COL380 A3:- Matrix Rearrangement using CUDA

Yash Bansal (2022CS51133)

1 Implementation Details

This section provides a detailed explanation of the CUDA-based implementation for processing matrices efficiently.

1. **Memory Allocation:** We begin by allocating necessary memory spaces on the GPU device:
   - `cuda_mat`: A flattened 1D representation of the input matrix.
   - `cuda_freq_array`: An array to store the frequency of each element in the matrix.
   - `cuda_prefix_sum`: An array to store the prefix sum of frequencies.
2. **Data Transfer to GPU:** We iterate over all rows of the original matrix in CPU memory and asynchronously copy each row to the appropriate location in the `cuda_mat` array. Additionally, we initialize the frequency array `cuda_freq_array` to zero asynchronously. We synchronize the CUDA stream to ensure that these asynchronous operations complete before proceeding further.
3. **Frequency Calculation:** We launch the `fill_freqs` kernel using a block size of 1024 threads. Each thread determines its corresponding index in the matrix using the formula:

   $$i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

   Each thread atomically increments the corresponding index in the frequency array based on the element's value in the matrix. After this kernel execution, we synchronize the CUDA stream to ensure accurate results.
4. **Prefix Sum Calculation:** Since the maximum range of values is limited, we compute the prefix sum linearly on the CPU. This approach was chosen after empirical testing, as other methods resulted in longer execution times. The prefix sum is then copied back to the GPU memory.
5. **Reconstruction of Matrix:** We launch the `fill_matrix` kernel to repopulate the `cuda_mat` array with sorted values. This function operates as follows:
   - Each thread processes a unique value within the range.
   - It retrieves the frequency of the value from `cuda_freq_array`.
   - It determines the starting index from `cuda_prefix_sum` and places the element in `cuda_mat` at appropriate positions within the given matrix size.

   The kernel runs with a block size of 1024 threads to maximize parallel execution.
6. **Data Transfer to CPU:** After synchronizing the CUDA device, each row of the processed matrix is asynchronously copied back from `cuda_mat` to CPU memory.
7. **Cleanup:** Finally, we destroy the CUDA stream and free all allocated device memory to prevent memory leaks and ensure optimal resource utilization.

2 Performance Analysis

The execution times were obtained from `data.csv`, where each data point represents an average of five runs of the code.

$r \times c$	$\lvert M \rvert$	**Max. Value** ($\max_{r \in R} r$)			
		1024	**4096**	$\mathbf{10^5}$	$\mathbf{10^8}$
$10^3 \times 10^3$	1	66.92	68.64	73.60	728.22
$10^3 \times 10^3$	10	222.78	215.56	222.84	9660.97
$10^4 \times 10^4$	1	882.71	599.68	670.45	1586.80
$10^4 \times 10^4$	10	9243.80	6304.16	8580.46	16464.10
$10^4 \times 10^5$	1	13842.61	12685.40	6759.22	7831.51

Table 1: Performance comparison of different matrix sizes and maximum range (in ms)

The observations from these runs are as follows:

- For the range values 1024, 4096, and 10^5, the execution time remains approximately the same. However, when the range increases to 10^8, the execution time increases significantly, by a factor of 2 to 5.
- When processing the same total input size, execution time is lower for a single matrix compared to multiple matrices. This is primarily due to the overheads associated with synchronization, CUDA stream creation, and data transfers between the host and the device.