COL380_A3_REPORT

In the correctness version, the solution begins by flattening each 2D matrix into a 1D array and then uses a CUDA kernel (countFreqKernel) to count the frequency of each value. Each thread processes one element from the flattened matrix and updates a frequency array (d_freq) using atomic operations. After this, the host code copies the frequency array back from the device and computes the prefix sum using a simple loop. This prefix sum, which tells us where each value should be placed in the final sorted order, is then copied back to the device. Finally, another CUDA kernel (assignSortedKernel) is used to assign values to the output array. In this kernel, each thread performs a linear search through the prefix sum array, starting from value 1 and incrementing until it finds the correct bucket where the index falls. While this approach is straightforward and works well for smaller ranges, it runs in O(range) time per thread, which can be inefficient for large ranges.

The optimized version addresses this inefficiency by replacing the linear search in the assignment kernel with a binary search. Instead of iterating through all possible values one by one, the optimized code introduces a device function called binarySearch, which quickly finds the correct bucket in O(log(range)) time. This change significantly reduces the number of comparisons and iterations each thread must perform, making the assignment kernel much more scalable and efficient, especially when the input range is very large (for example, 10^5 or higher). The rest of the code frequency counting, memory management, host-side prefix sum computation, and use of CUDA streams—remains unchanged, ensuring that the overall structure and parallelism of the solution are preserved.

In summary, both versions of the code achieve the assignment's goal: they flatten the matrices, count the frequencies of elements, compute a prefix sum, and then assign the sorted values to generate a new matrix that satisfies the required property. The key difference is that the optimized version uses binary search for value assignment, reducing the computational complexity from O(range) to O(log(range)) per thread. This improvement not only boosts performance for large input ranges but also enhances the robustness and scalability of the solution when processing high-range inputs.