**Question-1:**

(GPU: Tesla K80)

| **Dataset size (N)** | **Epsilon (𝞊)** | **CPU Version Response Time (seconds)**  **(Brute-force, with -O3, CPU cores = 6)** | **GPU Version Response Time (seconds)**  **(Brute-force, Global memory only)** |
| --- | --- | --- | --- |
| 100 | 5.0 | 0.000166 | Total = 0.1195  Kernel = 0.000007 |
| 1000 | 5.0 | 0.00145 | Total = 0.1235  Kernel = 0.000007 |
| 10,000 | 5.0 | 0.0325 | Total = 0.1601  Kernel = 0.0000073 |
| 100,000 | 5.0 | 2.694 | Total = 3.7422  Kernel = 0.0000073 |
| 100 | 10.0 | 0.000161 | Total = 0.1162  Kernel = 0.000007 |
| 1000 | 10.0 | 0.000428 | Total = 0.1201  Kernel = 0.000007 |
| 10,000 | 10.0 | 0.0291 | Total = 0.1557  Kernel = 0.000007 |
| 100,000 | 10.0 | 2.731 | Total = 3.7825  Kernel = 0.0000073 |

From the data presented in the above table, it is apparent that the response times of the CPU version of the brute force algorithm get progressively higher as the value of **N** increases from 100 to 100,000 for both epsilon = 5 and epsilon = 10. As for the GPU implementation of the brute force algorithm, at first glance, the overall response times (including the data transfer between host and device) seem to be getting higher with increasing N values and slower than the CPU version. However, when we look at the kernel response times only, we can see that the response times of the GPU kernel portion are significantly faster (23.71 to 390142.86 times faster) than the corresponding CPU version response times.

The GPU seems to handle large N values far better than the CPU version. Even though the overall response times of the GPU implementations increase as the sizes of the datasets increase due to data transfers between the host and the device, the kernel response times across all N values interestingly remain quite consistent.

As the CPU version ran the programs with only 6 available cores (hence only 6 threads), its performance starts to deteriorate as the value of **N** increased as each thread had to do more work. On the other hand, the GPU version had 1024 threads available in each block (1 block for N = 100 and 977 blocks for N = 1,000,000). Each GPU thread performed the same amount of calculations across all **N** values, hence the GPU version outperforms the CPU version significantly as the value of N increases.

**Question-2:**

(GPU: Tesla K80)

| **Dataset size (N)** | **Epsilon (𝞊)** | **GPU Version Response Time (seconds)**  **(Brute-force, Global memory only)** | **GPU Version Response Time (seconds)**  **(Optimized with reduced calculations)** |
| --- | --- | --- | --- |
| 100 | 5.0 | Total = 0.1195  Kernel = 0.000007 | Total = 0.1655  Kernel = 0.000007 |
| 1000 | 5.0 | Total = 0.1235  Kernel = 0.000007 | Total = 0.1631  Kernel = 0.0000073 |
| 10,000 | 5.0 | Total = 0.1601  Kernel = 0.0000073 | Total = 0.2064  Kernel = 0.0000077 |
| 100,000 | 5.0 | Total = 3.7422  Kernel = 0.0000073 | Total = 2.0479  Kernel = 0.0000073 |
| 1,000,000 | 5.0 | Total = 370.94  Kernel = 0.000009 | Total = 189.4190  Kernel = 0.000015 |
| 100 | 10.0 | Total = 0.1162  Kernel = 0.000007 | Total = 0.1429  Kernel = 0.0000073 |
| 1000 | 10.0 | Total = 0.1201  Kernel = 0.000007 | Total = 0.1502  Kernel = 0.000007 |
| 10,000 | 10.0 | Total = 0.1557  Kernel = 0.000007 | Total = 0.1823  Kernel = 0.0000073 |
| 100,000 | 10.0 | Total = 3.7825  Kernel = 0.0000073 | Total = 2.0280  Kernel = 0.000008 |
| 1,000,000 | 10.0 | Total = 370.761  Kernel = 0.000009 | Total = 189.4663  Kernel = 0.0000153 |

For implementing a more efficient version of the brute force algorithm, I first tried implementing the optimized method developed in Assignment 5. However, that solution requires making calls to user-defined functions from inside the GPU kernel function. According to my understanding, CUDA does not allow calling user-defined external functions from inside the GPU kernel function.

As a result, I developed a simpler solution by sorting the data array before copying it to the device’s (GPU) global memory. Inside the kernel functions, each thread (**Thread 0** to **Thread (N-1)**) calculates the distance between itself and the points starting from its position in the sorted data array till the last element. For example:

Thread 0 will perform N number of distance calculations and perform a double update if the other point is within epsilon distance

Thread 1 will perform N-1 number of distance calculations and perform a double update if the other point is within epsilon distance

Thread 2 will perform N-2 number of distance calculations and perform a double update if the other point is within epsilon distance

Thread 3 will perform N-3 number of distance calculations and perform a double update if the other point is within epsilon distance

.

.

.

.

.

Thread (N-1) will perform 1 distance calculation and perform a double update if the other point is within epsilon distance

So, the total number of calculations is, **Soptimized =** N + (N-1) + (N-2) + (N-3) + …………… + 1 = (N \* (N+1))/2

On the other hand, the brute force algorithm requires performing in total **Sbrute-force** = N**2** distance calculations.

So, the decrease in the number of floating point distance calculations is **Sbrute-force** - **Soptimized** = N2 - (N \* (N+1))/2 = (N2 - 1)/2

However, as seen in the above table, this optimized version performs quite similarly to the brute force GPU version across all values of N and epsilon. At N = 1,000,000 - even though the overall response times of the optimized GPU version are better than the response times of the brute-force GPU version, their kernel response times are almost similar. I think as the GPU supports parallelism at a massive scale (availability of a much higher number of threads on the GPU compared to typical CPUs), the GPU brute-force algorithm and the optimized algorithm’s performance will be similar across larger values of N, too.

However, as the slurm job on Monsoon kept getting throttled because of the time limit for the brute-force version of the algorithm, I could not verify this beyond any values of N > 1,000,000