the Expansion phase, items with the same row index are placed in adjacent positions. However in the segment of data corresponding to the same row index, their column indexes may be duplicated, and they are not arranged in the order from the smallest to the largest. In order to get the COO form of the matrix multiplication result, we need to arrange the column index of each row in the sorting stage.

The algorithm is straightforward, as shown in Fig. 16; it just applies **key_value_sort** to the data segments corresponding to each row, but the implementation is not trivial. The referenced paper underscores that in implementing the ESC algorithm, the Thrust sorting algorithms allocate and free large amounts of temporary global memory each time they are invoked, which represents a non-trivial cost. The inef-

parameters: A, B
return: $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$

foreach row i in C **do**
  for $k \in NNZ(A_{row_i})$ {Note $A_{i,k}$ is stored in shared memory for reuse}
    for $j \in NNZ(B_{row_k})$
      $\hat{I} \leftarrow [\hat{I},\ i]$ {implicit row index}
      $\hat{J} \leftarrow [\hat{J},\ j]$ {append column index}
      $\hat{V} \leftarrow [\hat{V},\ A_{i,k} * B_{k,j}]$ {append value}

Figure 15: Algorithm: Expansion Phase

ficient sorting performance is a potential bottleneck of the algorithm. Besides, they analyzed some faster global sorting algorithms and exploited the knowledge about the range of input values. However, the results highlight the limited gains achieved by focusing on improving global sorting methods. Therefore, sorting within the GPU's higher-bandwidth shared memory is concentrated on in the modified ESC algorithm to enhance the sorting performance.

The modified ESC algorithm sorts the regions corresponding to each matrix row in parallel within shared memory, which is a feasible and efficient method because: first, the rows of the matrix are independent of each other. Processing one row does not affect the other rows; the global sorting operations over millions of entries are replaced by numerous operations over possibly tens to thousands of entries; sorting the intermediate entries using shared memory reduces global memory operations. In the described implementation, the referenced paper further considered the possible inefficiency of attempting to sort a highly varying workload using a static number of threads. To address this problem, we proportionally scale the number of threads per row of C with the maximum number of entries produced during the expansion. Specifically, when $nnz(C)$ is less than 32, we will use one thread and bubble sorting network to sort this row. And when $nnz(C)$ is greater than 32 and less than 768, we will use a block and radix sorting algorithm to sort this row.

parameters: n: number of columns in B
$\hat{C} = (\hat{I}, \hat{J}, \hat{V})$: column indices, $\hat{J}$, reside in shared memory
return: $\hat{C}, P$ {$\hat{J}$ sorted row-wise and permutation vector P}

foreach row i in C **do**
  $J, V \leftarrow$ `extract`$_i$ $\hat{J}, \hat{V}$ {extract entries where $\hat{I} \equiv i$}
  $m \leftarrow$ `nnz`(J) {m is the number of expanded entries}
  $J, P \leftarrow$ `key_value_sort`$(J, [0, m])$ {keys-value sort}

Figure 16: Algorithm: Sorting Phase

The code responsible for the sorting phase is shown in Fig. 17 & 20 line 39-50. The sorting operations are applied on the shared arrays obtained from the expansion phase. When $nnz(C_{row_i}) \leqslant 32$, we use the **bubbleSortNetwork** within a thread to sort data segments corresponding to row_i. Otherwise, we use the **cub::BlockRadixSort** within a block to do it. The results of the sorting phase are presented in Fig. 11 line 41-44.

3.4 Contraction

In the contraction phase, we contract the values associated with duplicate column indices in $\hat{C}$ using pairwise addition. In contrast to the predictable nature of the total work required to construct $\hat{C}$ in the expansion phase, the number of duplicates to form any row of C is challenging to compute and often varies significantly between rows in $\hat{C}$. To address the inefficiency caused by the irregularity of work required, the ESC algorithms try to avoid imbalance by employing the **reduce_by_key** function in Thrust. However, the modified ESC algorithm focuses more on the possible optimizations associated with utilizing shared memory storage following the

```
__device__
void bubbleSortSwap(int* col_ind, int i, int j) {
    int temp = col_ind[i];
    col_ind[i] = col_ind[j];
    col_ind[j] = temp;
}

__device__
void bubbleSortSwap(float* ety_val, int i, int j) {
    float temp = ety_val[i];
    ety_val[i] = ety_val[j];
    ety_val[j] = temp;
}

__device__
void bubbleSortNetwork(int* col_ind, float* ety_val, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (col_ind[i] > col_ind[j]) {
                bubbleSortSwap(col_ind, i, j);
                bubbleSortSwap(ety_val, i, j);
            }
        }
    }
}
```

Figure 17: Code: Bubble Sort Network

sorting results stored in the shared memory. The pseudo-code of the contraction algorithm is presented in Fig. 18.

parameters: $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$, P — $\{P$ permutation which sorts $\hat{C}$ row-wise$\}$
return: C

foreach row i in C **do**
1 $v \leftarrow 0$ — {initialize output value}
$J, V \leftarrow \texttt{extract}_i\ \hat{J}, \hat{V}$ — $\{$extract entries where $\hat{I} \equiv i\}$
for $j = 0, \ldots, \texttt{nnz}(\hat{C}_{row_i})$ — $\{$Segmented scan by keys, J, over values in $V\}$
2 $v \leftarrow v + V[P_j]$ — {reduce consecutive values}
if $J[j] \neq J[j+1]$ — $\{J[j+1]$ marks beginning of new nonzero entry$\}$
3 $C_{i,J[j]} \leftarrow v$ — {Store accumulated value to output row}
4 $v \leftarrow 0$ — {re-initialize output value}

Figure 18: Algorithm: Contraction Phase

The code used to implement the contraction phase is rendered in Fig. 19 & 20 line 52. We apply **contractionOperation** to the shared arrays obtained from the sorting phase. Specifically, we assign one thread to one element and compare the element with the adjacent element. The results of the contraction phases are presented in Fig. 11 line 45-52.

```
__device__
void contractionOperation(int* col_ind, float* ety_val, int n) {
    int tid = threadIdx.x;
    if (tid < n) {
        if (tid == 0 || col_ind[tid] != col_ind[tid - 1]) {
            float res = ety_val[tid];
            for (int j = tid + 1; j < n && col_ind[j] == col_ind[tid]; j++) {
                res += ety_val[j];
                col_ind[j] = 0;
                ety_val[j] = 0;
            }
            ety_val[tid] = res;
        }
    }
}
```

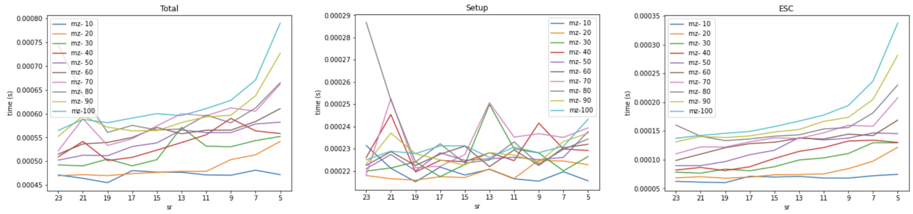
Figure 19: Code: Contraction Kernel

```
__global__
void expansionSortingContractionKernel(int* order, int* counting, int* operations,
                                        int* A_ptr, int* A_ind, float* A_val, int A_row, int A_col, int A_nnz,
                                        int* B_ptr, int* B_ind, float* B_val, int B_row, int B_col, int B_nnz,
                                        int* resulting_row, int* resulting_col, float* resulting_val) {
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    int block_size = blockDim.x;

    if (bid < A_row) {
        int row = order[bid];
        int A_row_start = A_ptr[row];
        int A_row_end = A_ptr[row + 1];
        int num_C_row_nnz = counting[bid];
        __shared__ float C_row_val[768];
        __shared__ int C_row_col[768];
        __shared__ int C_row_ind;
        if (tid == 0) {
            C_row_ind = 0;
        }
        __syncthreads();

        for (int entryIdx = tid + A_row_start; entryIdx < A_row_end; entryIdx += block_size) {
            int k = A_ind[entryIdx];
            int A_ik = A_val[entryIdx];
            int B_rowk_start = B_ptr[k];
            int B_rowk_end = B_ptr[k + 1];
            for (int i = B_rowk_start; i < B_rowk_end; i++) {
                int j = B_ind[i];
                int B_kj = B_val[i];
                int pos = atomicAdd(&C_row_ind, 1);
                C_row_val[pos] = A_ik * B_kj;
                C_row_col[pos] = j;
            }
        }

        __syncthreads();

        if (num_C_row_nnz <= 32) {
            if (tid == 0) {
                bubbleSortNetwork(C_row_col, C_row_val, num_C_row_nnz);
            }
        }
        else {
            typedef cub::BlockRadixSort<int, N_THREADS_PER_BLOCK, 6, float> BlockRadixSort;
            // Allocate shared memory for BlockRadixSort
            __shared__ typename BlockRadixSort::TempStorage temp_storage;
            // Collectively sort the keys
            BlockRadixSort(temp_storage).Sort(*static_cast<int(*)[6]>(static_cast<void*>(C_row_col + 6 * threadIdx.x)), *static_cast<float(*)[6]>(static_cast<void*>(C_row_val + 6 * threadIdx.x)));
        }

        contractionOperation(C_row_col, C_row_val, num_C_row_nnz);
        for (int resIdx = tid; resIdx < num_C_row_nnz; resIdx += block_size) {
            if (C_row_val[resIdx] != 0.f) {
                resulting_row[operations[bid] + resIdx] = row;
                resulting_col[operations[bid] + resIdx] = C_row_col[resIdx];
                resulting_val[operations[bid] + resIdx] = C_row_val[resIdx];
            }
        }
    }
    __syncthreads();
}
```

Figure 20: Code: Expansion Sorting and Contraction Kernel

4 COMPUTATIONAL RESULT

We employed two dimensions to evaluate our performance: matrix size and sparse ratio. The sparse ratio represents the ratio of the total number of entries in the matrix to the number of nonzero entries. The performance results are presented in Fig. 21(a). As observed in the figure, the time required to perform matrix multiplication increases with an increase in the matrix size, ranging from 10 to 100. Additionally, the time taken for the multiplication process increases significantly as the sparse ratio decreases. Fig. 21(b) illustrates our performance during the setup phase. While the patterns of the processing time are not entirely clear with respect to matrix size and sparse ratio, the processing time falls within a narrow range of $2.2 * 10^{-4}$ to $2.6 * 10^{-4}$ seconds. Therefore, our findings indicate that it takes a comparable amount of time to process the setup phase for matrix sizes ranging from 10 to 100 and sparse ratios ranging from 5 to 23. The performance of the ESC algorithm is presented in Fig. 21(c), where the processing time is plotted against the matrix size and sparse ratio. The figure clearly indicates that the processing time of the ESC algorithm increases with an increase in the matrix size, which aligns with our expectations. Moreover, we observe that the processing time tends to increase as the sparse ratio decreases, and this trend becomes more prominent for larger matrix sizes.



(a) Total Time (b) Setup Time (c) ESC Kernel Time

Figure 21: Computational Results: The Modified ESC Algorithm

We conducted a performance comparison of our implementation with two libraries developed by the authors, namely **CUSP** and **CUSParse**, both of which are used for matrix multiplication. The comparison was performed using the same sparse matrices, and the resulting performance is organized in Table. 1. Surprisingly, our implementation outperformed both libraries, with **cusparseSpGEMM_compute** demonstrating better performance than **cusp::multiply**. It is important to note that the total processing time for our implementation ranged from $4 * 10^{-4}$ to $8 * 10^{-4}$, whereas CUSParse and CUSP libraries ranged from $8 * 10^{-4}$ to $1.5 * 10^{-3}$ and 110^{-3} to $6 * 10^{-3}$, respectively.

Matrix Size	The Modified ESC		CUSparse		CUSP
	Total	ESC Kernel	Total	Compute	Total
10	0.000472	0.000068	0.000940	0.000292	0.000881
20	0.000488	0.000080	0.000961	0.000327	0.001626
30	0.000521	0.000098	0.001009	0.000346	0.001343
40	0.000539	0.000107	0.000946	0.000340	0.001445
50	0.000544	0.000120	0.000952	0.000333	0.001607
60	0.000559	0.000132	0.000977	0.000340	0.001860
70	0.000584	0.000143	0.001055	0.000355	0.001586
80	0.000611	0.000157	0.001002	0.000327	0.001491
90	0.000598	0.000168	0.001003	0.000336	0.001815
100	0.000622	0.000184	0.001095	0.000341	0.001785

Table 1: Computational Results

5 CONCLUSION

In the project, we reproduce the modified ESC algorithm from scratch, which exhibits notable speedup compared with the previous version. The setup phase, which contains a reordering scheme, is employed to process the intermediate matrix more effectively, specifically to reduce the load balance between adjacent blocks.

The number of total operations per row of the intermediate matrix is computed in the setup phase, and accordingly, the permutation is made. Besides, the shared memory is thoroughly utilized in the sorting and contraction phase. By parallelly sorting and contracting each part of the intermediate matrix in the shared memory, the time-consuming global operations are reduced, finally leading to performance improvement. In contrast, it is clear that the modified ESC algorithm, which relies on many operations performed in shared memory for speedup, does not behave better in all cases, especially when the shared memory size becomes a bottleneck. The referenced paper mentioned the ecumenical performance with a larger matrix size and small sparse ratio. Due to the computational capacity of the machine, such cases were not included in our tests. However, this will be the direction for our further exploration and enhancement.

6 REFERENCE

Bell, N., Dalton, S., & Olson, L. N. (2012). Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. SIAM Journal on Scientific Computing, 34(4), C123–C152.
https://doi.org/10.1137/110838844

Bell, N., & Garland, M. (2008). Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Corporation.

Chan, T. M. (2007). More Algorithms for All-Pairs Shortest Paths in Weighted Graphs. Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, 590-598.
https://doi.org/10.1145/1250790.1250877

Cohen, E. (1996). On Optimizing Multiplications of Sparse Matrices. Conference on Integer Programming and Combinatorial Optimization.

Davis, T. A. (2018). Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss. 2018 IEEE High Performance Extreme Computing Conference (HPEC), 1–6.
https://doi.org/10.1109/HPEC.2018.8547538

Dalton, S., Olson, L., & Bell, N. (2015). Optimizing Sparse Matrix—Matrix Multiplication for the GPU. ACM Transactions on Mathematical Software, 41(4), 1–20.
https://doi.org/10.1145/2699470

Gremse, F., Höfter, A., Schwen, L. O., Kiessling, F., & Naumann, U. (2015). GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. SIAM Journal on Scientific Computing, 37(1), C54–C71.
https://doi.org/10.1137/130948811

Gao, J., Ji, W., Tan, Z., & Zhao, Y. (2023). A Systematic Survey of General Sparse Matrix-Matrix Multiplication. ACM Computing Surveys, 55(12), 1–36.
https://doi.org/10.1145/3571157

Garland, M., Krik, D. B. (2010). Understanding throughput-oriented architectures. Communications of the ACM, 53(11), 58-66.
https://doi.org/10.1145/1839676.1839694

Kaplan, H., Sharir, M., & Verbin, E. (2006). Colored intersection searching via sparse rectangular matrix multiplication. Proceedings of the Twenty-Second Annual Symposium on Computational Geometry, 52–60.
https://doi.org/10.1145/1137856.1137866

Liu, W., & Vinter, B. (2015). A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. Journal of Parallel and Distributed Computing, 85, 47–61.
https://doi.org/10.1016/j.jpdc.2015.06.010

Li, K., Yang, W., & Li, K. (2015). Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling. IEEE Transactions on Parallel and Distributed Systems, 26(1), 196-205.
https://doi.org/10.1109/TPDS.2014.2308221

Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T.-A., Pham, K., Kemper, A., Neumann, T., & Vo, H. T. (2014). The more the merrier: Efficient multi-source graph traversal. Proceedings of the VLDB Endowment, 8(4), 449–460.
https://doi.org/10.14778/2735496.2735507

7 APPENDIX

This section presented all codes used to implement and test the algorithm, as well as the sample output of the algorithm. It consists of four parts:

- 7.1: implementing and testing the modified ESC algorithm
- 7.2: implementing and testing the **cusparseSpGEMM_compute**
- 7.3: implementing and testing the **cusp::multiply**
- 7.4: demonstration used in the Sec. 3.

7.1 The Modified ESC Algorithm: Implementation, Test, and Sample Output

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <thrust/device_ptr.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/scan.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/iterator/zip_iterator.h>
#include <cub/cub.cuh>

#include <chrono>

using namespace std::chrono;

#define N_WARPS_PER_BLOCK (1 << 2)
#define WARP_SIZE (1 << 5)
#define N_THREADS_PER_BLOCK (1 << 7)

void printVector(float* counting, int nrow) {
    for (int i = 0; i < nrow; i++) {
        std::cout << counting[i] << " ";
    }
    std::cout << "\n";
}

int generateASparseMatrixRandomly(int nrow, int ncol, float** result_matrix, int
     sparse_ratio) {
    float* A = (float*)malloc(sizeof(float) * nrow * ncol);
    int nnz = 0;
    int r;
    float* cur;
    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            r = rand();
            cur = A + (i * ncol + j);
            if (r % sparse_ratio == 0) { *cur = 10.0 * (r / (double)RAND_MAX);; }
            else { *cur = 0.0f; }
            if (*cur != 0.0f) { nnz++; }
        }
    }
    *result_matrix = A;
    return nnz;
}

void convertToCSRFormat(float* mat, int nrow, int ncol, int nnz, int** ptr, int** indices,
      float** data) {
    int* row_ptr = (int*)malloc(sizeof(int) * (nrow + 1));
    int* col_ind = (int*)malloc(sizeof(int) * nnz);
    float* nz_val = (float*)malloc(sizeof(float) * nnz);
```

```
    float* cur;
    int count = 0;
    for (int i = 0; i < nrow; i++) {
        row_ptr[i] = count;
        for (int j = 0; j < ncol; j++) {
            cur = mat + (i * ncol + j);
            if (*cur != 0.0f) {
                col_ind[count] = j;
                nz_val[count] = *cur;
                count++;
            }
        }
    }
    row_ptr[nrow] = count;

    *ptr = row_ptr;
    *indices = col_ind;
    *data = nz_val;
    return;
}

__global__
void countingKernel(int n, int* d_counting, int* A_ptr, int* A_ind, int* B_ptr) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    __syncthreads();
    if (idx < n) {
        int start = A_ptr[idx];
        int end = A_ptr[idx + 1];
        for (int i = start; i < end; i++) {
            int j = A_ind[i];
            d_counting[idx] = d_counting[idx] + B_ptr[j + 1] - B_ptr[j];
        }
    }
    __syncthreads();
}

__device__
void bubbleSortSwap(int* col_ind, int i, int j) {
    int temp = col_ind[i];
    col_ind[i] = col_ind[j];
    col_ind[j] = temp;
}

__device__
void bubbleSortSwap(float* ety_val, int i, int j) {
    float temp = ety_val[i];
    ety_val[i] = ety_val[j];
    ety_val[j] = temp;
}

__device__
void bubbleSortNetwork(int* col_ind, float* ety_val, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (col_ind[i] > col_ind[j]) {
                bubbleSortSwap(col_ind, i, j);
                bubbleSortSwap(ety_val, i, j);
            }
        }
    }
}

__device__
void contractionOperation(int* col_ind, float* ety_val, int n) {
    int tid = threadIdx.x;
    if (tid < n) {
        if (tid == 0 || col_ind[tid] != col_ind[tid - 1]) {
            float res = ety_val[tid];
            for (int j = tid + 1; j < n && col_ind[j] == col_ind[tid]; j++) {
                res += ety_val[j];
                col_ind[j] = 0;
```

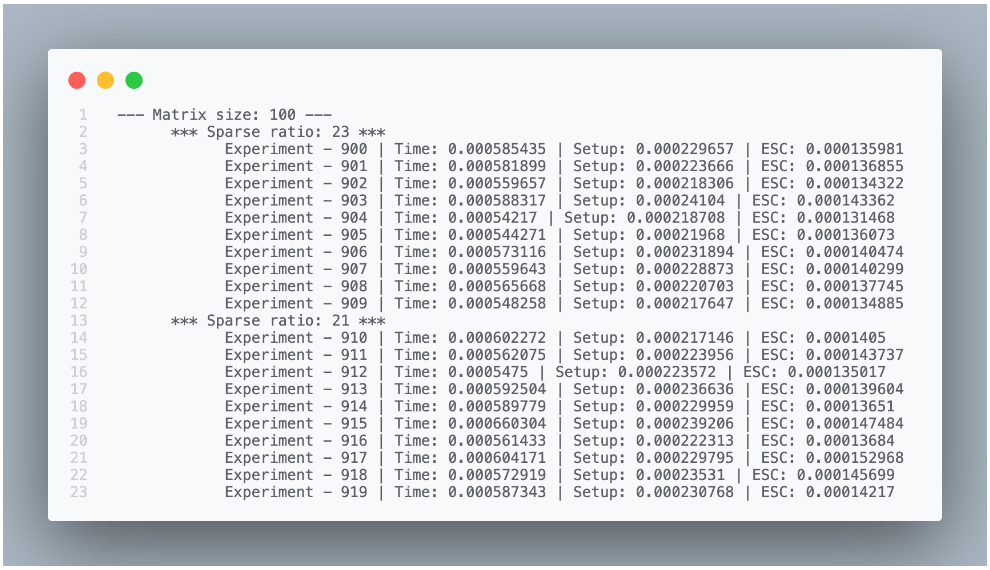
```
                ety_val[j] = 0;
            }
            ety_val[tid] = res;
        }
    }
}

__global__
void expansionSortingContractionKernel(int* order, int* counting, int* operations,
                                       int* A_ptr, int* A_ind, float* A_val, int A_row, int A_col, int A_nnz,
                                       int* B_ptr, int* B_ind, float* B_val, int B_row, int B_col, int B_nnz,
                                       int* resulting_row, int* resulting_col, float* resulting_val) {
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    int block_size = blockDim.x;

    if (bid < A_row) {
        int row = order[bid];
        int A_row_start = A_ptr[row];
        int A_row_end = A_ptr[row + 1];
        int num_C_row_nnz = counting[bid];
        __shared__ float C_row_val[768];
        __shared__ int C_row_col[768];
        __shared__ int C_row_ind;
        if (tid == 0) {
            C_row_ind = 0;
        }
        __syncthreads();

        for (int entryIdx = tid + A_row_start; entryIdx < A_row_end; entryIdx += block_size) {
            int k = A_ind[entryIdx];
            int A_ik = A_val[entryIdx];
            int B_rowk_start = B_ptr[k];
            int B_rowk_end = B_ptr[k + 1];
            for (int i = B_rowk_start; i < B_rowk_end; i++) {
                int j = B_ind[i];
                int B_kj = B_val[i];
                int pos = atomicAdd(&C_row_ind, 1);
                C_row_val[pos] = A_ik * B_kj;
                C_row_col[pos] = j;
            }
        }

        __syncthreads();

        if (num_C_row_nnz <= 32) {
            if (tid == 0) {
                bubbleSortNetwork(C_row_col, C_row_val, num_C_row_nnz);
            }
        }
        else {
            typedef cub::BlockRadixSort<int, N_THREADS_PER_BLOCK, 6, float> BlockRadixSort;
            // Allocate shared memory for BlockRadixSort
            __shared__ typename BlockRadixSort::TempStorage temp_storage;
            // Collectively sort the keys
            BlockRadixSort(temp_storage).Sort(*static_cast<int(*)[6]>(static_cast<void*>(C_row_col + 6 * threadIdx.x)), *static_cast<float(*)[6]>(static_cast<void*>(C_row_val + 6 * threadIdx.x)));
        }

        contractionOperation(C_row_col, C_row_val, num_C_row_nnz);
        for (int resIdx = tid; resIdx < num_C_row_nnz; resIdx += block_size) {
            if (C_row_val[resIdx] != 0.f) {
                resulting_row[operations[bid] + resIdx] = row;
                resulting_col[operations[bid] + resIdx] = C_row_col[resIdx];
                resulting_val[operations[bid] + resIdx] = C_row_val[resIdx];
```

```
            }
        }
    }
    __syncthreads();
}

int SPMMM(int* A_ptr, int* A_ind, float* A_val, int A_row, int A_col, int A_nnz,
            int* B_ptr, int* B_ind, float* B_val, int B_row, int B_col, int B_nnz,
            int** result_row, int** result_col, float** result_val,
            float* recorded_time) {


    auto tot_start_time = high_resolution_clock::now();

    int* d_A_ptr, * d_A_ind, * d_B_ptr, * d_B_ind;
    float* d_A_val, * d_B_val;
    cudaMalloc(&d_A_ptr, (A_row + 1) * sizeof(int));
    cudaMalloc(&d_A_ind, A_nnz * sizeof(int));
    cudaMalloc(&d_A_val, A_nnz * sizeof(float));
    cudaMalloc(&d_B_ptr, (B_row + 1) * sizeof(int));
    cudaMalloc(&d_B_ind, B_nnz * sizeof(int));
    cudaMalloc(&d_B_val, B_nnz * sizeof(float));
    cudaMemcpy(d_A_ptr, A_ptr, (A_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A_ind, A_ind, A_nnz * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A_val, A_val, A_nnz * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_ptr, B_ptr, (B_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_ind, B_ind, B_nnz * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_val, B_val, B_nnz * sizeof(float), cudaMemcpyHostToDevice);


    auto setup_start_time = high_resolution_clock::now();
    int* counting = (int*)malloc(A_row * sizeof(int));
    int* d_counting;
    cudaMalloc(&d_counting, A_row * sizeof(int));
    cudaMemset(d_counting, 0, A_row * sizeof(int));
    int set_up_n_blocks = (A_row + N_THREADS_PER_BLOCK - 1) / N_THREADS_PER_BLOCK;

    countingKernel <<< set_up_n_blocks, N_THREADS_PER_BLOCK >>> (A_row, d_counting,
     d_A_ptr, d_A_ind, d_B_ptr);
    cudaDeviceSynchronize();
    cudaMemcpy(counting, d_counting, A_row * sizeof(int), cudaMemcpyDeviceToHost);

    thrust::device_ptr<int> d_counting_ptr(d_counting);
    thrust::device_vector<int> d_counting_vec(d_counting_ptr, d_counting_ptr + A_row);
    thrust::device_vector<int> d_order_vec(A_row);
    thrust::sequence(d_order_vec.begin(), d_order_vec.end());
    thrust::stable_sort_by_key(d_counting_vec.begin(), d_counting_vec.end(), d_order_vec.
     begin(), thrust::greater<int>());
    thrust::device_vector<int> d_operations_vec(A_row + 1);
    thrust::exclusive_scan(d_counting_vec.begin(), d_counting_vec.end(), d_operations_vec.
     begin());
    int tot_operations = d_counting_vec.back() + d_operations_vec[A_row - 1];
    d_operations_vec[A_row] = tot_operations;
    auto setup_end_time = high_resolution_clock::now();

    int* order = (int*)malloc(A_row * sizeof(int));
    int* operations = (int*)malloc((A_row + 1) * sizeof(int));
    thrust::copy(d_order_vec.begin(), d_order_vec.end(), order);
    thrust::copy(d_operations_vec.begin(), d_operations_vec.end(), operations);
    thrust::copy(d_counting_vec.begin(), d_counting_vec.end(), counting);

    if (counting[0] > 768) {
        return 1;
    }

    int* d_order;
    int* d_operations;
    cudaMalloc(&d_order, A_row * sizeof(int));
    cudaMalloc(&d_operations, (A_row + 1) * sizeof(int));
    cudaMemcpy(d_order, order, A_row * sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_operations, operations, (A_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_counting, counting, A_row * sizeof(int), cudaMemcpyHostToDevice);

    int* d_resulting_row, * d_resulting_col;
    float* d_resulting_val;
    cudaMalloc(&d_resulting_row, tot_operations * sizeof(int));
    cudaMalloc(&d_resulting_col, tot_operations * sizeof(int));
    cudaMalloc(&d_resulting_val, tot_operations * sizeof(float));

    int enpansion_and_sorting_n_blocks = A_row;

    auto esc_start_time = high_resolution_clock::now();
    expansionSortingContractionKernel <<< enpansion_and_sorting_n_blocks, N_THREADS_PER_BLOCK >>> (d_order, d_counting, d_operations,
                d_A_ptr, d_A_ind, d_A_val, A_row, A_col, A_nnz,
                d_B_ptr, d_B_ind, d_B_val, B_row, B_col, B_nnz,
                d_resulting_row, d_resulting_col, d_resulting_val);
    cudaDeviceSynchronize();

    int* resulting_row = (int*)malloc(tot_operations * sizeof(int));
    int* resulting_col = (int*)malloc(tot_operations * sizeof(int));
    float* resulting_val = (float*)malloc(tot_operations * sizeof(float));
    cudaMemcpy(resulting_row, d_resulting_row, tot_operations * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(resulting_col, d_resulting_col, tot_operations * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(resulting_val, d_resulting_val, tot_operations * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_A_ptr);
    cudaFree(d_A_ind);
    cudaFree(d_A_val);
    cudaFree(d_B_ptr);
    cudaFree(d_B_ind);
    cudaFree(d_B_val);
    cudaFree(d_counting);
    cudaFree(d_order);
    cudaFree(d_operations);
    cudaFree(d_resulting_row);
    cudaFree(d_resulting_col);
    cudaFree(d_resulting_val);

    auto esc_end_time = high_resolution_clock::now();

    free(A_ptr);
    free(A_ind);
    free(A_val);
    free(B_ptr);
    free(B_ind);
    free(B_val);
    free(counting);
    free(order);
    free(operations);

    auto tot_end_time = high_resolution_clock::now();

    auto tot_elapsed_time = duration_cast<duration<double>>(tot_end_time - tot_start_time);
    auto setup_elapsed_time = duration_cast<duration<double>>(setup_end_time - setup_start_time);
    auto esc_elapsed_time = duration_cast<duration<double>>(esc_end_time - esc_start_time);
    float tot_elapsed_time_sec = static_cast<float>(tot_elapsed_time.count());
    float setup_elapsed_time_sec = static_cast<float>(setup_elapsed_time.count());
    float esc_elapsed_time_sec = static_cast<float>(esc_elapsed_time.count());

    *result_row = resulting_row;
```

```
    *result_col = resulting_col;
    *result_val = resulting_val;

    free(resulting_row);
    free(resulting_col);
    free(resulting_val);

    recorded_time[0] += tot_elapsed_time_sec;
    recorded_time[1] += setup_elapsed_time_sec;
    recorded_time[2] += esc_elapsed_time_sec;

    return 0;

}

int main() {

    int num_matrix_size = 10;
    int* matrix_size_list = (int*)malloc(num_matrix_size * sizeof(int));
    for (int i = 0; i < num_matrix_size; i++) {
        matrix_size_list[i] = 10 * (i + 1);
    }

    int num_sparse_ratio = 10;
    int* sparse_ratio_list = (int*)malloc(num_sparse_ratio * sizeof(int));
    for (int i = 0; i < num_sparse_ratio; i++) {
        sparse_ratio_list[i] = 23 - 2*i;
    }

    int repeat_times = 10;

    float* time1_tot = (float*)malloc(num_matrix_size * num_sparse_ratio * repeat_times *
     sizeof(float));
    float* time1_setup = (float*)malloc(num_matrix_size * num_sparse_ratio * repeat_times
     * sizeof(float));
    float* time1_esc = (float*)malloc(num_matrix_size * num_sparse_ratio * repeat_times *
     sizeof(float));

    for (int i = 0; i < num_matrix_size; i++) {
        std::cout << " --- Matrix size: " << matrix_size_list[i] << " --- \n";
        for (int j = 0; j < num_sparse_ratio; j++) {
            std::cout << "       *** Sparse ratio: " << sparse_ratio_list[j] << " *** \n";
            for (int k = 0; k < repeat_times; k++) {
                int mz = matrix_size_list[i];
                int sr = sparse_ratio_list[j];

                int A_row = mz;
                int A_col = mz;
                int B_row = mz;
                int B_col = mz;
                float* A;
                float* B;
                int A_nnz = generateASparseMatrixRandomly(A_row, A_col, &A, sr);
                int B_nnz = generateASparseMatrixRandomly(B_row, B_col, &B, sr);
                int* A_ptr;
                int* A_ind;
                float* A_val;
                convertToCSRFormat(A, A_row, A_col, A_nnz, &A_ptr, &A_ind, &A_val);
                int* B_ptr;
                int* B_ind;
                float* B_val;
                convertToCSRFormat(B, B_row, B_col, B_nnz, &B_ptr, &B_ind, &B_val);

                float* time = (float*)malloc(3 * sizeof(float));
                for (int i = 0; i < 3; i++) {
                    time[i] = 0;
                }
                int* resulting_row;
                int* resulting_col;
                float* resulting_val;
                int exit_code = SPMMM(A_ptr, A_ind, A_val, A_row, A_col, A_nnz,
```

```cpp
                                    B_ptr, B_ind, B_val, B_row, B_col, B_nnz,
                                    &resulting_row, &resulting_col, &resulting_val, time);

                if (exit_code == 0) {
                    time1_tot[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = time[0];
                    time1_setup[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = time[1];
                    time1_esc[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = time[2];
                    std::cout << "             Experiment - " << i * num_sparse_ratio * repeat_times + j * repeat_times + k << " | Time: " << time[0] << " | Setup: " << time[1] << " | ESC: " << time[2] << " \n";
                }
                else {
                    time1_tot[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = 0;
                    time1_setup[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = 0;
                    time1_esc[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = 0;
                    std::cout << "             Experiment - " << i * num_sparse_ratio * repeat_times + j * repeat_times + k << " | ERROR -> TRY NEXT \n";
                }
            }
        }
    }

    std::cout << " ####################### \n";
    std::cout << " ### PRINT THE TIME1 ### \n";
    std::cout << " ####################### \n";
    std::cout << " --- Time1 TOT --- \n";
    printVector(time1_tot, num_matrix_size * num_sparse_ratio * repeat_times);
    std::cout << " --- Time1 SETUP --- \n";
    printVector(time1_setup, num_matrix_size * num_sparse_ratio * repeat_times);
    std::cout << " --- Time1 ESC --- \n";
    printVector(time1_esc, num_matrix_size * num_sparse_ratio * repeat_times);

    return 0;
}
```



```
 --- Matrix size: 100 ---
      *** Sparse ratio: 23 ***
            Experiment - 900 | Time: 0.000585435 | Setup: 0.000229657 | ESC: 0.000135981
            Experiment - 901 | Time: 0.000581899 | Setup: 0.000223666 | ESC: 0.000136855
            Experiment - 902 | Time: 0.000559657 | Setup: 0.000218306 | ESC: 0.000134322
            Experiment - 903 | Time: 0.000588317 | Setup: 0.00024104 | ESC: 0.000143362
            Experiment - 904 | Time: 0.00054217 | Setup: 0.000218708 | ESC: 0.000131468
            Experiment - 905 | Time: 0.000544271 | Setup: 0.00021968 | ESC: 0.000136073
            Experiment - 906 | Time: 0.000573116 | Setup: 0.000231894 | ESC: 0.000140474
            Experiment - 907 | Time: 0.000559643 | Setup: 0.000228873 | ESC: 0.000140299
            Experiment - 908 | Time: 0.000565668 | Setup: 0.000220703 | ESC: 0.000137745
            Experiment - 909 | Time: 0.000548258 | Setup: 0.000217647 | ESC: 0.000134885
      *** Sparse ratio: 21 ***
            Experiment - 910 | Time: 0.000602272 | Setup: 0.000217146 | ESC: 0.0001405
            Experiment - 911 | Time: 0.000562075 | Setup: 0.000223956 | ESC: 0.000143737
            Experiment - 912 | Time: 0.0005475 | Setup: 0.000223572 | ESC: 0.000135017
            Experiment - 913 | Time: 0.000592504 | Setup: 0.000236636 | ESC: 0.000139604
            Experiment - 914 | Time: 0.000589779 | Setup: 0.000229959 | ESC: 0.00013651
            Experiment - 915 | Time: 0.000660304 | Setup: 0.000239206 | ESC: 0.000147484
            Experiment - 916 | Time: 0.000561433 | Setup: 0.000222313 | ESC: 0.00013684
            Experiment - 917 | Time: 0.000604171 | Setup: 0.000229795 | ESC: 0.000152968
            Experiment - 918 | Time: 0.000572919 | Setup: 0.00023531 | ESC: 0.000145699
            Experiment - 919 | Time: 0.000587343 | Setup: 0.000230768 | ESC: 0.00014217
```

Figure 22: Sample Output

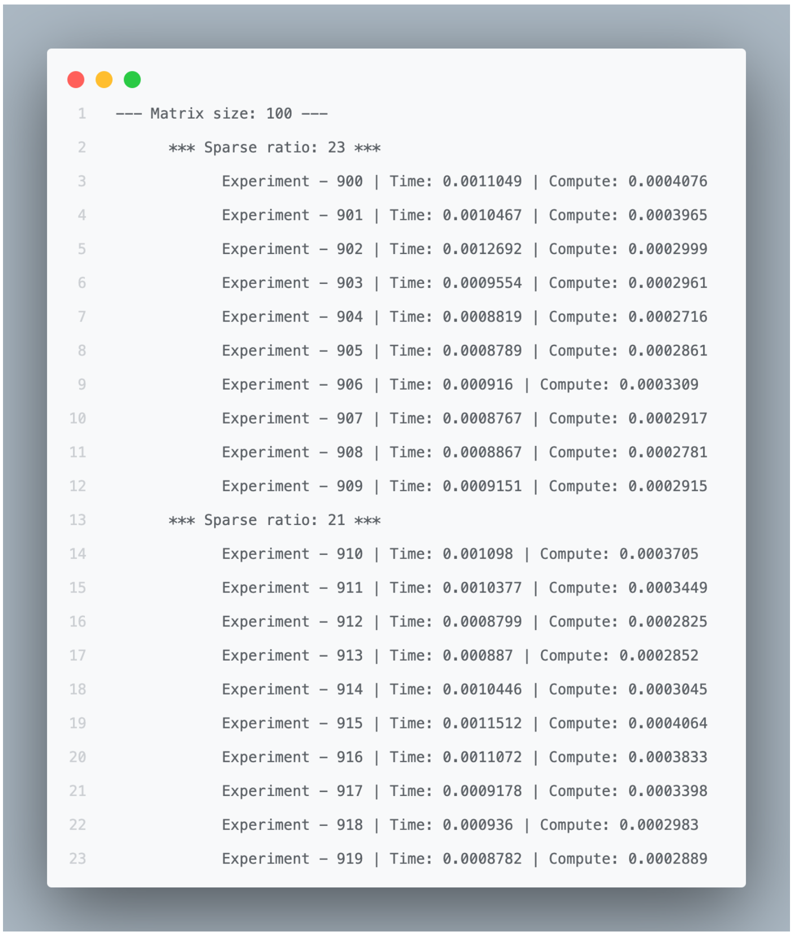
7.2 The *cusparseSpGEMM_compute*: Implementation, Test, and Sample Output

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <cusparse_v2.h>
#include <cuda.h>

#include <chrono>
using namespace std::chrono;

int generateASparseMatrixRandomly(int nrow, int ncol, float** result_matrix, int sparse_ratio) {
    float* A = (float*)malloc(sizeof(float) * nrow * ncol);
    int nnz = 0;
    int r;
    float* cur;
    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            r = rand();
            cur = A + (i * ncol + j);
            if (r % sparse_ratio == 0) { *cur = 10.0 * (r / (double)RAND_MAX);; }
            else { *cur = 0.0f; }
            if (*cur != 0.0f) { nnz++; }
        }
    }
    *result_matrix = A;
    return nnz;
}

void convertToCSRFormat(float* mat, int nrow, int ncol, int nnz, int** ptr, int** indices, float** data) {
    int* row_ptr = (int*)malloc(sizeof(int) * (nrow + 1));
    int* col_ind = (int*)malloc(sizeof(int) * nnz);
    float* nz_val = (float*)malloc(sizeof(float) * nnz);
    float* cur;
    int count = 0;
    for (int i = 0; i < nrow; i++) {
        row_ptr[i] = count;
        for (int j = 0; j < ncol; j++) {
            cur = mat + (i * ncol + j);
            if (*cur != 0.0f) {
                col_ind[count] = j;
                nz_val[count] = *cur;
                count++;
            }
        }
    }
    row_ptr[nrow] = count;

    *ptr = row_ptr;
    *indices = col_ind;
    *data = nz_val;
    return;
}

void SPMMM_COMP(int* A_ptr, int* A_ind, float* A_val, int A_row, int A_col, int A_nnz,
    int* B_ptr, int* B_ind, float* B_val, int B_row, int B_col, int B_nnz,
    int** result_row, int** result_col, float** result_val,
    float* recorded_time) {

    auto tot_start_time = high_resolution_clock::now();

    float alpha = 1.0f, beta = 0.0f;
    cusparseOperation_t opA = CUSPARSE_OPERATION_NON_TRANSPOSE;
    cusparseOperation_t opB = CUSPARSE_OPERATION_NON_TRANSPOSE;
    cudaDataType computeType = CUDA_R_32F;

    int* dA_csrOffsets, * dA_columns, * dB_csrOffsets, * dB_columns, * dC_csrOffsets, * dC_columns;
```

```
    float* dA_values, * dB_values, * dC_values;

    cudaMalloc((void**)&dA_csrOffsets, sizeof(int) * (A_row + 1));
    cudaMalloc((void**)&dA_columns, sizeof(int) * A_nnz);
    cudaMalloc((void**)&dA_values, sizeof(float) * A_nnz);
    cudaMalloc((void**)&dB_csrOffsets, sizeof(int) * (B_row + 1));
    cudaMalloc((void**)&dB_columns, sizeof(int) * B_nnz);
    cudaMalloc((void**)&dB_values, sizeof(float) * B_nnz);
    cudaMalloc((void**)&dC_csrOffsets, sizeof(int) * (A_row + 1));

    cudaMemcpy(dA_csrOffsets, A_ptr, sizeof(int) * (A_row + 1), cudaMemcpyHostToDevice);
    cudaMemcpy(dA_columns, A_ind, sizeof(int) * A_nnz, cudaMemcpyHostToDevice);
    cudaMemcpy(dA_values, A_val, sizeof(float) * A_nnz, cudaMemcpyHostToDevice);
    cudaMemcpy(dB_csrOffsets, B_ptr, sizeof(int) * (B_row + 1), cudaMemcpyHostToDevice);
    cudaMemcpy(dB_columns, B_ind, sizeof(int) * B_nnz, cudaMemcpyHostToDevice);
    cudaMemcpy(dB_values, B_val, sizeof(float) * B_nnz, cudaMemcpyHostToDevice);

    cusparseHandle_t handle = NULL;
    cusparseSpMatDescr_t matA, matB, matC;
    void* dBuffer1 = NULL, * dBuffer2 = NULL;
    size_t bufferSize1 = 0, bufferSize2 = 0;
    cusparseCreate(&handle);

    cusparseCreateCsr(&matA, A_row, A_col, A_nnz, dA_csrOffsets, dA_columns, dA_values,
     CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I, CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
    cusparseCreateCsr(&matB, B_row, B_col, B_nnz, dB_csrOffsets, dB_columns, dB_values,
     CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I, CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
    cusparseCreateCsr(&matC, A_row, B_col, 0, NULL, NULL, NULL, CUSPARSE_INDEX_32I,
     CUSPARSE_INDEX_32I, CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);

    auto compute_start_time = high_resolution_clock::now();
    cusparseSpGEMMDescr_t spgemmDesc;
    cusparseSpGEMM_createDescr(&spgemmDesc);
    cusparseSpGEMM_workEstimation(handle, opA, opB, &alpha, matA, matB, &beta, matC,
     computeType, CUSPARSE_SPGEMM_DEFAULT, spgemmDesc, &bufferSize1, NULL);
    cudaMalloc((void**)&dBuffer1, bufferSize1);
    cusparseSpGEMM_workEstimation(handle, opA, opB, &alpha, matA, matB, &beta, matC,
     computeType, CUSPARSE_SPGEMM_DEFAULT, spgemmDesc, &bufferSize1, dBuffer1);
    cusparseSpGEMM_compute(handle, opA, opB, &alpha, matA, matB, &beta, matC, computeType,
      CUSPARSE_SPGEMM_DEFAULT, spgemmDesc, &bufferSize2, NULL);
    cudaMalloc((void**)&dBuffer2, bufferSize2);
    cusparseSpGEMM_compute(handle, opA, opB, &alpha, matA, matB, &beta, matC, computeType,
      CUSPARSE_SPGEMM_DEFAULT, spgemmDesc, &bufferSize2, dBuffer2);
    int64_t C_num_rows1, C_num_cols1, C_nnz1;
    cusparseSpMatGetSize(matC, &C_num_rows1, &C_num_cols1, &C_nnz1);
    cudaMalloc((void**)&dC_columns, C_nnz1 * sizeof(int));
    cudaMalloc((void**)&dC_values, C_nnz1 * sizeof(float));
    cusparseCsrSetPointers(matC, dC_csrOffsets, dC_columns, dC_values);
    cusparseSpGEMM_copy(handle, opA, opB, &alpha, matA, matB, &beta, matC, computeType,
     CUSPARSE_SPGEMM_DEFAULT, spgemmDesc);

    cusparseSpGEMM_destroyDescr(spgemmDesc);
    cusparseDestroySpMat(matA);
    cusparseDestroySpMat(matB);
    cusparseDestroySpMat(matC);
    cusparseDestroy(handle);
    auto compute_end_time = high_resolution_clock::now();


    int* C_ptr = (int*)malloc(sizeof(int) * (A_row + 1));
    int* C_ind = (int*)malloc(sizeof(int) * C_nnz1);
    float* C_val = (float*)malloc(sizeof(float) * C_nnz1);

    cudaMemcpy(C_ptr, dC_csrOffsets, sizeof(int) * (A_row + 1), cudaMemcpyDeviceToHost);
    cudaMemcpy(C_ind, dC_columns, sizeof(int) * C_nnz1, cudaMemcpyDeviceToHost);
    cudaMemcpy(C_val, dC_values, sizeof(float) * C_nnz1, cudaMemcpyDeviceToHost);

    cudaFree(dBuffer1);
    cudaFree(dBuffer2);
    cudaFree(dA_csrOffsets);
    cudaFree(dA_columns);
```

```
    cudaFree(dA_values);
    cudaFree(dB_csrOffsets);
    cudaFree(dB_columns);
    cudaFree(dB_values);
    cudaFree(dC_csrOffsets);
    cudaFree(dC_columns);
    cudaFree(dC_values);

    *result_row = C_ptr;
    *result_col = C_ind;
    *result_val = C_val;

    auto tot_end_time = std::chrono::high_resolution_clock::now();
    auto tot_elapsed_time = duration_cast<duration<double>>(tot_end_time - tot_start_time);
    float tot_elapsed_time_sec = static_cast<float>(tot_elapsed_time.count());

    auto compute_elapsed_time = duration_cast<duration<double>>(compute_end_time - compute_start_time);
    float compute_elapsed_time_sec = static_cast<float>(compute_elapsed_time.count());

    recorded_time[0] += tot_elapsed_time_sec;
    recorded_time[1] += compute_elapsed_time_sec;

}

int main() {

    int num_matrix_size = 10;
    int* matrix_size_list = (int*)malloc(num_matrix_size * sizeof(int));
    for (int i = 0; i < num_matrix_size; i++) {
        matrix_size_list[i] = 10 * (i + 1);
    }

    int num_sparse_ratio = 10;
    int* sparse_ratio_list = (int*)malloc(num_sparse_ratio * sizeof(int));
    for (int i = 0; i < num_sparse_ratio; i++) {
        sparse_ratio_list[i] = 23 - 2*i;
    }

    int repeat_times = 10;

    float* time2_tot = (float*)malloc(num_matrix_size * num_sparse_ratio * repeat_times * sizeof(float));
    float* time2_compute = (float*)malloc(num_matrix_size * num_sparse_ratio * repeat_times * sizeof(float));

    for (int i = 0; i < num_matrix_size; i++) {
        std::cout << " --- Matrix size: " << matrix_size_list[i] << " --- \n";
        for (int j = 0; j < num_sparse_ratio; j++) {
            std::cout << "       *** Sparse ratio: " << sparse_ratio_list[j] << " *** \n";
            for (int k = 0; k < repeat_times; k++) {
                int mz = matrix_size_list[i];
                int sr = sparse_ratio_list[j];

                int A_row = mz;
                int A_col = mz;
                int B_row = mz;
                int B_col = mz;
                float* A;
                float* B;
                int A_nnz = generateASparseMatrixRandomly(A_row, A_col, &A, sr);
                int B_nnz = generateASparseMatrixRandomly(B_row, B_col, &B, sr);
                int* A_ptr;
                int* A_ind;
                float* A_val;
                convertToCSRFormat(A, A_row, A_col, A_nnz, &A_ptr, &A_ind, &A_val);
                int* B_ptr;
                int* B_ind;
                float* B_val;
                convertToCSRFormat(B, B_row, B_col, B_nnz, &B_ptr, &B_ind, &B_val);
```

```
                float* time = (float*)malloc(2 * sizeof(float));
                for (int i = 0; i < 2; i++) {
                    time[i] = 0;
                }
                int* resulting_row;
                int* resulting_col;
                float* resulting_val;
                SPMMM_COMP(A_ptr, A_ind, A_val, A_row, A_col, A_nnz,
                        B_ptr, B_ind, B_val, B_row, B_col, B_nnz,
                        &resulting_row, &resulting_col, &resulting_val, time);

                time2_tot[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = time[0];
                time2_compute[i * num_sparse_ratio * repeat_times + j * repeat_times + k] = time[1];
                std::cout << "             Experiment - " << i * num_sparse_ratio * repeat_times + j * repeat_times + k << " | Time: " << time[0] << " | Compute: " << time[1] << " \n";

            }
        }
    }
}
```



```
--- Matrix size: 100 ---
     *** Sparse ratio: 23 ***
          Experiment - 900 | Time: 0.0011049 | Compute: 0.0004076
          Experiment - 901 | Time: 0.0010467 | Compute: 0.0003965
          Experiment - 902 | Time: 0.0012692 | Compute: 0.0002999
          Experiment - 903 | Time: 0.0009554 | Compute: 0.0002961
          Experiment - 904 | Time: 0.0008819 | Compute: 0.0002716
          Experiment - 905 | Time: 0.0008789 | Compute: 0.0002861
          Experiment - 906 | Time: 0.000916 | Compute: 0.0003309
          Experiment - 907 | Time: 0.0008767 | Compute: 0.0002917
          Experiment - 908 | Time: 0.0008867 | Compute: 0.0002781
          Experiment - 909 | Time: 0.0009151 | Compute: 0.0002915
     *** Sparse ratio: 21 ***
          Experiment - 910 | Time: 0.001098 | Compute: 0.0003705
          Experiment - 911 | Time: 0.0010377 | Compute: 0.0003449
          Experiment - 912 | Time: 0.0008799 | Compute: 0.0002825
          Experiment - 913 | Time: 0.000887 | Compute: 0.0002852
          Experiment - 914 | Time: 0.0010446 | Compute: 0.0003045
          Experiment - 915 | Time: 0.0011512 | Compute: 0.0004064
          Experiment - 916 | Time: 0.0011072 | Compute: 0.0003833
          Experiment - 917 | Time: 0.0009178 | Compute: 0.0003398
          Experiment - 918 | Time: 0.000936 | Compute: 0.0002983
          Experiment - 919 | Time: 0.0008782 | Compute: 0.0002889
```

Figure 23: Sample Output

7.3 The *cusp::multiply*: Implementation, Test, and Sample Output

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <thrust/device_ptr.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/scan.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/iterator/zip_iterator.h>
#include <cub/cub.cuh>
#include <cusp/array2d.h>
#include <cusp/csr_matrix.h>
#include <cusp/coo_matrix.h>
#include <cusp/multiply.h>
#include <cusp/gallery/poisson.h>
#include <cusp/gallery/random.h>
#include <time.h>
#include <chrono>

using namespace std::chrono;

#define N_WARPS_PER_BLOCK (1 << 2)
#define WARP_SIZE (1 << 5)
#define N_THREADS_PER_BLOCK (1 << 7)
int main() {
    for (size_t i = 1; i <= 10; i++)
    {
        std::cout << "--- Matrix size: " << i << " -- - " << std::endl;
        for (size_t j = 5; j < 24; j++)
        {
            std::cout << "  *** Sparse ratio:" << j << "* **" << std::endl;
            int mz = i * 10;
            cusp::csr_matrix<int, float, cusp::device_memory> c_a;
            cusp::gallery::random(c_a, mz, mz, mz * j / 100);
            cusp::csr_matrix<int, float, cusp::device_memory> c_b;
            cusp::gallery::random(c_b, mz, mz, mz * j / 100);
            cusp::csr_matrix<int, float, cusp::device_memory> c_c;

            std::chrono::time_point<std::chrono::system_clock> start, end;
            start = std::chrono::system_clock::now();
            cusp::multiply(c_a, c_b, c_c);
            end = std::chrono::system_clock::now();
            std::chrono::duration<double> elapsed_seconds = end - start;
            std::cout << "matrxi size: " << i * 10 << ", cusp::multiply: " <<
     elapsed_seconds.count() << " s" << std::endl;
        }
    }
    return 0;
}
```

```
--- Matrix size: 10 -- -
  *** Sparse ratio:5* **
  cusp::multiply:  0.0001258 s
  *** Sparse ratio:6* **
  cusp::multiply:  0.0001457 s
  *** Sparse ratio:7* **
  cusp::multiply:  0.0001257 s
  *** Sparse ratio:8* **
  cusp::multiply:  0.0001277 s
  *** Sparse ratio:9* **
  cusp::multiply:  0.0001045 s
  *** Sparse ratio:10* **
  cusp::multiply:  0.0008739 s
```

Figure 24: Sample Output

7.4 The Modified ESC Algorithm: Demonstration

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <thrust/device_ptr.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/scan.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/iterator/zip_iterator.h>
#include <cub/cub.cuh>

#define N_WARPS_PER_BLOCK (1 << 2)
#define WARP_SIZE (1 << 5)
#define N_THREADS_PER_BLOCK (1 << 7)

int generateASparseMatrixRandomly(int nrow, int ncol, float** result_matrix, int
     sparse_ratio) {
    float* A = (float*)malloc(sizeof(float) * nrow * ncol);
    int nnz = 0;
    int r;
    float* cur;
    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            r = rand();
            cur = A + (i * ncol + j);
            if (r % sparse_ratio == 0) { *cur = 10.0 * (r / (double)RAND_MAX);; }
            else { *cur = 0.0f; }
            if (*cur != 0.0f) { nnz++; }
        }
    }
    *result_matrix = A;
    return nnz;
}

void presentAMatrix(float* mat, int nrow, int ncol, int present_row, int present_col) {
    printf(" --- PRINT THE MATRIX ---- \n");
    for (int i = 0; i < present_row; i++) {
        for (int j = 0; j < present_col; j++) {
            printf("%3.0f", mat[i * ncol + j]);
        }
        printf("...\n");
    }
    printf("...\n");
}

void convertToCSRFormat(float* mat, int nrow, int ncol, int nnz, int** ptr, int** indices,
      float** data) {
    int* row_ptr = (int*)malloc(sizeof(int) * (nrow + 1));
    int* col_ind = (int*)malloc(sizeof(int) * nnz);
    float* nz_val = (float*)malloc(sizeof(float) * nnz);
    float* cur;
    int count = 0;
    for (int i = 0; i < nrow; i++) {
        row_ptr[i] = count;
        for (int j = 0; j < ncol; j++) {
            cur = mat + (i * ncol + j);
            if (*cur != 0.0f) {
                col_ind[count] = j;
                nz_val[count] = *cur;
                count++;
            }
        }
    }
    row_ptr[nrow] = count;

```

```
    *ptr = row_ptr;
    *indices = col_ind;
    *data = nz_val;
    return;
}

void presentCSR(int* ptr, int* indices, float* data, int nnz, int nrow) {
    printf(" --- PRINT THE CSR FORMAT MATRIX ---- \n");
    printf("ptr - ");
    for (int i = 0; i <= nrow; i++) {
        printf("%+5d", ptr[i]);
    }
    printf("\n");
    printf("indices - ");
    for (int i = 0; i < nnz; i++) {
        printf("%+5d", indices[i]);
    }
    printf("\n");
    printf("data - ");
    for (int i = 0; i < nnz; i++) {
        printf("%+5f", data[i]);
    }
    printf("\n");
}

void print(const thrust::device_vector<int>& v)
{
    for (size_t i = 0; i < v.size(); i++)
        std::cout << " " << v[i];
    std::cout << "\n";
}

void print(const thrust::host_vector<int>& v)
{
    for (size_t i = 0; i < v.size(); i++)
        std::cout << " " << v[i];
    std::cout << "\n";
}

void print(const thrust::device_vector<float>& v)
{
    for (size_t i = 0; i < v.size(); i++)
        std::cout << " " << std::fixed << std::setprecision(1) << v[i];
    std::cout << "\n";
}

void print(const thrust::host_vector<float>& v)
{
    for (size_t i = 0; i < v.size(); i++)
        std::cout << " " << std::fixed << std::setprecision(1) << v[i];
    std::cout << "\n";
}

void print(thrust::device_vector<int>& v1, thrust::device_vector<int>& v2)
{
    for (size_t i = 0; i < v1.size(); i++)
        std::cout << " (" << v1[i] << "," << std::setw(2) << v2[i] << ")";
    std::cout << "\n";
}

void printVector(int* counting, int nrow) {
    for (int i = 0; i < nrow; i++) {
        std::cout << counting[i] << " ";
    }
    std::cout << "\n";
}

void printVector(float* counting, int nrow) {
    for (int i = 0; i < nrow; i++) {
        std::cout << counting[i] << " ";
    }
```

```
    std::cout << "\n";
}

__device__
void printVectorDevice(int* vec, int n) {
    for (int i = 0; i < n; ++i) {
        printf("%d ", vec[i]);
    }
    printf("\n");
}

__device__
void printVectorDevice(float* vec, int n) {
    for (int i = 0; i < n; ++i) {
        printf("%.2f ", vec[i]);
    }
    printf("\n");
}

__global__
void countingKernel(int n, int* d_counting, int* A_ptr, int* A_ind, int* B_ptr) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    __syncthreads();
    if (idx < n) {
        int start = A_ptr[idx];
        int end = A_ptr[idx + 1];
        for (int i = start; i < end; i++) {
            int j = A_ind[i];
            d_counting[idx] = d_counting[idx] + B_ptr[j + 1] - B_ptr[j];
        }
    }
    __syncthreads();
}

__device__
void bubbleSortSwap(int* col_ind, int i, int j) {
    int temp = col_ind[i];
    col_ind[i] = col_ind[j];
    col_ind[j] = temp;
}

__device__
void bubbleSortSwap(float* ety_val, int i, int j) {
    float temp = ety_val[i];
    ety_val[i] = ety_val[j];
    ety_val[j] = temp;
}

__device__
void bubbleSortNetwork(int* col_ind, float* ety_val, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (col_ind[i] > col_ind[j]) {
                bubbleSortSwap(col_ind, i, j);
                bubbleSortSwap(ety_val, i, j);
            }
        }
    }
}

__device__
void contractionOperation(int* col_ind, float* ety_val, int n) {
    int tid = threadIdx.x;
    if (tid < n) {
        if (tid == 0 || col_ind[tid] != col_ind[tid - 1]) {
            float res = ety_val[tid];
            for (int j = tid + 1; j < n && col_ind[j] == col_ind[tid]; j++) {
                res += ety_val[j];
                col_ind[j] = 0;
                ety_val[j] = 0;
            }
```

```
            ety_val[tid] = res;
        }
    }
}

__global__
void expansionSortingContractionKernel(int* order, int* counting, int* operations,
    int* A_ptr, int* A_ind, float* A_val, int A_row, int A_col, int A_nnz,
    int* B_ptr, int* B_ind, float* B_val, int B_row, int B_col, int B_nnz,
    int* phase1_row, int* phase1_col, float* phase1_val,
    int* phase2_row, int* phase2_col, float* phase2_val,
    int* resulting_row, int* resulting_col, float* resulting_val) {
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    int block_size = blockDim.x;

    if (bid < A_row) {
        int row = order[bid];
        int A_row_start = A_ptr[row];
        int A_row_end = A_ptr[row + 1];
        int num_C_row_nnz = counting[bid];
        __shared__ float C_row_val[768];
        __shared__ int C_row_col[768];
        __shared__ int C_row_ind;
        if (tid == 0) {
            C_row_ind = 0;
        }
        __syncthreads();

        for (int entryIdx = tid + A_row_start; entryIdx < A_row_end; entryIdx +=
     block_size) {
            int k = A_ind[entryIdx];
            int A_ik = A_val[entryIdx];
            int B_rowk_start = B_ptr[k];
            int B_rowk_end = B_ptr[k + 1];
            for (int i = B_rowk_start; i < B_rowk_end; i++) {
                int j = B_ind[i];
                int B_kj = B_val[i];
                int pos = atomicAdd(&C_row_ind, 1);
                C_row_val[pos] = A_ik * B_kj;
                C_row_col[pos] = j;
            }
        }


        for (int resIdx = tid; resIdx < num_C_row_nnz; resIdx += block_size) {
            phase1_row[operations[bid] + resIdx] = row;
            phase1_col[operations[bid] + resIdx] = C_row_col[resIdx];
            phase1_val[operations[bid] + resIdx] = C_row_val[resIdx];
        }

        __syncthreads();

        if (num_C_row_nnz <= 32) {
            if (tid == 0) {
                bubbleSortNetwork(C_row_col, C_row_val, num_C_row_nnz);
            }
        }
        else {
            typedef cub::BlockRadixSort<int, N_THREADS_PER_BLOCK, 6, float> BlockRadixSort
     ;
            // Allocate shared memory for BlockRadixSort
            __shared__ typename BlockRadixSort::TempStorage temp_storage;
            // Collectively sort the keys
            BlockRadixSort(temp_storage).Sort(*static_cast<int(*)[6]>(static_cast<void*>(
     C_row_col + 6 * threadIdx.x)), *static_cast<float(*)[6]>(static_cast<void*>(C_row_val
      + 6 * threadIdx.x)));
        }

        for (int resIdx = tid; resIdx < num_C_row_nnz; resIdx += block_size) {
            phase2_row[operations[bid] + resIdx] = row;
```

```
            phase2_col[operations[bid] + resIdx] = C_row_col[resIdx];
            phase2_val[operations[bid] + resIdx] = C_row_val[resIdx];
        }

        contractionOperation(C_row_col, C_row_val, num_C_row_nnz);
        for (int resIdx = tid; resIdx < num_C_row_nnz; resIdx += block_size) {
            if (C_row_val[resIdx] != 0.f) {
                resulting_row[operations[bid] + resIdx] = row;
                resulting_col[operations[bid] + resIdx] = C_row_col[resIdx];
                resulting_val[operations[bid] + resIdx] = C_row_val[resIdx];
            }
        }
    }
    __syncthreads();
}

struct removed_item {
    template <typename Tuple>
    __host__ __device__
    bool operator()(const Tuple& t) {
        return (thrust::get<2>(t) == 0);
    }
};


int main() {

    int A_row = 7;
    int A_col = 7;
    int B_row = 7;
    int B_col = 7;
    int present_row = 7;
    int present_col = 7;
    float* A;
    float* B;
    int A_nnz = generateASparseMatrixRandomly(A_row, A_col, &A, 4);
    int B_nnz = generateASparseMatrixRandomly(B_row, B_col, &B, 4);
    presentAMatrix(A, A_row, A_col, present_row, present_col);
    presentAMatrix(B, B_row, B_col, present_row, present_col);
    int* A_ptr;
    int* A_ind;
    float* A_val;
    convertToCSRFormat(A, A_row, A_col, A_nnz, &A_ptr, &A_ind, &A_val);
    presentCSR(A_ptr, A_ind, A_val, A_nnz, A_row);
    int* B_ptr;
    int* B_ind;
    float* B_val;
    convertToCSRFormat(B, B_row, B_col, B_nnz, &B_ptr, &B_ind, &B_val);
    presentCSR(B_ptr, B_ind, B_val, B_nnz, B_row);

    int* d_A_ptr, * d_A_ind, * d_B_ptr, * d_B_ind;
    float* d_A_val, * d_B_val;
    cudaMalloc(&d_A_ptr, (A_row + 1) * sizeof(int));
    cudaMalloc(&d_A_ind, A_nnz * sizeof(int));
    cudaMalloc(&d_A_val, A_nnz * sizeof(float));
    cudaMalloc(&d_B_ptr, (B_row + 1) * sizeof(int));
    cudaMalloc(&d_B_ind, B_nnz * sizeof(int));
    cudaMalloc(&d_B_val, B_nnz * sizeof(float));
    cudaMemcpy(d_A_ptr, A_ptr, (A_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A_ind, A_ind, A_nnz * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A_val, A_val, A_nnz * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_ptr, B_ptr, (B_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_ind, B_ind, B_nnz * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B_val, B_val, B_nnz * sizeof(float), cudaMemcpyHostToDevice);

    int* counting = (int*)malloc(A_row * sizeof(int));
    int* d_counting;
    cudaMalloc(&d_counting, A_row * sizeof(int));
    cudaMemset(d_counting, 0, A_row * sizeof(int));
    int set_up_n_blocks = (A_row + N_THREADS_PER_BLOCK - 1) / N_THREADS_PER_BLOCK;

```

```
countingKernel <<< set_up_n_blocks, N_THREADS_PER_BLOCK >>> (A_row, d_counting, d_A_ptr, d_A_ind, d_B_ptr);
cudaDeviceSynchronize();
cudaMemcpy(counting, d_counting, A_row * sizeof(int), cudaMemcpyDeviceToHost);
std::cout << " --- SET UP --- " << "\n";
std::cout << "original counting: ";
printVector(counting, A_row);

thrust::device_ptr<int> d_counting_ptr(d_counting);
thrust::device_vector<int> d_counting_vec(d_counting_ptr, d_counting_ptr + A_row);
thrust::device_vector<int> d_order_vec(A_row);
thrust::sequence(d_order_vec.begin(), d_order_vec.end());
thrust::stable_sort_by_key(d_counting_vec.begin(), d_counting_vec.end(), d_order_vec.begin(), thrust::greater<int>());
thrust::device_vector<int> d_operations_vec(A_row + 1);
thrust::exclusive_scan(d_counting_vec.begin(), d_counting_vec.end(), d_operations_vec.begin());
int tot_operations = d_counting_vec.back() + d_operations_vec[A_row - 1];
d_operations_vec[A_row] = tot_operations;

std::cout << " --- AFTER REORDER - THRUST VECTOR" << "\n";
std::cout << "d_counting_vec: ";
print(d_counting_vec);
std::cout << "d_order_vec: ";
print(d_order_vec);
std::cout << "d_operations_vec: ";
print(d_operations_vec);

int* order = (int*)malloc(A_row * sizeof(int));
int* operations = (int*)malloc((A_row + 1) * sizeof(int));
thrust::copy(d_order_vec.begin(), d_order_vec.end(), order);
thrust::copy(d_operations_vec.begin(), d_operations_vec.end(), operations);
thrust::copy(d_counting_vec.begin(), d_counting_vec.end(), counting);
std::cout << " --- AFTER REORDER - CONVERT TO ptr" << "\n";
std::cout << "order: ";
printVector(order, A_row);
std::cout << "counting: ";
printVector(counting, A_row);
std::cout << "operations: ";
printVector(operations, A_row + 1);
int* d_order;
int* d_operations;
cudaMalloc(&d_order, A_row * sizeof(int));
cudaMalloc(&d_operations, (A_row + 1) * sizeof(int));
cudaMemcpy(d_order, order, A_row * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_operations, operations, (A_row + 1) * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_counting, counting, A_row * sizeof(int), cudaMemcpyHostToDevice);

int* d_resulting_row, * d_resulting_col;
float* d_resulting_val;
cudaMalloc(&d_resulting_row, tot_operations * sizeof(int));
cudaMalloc(&d_resulting_col, tot_operations * sizeof(int));
cudaMalloc(&d_resulting_val, tot_operations * sizeof(float));

int* d_phase1_row, * d_phase1_col;
float* d_phase1_val;
cudaMalloc(&d_phase1_row, tot_operations * sizeof(int));
cudaMalloc(&d_phase1_col, tot_operations * sizeof(int));
cudaMalloc(&d_phase1_val, tot_operations * sizeof(float));

int* d_phase2_row, * d_phase2_col;
float* d_phase2_val;
cudaMalloc(&d_phase2_row, tot_operations * sizeof(int));
cudaMalloc(&d_phase2_col, tot_operations * sizeof(int));
cudaMalloc(&d_phase2_val, tot_operations * sizeof(float));


int enpansion_and_sorting_n_blocks = A_row;
expansionSortingContractionKernel <<< enpansion_and_sorting_n_blocks, N_THREADS_PER_BLOCK >>> (d_order, d_counting, d_operations,
```

```
        d_A_ptr, d_A_ind, d_A_val, A_row, A_col, A_nnz,
        d_B_ptr, d_B_ind, d_B_val, B_row, B_col, B_nnz,
        d_phase1_row, d_phase1_col, d_phase1_val,
        d_phase2_row, d_phase2_col, d_phase2_val,
        d_resulting_row, d_resulting_col, d_resulting_val);
    cudaDeviceSynchronize();
    int* resulting_row = (int*)malloc(tot_operations * sizeof(int));
    int* resulting_col = (int*)malloc(tot_operations * sizeof(int));
    float* resulting_val = (float*)malloc(tot_operations * sizeof(float));
    cudaMemcpy(resulting_row, d_resulting_row, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(resulting_col, d_resulting_col, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(resulting_val, d_resulting_val, tot_operations * sizeof(float),
     cudaMemcpyDeviceToHost);

    int* phase1_row = (int*)malloc(tot_operations * sizeof(int));
    int* phase1_col = (int*)malloc(tot_operations * sizeof(int));
    float* phase1_val = (float*)malloc(tot_operations * sizeof(float));
    cudaMemcpy(phase1_row, d_phase1_row, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(phase1_col, d_phase1_col, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(phase1_val, d_phase1_val, tot_operations * sizeof(float),
     cudaMemcpyDeviceToHost);

    int* phase2_row = (int*)malloc(tot_operations * sizeof(int));
    int* phase2_col = (int*)malloc(tot_operations * sizeof(int));
    float* phase2_val = (float*)malloc(tot_operations * sizeof(float));
    cudaMemcpy(phase2_row, d_phase2_row, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(phase2_col, d_phase2_col, tot_operations * sizeof(int),
     cudaMemcpyDeviceToHost);
    cudaMemcpy(phase2_val, d_phase2_val, tot_operations * sizeof(float),
     cudaMemcpyDeviceToHost);

    std::cout << " --- AFTER EXPANSION and BEFORE SORTING" << "\n";
    std::cout << "phase1_row: ";
    printVector(phase1_row, tot_operations);
    std::cout << "phase1_col: ";
    printVector(phase1_col, tot_operations);
    std::cout << "phase1_val: ";
    printVector(phase1_val, tot_operations);

    std::cout << " --- AFTER SORTING and BEFORE CONTRACTION" << "\n";
    std::cout << "phase2_row: ";
    printVector(phase2_row, tot_operations);
    std::cout << "phase2_col: ";
    printVector(phase2_col, tot_operations);
    std::cout << "phase2_val: ";
    printVector(phase2_val, tot_operations);

    std::cout << " --- FINALLY --- " << "\n";
    std::cout << "resulting_row: ";
    printVector(resulting_row, tot_operations);
    std::cout << "resulting_col: ";
    printVector(resulting_col, tot_operations);
    std::cout << "resulting_val: ";
    printVector(resulting_val, tot_operations);

    std::cout << "--- COO FORMAT --- " << "\n";
    thrust::host_vector<int> coo_row(resulting_row, resulting_row + tot_operations);
    thrust::host_vector<int> coo_col(resulting_col, resulting_col + tot_operations);
    thrust::host_vector<float> coo_val(resulting_val, resulting_val + tot_operations);

    thrust::device_vector<int> d_coo_row = coo_row;
    thrust::device_vector<int> d_coo_col = coo_col;
    thrust::device_vector<float> d_coo_val = coo_val;

    typedef thrust::device_vector<int>::iterator IntIterator;
    typedef thrust::device_vector<float>::iterator FloatIterator;
```

```
    typedef thrust::tuple<IntIterator, IntIterator, FloatIterator> IteratorTuple;
    typedef thrust::zip_iterator<IteratorTuple> ZipIterator;

    ZipIterator first = thrust::make_zip_iterator(thrust::make_tuple(d_coo_row.begin(),
     d_coo_col.begin(), d_coo_val.begin()));
    ZipIterator last = thrust::make_zip_iterator(thrust::make_tuple(d_coo_row.end(),
     d_coo_col.end(), d_coo_val.end()));

    ZipIterator new_last = thrust::remove_if(first, last, removed_item());

    d_coo_row.erase(new_last.get_iterator_tuple().get<0>(), d_coo_row.end());
    d_coo_col.erase(new_last.get_iterator_tuple().get<1>(), d_coo_col.end());
    d_coo_val.erase(new_last.get_iterator_tuple().get<2>(), d_coo_val.end());

    thrust::host_vector<int> h_coo_row = d_coo_row;
    thrust::host_vector<int> h_coo_col = d_coo_col;
    thrust::host_vector<float> h_coo_val = d_coo_val;

    std::cout << "coo_row: ";
    print(h_coo_row);
    std::cout << "coo_col: ";
    print(h_coo_col);
    std::cout << "coo_val: ";
    print(h_coo_val);
    return 0;
}
```