Small Scale Parallel Programming Assignment

Aritra Das Ray s426160

February 2024

Contents

1 Introduction

1.1 Aim

The aim of the work described in this report is to explore Sparse Matrix and their Multiplication with Vectors of different sizes OpenMP and CUDA parallel programming paradigms and analyse their performance.

1.2 Objective

To Achieve our Aims we breakdown the tasks into the following steps:

1. Download matrices in Matrix market Format form https://sparse.tamu.edu/ .
2. Convert the matrices to CSR format
3. Generate vectors having varying columns.
4. Implement matrix vector multiplication in OpenMP and CUDA programming paradigms.
5. Measure time for execution of each operation to analyse performance.

2 Methodology

Sparse Matrix:
In the field of Machine Learning, working with large matrices is common. However, these matrices often contain a significant number of 0s, leading to what is known as a Sparse Matrix. As these 0s do not convey useful information and occupy considerable space, it becomes inefficient to compute them. To address this issue, various storage formats are employed to significantly reduce storage and computing needs. Depending on the distribution and quantity of non-zero entries, different data structures can be utilized for substantial memory savings compared to standard approaches. This trade-off requires more complex access to individual elements and necessitates additional structures for unambiguous recovery of the original matrix.[1]

Sparse Matrix Formats
Sparse Matrix can be stored in several different formats, following methods are the widely used formats:

- Dictionary of Keys (DOK)
- List of Lists
- Coordinated List
- Compressed Sparse Row (CSR)

- Compressed Sparse Column (CSC)

We will utilize the Compressed Sparse Row Method for our project as it allows for effective access and manipulation of matrices.[2]

CSR

The compressed row storage, also known as the compressed sparse row or Yale format, is a method for representing a matrix M using three one-dimensional arrays. These arrays store the nonzero values, row extents, and column indices of the matrix.[2]

The matrix M with dimensions m x n and nnz non-zero elements is represented using three one-dimensional arrays (V, column index, row index). The array V stores the non-zero values, the column index array stores their respective column positions and both arrays are of equal length. Meanwhile, the row index array has a length of m+1 and indicates the starting index of each row in V and the column index. The last element in the row index represents nnz which acts as a placeholder for proper loop iteration.[2]

Parallel Programming

Parallel programming involves executing multiple calculations or processes simultaneously to improve computational efficiency and reduce the time required for running complex tasks. This approach is particularly relevant in the context of CPU (Central Processing Unit) and GPU (Graphics Processing Unit) computing, where different architectures and capabilities are leveraged for optimal performance. Parallel programming with multi-threaded shared memory is a key strategy for maximizing the utilization of these hardware resources.[5]

CPU Parallel Programming

CPUs are general-purpose processors designed to handle a wide range of computing tasks. They are composed of a few cores with high clock speeds capable of executing a variety of operations. Parallel programming on CPUs often involves creating multiple threads within an application, where each thread can run concurrently on different cores, allowing for multitasking within the same application. This is achieved through the use of shared memory, where multiple threads can access and modify the same memory space, enabling efficient communication and data sharing among them.[4]

The main challenge in CPU parallel programming is managing the synchronization between threads to ensure data consistency and prevent race conditions, where two or more threads attempt to modify the same data simultaneously. Tools and libraries such as OpenMP (Open Multi-Processing) and Pthreads (POSIX threads) are commonly used for facilitating multi-threaded programming on CPUs by providing APIs for thread creation, synchronization, and management.[4]

GPU Parallel Programming

GPUs, on the other hand, are specialized processors designed to handle the

massive parallelism required for rendering graphics and video processing. Unlike CPUs, GPUs consist of thousands of smaller cores designed to perform simple calculations on large data sets simultaneously. This makes them highly efficient for algorithms that can be parallelized at a fine-grained level.
Parallel programming on GPUs involves dividing a task into small work items that can be executed simultaneously across many GPU cores. This is achieved through the use of a different programming model and memory hierarchy, including local (private), global, and shared memory spaces. Shared memory on a GPU is a small but fast memory accessible by threads within the same thread block, facilitating fast data exchange and synchronization among these threads.[3]
CUDA (Compute Unified Device Architecture) by NVIDIA and OpenCL (Open Computing Language) are popular frameworks for GPU programming, allowing developers to write programs that execute across both CPUs and GPUs. These frameworks provide the necessary tools for managing memory, defining parallel kernels (functions), and controlling thread execution.[3]

Multi-Threaded Shared Memory Model
The multi-threaded shared memory model in parallel programming exploits the shared memory architecture of both CPUs and GPUs to allow multiple threads to access common memory locations. This model is critical for achieving high performance in computing tasks that require frequent data exchange and synchronization between threads. Efficient use of shared memory can significantly reduce the latency associated with memory accesses and improve the overall speed of parallel algorithms.[9]
However, the efficient design of parallel programs that use shared memory requires careful consideration of memory access patterns, synchronization mechanisms, and the avoidance of bottlenecks that can arise from contention over shared resources. Techniques such as lock-free algorithms, atomic operations, and barrier synchronization are often employed to manage these challenges.[9]

3 Algorithm and Code Description

3.1 Main function

The general architecture of the "main" function is as follows:

- **Initialization:** The function start by defining a Sparse_CSR structure for storing the matrix in Compressed Sparse Row format. It also initialize a seed for random number generation, used later for vector generation.

- **Matrix Reading:** Then it reads a sparse matrix from the .mtx file using read_mtx_file function and stores it in CSR format within csr_matrix Struct.

- **Vector Generation:** Vectors of varying sizes (vec_k1, vec_k2, vec_k3, vec_k6) are generated using vec_gen. The sizes are based on the number of columns in the CSR matrix and a scaling factor (1, 2, 3, 6), signifying the number of columns of the vector with the third parameter as the seed for randomness.
- **Matrix Vector Multiplication:** The matrix is multiplied to the vectors generated in the previous steps using parallelization with OpenMP and CUDA and the results are stored in result_k1, result_k2, result_k3, result_k6. The OpenMP parallelization is achieved using the multiplication_openmp, while CUDA parallelization is achieved through multiplication_openmp.
- **Printing Results:** Results are printed by looping through the result arrays.
- **Freeing memory:** The memory is freed from csr_matrix, generated vectors and results of multiplication to prevent memory leaks.

3.2 Reading .MTX file and CSR conversion

The read_mtx_file function is designed to read a sparse matrix from a .mtx file and convert it into Compressed Sparse Row (CSR) format. This process involves several steps, each crucial for efficiently representing sparse matrices in memory. The function's operation can be summarized as follows:

Parameters

- **const char *filename:** A string that represents the path to the .mtx file to be read.
- **Sparse_CSR csr_matrix:** A pointer to a Sparse_CSR structure where the matrix in CSR format will be stored.

Process and Algorithms

1. **Opening the File:** Opens the .mtx file for reading. If the file cannot be opened, it reports an error and exits the program.
2. **Skipping Comments:** Reads lines from the file until it encounters a line that does not start with %, indicating the end of comments and the start of matrix dimensions.
3. **Reading Matrix Dimensions:** Reads the first non-comment line to extract the number of rows, columns, and non-zero elements in the matrix.
4. **Allocating Memory for COO:** Allocates memory for arrays to temporarily store the matrix in Coordinate (COO) format (row indices, column indices, and values of non-zero elements).

5. **Reading Matrix Entries:** Reads the file's entries (row index, column index, value) into the COO format arrays.

6. **Initializing CSR Structure:** Stores the read dimensions in the csr_matrix structure and allocates memory for CSR arrays (row_ptrs, col_indices, values).

7. **Converting to CSR Format:**

   - **Counting Non-Zeros per Row:** Iterates over the COO arrays to count the number of non-zero elements in each row, storing these counts in the row_ptrs array.
   - **Computing Row Pointer Indices:** Transforms the counts in row_ptrs into starting indices for each row in the CSR arrays.
   - **Filling CSR Arrays:** Iterates over the COO arrays again to fill the col_indices and values arrays, adjusting the row_ptrs indices as needed.

8. **Freeing Temporary Memory:** Releases the memory allocated for the COO format arrays.

Output
The function outputs by filling the passed Sparse_CSR structure with the matrix's CSR representation, including:

- The number of rows (n_row), columns (n_cols), and non-zero elements (n_nz).
- Arrays for the row pointers (row_ptrs), column indices (col_indices), and values of non-zero elements (values).

3.3 Generating Vectors based on input columns

The vec_gen function generates a matrix or vector with specified dimensions and fills it with random integer values. It's designed to create test vectors or matrices for operations such as matrix-vector multiplication, with the ability to control the randomness for reproducibility through a seed value. Here's a breakdown of its operation:

Parameters

- **int vec_row:** The number of rows in the vector or matrix to be generated.
- **int vec_col:** The number of columns in the vector or matrix. When vec_col is 1, the function generates a traditional vector. Otherwise, it generates a matrix.
- **unsigned int seed:** A seed value for the random number generator, ensuring reproducible results across different runs when the same seed is used.

Algorithm

1. **Initialization of Random Number Generator:** Uses the seed parameter to initialize the standard C library's random number generator with srand, ensuring that subsequent calls to rand produce a sequence of numbers that can be replicated by using the same seed.

2. **Memory Allocation:** Allocates memory for the vector or matrix based on the specified dimensions (vec_row * vec_col). It calculates the total number of elements and allocates enough space to store int values for each.

3. **Filling with Random Values:** Iterates over each element in the allocated space, assigning it a random integer value between 0 and 9 (inclusive). This is done by using the rand function and taking the remainder when divided by 10 (rand() % 10), ensuring that all values fall within the specified range.

4. **Error Handling:** Checks if memory allocation failed (if(vector == NULL)). If so, it returns NULL to indicate an error, allowing the calling function to handle the situation appropriately.

Output
Returns a pointer to the first element of the allocated and filled vector or matrix. This pointer can then be used in further computations or for debugging purposes (e.g., printing the vector/matrix).

3.4 Multiplication with OpenMP

The multiplication_openmp function performs matrix-vector multiplication using the Compressed Sparse Row (CSR) format for the matrix and supports parallel computation with OpenMP. This function is designed to work efficiently on multi-core processors by leveraging OpenMP to distribute the workload among multiple threads.

Parameters

- **const Sparse_CSR* csr_matrix:** A pointer to the sparse matrix in CSR format, which includes arrays for row pointers, column indices, and non-zero values, along with the matrix's dimensions.

- **const int* vector:** A pointer to the vector (or matrix when vec_col > 1) to be multiplied with the CSR matrix.

- **int vec_col:** The number of columns in the vector. This allows the function to support both traditional vector multiplication (when vec_col $= 1$) and matrix multiplication (when vec_col > 1).

Algorithm

1. **Memory Allocation for Result:** Allocates memory for the result of the multiplication. The size is determined by the number of rows in the CSR matrix and the number of columns in the vector.

2. **Parallel Computation:**

   - Retrieves the number of threads to be used from the environment variable OMP_NUM_THREADS and parses it to an integer.
   - Uses OpenMP directives (#pragma omp parallel for) to parallelize the outer loops over the vector's columns and the CSR matrix's rows.
   - Initializes private variables for each thread to avoid race conditions. These variables include loop indices and temporary variables for intermediate calculations.
   - Each thread calculates a portion of the result matrix, iterating through the non-zero elements of the CSR matrix, multiplying each by the corresponding vector value, and summing these products to form the result elements.

3. **OpenMP Barrier:** Ensures that all threads have completed their portion of the work before proceeding. This is implicitly provided by the end of the **#pragma omp parallel** for region.

Output
Returns a pointer to a dynamically allocated array of doubles that contains the result of the matrix-vector multiplication. The result is stored in row-major order if vec_col > 1, treating the vector as a matrix.

3.5 Multiplication with CUDA

The multiplication_cuda function performs matrix-vector multiplication leveraging CUDA for GPU acceleration, suitable for sparse matrices in Compressed Sparse Row (CSR) format. It demonstrates efficient GPU utilization for handling large-scale computations more quickly than CPU-based methods.
Parameters

- **const Sparse_CSR* csr_matrix:** A pointer to the sparse matrix in CSR format, which includes arrays for row pointers, column indices, and non-zero values, along with the matrix's dimensions.

- **const int* vector:** A pointer to the vector (or matrix when vec_col > 1) to be multiplied with the CSR matrix.

- **int vec_col:** The number of columns in the vector. This allows the function to support both traditional vector multiplication (when vec_col = 1) and matrix multiplication (when vec_col > 1).

Algorithm

1. **Memory Allocation for Result:** Allocates memory for the result of the multiplication. The size is determined by the number of rows in the CSR matrix and the number of columns in the vector.

2. **Memory Allocation on GPU:** Allocates GPU memory for the matrix's CSR components (d_row_ptr, d_col_ind, d_values), the input vector (d_vec), and the result vector (d_result).

3. **Data Transfer to GPU:** Copies the CSR matrix components and the input vector from host (CPU) memory to the allocated GPU memory.

4. **Kernel Initialization and Execution:**
   - Initializes CUDA events (start, stop) for timing the kernel execution.
   - Calculates the number of blocks and threads per block needed for the kernel launch, ensuring efficient GPU utilization.
   - Launches the matrix_vector_multiplication kernel, passing the necessary parameters for the computation.
   - Checks for kernel launch errors using cudaGetLastError.

5. **Data Transfer Back to Host:** Copies the result vector from GPU memory back to host memory.

6. **Timing and Cleanup:**
   - Records the elapsed time for the kernel execution using CUDA events.
   - Frees the allocated GPU memory to avoid memory leaks.

CUDA Kernel: matrix_vector_multiplication:

- Executed on the GPU, it calculates the product of the sparse matrix and the vector for each row. For matrices with multiple columns (vec_col > 1), it processes each column of the input vector in parallel.

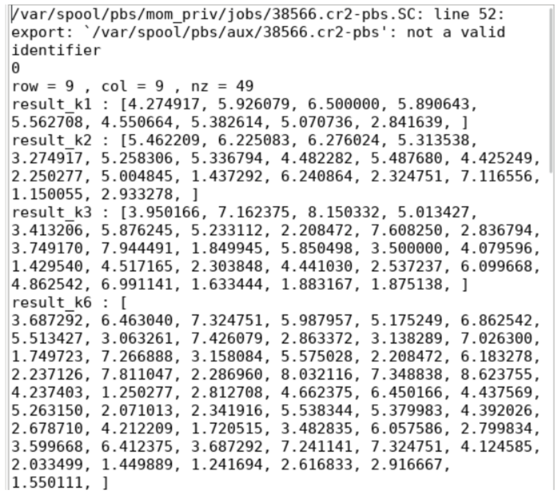
- Each thread computes one or more elements of the result vector, iterating over the non-zero elements of its assigned rows and calculating the dot product.

Output
Returns a pointer to the result vector (double*), which contains the outcome of the matrix-vector multiplication. The memory for the result vector is dynamically allocated and should be freed by the caller to prevent memory leaks.

4 Analysis

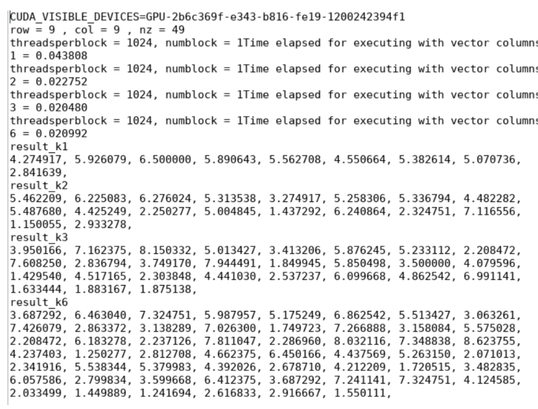
4.1 Funtional Test

For Functional testing, the results obtained from the OpenMP and CUDA code has been verified against an analytical solution. The analytical solution has been obtained from matrixcalc.org. The analytical Solution is shared in a pdf file along with the code files for verification Based on the figures it can be



```
/var/spool/pbs/mom_priv/jobs/38566.cr2-pbs.SC: line 52:
export: `/var/spool/pbs/aux/38566.cr2-pbs': not a valid
identifier
0
row = 9 , col = 9 , nz = 49
result_k1 : [4.274917, 5.926079, 6.500000, 5.890643,
5.562708, 4.550664, 5.382614, 5.070736, 2.841639, ]
result_k2 : [5.462209, 6.225083, 6.276024, 5.313538,
3.274917, 5.258306, 5.336794, 4.482282, 5.487680, 4.425249,
2.250277, 5.004845, 1.437292, 6.240864, 2.324751, 7.116556,
1.150055, 2.933278, ]
result_k3 : [3.950166, 7.162375, 8.150332, 5.013427,
3.413206, 5.876245, 5.233112, 2.208472, 7.608250, 2.836794,
3.749170, 7.944491, 1.849945, 5.850498, 3.500000, 4.079596,
1.429540, 4.517165, 2.303848, 4.441030, 2.537237, 6.099668,
4.862542, 6.991141, 1.633444, 1.883167, 1.875138, ]
result_k6 : [
3.687292, 6.463040, 7.324751, 5.987957, 5.175249, 6.862542,
5.513427, 3.063261, 7.426079, 2.863372, 3.138289, 7.026300,
1.749723, 7.266888, 3.158084, 5.575028, 2.208472, 6.183278,
2.237126, 7.811047, 2.286960, 8.032116, 7.348838, 8.623755,
4.237403, 1.250277, 2.812708, 4.662375, 6.450166, 4.437569,
5.263150, 2.071013, 2.341916, 5.538344, 5.379983, 4.392026,
2.678710, 4.212209, 1.720515, 3.482835, 6.057586, 2.799834,
3.599668, 6.412375, 3.687292, 7.241141, 7.324751, 4.124585,
2.033499, 1.449889, 1.241694, 2.616833, 2.916667,
1.550111, ]
```

Figure 1: Results from OpenMP code



```
CUDA_VISIBLE_DEVICES=GPU-2b6c369f-e343-b816-fe19-1200242394f1
row = 9 , col = 9 , nz = 49
threadsperblock = 1024, numblock = 1Time elapsed for executing with vector columns
1 = 0.043808
threadsperblock = 1024, numblock = 1Time elapsed for executing with vector columns
2 = 0.022752
threadsperblock = 1024, numblock = 1Time elapsed for executing with vector columns
3 = 0.020480
threadsperblock = 1024, numblock = 1Time elapsed for executing with vector columns
6 = 0.020992
result_k1
4.274917, 5.926079, 6.500000, 5.890643, 5.562708, 4.550664, 5.382614, 5.070736,
2.841639,
result_k2
5.462209, 6.225083, 6.276024, 5.313538, 3.274917, 5.258306, 5.336794, 4.482282,
5.487680, 4.425249, 2.250277, 5.004845, 1.437292, 6.240864, 2.324751, 7.116556,
1.150055, 2.933278,
result_k3
3.950166, 7.162375, 8.150332, 5.013427, 3.413206, 5.876245, 5.233112, 2.208472,
7.608250, 2.836794, 3.749170, 7.944491, 1.849945, 5.850498, 3.500000, 4.079596,
1.429540, 4.517165, 2.303848, 4.441030, 2.537237, 6.099668, 4.862542, 6.991141,
1.633444, 1.883167, 1.875138,
result_k6
3.687292, 6.463040, 7.324751, 5.987957, 5.175249, 6.862542, 5.513427, 3.063261,
7.426079, 2.863372, 3.138289, 7.026300, 1.749723, 7.266888, 3.158084, 5.575028,
2.208472, 6.183278, 2.237126, 7.811047, 2.286960, 8.032116, 7.348838, 8.623755,
4.237403, 1.250277, 2.812708, 4.662375, 6.450166, 4.437569, 5.263150, 2.071013,
2.341916, 5.538344, 5.379983, 4.392026, 2.678710, 4.212209, 1.720515, 3.482835,
6.057586, 2.799834, 3.599668, 6.412375, 3.687292, 7.241141, 7.324751, 4.124585,
2.033499, 1.449889, 1.241694, 2.616833, 2.916667, 1.550111,
```

Figure 2: Results from CUDA code

concluded that both the OpenMP and CUDA implementations give the same results. On comparison with the Analytical solution, it has been verified that the solutions are correct.

4.2 Performance Tests

The execution times of computing the matrix multiplication has been captured for 15 matrices. The data obtained has been shared along with the code files. However, Here are some key findings from the data:

1. For certain matrices and vector column configurations, CUDA shows either superior or comparable performance to OpenMP. This can be attributed to CUDA's efficient utilization of GPU resources, which can significantly accelerate computations for parallelizable tasks.
2. As the number of kernels increases from 1 to 8, there's a general trend of decreasing average execution time, suggesting that using more kernels can lead to faster execution times up to a certain point.
3. the reduction in execution time does not always occur linearly with the increase in the number of kernels, indicating that there might be an optimal number of kernels beyond which the performance gain diminishes or becomes inconsistent due to overhead or other factors.

References

[1] Jason Brownlee. A Gentle Introduction to Sparse Matrices for Machine Learning[Internet]. Guiding Tech Media; 2019 Aug 9 [cited 2024 Feb 25]. Available from: https://machinelearningmastery.com/sparse-matrices-for-machine-learning/

[2] Wikipedia contributors. Sparse matrix [Internet]. Wikipedia, The Free Encyclopedia; 2024 Feb 8, 18:29 [cited 2024 Feb 25]. Available from: https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=1205044030.

[3] NVIDIA. CUDA Quick Start Guide [Internet]. California, USA: NVIDIA; 2023 Nov 14 [cited 2024 Feb 25]. Available from: https://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html

[4] Wikipedia contributors. OpenMP [Internet]. Wikipedia, The Free Encyclopedia; 2024 Jan 28, 06:11 UTC [cited 2024 Feb 25]. Available from: https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1199889263.

[5] Wikipedia contributors. Parallel computing [Internet]. Wikipedia, The Free Encyclopedia; 2024 Feb 1, 00:45 UTC [cited 2024 Feb 25]. Available from: https://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=1201602893.

[6] Szorbasc. mat_mult.cu in CUDA-CSR-matrix-vector multiplication [Internet]. GitHub; [cited 2024 Feb 25]. Available from: https://github.com/szorbasc/CUDA-CSR-matrix-vector-multiplication/blob/master/mat_mult.cu

[7] Szorbasc. mat_mult.c in OpenMP-CSR-matrix-vector multiplication [Internet]. GitHub; [cited 2024 Feb 25]. Available from: https://github.com/szorbasc/OpenMP-CSR-matrix-vector-multiplication/blob/master/mat_mult.c

[8] OpenMP Architecture Review Board. OpenMP Application Programming Interface. [Internet]. Gemini Dr, USA: 2024 Jan 28, 06:11 UTC [cited 2024 Feb 25]. Available from: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[9] Herlihy M, Shavit N. The Art of Multiprocessor Programming. San Francisco: Morgan Kaufmann; 2011.[cited 2024 Feb 25]

A Appendix

A.1 C Scripts

A.1.1 Matrix Vector Multiplication CUDA

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <cuda_runtime.h>

typedef struct Sparse_CSR {
    int n_row;
    int n_cols;
    int n_nz;
    int* row_ptrs;
    int* col_indices;
    double* values;
} Sparse_CSR;

void read_mtx_file(const char *filename, Sparse_CSR *csr_matrix){

  //open file
  FILE *file = fopen(filename, "r");
  if(!file) {
    perror("Unable to open file!");
    exit(EXIT_FAILURE);
  }

  //skip comments
  char line[256];
  while (fgets(line, sizeof(line), file)) {
          if (line[0] != '%') break;
      }

  //initialize variable to store coo matrix from file
  int coo_rown, coo_coln, coo_nnz;

  //input no of rows, columns and non-zeros
  sscanf(line, "%d %d %d", &coo_rown, &coo_coln, &coo_nnz);
  printf("row = %d , col = %d , nz = %d\n", coo_rown, coo_coln,
      coo_nnz);

  //allocate memory to store the coo values
  int* coo_rows  = (int *)malloc(coo_nnz * sizeof(int));
  int* coo_cols = (int *)malloc(coo_nnz * sizeof(int));
  double* coo_val = (double *)malloc(coo_nnz * sizeof(double));

  if (!coo_rows || !coo_cols || !coo_val){
    perror("Memory Allocation failed!");
    fclose(file);
    exit(EXIT_FAILURE);
  }

  for(int i = 0; i < coo_nnz ; i++){
```

```
    fscanf(file, "%d %d %lf" , &coo_rows[i], &coo_cols[i], &coo_val[i]);
  }

  //variable assignment
  csr_matrix->n_row = coo_rown;
  csr_matrix->n_cols = coo_coln;
  csr_matrix->n_nz = coo_nnz;


  // Allocate CSR format arrays
  csr_matrix->row_ptrs = (int *)malloc((coo_rown + 1) * sizeof(int));
  memset(csr_matrix->row_ptrs, 0, (coo_rown + 1) * sizeof(int)); // Initialize to 0

  csr_matrix->col_indices = (int *)malloc(coo_nnz * sizeof(int));
  csr_matrix->values = (double *)malloc(coo_nnz * sizeof(double));

  // Step 1: Count non-zero elements per row
  for (int i = 0; i < coo_nnz; i++) {
    csr_matrix->row_ptrs[coo_rows[i]]++;
  }

  // Convert counts to starting indices
      int sum = 0;
      for (int i = 0; i <= coo_rown; i++) {
          int temp = csr_matrix->row_ptrs[i];
          csr_matrix->row_ptrs[i] = sum;
          sum += temp;
  }

  // Step 3: Fill CSR format data
  for (int i = 0; i < coo_nnz; i++) {
          int row = coo_rows[i];
          int dest = csr_matrix->row_ptrs[row];

          csr_matrix->col_indices[dest] = coo_cols[i];
          csr_matrix->values[dest] = coo_val[i];

          csr_matrix->row_ptrs[row]++;
  }


  //free temporary allocations
  free(coo_rows);
  free(coo_cols);
  free(coo_val);

}

int print_sparse_csr(const Sparse_CSR* csr_matrix) {
    printf("nz_id\trow\tcol\tmat_val\n");
    printf("----\n");

        for (size_t i=0; i<csr_matrix->n_row; ++i) {
            size_t nz_start = csr_matrix->row_ptrs[i];
```

```
            size_t nz_end = csr_matrix->row_ptrs[i+1];
            for (size_t nz_id=nz_start; nz_id<nz_end; ++nz_id) {
                size_t j = csr_matrix->col_indices[nz_id];
                double val = csr_matrix->values[nz_id];
                printf("%d\t%d\t%d\t%lf\n",nz_id, i, j, val);
            }
        }

    return EXIT_SUCCESS;
}

int free_sparse_csr(Sparse_CSR* csr_matrix) {
    free(csr_matrix->row_ptrs);
    free(csr_matrix->col_indices);
    free(csr_matrix->values);

    return EXIT_SUCCESS;
}

int* vec_gen(int vec_row, int vec_col, unsigned int seed){
  srand(seed);

  int* vector = (int*)malloc(vec_row * vec_col * sizeof(int));

  if(vector == NULL){
    return NULL;
  }

  for (int i = 0; i < vec_row; i++){
    for (int j = 0; j < vec_col; j++){
      vector[i * vec_col + j] = rand()%10;
    }
  }

  return vector;
}

double* multiplication_cuda(const Sparse_CSR* csr_matrix, const int * vector, int vec_col);

__global__ void matrix_vector_multiplication(const int num_rows, const int vec_col, const int *row_ptrs, const int *col_indices, const double *mat_val, const int *vec_val, double* res);


int main(int argc, char** argv) {
  Sparse_CSR csr_matrix;
  unsigned int seed = 1;

  read_mtx_file( "cage4.mtx" , &csr_matrix);

  //printf("Printing CSR Matrix after storing it");
  //print_sparse_csr(&csr_matrix);

  int* vec_k1 = vec_gen(csr_matrix.n_cols, 1, seed);
  int* vec_k2 = vec_gen(csr_matrix.n_cols, 2, seed);
  int* vec_k3 = vec_gen(csr_matrix.n_cols, 3, seed);
```

```c
  int* vec_k6 = vec_gen(csr_matrix.n_cols, 6, seed);

  double* result_k1 = multiplication_cuda(&csr_matrix, vec_k1, 1);
  double* result_k2 = multiplication_cuda(&csr_matrix, vec_k2, 2);
  double* result_k3 = multiplication_cuda(&csr_matrix, vec_k3, 3);
  double* result_k6 = multiplication_cuda(&csr_matrix, vec_k6, 6);

  printf("result_k1 \n");
  for(int i = 0; i < (csr_matrix.n_row * 1) ; i++){
    printf("%lf, " , result_k1[i]);
  }
  printf("\n");

  printf("result_k2 \n");
  for(int i = 0; i < (csr_matrix.n_row * 2) ; i++){
    printf("%lf, " , result_k2[i]);
  }
  printf("\n");

  printf("result_k3 \n");
  for(int i = 0; i < (csr_matrix.n_row * 3) ; i++){
    printf("%lf, " , result_k3[i]);
  }
  printf("\n");

  printf("result_k6 \n");
  for(int i = 0; i < (csr_matrix.n_row * 6) ; i++){
    printf("%lf, " , result_k6[i]);
  }
  printf("\n");

  free_sparse_csr(&csr_matrix);
  free(vec_k1);
  free(vec_k2);
  free(vec_k3);
  free(vec_k6);
  free(result_k1);
  free(result_k2);
  free(result_k3);
  free(result_k6);

  return EXIT_SUCCESS;
}

double* multiplication_cuda(const Sparse_CSR* csr_matrix, const int * vector, int vec_col){
  double* result = (double*)calloc(csr_matrix->n_row * vec_col , sizeof(double));

  if (result == NULL) {
          fprintf(stderr, "Failed to allocate memory for result\n");
          exit(EXIT_FAILURE);
  }

  //variable intialization
  int *d_row_ptr, *d_col_ind, *d_vec;
```

```
  double *d_values, *d_result;

// Allocating kernel memory and checking for errors
  cudaError_t err;

  //allocating kernal memory
  cudaMalloc(&d_row_ptr, (csr_matrix->n_row+1) * sizeof(int));
  cudaMalloc(&d_col_ind, csr_matrix->n_nz * sizeof(int));
  cudaMalloc(&d_values, csr_matrix->n_nz * sizeof(double));
  cudaMalloc(&d_vec, (csr_matrix->n_cols*vec_col) * sizeof(int));
  cudaMalloc(&d_result, (csr_matrix->n_row*vec_col) * sizeof(double));

  //moving data from CPU to GPU
  cudaMemcpy(d_row_ptr, csr_matrix->row_ptrs, (csr_matrix->n_row+1)*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_col_ind, csr_matrix->col_indices, csr_matrix->n_nz*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_values, csr_matrix->values, csr_matrix->n_nz*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(d_vec, vector, (csr_matrix->n_cols*vec_col)*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_result, result, (csr_matrix->n_row*vec_col)*sizeof(double), cudaMemcpyHostToDevice);

  //cuda event intialization
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start);
  //initializing kernel variables
  int threadsperblock  = 1024;
  int numblock = ceil(csr_matrix->n_row/threadsperblock);

  if( ceil(csr_matrix->n_row/threadsperblock) == 0) {
    numblock = 1;
  } else {
    numblock = ceil(csr_matrix->n_row/threadsperblock);
  }


  printf("threadsperblock = %d, numblock = %d", threadsperblock, numblock);

  //launching kernels

  matrix_vector_multiplication<<< numblock , threadsperblock >>>(csr_matrix->n_row, vec_col, d_row_ptr, d_col_ind, d_values, d_vec, d_result);
  // Check for kernel launch errors
  err = cudaGetLastError();
  if (err != cudaSuccess) {
          fprintf(stderr, "Failed to launch matrix_vector_multiplication kernel: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
```

```
  }

  cudaMemcpy(result, d_result, (csr_matrix->n_row*vec_col)*sizeof(
      double), cudaMemcpyDeviceToHost);

  cudaEventRecord(stop);
  cudaEventSynchronize(stop);

  cudaFree(d_row_ptr);
  cudaFree(d_col_ind);
  cudaFree(d_values);
  cudaFree(d_vec);
  cudaFree(d_result);

  return result;
}


__global__ void matrix_vector_multiplication(const int num_rows,
    const int vec_col, const int *row_ptrs, const int *col_indices,
     const double *mat_val, const int *vec_val, double* res){
  int row = blockIdx.x * blockDim.x + threadIdx.x;

  //printf("row = %d , col = %d, num_rows = %d", row, col, num_rows
      );

  int i, row_start, row_end;
  double dot;

  if (row < num_rows){
    for(int k = 0; k<vec_col; k++){
      dot = 0.0;
      row_start = row_ptrs[row];
      row_end = row_ptrs[row+1];
      for(i = row_start; i<row_end ; i++){
        dot += mat_val[i] * vec_val[(col_indices[i]-1)*vec_col+k];
      }
      res[row*vec_col+k] = dot;
    }
  }
}
```

Listing 1: mat_vec_multi_cuda.c

A.1.2 Matrix Vector Multiplication CUDA Performance

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <cuda_runtime.h>

typedef struct Sparse_CSR {
    int n_row;
    int n_cols;
```

```
    int n_nz;
    int* row_ptrs;
    int* col_indices;
    double* values;
} Sparse_CSR;

void read_mtx_file(const char *filename, Sparse_CSR *csr_matrix){

  //open file
  FILE *file = fopen(filename, "r");
  if(!file) {
    perror("Unable to open file!");
    exit(EXIT_FAILURE);
  }

  //skip comments
  char line[256];
  while (fgets(line, sizeof(line), file)) {
          if (line[0] != '%') break;
      }

  //initialize variable to store coo matrix from file
  int coo_rown, coo_coln, coo_nnz;

  //input no of rows, columns and non-zeros
  sscanf(line, "%d %d %d", &coo_rown, &coo_coln, &coo_nnz);
  printf("row = %d , col = %d , nz = %d\n", coo_rown, coo_coln, coo_nnz);

  //allocate memory to store the coo values
  int* coo_rows  = (int *)malloc(coo_nnz * sizeof(int));
  int* coo_cols = (int *)malloc(coo_nnz * sizeof(int));
  double* coo_val = (double *)malloc(coo_nnz * sizeof(double));

  if (!coo_rows || !coo_cols || !coo_val){
    perror("Memory Allocation failed!");
    fclose(file);
    exit(EXIT_FAILURE);
  }

  for(int i = 0; i < coo_nnz ; i++){
    fscanf(file, "%d %d %lf" , &coo_rows[i], &coo_cols[i], &coo_val[i]);
  }

  //variable assignment
  csr_matrix->n_row = coo_rown;
  csr_matrix->n_cols = coo_coln;
  csr_matrix->n_nz = coo_nnz;


  // Allocate CSR format arrays
  csr_matrix->row_ptrs = (int *)malloc((coo_rown + 1) * sizeof(int));
  memset(csr_matrix->row_ptrs, 0, (coo_rown + 1) * sizeof(int)); // Initialize to 0

```

```
  csr_matrix->col_indices = (int *)malloc(coo_nnz * sizeof(int));
  csr_matrix->values = (double *)malloc(coo_nnz * sizeof(double));

  // Step 1: Count non-zero elements per row
  for (int i = 0; i < coo_nnz; i++) {
    csr_matrix->row_ptrs[coo_rows[i]]++;
  }

  // Convert counts to starting indices
      int sum = 0;
      for (int i = 0; i <= coo_rown; i++) {
          int temp = csr_matrix->row_ptrs[i];
          csr_matrix->row_ptrs[i] = sum;
          sum += temp;
  }

  // Step 3: Fill CSR format data
  for (int i = 0; i < coo_nnz; i++) {
          int row = coo_rows[i];
          int dest = csr_matrix->row_ptrs[row];

          csr_matrix->col_indices[dest] = coo_cols[i]-1;
          csr_matrix->values[dest] = coo_val[i];

          csr_matrix->row_ptrs[row]++;
  }


  //free temporary allocations
  free(coo_rows);
  free(coo_cols);
  free(coo_val);

}

int print_sparse_csr(const Sparse_CSR* csr_matrix) {
    printf("nz_id\trow\tcol\tmat_val\n");
    printf("----\n");

        for (size_t i=0; i<csr_matrix->n_row; ++i) {
            size_t nz_start = csr_matrix->row_ptrs[i];
            size_t nz_end = csr_matrix->row_ptrs[i+1];
            for (size_t nz_id=nz_start; nz_id<nz_end; ++nz_id) {
                size_t j = csr_matrix->col_indices[nz_id];
                double val = csr_matrix->values[nz_id];
                printf("%d\t%d\t%d\t%lf\n",nz_id, i, j, val);
            }
        }

    return EXIT_SUCCESS;
}

int free_sparse_csr(Sparse_CSR* csr_matrix) {
    free(csr_matrix->row_ptrs);
    free(csr_matrix->col_indices);
    free(csr_matrix->values);

```

```c
    return EXIT_SUCCESS;
}

int* vec_gen(int vec_row, int vec_col, unsigned int seed){
  srand(seed);

  int* vector = (int*)malloc(vec_row * vec_col * sizeof(int));

  if(vector == NULL){
    return NULL;
  }

  for (int i = 0; i < vec_row; i++){
    for (int j = 0; j < vec_col; j++){
      vector[i * vec_col + j] = rand()%10;
    }
  }

  return vector;
}

double* multiplication_cuda(const Sparse_CSR* csr_matrix, const int * vector, int vec_col);

__global__ void matrix_vector_multiplication(const int num_rows, const int vec_col, const int *row_ptrs, const int *col_indices, const double *mat_val, const int *vec_val, double* res);


int main(int argc, char** argv) {
  Sparse_CSR csr_matrix;
  unsigned int seed = 1;

  read_mtx_file( "mhd4800a.mtx" , &csr_matrix);

  //printf("Printing CSR Matrix after storing it");
  //print_sparse_csr(&csr_matrix);

  int* vec_k1 = vec_gen(csr_matrix.n_cols, 1, seed);
  int* vec_k2 = vec_gen(csr_matrix.n_cols, 2, seed);
  int* vec_k3 = vec_gen(csr_matrix.n_cols, 3, seed);
  int* vec_k6 = vec_gen(csr_matrix.n_cols, 6, seed);

  double* result_k1 = multiplication_cuda(&csr_matrix, vec_k1, 1);
  double* result_k2 = multiplication_cuda(&csr_matrix, vec_k2, 2);
  double* result_k3 = multiplication_cuda(&csr_matrix, vec_k3, 3);
  double* result_k6 = multiplication_cuda(&csr_matrix, vec_k6, 6);


  free_sparse_csr(&csr_matrix);
  free(vec_k1);
  free(vec_k2);
  free(vec_k3);
  free(vec_k6);
  free(result_k1);
  free(result_k2);
  free(result_k3);
```

```
  free(result_k6);

  return EXIT_SUCCESS;
}

double* multiplication_cuda(const Sparse_CSR* csr_matrix, const int * vector, int vec_col){
  double* result = (double*)calloc(csr_matrix->n_row * vec_col , sizeof(double));

  if (result == NULL) {
          fprintf(stderr, "Failed to allocate memory for result\n");
          exit(EXIT_FAILURE);
  }

  //variable intialization
  int *d_row_ptr, *d_col_ind, *d_vec;
  double *d_values, *d_result;

// Allocating kernel memory and checking for errors
  cudaError_t err;

  //allocating kernal memory
  cudaMalloc(&d_row_ptr, (csr_matrix->n_row+1) * sizeof(int));
  cudaMalloc(&d_col_ind, csr_matrix->n_nz * sizeof(int));
  cudaMalloc(&d_values, csr_matrix->n_nz * sizeof(double));
  cudaMalloc(&d_vec, (csr_matrix->n_cols*vec_col) * sizeof(int));
  cudaMalloc(&d_result, (csr_matrix->n_row*vec_col) * sizeof(double));

  //moving data from CPU to GPU
  cudaMemcpy(d_row_ptr, csr_matrix->row_ptrs, (csr_matrix->n_row+1)*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_col_ind, csr_matrix->col_indices, csr_matrix->n_nz*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_values, csr_matrix->values, csr_matrix->n_nz*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(d_vec, vector, (csr_matrix->n_cols*vec_col)*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_result, result, (csr_matrix->n_row*vec_col)*sizeof(double), cudaMemcpyHostToDevice);

  //cuda event intialization
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start);
  //initializing kernel variables
  int threadsperblock  = 1024;
  int numblock = ceil(csr_matrix->n_row/threadsperblock);

  if( ceil(csr_matrix->n_row/threadsperblock) == 0) {
    numblock = 1;
  } else {
    numblock = ceil(csr_matrix->n_row/threadsperblock);
```

```
  }


  //printf("threadsperblock = %d, numblock = %d", threadsperblock, numblock);

  //launching kernels

  matrix_vector_multiplication<<< numblock , threadsperblock >>>(csr_matrix->n_row, vec_col, d_row_ptr, d_col_ind, d_values, d_vec, d_result);
  // Check for kernel launch errors
  err = cudaGetLastError();
  if (err != cudaSuccess) {
          fprintf(stderr, "Failed to launch matrix_vector_multiplication kernel: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }

  cudaMemcpy(result, d_result, (csr_matrix->n_row*vec_col)*sizeof(double), cudaMemcpyDeviceToHost);

  cudaEventRecord(stop);
  cudaEventSynchronize(stop);

  float time_elapsed;
  cudaEventElapsedTime(&time_elapsed, start, stop);
  printf("Time elapsed for executing with vector columns %d = %lf\n", vec_col, time_elapsed);

  cudaFree(d_row_ptr);
  cudaFree(d_col_ind);
  cudaFree(d_values);
  cudaFree(d_vec);
  cudaFree(d_result);

  return result;
}


__global__ void matrix_vector_multiplication(const int num_rows, const int vec_col, const int *row_ptrs, const int *col_indices, const double *mat_val, const int *vec_val, double* res){
  int row = blockIdx.x * blockDim.x + threadIdx.x;

  //printf("row = %d , col = %d, num_rows = %d", row, col, num_rows);

  int i, row_start, row_end;
  double dot;

  if (row < num_rows){
    for(int k = 0; k<vec_col; k++){
      dot = 0.0;
      row_start = row_ptrs[row];
      row_end = row_ptrs[row+1];
```

```
      for(i = row_start; i<row_end ; i++){
        dot += mat_val[i] * vec_val[(col_indices[i])*vec_col+k];
      }
      res[row*vec_col+k] = dot;
    }
  }
}
```

Listing 2: mat_vec_multi_cuda_performance.c

A.1.3 Matrix Vector Multiplication OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <omp.h>

typedef struct Sparse_CSR {
    int n_row;
    int n_cols;
    int n_nz;
    int* row_ptrs;
    int* col_indices;
    double* values;
} Sparse_CSR;

void read_mtx_file(const char *filename, Sparse_CSR *csr_matrix){

  //open file
  FILE *file = fopen(filename, "r");
  if(!file) {
    perror("Unable to open file!");
    exit(EXIT_FAILURE);
  }

  //skip comments
  char line[256];
  while (fgets(line, sizeof(line), file)) {
            if (line[0] != '%') break;
      }

  //initialize variable to store coo matrix from file
  int coo_rown, coo_coln, coo_nnz;

  //input no of rows, columns and non-zeros
  sscanf(line, "%d %d %d", &coo_rown, &coo_coln, &coo_nnz);
  printf("row = %d , col = %d , nz = %d\n", coo_rown, coo_coln,
      coo_nnz);

  //allocate memory to store the coo values
  int* coo_rows  = (int *)malloc(coo_nnz * sizeof(int));
  int* coo_cols = (int *)malloc(coo_nnz * sizeof(int));
  double* coo_val = (double *)malloc(coo_nnz * sizeof(double));

```

```
  if (!coo_rows || !coo_cols || !coo_nnz){
    perror("Memory Allocation failed!");
    fclose(file);
    exit(EXIT_FAILURE);
  }

  for(int i = 0; i < coo_nnz ; i++){
    fscanf(file, "%d %d %lf" , &coo_rows[i], &coo_cols[i], &coo_val[i]);
  }

  //variable assignment
  csr_matrix->n_row = coo_rown;
  csr_matrix->n_cols = coo_coln;
  csr_matrix->n_nz = coo_nnz;


  // Allocate CSR format arrays
  csr_matrix->row_ptrs = (int *)malloc((coo_rown + 1) * sizeof(int));
  memset(csr_matrix->row_ptrs, 0, (coo_rown + 1) * sizeof(int)); // Initialize to 0

  csr_matrix->col_indices = (int *)malloc(coo_nnz * sizeof(int));
  csr_matrix->values = (double *)malloc(coo_nnz * sizeof(double));

  // Step 1: Count non-zero elements per row
  for (int i = 0; i < coo_nnz; i++) {
    csr_matrix->row_ptrs[coo_rows[i]]++;
  }

  // Convert counts to starting indices
      int sum = 0;
      for (int i = 0; i <= coo_rown; i++) {
          int temp = csr_matrix->row_ptrs[i];
          csr_matrix->row_ptrs[i] = sum;
          sum += temp;
  }

  // Step 3: Fill CSR format data
  for (int i = 0; i < coo_nnz; i++) {
          int row = coo_rows[i];
          int dest = csr_matrix->row_ptrs[row];

          csr_matrix->col_indices[dest] = coo_cols[i]-1;
          csr_matrix->values[dest] = coo_val[i];

          csr_matrix->row_ptrs[row]++;
  }


  //free temporary allocations
  free(coo_rows);
  free(coo_cols);
  free(coo_val);

}
```

```
int print_sparse_csr(const Sparse_CSR* csr_matrix) {
    printf("nz_id\trow\tcol\tmat_val\n");
    printf("----\n");

        for (size_t i=0; i<csr_matrix->n_row; ++i) {
            size_t nz_start = csr_matrix->row_ptrs[i];
            size_t nz_end = csr_matrix->row_ptrs[i+1];
            for (size_t nz_id=nz_start; nz_id<nz_end; ++nz_id) {
                size_t j = csr_matrix->col_indices[nz_id];
                double val = csr_matrix->values[nz_id];
                printf("%d\t%d\t%d\t%lf\n",nz_id, i, j, val);
            }
        }

    return EXIT_SUCCESS;
}

int free_sparse_csr(Sparse_CSR* csr_matrix) {
    free(csr_matrix->row_ptrs);
    free(csr_matrix->col_indices);
    free(csr_matrix->values);

    return EXIT_SUCCESS;
}

int* vec_gen(int vec_row, int vec_col, unsigned int seed){
  srand(seed);

  int* vector = (int*)malloc(vec_row * vec_col * sizeof(int));

  if(vector == NULL){
    return NULL;
  }

  for (int i = 0; i < vec_row; i++){
    for (int j = 0; j < vec_col; j++){
      vector[i * vec_col + j] = rand()%10;
    }
  }

  return vector;
}

double* multiplication_openmp(const Sparse_CSR* csr_matrix, const
    int* vector, int vec_col);

int main(int argc, char** argv) {
  Sparse_CSR csr_matrix;
  unsigned int seed = 1;

  read_mtx_file("cage4.mtx" , &csr_matrix);

  //printf("Printing CSR Matrix after storing it");
  //print_sparse_csr(&csr_matrix);
  //printf("-------------------------------------------------");

```

```c
  int* vec_k1 = vec_gen(csr_matrix.n_cols, 1, seed);
  int* vec_k2 = vec_gen(csr_matrix.n_cols, 2, seed);
  int* vec_k3 = vec_gen(csr_matrix.n_cols, 3, seed);
  int* vec_k6 = vec_gen(csr_matrix.n_cols, 6, seed);

  double* result_k1 = multiplication_openmp(&csr_matrix, vec_k1, 1);
  double* result_k2 = multiplication_openmp(&csr_matrix, vec_k2, 2);
  double* result_k3 = multiplication_openmp(&csr_matrix, vec_k3, 3);
  double* result_k6 = multiplication_openmp(&csr_matrix, vec_k6, 6);
/*
  printf("vec_k1 : [");
  for(int i = 0; i < (csr_matrix.n_cols * 1) ; i++){
    printf("%d, " , vec_k1[i]);
  }
  printf("]\n");

  printf("vec_k2 : [");
  for(int i = 0; i < (csr_matrix.n_cols * 2) ; i++){
    printf("%d, " , vec_k2[i]);
  }
  printf("]\n");

  printf("vec_k3 : [");
  for(int i = 0; i < (csr_matrix.n_cols * 3) ; i++){
    printf("%d, " , vec_k3[i]);
  }
  printf("]\n");

  printf("vec_k6 : [");
  for(int i = 0; i < (csr_matrix.n_cols * 6) ; i++){
    printf("%d, " , vec_k6[i]);
  }
  printf("]\n");
  */
  //printf("-----------------------------------------------------------\n");

  printf("result_k1 : [");
  for(int i = 0; i < (csr_matrix.n_row * 1) ; i++){
    printf("%lf, " , result_k1[i]);
  }
  printf("]\n");

  printf("result_k2 : [");
  for(int i = 0; i < (csr_matrix.n_row * 2) ; i++){
    printf("%lf, " , result_k2[i]);
  }
  printf("]\n");

  printf("result_k3 : [");
  for(int i = 0; i < (csr_matrix.n_row * 3) ; i++){
    printf("%lf, " , result_k3[i]);
```

```c
  }
  printf("]\n");

  printf("result_k6 : [\n");
  for(int i = 0; i < (csr_matrix.n_row * 6) ; i++){
    printf("%lf, " , result_k6[i]);
  }
  printf("]\n");

  free_sparse_csr(&csr_matrix);
  free(vec_k1);
  free(vec_k2);
  free(vec_k3);
  free(vec_k6);
  free(result_k1);
  free(result_k2);
  free(result_k3);
  free(result_k6);

  return EXIT_SUCCESS;
}

double* multiplication_openmp(const Sparse_CSR* csr_matrix, const
    int* vector, int vec_col){
  double* result = (double *)malloc(csr_matrix->n_row * vec_col *
      sizeof(double));

  if(result == NULL){
    perror("Failed to allocate memory for result");
    exit(EXIT_FAILURE);
  }

  int i, j, k, nz_start, nz_end, nz_id, vec_val;
  double mat_val, temp;
  //printf("k\ti\tnz_id\tj\tmat_val\tvec_val\ttemp\n");
  #pragma omp parallel for private (i, j, k, nz_start, nz_end,
      nz_id, mat_val, vec_val, temp)
  for(k = 0; k<vec_col; k++){
    for(i=0 ; i < csr_matrix->n_row; i++){
      temp = 0.0;
      nz_start = csr_matrix->row_ptrs[i];
      nz_end = csr_matrix->row_ptrs[i+1];
      for(nz_id = nz_start ; nz_id < nz_end ; nz_id++){
        j = csr_matrix->col_indices[nz_id];
        mat_val = csr_matrix->values[nz_id];
        vec_val = vector[(j)*vec_col + k];
        temp += (mat_val*vec_val);
        //printf("%d\t%d\t%d\t%d\t%lf\t%d\t%lf\n ", k, i , nz_id, j
            , mat_val, vec_val, temp);
      }
      result[i*vec_col+k] = temp;
      //printf("result[%d] = %lf", (i*vec_col+k), result[i*vec_col+
          k]);
    }
  }

  return result;
```

```
}
```

Listing 3: mat_vec_multi_openmp.c

A.1.4 Matrix Vector Multiplication OpenMP Performance

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <omp.h>

typedef struct Sparse_CSR {
    int n_row;
    int n_cols;
    int n_nz;
    int* row_ptrs;
    int* col_indices;
    double* values;
} Sparse_CSR;

void read_mtx_file(const char *filename, Sparse_CSR *csr_matrix){

  //open file
  FILE *file = fopen(filename, "r");
  if(!file) {
    perror("Unable to open file!");
    exit(EXIT_FAILURE);
  }

  //skip comments
  char line[256];
  while (fgets(line, sizeof(line), file)) {
          if (line[0] != '%') break;
      }

  //initialize variable to store coo matrix from file
  int coo_rown, coo_coln, coo_nnz;

  //input no of rows, columns and non-zeros
  sscanf(line, "%d %d %d", &coo_rown, &coo_coln, &coo_nnz);
  printf("row = %d , col = %d , nz = %d\n", coo_rown, coo_coln,
      coo_nnz);

  //allocate memory to store the coo values
  int* coo_rows  = (int *)malloc(coo_nnz * sizeof(int));
  int* coo_cols = (int *)malloc(coo_nnz * sizeof(int));
  double* coo_val = (double *)malloc(coo_nnz * sizeof(double));

  if (!coo_rows || !coo_cols || !coo_nnz){
    perror("Memory Allocation failed!");
    fclose(file);
    exit(EXIT_FAILURE);
  }

```

```
  for(int i = 0; i < coo_nnz ; i++){
    fscanf(file, "%d %d %lf" , &coo_rows[i], &coo_cols[i], &coo_val
        [i]);
  }

  //variable assignment
  csr_matrix->n_row = coo_rown;
  csr_matrix->n_cols = coo_coln;
  csr_matrix->n_nz = coo_nnz;


  // Allocate CSR format arrays
  csr_matrix->row_ptrs = (int *)malloc((coo_rown + 1) * sizeof(int)
      );
  memset(csr_matrix->row_ptrs, 0, (coo_rown + 1) * sizeof(int)); //
       Initialize to 0

  csr_matrix->col_indices = (int *)malloc(coo_nnz * sizeof(int));
  csr_matrix->values = (double *)malloc(coo_nnz * sizeof(double));

  // Step 1: Count non-zero elements per row
  for (int i = 0; i < coo_nnz; i++) {
    csr_matrix->row_ptrs[coo_rows[i]]++;
  }

  // Convert counts to starting indices
      int sum = 0;
      for (int i = 0; i <= coo_rown; i++) {
          int temp = csr_matrix->row_ptrs[i];
          csr_matrix->row_ptrs[i] = sum;
          sum += temp;
  }

  // Step 3: Fill CSR format data
  for (int i = 0; i < coo_nnz; i++) {
          int row = coo_rows[i];
          int dest = csr_matrix->row_ptrs[row];

          csr_matrix->col_indices[dest] = coo_cols[i]-1;
          csr_matrix->values[dest] = coo_val[i];

          csr_matrix->row_ptrs[row]++;
  }


  //free temporary allocations
  free(coo_rows);
  free(coo_cols);
  free(coo_val);

}

int print_sparse_csr(const Sparse_CSR* csr_matrix) {
    printf("nz_id\trow\tcol\tmat_val\n");
    printf("----\n");

        for (size_t i=0; i<csr_matrix->n_row; ++i) {
```

```c
            size_t nz_start = csr_matrix->row_ptrs[i];
            size_t nz_end = csr_matrix->row_ptrs[i+1];
            for (size_t nz_id=nz_start; nz_id<nz_end; ++nz_id) {
                size_t j = csr_matrix->col_indices[nz_id];
                double val = csr_matrix->values[nz_id];
                printf("%d\t%d\t%d\t%lf\n",nz_id, i, j, val);
            }
        }

    return EXIT_SUCCESS;
}

int free_sparse_csr(Sparse_CSR* csr_matrix) {
    free(csr_matrix->row_ptrs);
    free(csr_matrix->col_indices);
    free(csr_matrix->values);

    return EXIT_SUCCESS;
}

int* vec_gen(int vec_row, int vec_col, unsigned int seed){
  srand(seed);

  int* vector = (int*)malloc(vec_row * vec_col * sizeof(int));

  if(vector == NULL){
    return NULL;
  }

  for (int i = 0; i < vec_row; i++){
    for (int j = 0; j < vec_col; j++){
      vector[i * vec_col + j] = rand()%10;
    }
  }

  return vector;
}

double* multiplication_openmp(const Sparse_CSR* csr_matrix, const
    int* vector, int vec_col);

int main(int argc, char** argv) {
  Sparse_CSR csr_matrix;
  unsigned int seed = 1;

  read_mtx_file("mhd4800a.mtx" , &csr_matrix);

  //printf("Printing CSR Matrix after storing it");
  //print_sparse_csr(&csr_matrix);
  //printf("---------------------------------------------------");

  int* vec_k1 = vec_gen(csr_matrix.n_cols, 1, seed);
  int* vec_k2 = vec_gen(csr_matrix.n_cols, 2, seed);
  int* vec_k3 = vec_gen(csr_matrix.n_cols, 3, seed);
  int* vec_k6 = vec_gen(csr_matrix.n_cols, 6, seed);

```

```c
  double* result_k1 = multiplication_openmp(&csr_matrix, vec_k1, 1);
  double* result_k2 = multiplication_openmp(&csr_matrix, vec_k2, 2);
  double* result_k3 = multiplication_openmp(&csr_matrix, vec_k3, 3);
  double* result_k6 = multiplication_openmp(&csr_matrix, vec_k6, 6);

  free_sparse_csr(&csr_matrix);
  free(vec_k1);
  free(vec_k2);
  free(vec_k3);
  free(vec_k6);
  free(result_k1);
  free(result_k2);
  free(result_k3);
  free(result_k6);

  return EXIT_SUCCESS;
}

double* multiplication_openmp(const Sparse_CSR* csr_matrix, const int* vector, int vec_col){
  double* result = (double *)malloc(csr_matrix->n_row * vec_col * sizeof(double));

  if(result == NULL){
    perror("Failed to allocate memory for result");
    exit(EXIT_FAILURE);
  }

  int i, j, k, nz_start, nz_end, nz_id, vec_val;
  double mat_val, temp;

  char* ompThreads = getenv("OMP_NUM_THREADS");
  int numThreads = atoi(ompThreads);
  double start, end, execution_time;
  start = omp_get_wtime();

  #pragma omp parallel for private (i, j, k, nz_start, nz_end, nz_id, mat_val, vec_val, temp)
  for(k = 0; k<vec_col; k++){
    for(i=0 ; i < csr_matrix->n_row; i++){
      temp = 0.0;
      nz_start = csr_matrix->row_ptrs[i];
      nz_end = csr_matrix->row_ptrs[i+1];
      for(nz_id = nz_start ; nz_id < nz_end ; nz_id++){
        j = csr_matrix->col_indices[nz_id];
        mat_val = csr_matrix->values[nz_id];
        vec_val = vector[(j)*vec_col + k];
        temp += (mat_val*vec_val);
      }
      result[i*vec_col+k] = temp;
    }
  }
  #pragma omp barrier
```

```c
  end = omp_get_wtime();
  execution_time = end - start;
  printf("Execution time for %d threads and %d columns = %lf
      seconds.\n", numThreads, vec_col, execution_time);

  return result;
}
```

Listing 4: mat_vec_multi_openmp_performance.c