PA3: K Nearest Neighbor

Hannah Munoz

2/15/18

Introduction

In this project, we implemented the sequential and GPU algorithms for k nearest neighbor (KNN) and the compare the results. The speedup, throughput, and timing of the parallel implementation is calculated.

Method

Sequential Algorithm

The clock starts. The sequential algorithm works by iterating through each row. If a missing value is found, then the kdistance() function is called. The Euclidean distance in Equation 1 is calculated for every row. The result is put into a vector of all distances.

Equation 1:

$$Euclidean\ Distance\ = \sqrt{\sum_{0}^{maxrow} (row_i - row_j)^2}$$

The vector is sorted using stable sort. The first k values, in this case 5, are summed and divided by k. The result is pushed into a pair vector with its related row. The clock is stopped and the calculated results are output.

Parallel Algorithm

Because vectors don't exist on CUDA devices, the parallel algorithm was very different, however it starts the same way. The clock is started and each row is iterated through to find a missing value. When one is found, the kDistance() function is called on the GPU. The thread calculates its global id. This is the row number it's working on. It calculates that row's Euclidean distance from the missing value's row using Eq. 1. The final result is put into an array,

The thread whose global id is the missing value's rows waits for everyone the finish calculation and then sorts the results array using bubble sort. The first 5 values are the summed and divided by 5. The resulting value is passed back to the CPU in an array. Once all missing values have been found the clock is stopped and the result is output.

Results and Conclusion

Sequential

The sequential algorithm's execution times in Table 1 and Fig. 1. 15 runs were done and an average run time was calculated

Table 1: The sequential execution times in ms

Run	Execution Time (ms)
1	6.651533
2	8.29462
3	5.83072
4	6.14224
5	6.07869
6	8.57507
7	6.10909
8	10.03320
9	5.44608
10	6.08224
11	6.09747
12	6.08371
13	5.92336
14	6.79779
15	6.35440
Average	6.70001

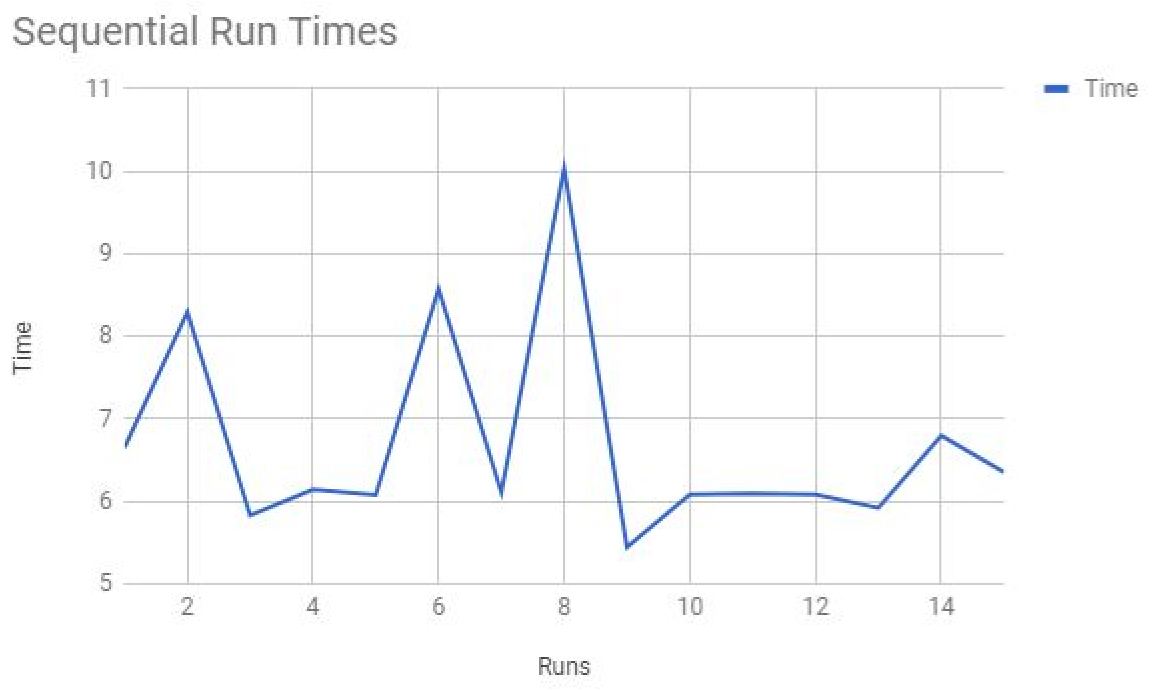

Figure 1: A graph of the sequential execution times found in Table 1.

Parallel

The parallel algorithms execution times are in Table 2 and Fig. 2. Each different thread/block count was run 5 times and an average value was used

Table 2: The parallel execution times

Threads per Block / Runs	1	3	6	9	18	27
1	11.705	11.5524	11.5331	11.7413	11.7398	11.7098
2	11.7526	11.8708	11.7288	11.7696	11.8994	11.7378
3	11.7394	11.7785	11.6561	11.7653	11.7936	11.9729
4	11.5276	11.631	11.772	11.6276	11.7683	11.6531
5	11.7184	11.6625	11.5928	11.6876	11.8084	11.6828
Average	11.6886	11.69904	11.65656	11.71828	11.8019	11.75128

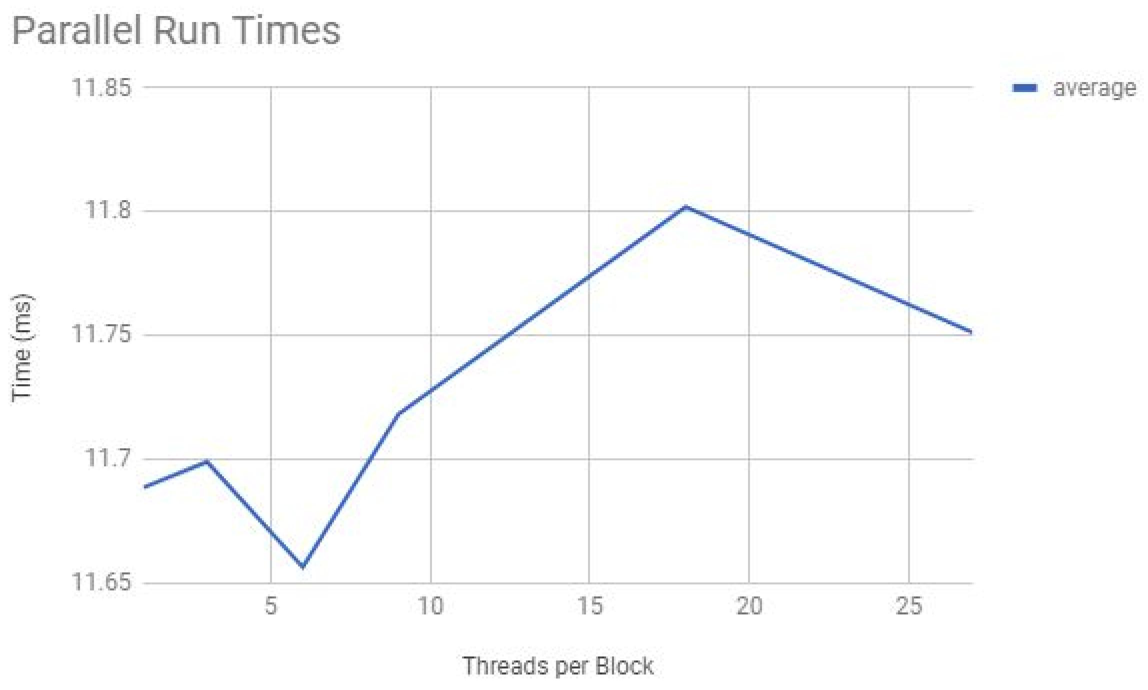

Figure 3: The GPU execution times over varying threads per block.

For the most parts, the run times between different thread and block counts does not change, however there is a slight increase in runtime as the threads per block increases. This is most likely because of two reasons; the small data set and use of bubble sort. Using unified memory with such a small data set gives significant overhead. The sequential algorithm would not need to deal with paging memory between the GPU and CPU. Bubble sort was used to find the minimum result values and was implemented simply because it was the easiest. The sequential algorithm used vectors and the algorithm header file to sort the array. Stable sort has a time complexity of O(n log n). Bubble sort has a time complexity of O(n^2).

The KNN values compare to the original values are displayed in Table 4 and Fig.4

Table 4: A comparison of the KNN values and the original values

Row #	Original	Sequential	Parallel
9	2.179	255.32	187.13
20	1.411	39.5	39.21
38	1.246	350.55	316.37
42	2.854	65.02	64.8
52	2.131	96.27	96.18
59	5.047	293.06	286.89
87	7.415	322.54	322.49
105	0.189	298.67	298.6
112	3.888	257.07	256.99
115	2.024	91.04	90.93
122	3.119	88.91	88.77
125	1.709	67.84	67.68
126	2.124	79.45	79.28
141	-3.55	89.7	89.57
157	-3.55	282.41	126.91
158	-2.793	390.66	343.02

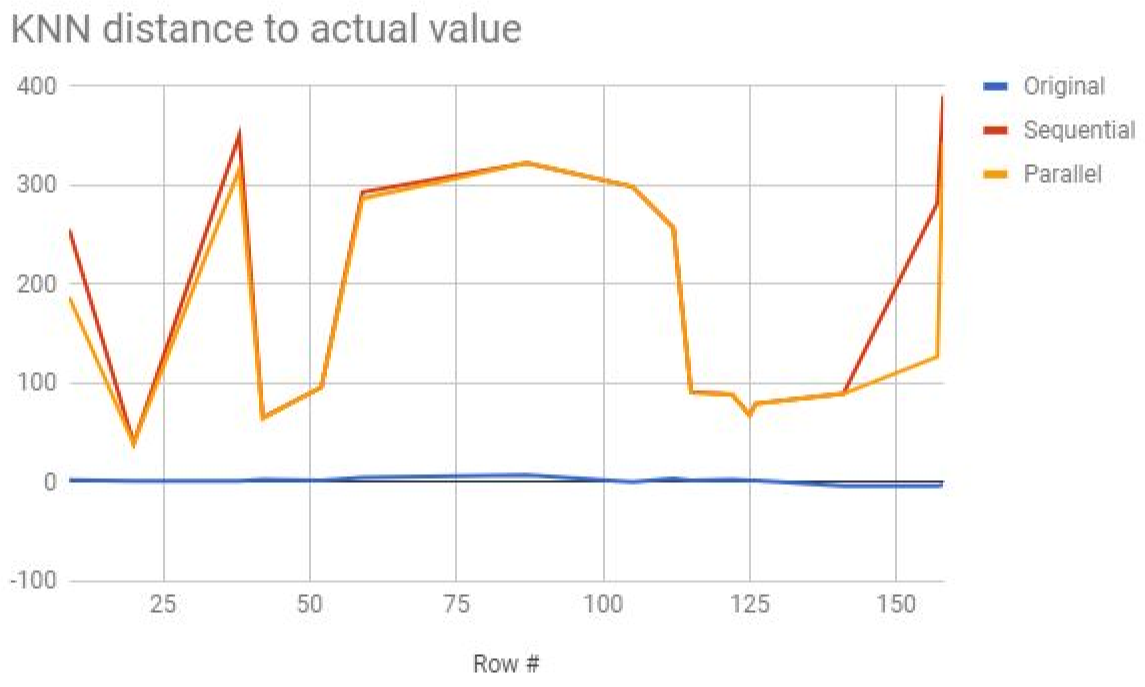

Figure 4: The output KNN values compared to the actual values.

The KNN results output were so far off from the original values because of outliers in the data set. Many columns of the data set were sparsely populated. If they were populated, they had much much larger values than other columns. These outlier probably affect the results.

The sequential and parallel algorithms had slightly different results because of the different ways they were implemented. It could be caused by the floating points on the CPU and GPU being stored in memory differently.

The equation used to calculate speedup in Table 5 and Fig.5 is given in Eq. 2.

Equation 2:
$$speedup = \frac{sequential\ time}{GPU\ time}$$

Table 5: The speedup factor of different matrix sizes and thread counts

Thread per block	Speedup
1	0.5732092979
3	0.5726977769
6	0.5747848593
9	0.5717574764
18	0.5677064032
27	0.5701518643

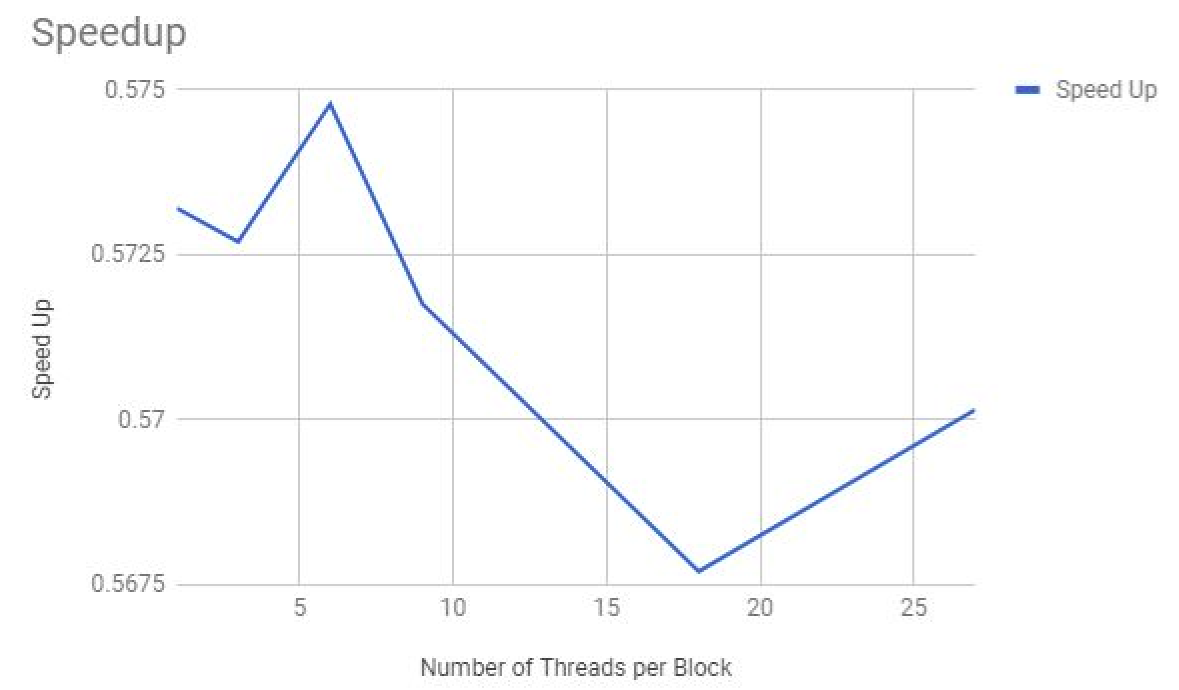

Figure 5: The speedup factor of the parallel algorithm

Max speedup was found to be 0.5747848593 with 27 blocks and 6 threads per block. Average speedup was found to be 0.5717179463. No speed up was found because of the GPU algorithm using bubble sort.

The equation used to calculate throughput in Table 6 and Fig. 6 is given in Eq. 3

Equation 3: $$throughput = \frac{CSV\ Size\ (N\ x\ M)}{GPU\ execution\ time\ (s)}$$

Table 5: The throughput of different threads per block in bit/sec

Thread per block	Throughput
1	1774.03624
3	1772.453124
6	1778.912475
9	1769.54297
18	1757.005228
27	1764.573732

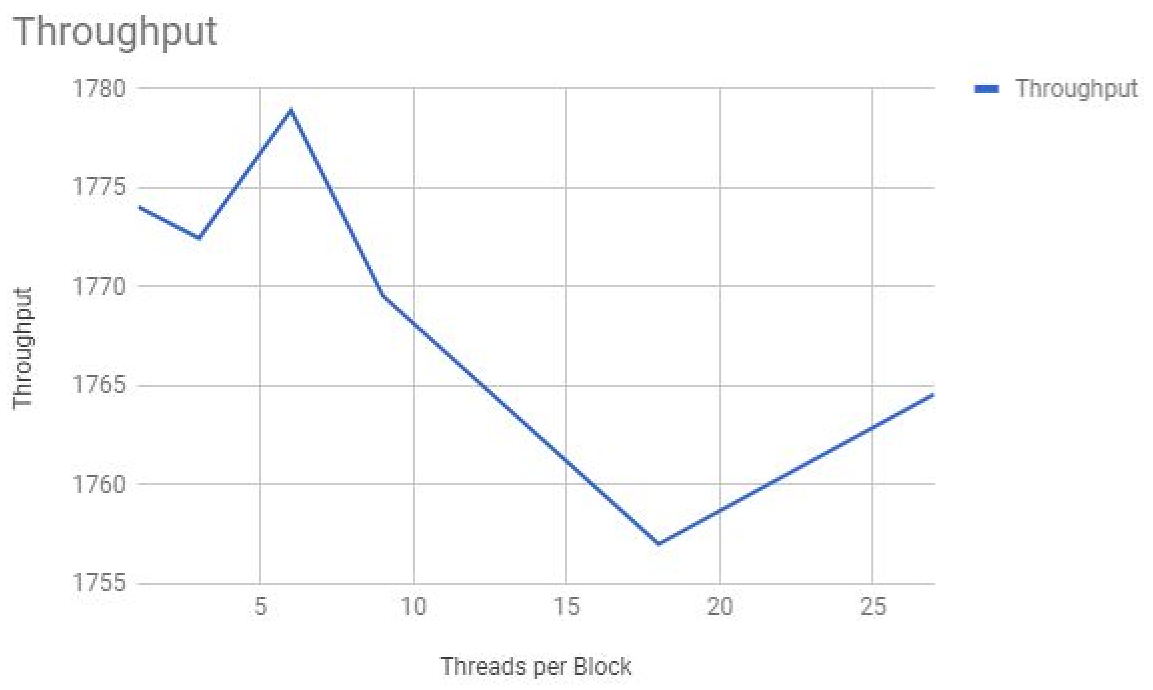

Figure 5: The throughput in Table 5 graphed.

Max throughput was found to be 1778.912475 bits/sec with 27 blocks and 6 threads per block. Average throughput was 1769.420628 bits/sec.