CS147 Final Project Report

CUDA Parallelized KNN

Team: **LGS**
Sabaipon Phimmala, Linda Ghunaim, Glider Mapalad

GitHub Repository: https://github.com/UCR-CSEE147/final-project-sp2025-lgs
Demo Recording | Presentation Slides

Project Overview

LGS used CUDA to parallelize a K-Nearest neighbor (KNN) classifier that predicts the 'star-rating' of Amazon reviews for digital music products. The classifier used TF-IDF feature vectors extracted from the review text. Our project optimizes and improves Glider's portion of a project from CS105. Below are links to the original project and dataset we used.

Original Project:
https://colab.research.google.com/drive/19okzU6i1WmpZDTi0z8zsshTYogO_CSpa?usp=sharing

Dataset:
https://www.google.com/url?q=https%3A%2F%2Fhuggingface.co%2Fdatasets%2FMcAuley-Lab%2FAmazon-Reviews-2023

Stratified Dataset:
https://docs.google.com/spreadsheets/d/13_LRyNGp4SQ74kcQfe5plDogiOQ_Be6hhIiEPIni6g8/edit?usp=sharing

We successfully accelerated the distance computation step of the KNN classifier using GPU kernels. Overall performance and runtime was improved.

GPU Acceleration of the Application

Each thread in the CUDA kernel handles the cosine similarity (distance) computation between the test vector and one training vector.

We defined our problem space with 25 thread blocks of `BLOCK_SIZE = 32`. This can be seen below in our Implementation section.

The GPU accelerates the KNN classifier by running the distance computation on all the training samples parallel across the GPU. This is the only stage of the software pipeline that is parallelized. In the naive implementation, this portion of the pipeline is very slow compared to the rest, therefore parallelizing the cosine similarity computation resulted in faster runtime.

Implementation

Our project consists of four key components: `main.py`, `naive_knn.py`, `naive_K_search.py`, and `parallelKNN.py`. Within these components, we use NumPy, Numba, Sklearn, and Re respectively. For context, some print statements were removed in the code snippets below for readability and length of the report. We utilize

Our `main.py` file was implemented to handle the cleaning and partitioning of the dataset prior to calling each different implementation of the KNN classification algorithm. Before using Sklearn to TF-IDF vectorize the models, main combines the ‘title’ and ‘text’ fields of every review to create a larger text body. Additionally, it uses sklearn’s `TruncatedSVD` to turn the sparse TF-IDF vectors into dense vectors (3000 sparse features to 100 dense features).

The `naive_knn.py` file implements a serialized version of the KNN classifier utilizing a cosine similarity function. The `run_naive_knn` function is responsible for timing the serialized version of KNN, calling the `predict_knn_cpu` function that does the actual classification, and testing both the accuracy and ranged accuracy of the classifier once it finishes. The `predict_knn_cpu` function calculates the K-nearest neighbors for every test sample in a loop, utilizing the cosine similarity function to calculate the distance between each item in the training vector with the entire test vector. The cosine similarity is stored in a vector each iteration, and after all cosine similarities have been calculated, the function then sorts the reviews in descending order, meaning the most similar (closest cosine distanced) reviews are first. Then, looking at the K-nearest reviews, the most common score of these reviews is the score that the test review is assigned. This process is identical in the parallelized version.

```python
def cosine_similarity(a, b):
   dot = np.dot(a, b)
   A_norm = np.linalg.norm(a)
   B_norm = np.linalg.norm(b)
   if A_norm == 0 or B_norm == 0:
       return 0
   return dot / (A_norm * B_norm)
```

```python
def predict_knn_cpu(X_train, Y_train, X_test, k):

   distances = []
   for i in range(len(X_train)):
       similarity = cosine_similarity(X_train[i], X_test)
       distances.append((similarity, Y_train[i]))

   #now we sort so that the first one is the highest similarity
   distances.sort(reverse = True)
   top_k = [label for _, label in distances[:k]]
   most_common = Counter(top_k).most_common(1)[0][0]
   return most_common
```

```
def run_naive_knn(X_train, Y_train, X_val, Y_val, k):
   print("\n=== Running Naive CPU Version of KNN ===\n")
   start = time.time()

   correct = 0
   ranged_correct = 0

   for i in range(len(X_val)):
       pred = predict_knn_cpu(X_train, Y_train, X_val[i], k)
       actual = Y_val[i]
       if pred == actual:
           correct += 1
       if abs(pred - actual) <= 1:
           ranged_correct += 1

   total = len(X_val)
   end = time.time()

   # print strict accuracy, ranged accuracy, and GPU runtime
   print("\n=== Ending Naive CPU Version of KNN ===\n")
```

The `naive_K_search.py` file was used to find the best possible K in our dataset. Main calls this function. It takes a prediction function as a parameter called “predict_device” so that we can call this function using CPU and GPU methods. This way, we can easily make comparisons.

```
def naive_bestKsearch(x_train, y_train, x_val, y_val, krange,
predict_device):
    BEST_K = -1
    BEST_ACCURACY = -1
    BEST_RANGED_K = -1
    BEST_RANGED_ACCURACY = -1

    start_overall = time.time()
    #loop through all k ranges
    for k in range(1,krange + 1):
        correct = 0
        ranged_correct = 0
        start_individual = time.time()
      #calculate KNN
        for i in range(len(x_val)):
            pred = predict_device(x_train, y_train, x_val[i], k)
            actual = y_val[i]

            if pred == actual:
                correct += 1
            if abs(pred - actual) <= 1:
                ranged_correct += 1

        end_individual = time.time()
        accuracy = correct / len(y_val)
        ranged_accuracy = ranged_correct / len(y_val)

        # print K, accuracy, ranged accuracy, and time

        if accuracy > BEST_ACCURACY:
            BEST_ACCURACY = accuracy
            BEST_K = k
        if ranged_accuracy > BEST_RANGED_ACCURACY:
            BEST_RANGED_ACCURACY = ranged_accuracy
            BEST_RANGED_K = k

    end_overall = time.time()
    # print strict accuracy, ranged accuracy, and time
```

Our `parallelKNN.py` file implements a parallelized version of the KNN classifier along with the cosine similarity function. The function `numbaParallelKNN` starts the timer for the classifier, which we use to compare to the naive implementation. It uploads the training data to the GPU memory via `cuda.to_device.` Then, for each validation review, that data gets uploaded to the GPU as well before initializing and launching the GPU kernel. We initialize the GPU by setting the `BLOCK_SIZE = 32` and the `dim_grid = (x_train.shape[0] + BLOCK_SIZE - 1 // BLOCK_SIZE`, which is equivalent to 25 thread blocks. The parallelized function `cosineSim` is then called; threads here run in parallel to compute their cosine similarities. Finally, the function sorts the closest reviews in descending order, then gets a vote from the k closest reviews. The most common occurring “review score” is the score that the review being tested is assigned.

```
@cuda.jit
def cosineSim(train_vectors, test_vector, distances):
   idx = cuda.grid(1)

   if idx < train_vectors.shape[0]:
      dot = 0.0
      normal_a = 0.0
      normal_b = 0.0

      for j in range(train_vectors.shape[1]):
          a = train_vectors[idx, j]
          b = test_vector[0, j]
          dot += a * b
          normal_a += a * a
          normal_b += b * b

      denominator = (normal_a ** 0.5) * (normal_b ** 0.5)
      distances[idx] = 1 - (dot / denominator) if denominator != 0 else 1.0
```

```
def numbaParallelKNN(x_train, y_train, x_val, y_val, k):
   #START TIME
      print("\n=== Running GPU KNN ===")
      start = time.time()
      print("\nGPU PARALLELIZED KNN KERNEL")

      CORRECT = 0;
      RANGED_CORRECT = 0;

      train_d = cuda.to_device(x_train.astype(np.float32))

      #for each validation review, get cosine distance from each training
tf-idf vector
      for i in range(len(x_val)):
         testVector = x_val[i:i+1] #the review to test cosine distance
         test_d = cuda.to_device(testVector.astype(np.float32))
         distances_d = cuda.device_array(x_train.shape[0],dtype=np.float32)

         BLOCK_SIZE = 32
         dim_grid = (x_train.shape[0] + BLOCK_SIZE - 1) // BLOCK_SIZE

         #invoke numba kernel
         cosineSim[dim_grid , BLOCK_SIZE](train_d, test_d, distances_d)

         cosineDistances = distances_d.copy_to_host()
         closestReviews = np.argsort(cosineDistances)[:k]
         closestLabels = y_train[closestReviews]

         #MAKE PREDICTION BY MOST COMMON CLASS OF K NEIGHBORS
         predictedRating = Counter(closestLabels).most_common(1)[0][0]
```

```
        #KEEP TRACK OF HOW MANY CORRECT
        if predictedRating == y_val[i]:
           CORRECT+=1
        #KEEP TRACK OF HOW MANY CORRECT (with tolerance)
        if predictedRating == y_val[i] + 1 or predictedRating == y_val[i]
- 1 or predictedRating == y_val[i]:
           RANGED_CORRECT+=1

     total = len(x_val)
     end = time.time()
     # print strict accuracy, ranged accuracy, and GPU runtime
     print("\n=== Running GPU KNN ===")
```

```
def predict_knn_gpu(x_train, y_train, x_test_row, k):
   BLOCK_SIZE = 32
   dim_grid = (x_train.shape[0] + BLOCK_SIZE - 1) // BLOCK_SIZE

   train_d = cuda.to_device(x_train.astype(np.float32))
   test_d = cuda.to_device(x_test_row[np.newaxis, :].astype(np.float32))
   distances_d = cuda.device_array(x_train.shape[0], dtype=np.float32)

   cosineSim[dim_grid , BLOCK_SIZE](train_d, test_d, distances_d)

   cosineDistances = distances_d.copy_to_host()
   closestReviews = np.argsort(cosineDistances)[:k]
   closestLabels = y_train[closestReviews]
   predictedRating = Counter(closestLabels).most_common(1)[0][0]
   return predictedRating
```

Documentation

We ran our program on the Bender server provided by BCOE Systems. After signing in, the user must `git clone` our repository. Afterwards, they must run this command to get into the Apptainer:
`apptainer shell --nv /singularity/cs217/cs217.2024-12-12.sif`.

Next, import the following dependencies by running:
`python`
`import numba`

`import pandas`.

Exit out of the Python specific container. Now the project is runnable through the Apptainer terminal by running:
`python3 main.py`

Evaluation & Results

Defining STRICT vs RANGED Accuracy

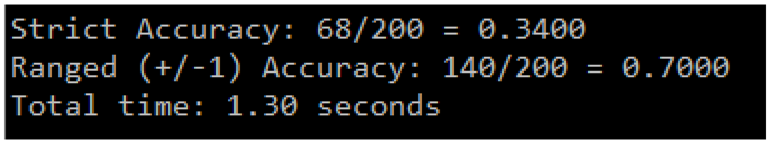
Before comparing the results between CPU and GPU based implementations of KNN, we must define the two accuracy metrics:

- **Strict Accuracy**
    - Percentage of predictions that EXACTLY match the true labels. Predictions count as “correct” only when the prediction matches the true label

- **Ranged Accuracy**
    - Percentage of predictions that are within a ±1 tolerance of the true label. Predictions count as correct when they are:
        - +1 star rating above the true label
        - -1 star rating below the true label

CPU-Only KNN Classification Results (K = 3)

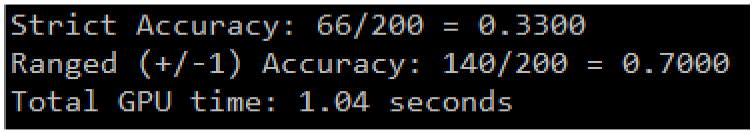
Below is the resulting strict & ranged accuracies, as well as total time training/testing the **Naive CPU** version of KNN, with K = 3



```
Strict Accuracy: 68/200 = 0.3400
Ranged (+/-1) Accuracy: 140/200 = 0.7000
Total time: 1.30 seconds
```

GPU-Parallelized KNN Classification Results (K = 3)

Below is the resulting strict & ranged accuracies, as well as total time training/testing the **GPU Parallelized** version of KNN, with K = 3



```
Strict Accuracy: 66/200 = 0.3300
Ranged (+/-1) Accuracy: 140/200 = 0.7000
Total GPU time: 1.04 seconds
```

Between the Naive CPU and GPU Parallelized implementations, there is **approximately a 20% reduction in time** spent calculating the accuracies of K = 3.

It is important to note that this reduction in time is not guaranteed in different running sessions. We have not been able to identify a direct cause for this, but we believe that latency with the bender server causes this discrepancy.

Despite this, when running main, it is expected to see on average a 20% reduction in time.

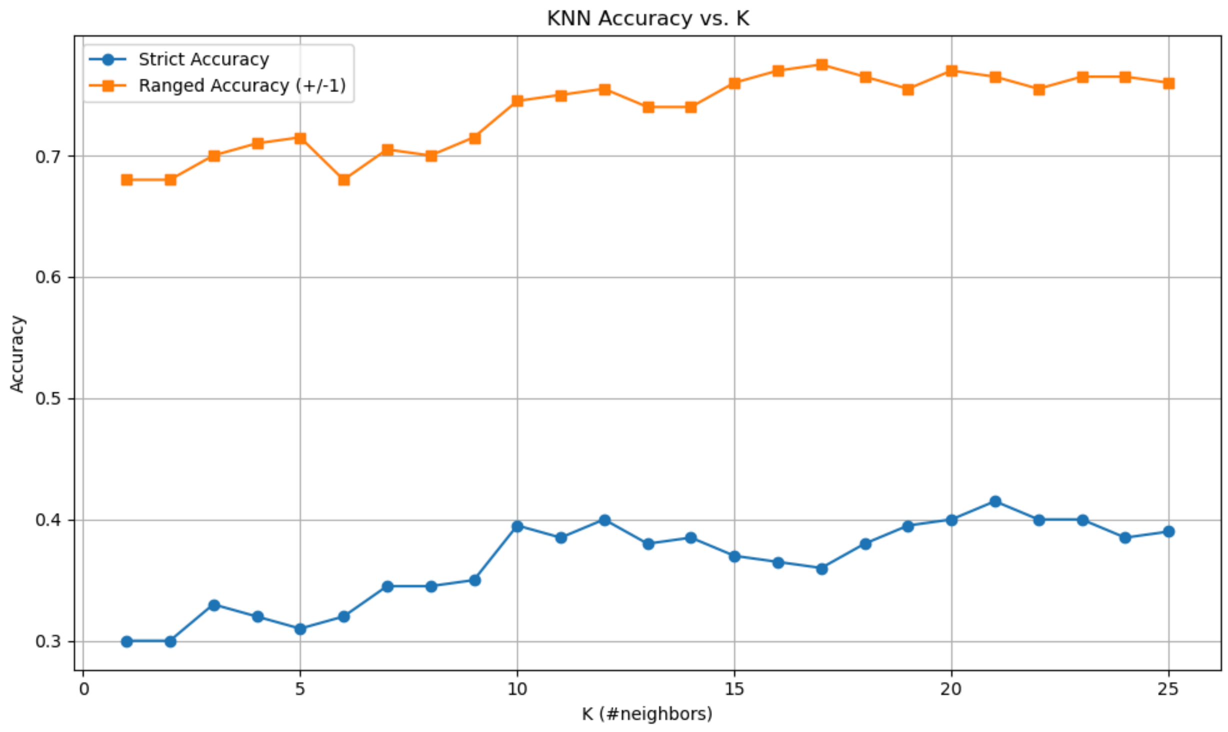
Results in Finding the Best K

The bestKsearch function takes KNN functions as input, allowing for both predict_knn_cpu and predict_knn_gpu to be used to find the best K values. In our main function, we try every K value from 1 to 25, however this can be set to any range as long as it is below the size of the training set.

Before comparing the times for each prediction kernel, it is important to note that both kernels found the same K which gives the highest strict accuracy AND ranged accuracies:

- Best Strict K: **21**
- Best Ranged K**: 17**

This is also reflected in the graph below:

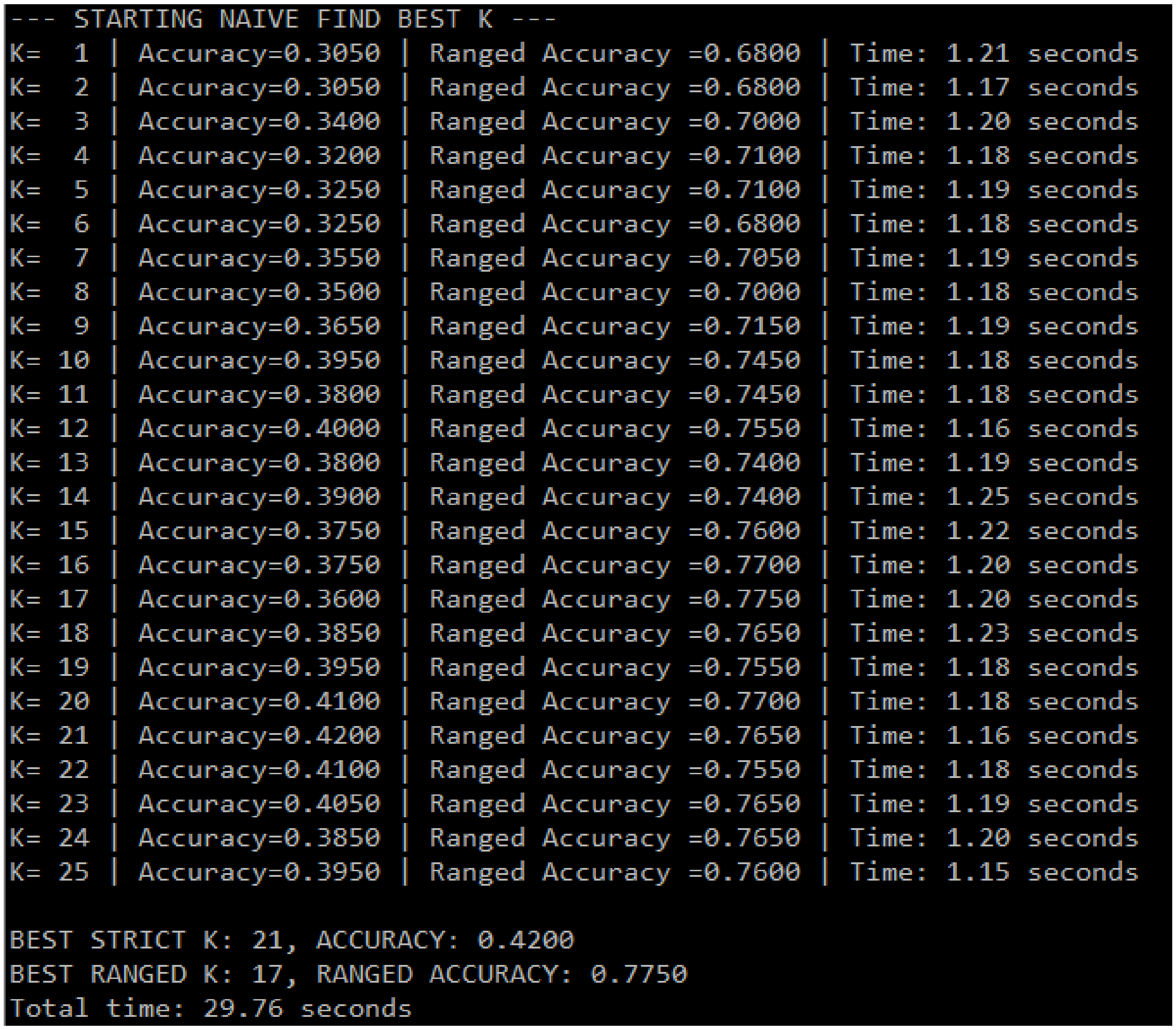


Best K Search Times:

Our program prints the strict and ranged accuracies for each K value, and records the time taken on each K value.

This is followed by a print of the best K values for each respective accuracy, as well as the total time to run the kernel for all K values.

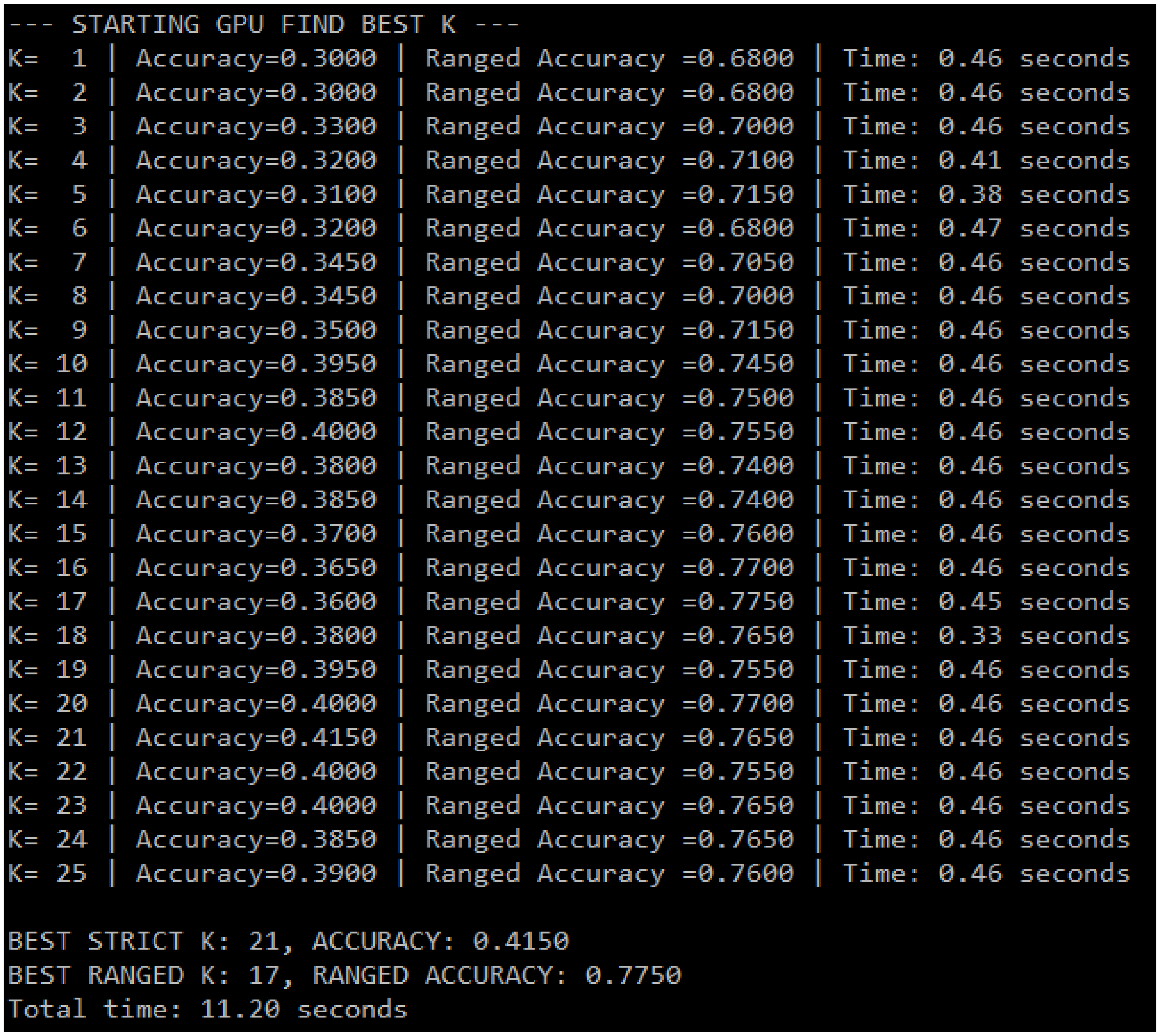
CPU



```
--- STARTING NAIVE FIND BEST K ---
K=  1 | Accuracy=0.3050 | Ranged Accuracy =0.6800 | Time: 1.21 seconds
K=  2 | Accuracy=0.3050 | Ranged Accuracy =0.6800 | Time: 1.17 seconds
K=  3 | Accuracy=0.3400 | Ranged Accuracy =0.7000 | Time: 1.20 seconds
K=  4 | Accuracy=0.3200 | Ranged Accuracy =0.7100 | Time: 1.18 seconds
K=  5 | Accuracy=0.3250 | Ranged Accuracy =0.7100 | Time: 1.19 seconds
K=  6 | Accuracy=0.3250 | Ranged Accuracy =0.6800 | Time: 1.18 seconds
K=  7 | Accuracy=0.3550 | Ranged Accuracy =0.7050 | Time: 1.19 seconds
K=  8 | Accuracy=0.3500 | Ranged Accuracy =0.7000 | Time: 1.18 seconds
K=  9 | Accuracy=0.3650 | Ranged Accuracy =0.7150 | Time: 1.19 seconds
K= 10 | Accuracy=0.3950 | Ranged Accuracy =0.7450 | Time: 1.18 seconds
K= 11 | Accuracy=0.3800 | Ranged Accuracy =0.7450 | Time: 1.18 seconds
K= 12 | Accuracy=0.4000 | Ranged Accuracy =0.7550 | Time: 1.16 seconds
K= 13 | Accuracy=0.3800 | Ranged Accuracy =0.7400 | Time: 1.19 seconds
K= 14 | Accuracy=0.3900 | Ranged Accuracy =0.7400 | Time: 1.25 seconds
K= 15 | Accuracy=0.3750 | Ranged Accuracy =0.7600 | Time: 1.22 seconds
K= 16 | Accuracy=0.3750 | Ranged Accuracy =0.7700 | Time: 1.20 seconds
K= 17 | Accuracy=0.3600 | Ranged Accuracy =0.7750 | Time: 1.20 seconds
K= 18 | Accuracy=0.3850 | Ranged Accuracy =0.7650 | Time: 1.23 seconds
K= 19 | Accuracy=0.3950 | Ranged Accuracy =0.7550 | Time: 1.18 seconds
K= 20 | Accuracy=0.4100 | Ranged Accuracy =0.7700 | Time: 1.18 seconds
K= 21 | Accuracy=0.4200 | Ranged Accuracy =0.7650 | Time: 1.16 seconds
K= 22 | Accuracy=0.4100 | Ranged Accuracy =0.7550 | Time: 1.18 seconds
K= 23 | Accuracy=0.4050 | Ranged Accuracy =0.7650 | Time: 1.19 seconds
K= 24 | Accuracy=0.3850 | Ranged Accuracy =0.7650 | Time: 1.20 seconds
K= 25 | Accuracy=0.3950 | Ranged Accuracy =0.7600 | Time: 1.15 seconds

BEST STRICT K: 21, ACCURACY: 0.4200
BEST RANGED K: 17, RANGED ACCURACY: 0.7750
Total time: 29.76 seconds
```

GPU



```
--- STARTING GPU FIND BEST K ---
K=  1 | Accuracy=0.3000 | Ranged Accuracy =0.6800 | Time: 0.46 seconds
K=  2 | Accuracy=0.3000 | Ranged Accuracy =0.6800 | Time: 0.46 seconds
K=  3 | Accuracy=0.3300 | Ranged Accuracy =0.7000 | Time: 0.46 seconds
K=  4 | Accuracy=0.3200 | Ranged Accuracy =0.7100 | Time: 0.41 seconds
K=  5 | Accuracy=0.3100 | Ranged Accuracy =0.7150 | Time: 0.38 seconds
K=  6 | Accuracy=0.3200 | Ranged Accuracy =0.6800 | Time: 0.47 seconds
K=  7 | Accuracy=0.3450 | Ranged Accuracy =0.7050 | Time: 0.46 seconds
K=  8 | Accuracy=0.3450 | Ranged Accuracy =0.7000 | Time: 0.46 seconds
K=  9 | Accuracy=0.3500 | Ranged Accuracy =0.7150 | Time: 0.46 seconds
K= 10 | Accuracy=0.3950 | Ranged Accuracy =0.7450 | Time: 0.46 seconds
K= 11 | Accuracy=0.3850 | Ranged Accuracy =0.7500 | Time: 0.46 seconds
K= 12 | Accuracy=0.4000 | Ranged Accuracy =0.7550 | Time: 0.46 seconds
K= 13 | Accuracy=0.3800 | Ranged Accuracy =0.7400 | Time: 0.46 seconds
K= 14 | Accuracy=0.3850 | Ranged Accuracy =0.7400 | Time: 0.46 seconds
K= 15 | Accuracy=0.3700 | Ranged Accuracy =0.7600 | Time: 0.46 seconds
K= 16 | Accuracy=0.3650 | Ranged Accuracy =0.7700 | Time: 0.46 seconds
K= 17 | Accuracy=0.3600 | Ranged Accuracy =0.7750 | Time: 0.45 seconds
K= 18 | Accuracy=0.3800 | Ranged Accuracy =0.7650 | Time: 0.33 seconds
K= 19 | Accuracy=0.3950 | Ranged Accuracy =0.7550 | Time: 0.46 seconds
K= 20 | Accuracy=0.4000 | Ranged Accuracy =0.7700 | Time: 0.46 seconds
K= 21 | Accuracy=0.4150 | Ranged Accuracy =0.7650 | Time: 0.46 seconds
K= 22 | Accuracy=0.4000 | Ranged Accuracy =0.7550 | Time: 0.46 seconds
K= 23 | Accuracy=0.4000 | Ranged Accuracy =0.7650 | Time: 0.46 seconds
K= 24 | Accuracy=0.3850 | Ranged Accuracy =0.7650 | Time: 0.46 seconds
K= 25 | Accuracy=0.3900 | Ranged Accuracy =0.7600 | Time: 0.46 seconds

BEST STRICT K: 21, ACCURACY: 0.4150
BEST RANGED K: 17, RANGED ACCURACY: 0.7750
Total time: 11.20 seconds
```

As seen in the displays above, both ‘bestKsearch” functions (using CPU and GPU prediction functions respectively), produce the same “best K” for strict and ranged accuracies.

There is approximately 62.4% reduction in time when the bestKsearch function utilizes the GPU prediction function.

Similar to the initial K = 3 output for the two kernels, this percent in time reduction is not guaranteed with every instance of the program being run, however it can be expected to see a percentile close to this reduction.

Issues

We had issues with using Numba since the default environment in Bender did not have the necessary packages. Dr. Wong explained how to use an Apptainer container environment on Bender, which allowed us to utilize Numba.

Task Breakdown

Task Breakdown	
Serialized, naive version of KNN	Linda - 100%, Glider - 0%, Sabaipon - 0%
Best K Search, Displaying Results	Linda - 0%, Glider - 100%, Sabaipon - 0%
Parallelized version of KNN	Linda - 0%, Glider - 0%, Sabaipon - 100%
Project Presentation	Linda - 33%, Glider - 33%, Sabaipon - 33%
Project Report	Linda - 33%, Glider - 33%, Sabaipon - 33%