Without any optimization, we were able to run the three algorithms in approximately 80 milliseconds.

For the histogram kernel, we created a shared memory space with the size of a number of partitions. The idea is after we calculated the radix value, we incremented the value from the shared memory histogram. Once all the data has been through, we would use atomic operations to add all the values from shared memory into our histogram.

We were able to implement thread coarsening by modifying the loop: for (int idx = threadId; idx < r_size; idx += stride) where stride is defined as the total number of threads across all blocks in the grid. The idea is that threads can handle multiple tasks by using the stride to distribute work across the entire dataset. Each thread processes its assigned data and then jumps by the stride to handle additional tasks. This way, threads are not limited to handling only work from their original block but instead collaborate across blocks to process the entire database This optimization helped reduce the runtime from 80 milliseconds to 60 milliseconds.

For the prefixSum kernel, we first set up the shared memory with the size based on the number of partitions; however, we initialize all the values of shared memory from our histogram.

The prefix sum is computed in the following loop "for (int stride = 1; stride < blockDim.x; stride *= 2)". During each iteration, threads read a value from stride steps earlier in shared memory (temp = shared_data[localIdx - stride]) and add it to their current position. The stride doubles each iteration, progressively building the sum across all threads. Two __syncthreads() calls ensure correctness: The first ensures all threads finish reading before any updates. The second ensures all updates are complete before the next iteration. By the end of this loop, shared memory contains the inclusive prefix sum of the histogram. In the final step, each thread (if not the first thread) replaces its current value with the value from the previous thread (shared_data[localIdx] = shared_data[localIdx - 1]). This converts the inclusive prefix sum into an exclusive prefix sum, excluding the current thread's original value in the result.". This optimization helped reduce the runtime from 50 milliseconds to 50 milliseconds.

For the reorder kernel, we implemented sharedmemory and parallel reduction. We created a shared memory space for the "prefixSum" with the size based on the number of partitions.

In the beginning we would have a for-loop to go through all the data and update the location of the output array while using atomic operations to increment the radix from shared memory. We also included "if (i >= size) return;" this line to address control divergence. At this point, each thread had created partial sums, and we need to combine all them into a single value per bucket. The next for-loop would perform parallel reduction "int stride = n_partitions / 2; stride > 0; stride /= 2)". The process is a binary reduction pattern. and the main function "shared_prefixSum[threadIdx.x] += shared_prefixSum[threadIdx.x + stride];" means within the shared memory, each thread would combine its own value, with the value ahead. The loop continues until there is one thread per bucket. The final for loop would use atomic operations to add all the values from the shared memory to the previously stored prefix sum. This optimization helped reduce the runtime from 50 milliseconds to 35 milliseconds.

Without any optimization, we were able to run the three algorithms in approximately 80 milliseconds, however, after optimization we were able to run the three kernels in approximately 35 milliseconds.

Unfortunately, because we had set the block size and shared memory size to be equivalent to the number of partitions, any attempt in adjusting the block size would only produce the output. So, the block size must be equivalent to the number of partitions