Cairo University - Faculty of Engineering
Computer Engineering Department -Spring 2024

# PARALLEL COMPUTING BIG ASSIGNMENT REPORT

## CUDA Implementation Integrated Machine Learning Algorithms of **K-Means** and **KNN** Algorithms

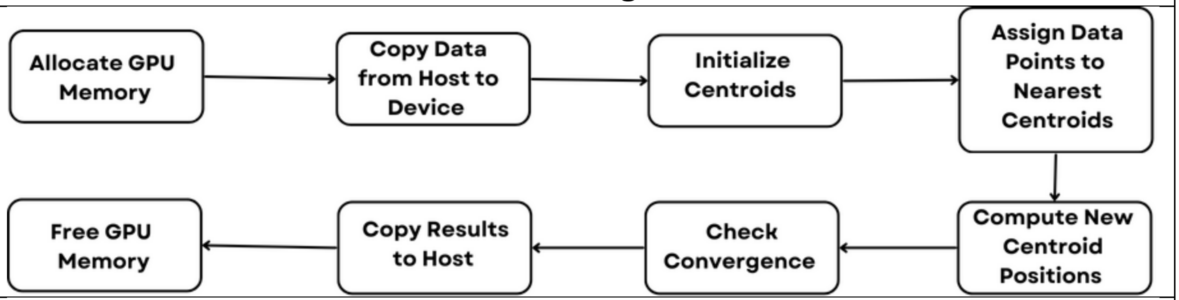
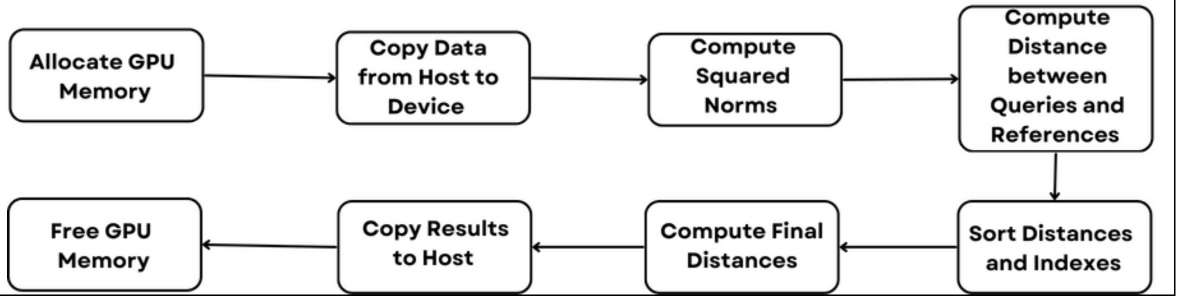
**Team members:**

| Name | Section | BN |
|---|---|---|
| Donia Gameel | 1 | 24 |
| Shaza Mohammed | 1 | 32 |
| Heba Ashraf Raslan | 2 | 32 |

**Project Description:**

Our project aims to implement an integrated solution using CUDA for parallel computation of two fundamental machine learning algorithms: K-Nearest Neighbors (KNN) and K-Means clustering. These algorithms are interrelated in their applications and can benefit from parallelization using CUDA, allowing for efficient processing of large datasets.

**Block Diagram:**

| Module | Block Diagram |
|---|---|
| K-Means |   Allocate GPU Memory<br>Copy Data from Host to Device<br>Initialize Centroids<br>Assign Data Points to Nearest Centroids<br>Compute New Centroid Positions<br>Check Convergence<br>Copy Results to Host<br>Free GPU Memory |
| KNN |   Allocate GPU Memory<br>Copy Data from Host to Device<br>Compute Squared Norms<br>Compute Distance between Queries and References<br>Sort Distances and Indexes<br>Compute Final Distances<br>Copy Results to Host<br>Free GPU Memory |

# K-Means

## Overview of Implementations and Measurements

K-means clustering is a popular unsupervised machine learning algorithm used for clustering data points into groups based on similarities. It aims to partition the data into K clusters where each data point belongs to the cluster with the nearest mean, serving as the cluster's centroid.

We have implemented the k-nearest neighbors (KNN) algorithm using three different approaches:

1. **CPU-based implementation in Python using sklearn**
2. **CPU-based implementation in C**
3. **GPU-based implementation in Python using TensorFlow**
4. **cuML-based implementation with cuDF in Python**
5. **CUDA-based implementation**
6. **CUDA-based implementation with streaming**

The objective is to compare the performance of the K-Meansalgorithm across these implementations, particularly focusing on the speedup achieved by utilizing GPU acceleration.

**CPU-based implementation in Python using sklearn:**

This implementation utilizes the **pandas**, **numpy**, and **scikit-learn** libraries for data manipulation, numerical operations, and machine learning algorithms, respectively. Here's a summary of the Python implementation:

1. **Data Loading and Preprocessing**: The script loads datasets of different sizes from CSV files, standardizes the features using **StandardScaler**, and iterates over different numbers of clusters.
2. **K-means Clustering**: It applies the K-means clustering algorithm using **KMeans** from **scikit-learn**, measures the time taken for clustering, and stores the results in a DataFrame.
3. **Results**: The clustering results, including data size, number of clusters, and time taken for clustering

| Kmeans sklearn | | |
|---|---|---|
| Data Size | Clusters | Time (seconds) |
| 10000 | 10 | 0.842065 |
| 10000 | 100 | 2.387318 |
| 10000 | 1000 | 41.758798 |
| 100000 | 10 | 4.466653 |
| 100000 | 100 | 16.375293 |
| 100000 | 1000 | 143.831143 |
| 1000000 | 10 | 10.407497 |
| 1000000 | 100 | 141.549031 |
| 1000000 | 1000 | 1593.802099 |

### CPU-based implementation in C

The C implementation directly reads CSV data and implements the K-means algorithm from scratch without relying on external libraries. Here's an overview of the C implementation:

1. **Data Reading**: It reads CSV data directly from files and dynamically allocates memory to store the data points.
2. **K-means Algorithm**: The algorithm consists of two main functions: **kMeansClusterAssignment** and **kMeansCentroidUpdate**, responsible for assigning data points to clusters and updating cluster centroids iteratively.
3. **Results**: After a predefined number of iterations, the total time taken for clustering is printed.

| Kmeans C code | | |
|---|---|---|
| Data Size | Clusters | Time (seconds) |

| | | |
|---|---|---|
| 10000 | 10 | 0.059939 |
| 10000 | 100 | 0.514360 |
| 10000 | 1000 | 3.335168 |
| 100000 | 10 | 0.404125 |
| 100000 | 100 | 3.388042 |
| 100000 | 1000 | 41.548951 |
| 1000000 | 10 | 5.506077 |
| 1000000 | 100 | 44.086444 |
| 1000000 | 1000 | 415.07164 |

### GPU-based implementation in Python using TensorFlow

The TensorFlow implementation utilizes TensorFlow's computational graph and automatic differentiation capabilities to perform K-means clustering efficiently. Here's a summary of the implementation:

1. **Data Loading and Preprocessing**: The script loads datasets of different sizes from CSV files using **pandas**, standardizes the features using **StandardScaler**, and converts the data into TensorFlow tensors for compatibility with TensorFlow operations.
2. **K-means Clustering Function**: The **kmeans** function defines the K-means clustering algorithm using TensorFlow operations. It initializes centroids randomly, iteratively assigns points to the nearest centroid, and updates centroids based on the mean of the assigned points.
3. **GPU Acceleration**: The implementation utilizes TensorFlow's GPU support to accelerate the computation of K-means clustering. It specifies GPU options and creates a TensorFlow session configured to use the GPU.
4. **Clustering Process**: The script iterates over different dataset sizes and numbers of clusters. For each combination of dataset size and number of clusters, it measures the time taken for K-means clustering on the GPU using TensorFlow.

5. **Results**: The clustering results, including data size, number of clusters, and time taken for clustering

| TensorFlow Python code | | |
|---|---|---|
| Data Size | Clusters | Time (seconds) |
| 10000 | 10 | 0.871729 |
| 10000 | 100 | 6.014540 |
| 10000 | 1000 | 49.04130 |
| 100000 | 10 | 0.471055 |
| 100000 | 100 | 4.323267 |
| 100000 | 1000 | 51.07692 |
| 1000000 | 10 | 0.604169 |
| 1000000 | 100 | 4.198499 |
| 1000000 | 1000 | 51.03058 |

**cuML-based implementation with cuDF in Python:**

The cuML implementation utilizes NVIDIA's GPU-accelerated K-means clustering algorithm to efficiently cluster large datasets on GPUs. Here's a summary of the implementation:

1. **Data Loading and Preprocessing**: The script loads datasets of different sizes from CSV files using **pandas**, standardizes the features using **StandardScaler** from **scikit-learn**, and prepares the data for clustering.
2. **K-means Clustering with cuML**: The implementation applies K-means clustering using the **KMeans** class from **cuml.cluster**. This class provides a GPU-accelerated implementation of the K-means algorithm, enabling faster clustering on NVIDIA GPUs.

3. **GPU Acceleration**: cuML leverages the parallel processing capabilities of NVIDIA GPUs to accelerate the computation of K-means clustering. By offloading computation to the GPU, it achieves significant speedups compared to CPU-based implementations.
4. **Clustering Process**: The script iterates over different dataset sizes and numbers of clusters, measures the time taken for K-means clustering using cuML, and stores the results in a list.
5. **Results**: The clustering results, including data size, number of clusters, and time taken for clustering

| cuML | | |
|---|---|---|
| Data Size | Clusters | Time (seconds) |
| 10000 | 10 | 1.278008 |
| 10000 | 100 | 0.072464 |
| 10000 | 1000 | 0.475428 |
| 100000 | 10 | 0.040021 |
| 100000 | 100 | 0.178397 |
| 100000 | 1000 | 1.900078 |
| 1000000 | 10 | 0.354767 |
| 1000000 | 100 | 1.561732 |
| 1000000 | 1000 | 14.652058 |

**CUDA-based implementation**

This Implementation leverages CUDA to parallelize the K-means clustering algorithm, improving performance for large datasets by taking advantage of the GPU's parallel processing capabilities.

**Execution Flow**

1. **Reading Data:**
   - The **readCSVData** function reads the CSV file and populates the **datapoints** array.
   - The number of points and dimensions are determined.
2. **Memory Allocation:**
   - Allocates memory on both host and device for data points, centroids, cluster assignments, and cluster sizes.
3. **Initialization:**
   - Initializes centroids randomly from the data points.
4. **K-means Iterations:**
   - For each iteration:
     - The **kMeansClusterAssignment** kernel assigns each data point to the nearest centroid.
     - The **kMeansCentroidUpdate** kernel updates the centroids by averaging the assigned points.
     - The **normalizeCentroids** kernel normalizes the centroids.
5. **Timing:**
   - CUDA events are used to record the start and stop times of the K-means iterations.
   - The total time is calculated and printed.
6. **Cleanup:**
   - Frees the allocated memory on both the host and device.

**CUDA Kernels in Detail**

1. **Cluster Assignment Kernel:**
   - Computes the distance from each point to each centroid.
   - Assigns the point to the nearest centroid.
2. **Centroid Update Kernel:**
   - Uses shared memory to accumulate the sum of points for each centroid.
   - Uses atomic operations to ensure correct summation in parallel.
3. **Normalization Kernel:**
   - Divides the accumulated sum by the number of points in each cluster to compute the mean.

| Kmeans GPU | | |
|---|---|---|
| Data Size | Clusters | Time (seconds) |
| 10000 | 10 | 0.162942 |
| 10000 | 100 | 0.002410 |
| 10000 | 1000 | 0.009373 |
| 100000 | 10 | 0.001900 |
| 100000 | 100 | 0.006472 |
| 100000 | 1000 | 0.047023 |
| 1000000 | 10 | 0.010275 |
| 1000000 | 100 | 0.055242 |
| 1000000 | 1000 | 0.351849 |
| 10000000 | 10 | 0.094760 |
| 10000000 | 100 | 0.409743 |
| 10000000 | 1000 | 2.251465 |

**CUDA-based implementation with streaming**

CUDA streams allow for concurrent execution of memory transfers and kernel execution. In this implementation, two streams are created to overlap computation with memory transfers

The implementation of k-means using CUDA with streaming demonstrates significant performance improvements over traditional CPU implementations. By leveraging the parallel computing capabilities of GPUs and using streams to overlap computation with memory transfers, the algorithm can handle large datasets more efficiently. This implementation highlights the importance of optimizing both computation and memory access patterns in GPU programming.

| Data Size | Clusters | No streaming | streaming |
|---|---|---|---|
| 10000 | 10 | 0.162942 | 0.000728 |
| 10000 | 100 | 0.002410 | 0.001503 |
| 10000 | 1000 | 0.009373 | 0.009231 |
| 100000 | 10 | 0.001900 | 0.001730 |
| 100000 | 100 | 0.006472 | 0.006322 |
| 100000 | 1000 | 0.047023 | 0.046692 |
| 1000000 | 10 | 0.010275 | 0.010131 |
| 1000000 | 100 | 0.055242 | 0.055107 |
| 1000000 | 1000 | 0.351849 | 0.352530 |
| 10000000 | 10 | 0.094760 | 0.094549 |
| 10000000 | 100 | 0.409743 | 0.380678 |
| 10000000 | 1000 | 2.251465 | 2.180443 |

**Speedup of the GPU over the CPU:**

| Data Size | Clusters | GPU Time | CPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 10000 | 10 | 0.162942 | 0.059939 | 0.3678x |
| 10000 | 100 | 0.002410 | 0.514360 | 213.42x |
| 10000 | 1000 | 0.009373 | 3.335168 | 355.82x |

|  |  |  |  |  |
|---|---|---|---|---|
| 100000 | 10 | 0.001900 | 0.404125 | 212.697x |
| 100000 | 100 | 0.006472 | 3.388042 | 523.492x |
| 100000 | 1000 | 0.047023 | 41.548951 | 883.586x |
| 1000000 | 10 | 0.010275 | 5.506077 | 535.871x |
| 1000000 | 100 | 0.055242 | 44.086444 | 798.06x |
| 1000000 | 1000 | 0.351849 | 415.071644 | 1179.686x |
| 10000000 | 10 | 0.094760 | 49.511440 | 522.493x |
| 10000000 | 100 | 0.409743 | 428.748454 | 1046.383x |
| 10000000 | 1000 | 2.251465 | 4156.124091 | 1845.964x |

**CPU benchmarks for kmeans:**

Kmeans benchmarks depend on several factors:

- Input data size (number of data points and dimensionality):
    - This has a notable effect on time, as increasing either the number of data points or number of features increases the execution time
- Number of clusters (K):
    - Increasing the numbers of clusters also increases the execution time substantially as more clusters generally require more iterations
- CPU architecture used, clock speed, and number of cores (whether single or multicore)
- Libraries and implementation:
    - Whether we use libraries like Scikit-learn (Python) or a code implementation of Kmeans affects performance

**Their GPU counterparts:**

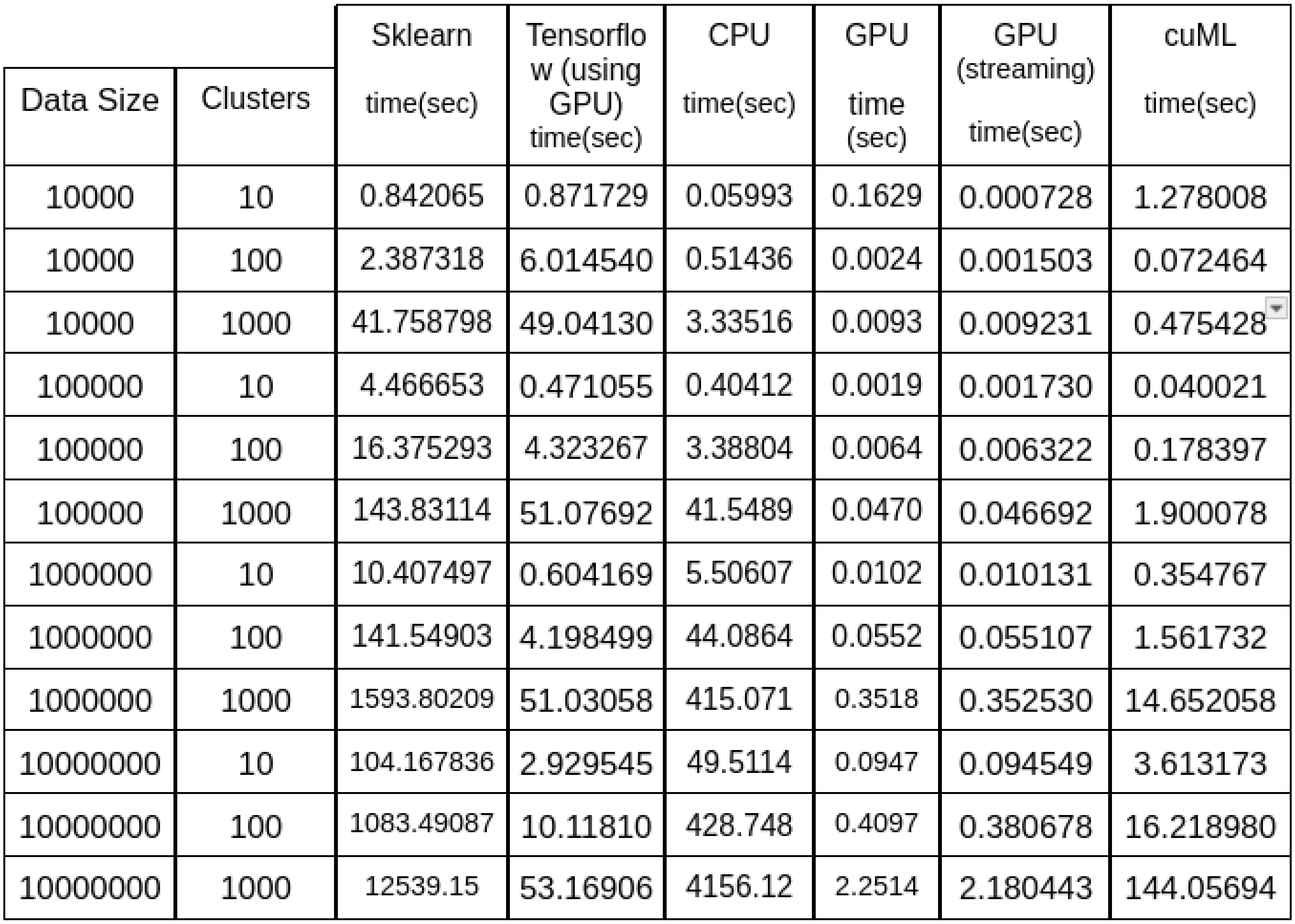
- Input data size (number of data points and dimensionality):
    - This has a notable effect on time, as increasing either the number of data points or number of features increases the execution time

- Number of clusters (K):
  - Increasing the numbers of clusters also increases the execution time substantially as more clusters generally require more iterations
- GPU architecture used, and number of blocks, number of threads per block, shared memory size
- Libraries and implementation:
  - Whether we use libraries like cuML (Python) or a code implementation of Kmeans affects performance

**GPU results compare to open-source peers:**

| Data Size | Clusters | GPU Time | cuML Time |
|---:|---:|---:|---:|
| 10000 | 10 | 0.162942 | 1.278008 |
| 10000 | 100 | 0.002410 | 0.072464 |
| 10000 | 1000 | 0.009373 | 0.475428 |
| 100000 | 10 | 0.001900 | 0.040021 |
| 100000 | 100 | 0.006472 | 0.178397 |
| 100000 | 1000 | 0.047023 | 1.900078 |
| 1000000 | 10 | 0.010275 | 0.354767 |
| 1000000 | 100 | 0.055242 | 1.561732 |
| 1000000 | 1000 | 0.351849 | 14.652058 |

### Comparison between implementations:



| Data Size | Clusters | Sklearn time(sec) | Tensorflow (using GPU) time(sec) | CPU time(sec) | GPU time (sec) | GPU (streaming) time(sec) | cuML time(sec) |
|---|---|---|---|---|---|---|---|
| 10000 | 10 | 0.842065 | 0.871729 | 0.05993 | 0.1629 | 0.000728 | 1.278008 |
| 10000 | 100 | 2.387318 | 6.014540 | 0.51436 | 0.0024 | 0.001503 | 0.072464 |
| 10000 | 1000 | 41.758798 | 49.04130 | 3.33516 | 0.0093 | 0.009231 | 0.475428 |
| 100000 | 10 | 4.466653 | 0.471055 | 0.40412 | 0.0019 | 0.001730 | 0.040021 |
| 100000 | 100 | 16.375293 | 4.323267 | 3.38804 | 0.0064 | 0.006322 | 0.178397 |
| 100000 | 1000 | 143.83114 | 51.07692 | 41.5489 | 0.0470 | 0.046692 | 1.900078 |
| 1000000 | 10 | 10.407497 | 0.604169 | 5.50607 | 0.0102 | 0.010131 | 0.354767 |
| 1000000 | 100 | 141.54903 | 4.198499 | 44.0864 | 0.0552 | 0.055107 | 1.561732 |
| 1000000 | 1000 | 1593.80209 | 51.03058 | 415.071 | 0.3518 | 0.352530 | 14.652058 |
| 10000000 | 10 | 104.167836 | 2.929545 | 49.5114 | 0.0947 | 0.094549 | 3.613173 |
| 10000000 | 100 | 1083.49087 | 10.11810 | 428.748 | 0.4097 | 0.380678 | 16.218980 |
| 10000000 | 1000 | 12539.15 | 53.16906 | 4156.12 | 2.2514 | 2.180443 | 144.05694 |

# KNN

## Overview of Implementations and Measurements

I have implemented the k-nearest neighbors (KNN) algorithm using three different approaches:

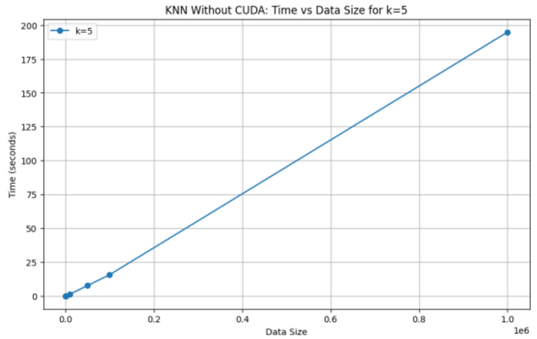
1. **CPU-based implementation in Python**
2. **CUDA-based implementation**
3. **cuML-based implementation with cuDF in Python**

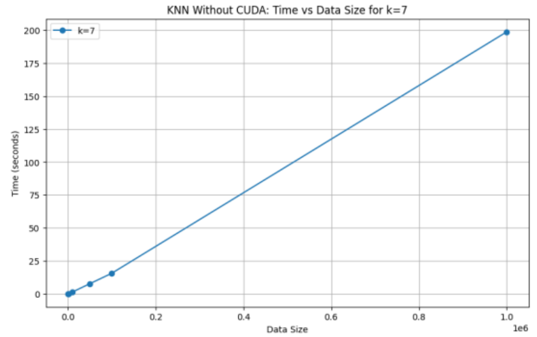
The objective is to compare the performance of the KNN algorithm across these implementations, particularly focusing on the speedup achieved by utilizing GPU acceleration.

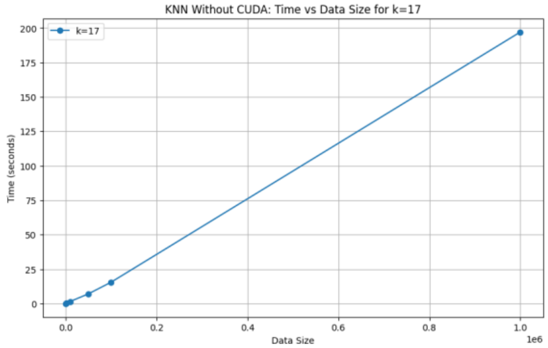
### 1. CPU-based KNN Implementation in Python

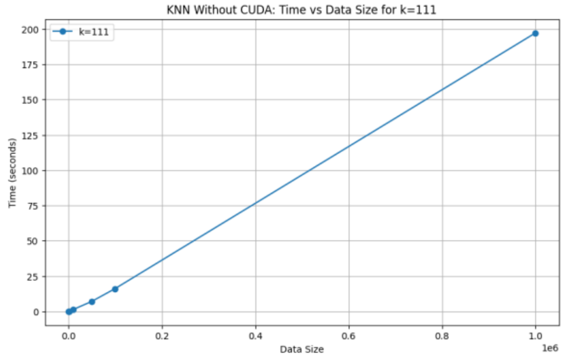
The CPU implementation is straightforward and involves calculating the Euclidean distance between each query point and all reference points, sorting these distances, and selecting the nearest **k** neighbors.

I measured the execution time of this implementation for different sizes of reference points and various values of **k**.





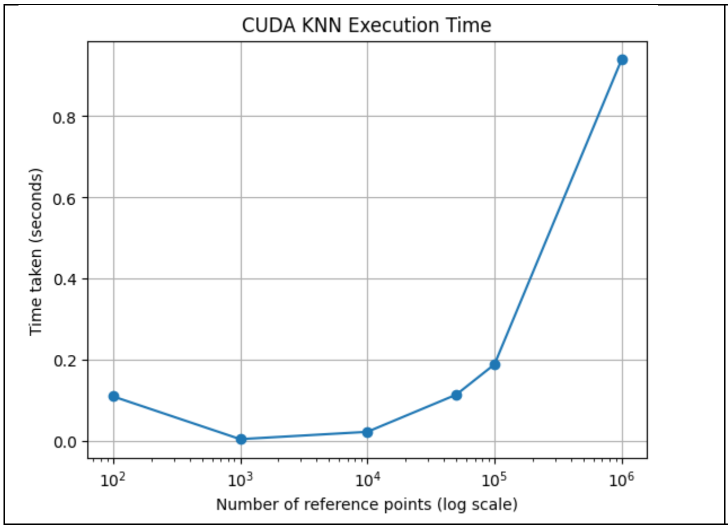


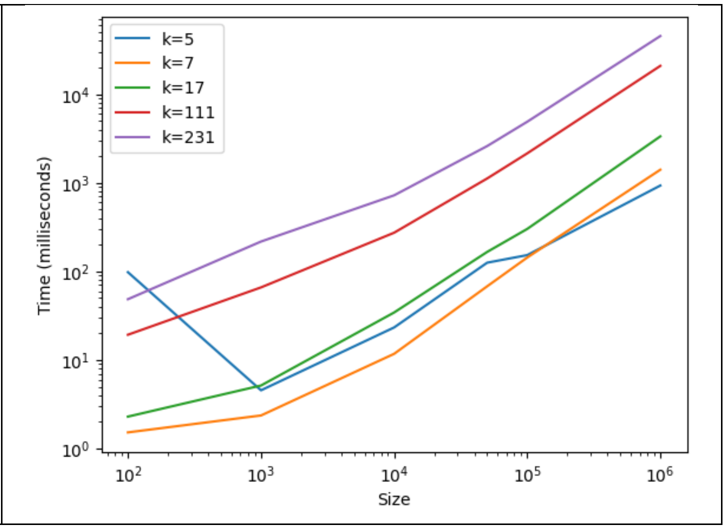


## 2. GPU-based KNN Implementation in Python

In the CUDA implementation of the K-Nearest Neighbors (KNN) algorithm, parallelism is achieved through the use of CUDA threads and blocks. Each thread in a CUDA block processes one query point. This means that if we have **m** query points, we will launch **m** threads.

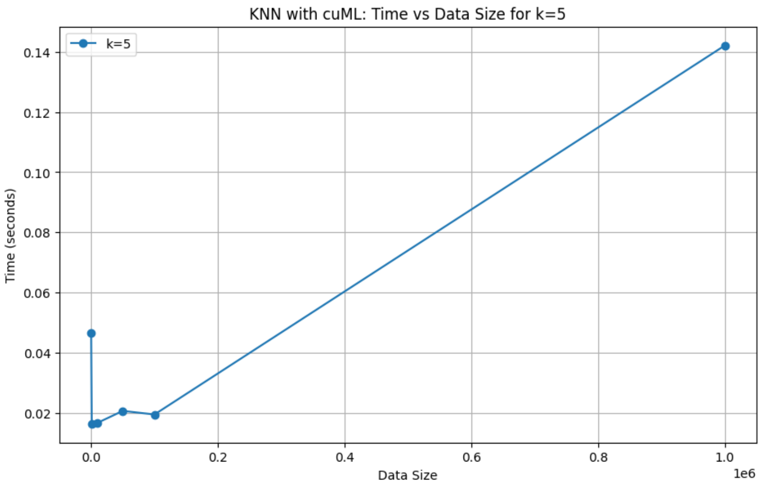
I measured the execution time of this implementation for different sizes of reference points and various values of **k**.





## 3. cuML-based KNN Implementation with cuML in Python

The cuML-based implementation uses RAPIDS AI libraries to perform KNN computations. This approach also utilizes GPU acceleration but through a higher-level API provided by cuML



# Performance Analysis

## 1. CPU Benchmarks for KNN Algorithms

- **Overview**

  The K-Nearest Neighbors (KNN) algorithm is widely used in graph learning to build a KNN graph by finding the K nearest neighbors for each of the N samples in a dataset. The performance of KNN on a CPU can be influenced by the dataset size (N), the number of dimensions (D), and the number of neighbors (K). In this analysis, we evaluate various KNN implementations using different datasets to understand the CPU performance across common scenarios.

- **Datasets and Scenarios**

  - **Mixed Gaussian Dataset:**
    - ￼￼￼
  - **Point Cloud Sample from ModelNet:**
    - ￼￼D=3
  - **MNIST Subsets:**
    - Small Subset: ￼￼￼*D*=784
    - Medium Subset: ￼￼*D*=784
    - Large Subset: ￼￼*D*=784

- **KNN Implementations**

  - **Brute Force (blas):** Uses matrix multiplication, memory inefficient but straightforward.
  - **kd-tree:** Efficient for low-dimensional data, implemented only on CPU.
  - **Brute Force (standard):** A simple implementation without optimization.
  - **Brute Force (shared memory):** Optimized for GPU, not applicable for CPU benchmarks.
  - **nn-descent:** An approximate algorithm with a trade-off between speed and accuracy.

- **Benchmark Results**

**1. Mixed Gaussian Dataset:**

| Model | CPU (K=8) | CPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.010 | 0.011 |
| kd-tree | 0.004 | 0.006 |
| Brute Force | 0.004 | 0.006 |
| nn-descent | 0.014 (R: 0.985) | 0.148 (R: 1.000) |

**2. Point Cloud Sample:**

| Model | CPU (K=8) | CPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.359 | 0.432 |
| kd-tree | 0.007 | 0.026 |
| Brute Force | 0.074 | 0.167 |
| nn-descent | 0.161 (R: 0.977) | 1.345 (R: 0.999) |

**3. MNIST Subsets:**

**Small MNIST:**

| Model | CPU (K=8) | CPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.014 | 0.015 |
| kd-tree | 0.179 | 0.182 |
| Brute Force | 0.173 | 0.228 |
| nn-descent | 0.060 (R: 0.878) | 1.077 (R: 0.999) |

**Medium MNIST:**

| Model | CPU (K=8) | CPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.897 | 0.970 |
| kd-tree | 18.902 | 18.928 |
| Brute Force | 14.495 | 17.652 |
| nn-descent | 0.804 (R: 0.755) | 14.108 (R: 0.999) |

**Large MNIST:**

| Model | CPU (K=8) | CPU (K=64) |
|---|---|---|

| | | |
|---|---|---|
| Brute Force (blas) | 21.829 | 22.135 |
| kd-tree | 542.688 | 573.379 |
| Brute Force | 373.823 | 432.963 |
| nn-descent | 4.995 (R: 0.658) | 75.487 (R: 0.999) |

- **Analysis**
    - **Brute Force (blas):**
        - Generally provides consistent and reasonable performance across all datasets and K values.
        - Performance deteriorates significantly with larger datasets due to memory inefficiency.
    - **kd-tree:**
        - Extremely efficient for low-dimensional data (D=3) but becomes impractically slow with high-dimensional data (D=784).
        - Not applicable for GPU, so not comparable in the GPU context.
    - **Brute Force:**
        - Standard implementation shows competitive performance in smaller datasets but lags behind optimized versions in larger datasets.
    - **nn-descent:**
        - Provides approximate results with varying recall rates.
        - Performance is competitive, especially in larger datasets, but depends on the required accuracy.

- **Conclusion**
    - For CPU implementations, the kd-tree algorithm is optimal for low-dimensional data due to its efficiency, while brute force (blas) provides a balanced performance for higher dimensions if memory constraints are not an issue. nn-descent offers a good trade-off between speed and accuracy, particularly useful for large datasets where exact precision is not critical.
    - By understanding these benchmarks, we can evaluate the effectiveness of KNN algorithms and make informed decisions about their application and optimization on CPUs.

Reference: https://docs.dgl.ai/en/0.9.x/api/python/knn_benchmark.html

## 2. GPU Benchmarks for KNN Algorithms

### Introduction

For the GPU benchmarks, we utilized the Amazon EC2 P3.2xlarge instance, which is equipped with 8 vCPUs, 61GB RAM, and one Tesla V100 GPU with 16GB RAM. The environment setup included DGL==0.7.0, PyTorch==1.8.1, and CUDA==11.1 on Ubuntu 18.04.5 LTS. We used the same datasets and configurations as in the CPU benchmarks to ensure consistency in our performance comparisons.

### Benchmark Results

Here, we present the performance metrics (time taken) for the GPU implementations of various KNN algorithms across different datasets. We include the results for different values of *K* to illustrate the impact of this parameter on performance.

**Mixed Gaussian Dataset (N = 1000, D = 3):**

| Model | GPU (K=8) | GPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.002 s | 0.003 s |
| Brute Force (shared memory) | 0.002 s | 0.003 s |
| Brute Force | 0.126 s | 0.009 s |
| nn-descent | 0.016 s (R: 0.973) | 0.077 s (R: 1.000) |

**Point Cloud Sample from ModelNet (N = 10000, D = 3):**

| Model | GPU (K=8) | GPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.010 s | 0.010 s |
| Brute Force (shared memory) | 0.004 s | 0.017 s |
| Brute Force | 0.008 s | 0.039 s |
| nn-descent | 0.086 s (R: 0.966) | 0.445 s (R: 0.999) |

**Small MNIST Subset (N = 1000, D = 784):**

| Model | GPU (K=8) | GPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.002 s | 0.002 s |
| Brute Force (shared memory) | 0.045 s | 0.054 s |
| Brute Force | 0.123 s | 0.170 s |
| nn-descent | 0.030 s (R: 0.952) | 0.457 s (R: 0.999) |

**Medium MNIST Subset (N = 10000, D = 784):**

| Model | GPU (K=8) | GPU (K=64) |
|---|---|---|
| Brute Force (blas) | 0.019 s | 0.023 s |
| Brute Force (shared memory) | 2.257 s | 2.524 s |
| Brute Force | 2.058 s | 2.588 s |
| nn-descent | 0.158 s (R: 0.900) | 1.794 s (R: 0.999) |

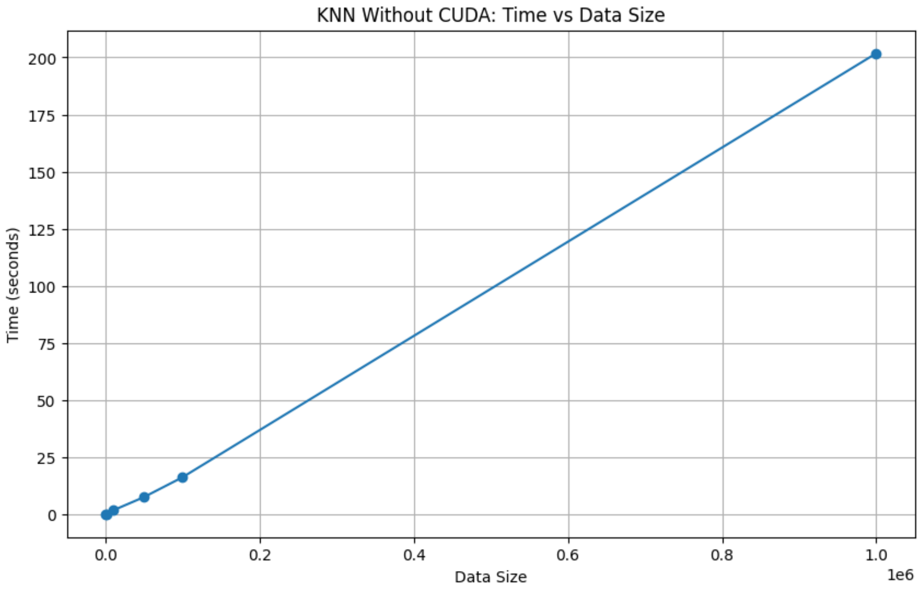
**Large MNIST Subset (N = 50000, D = 784):**

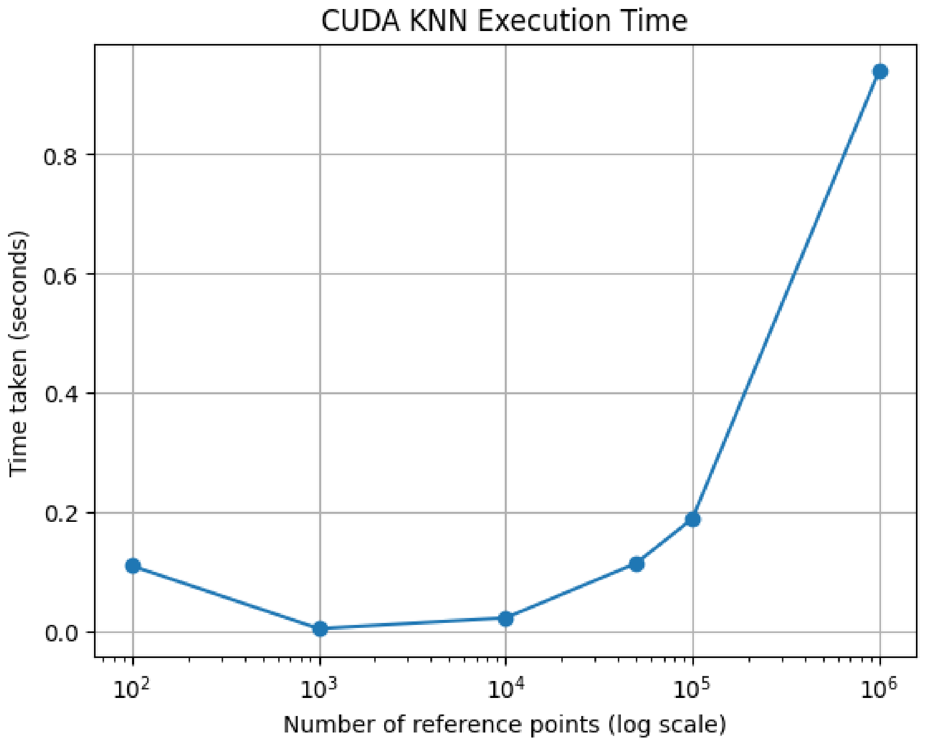
| Model | GPU (K=8) | GPU (K=64) |
|---|---|---|
| Brute Force (blas) | Out of Memory | Out of Memory |
| Brute Force (shared memory) | 53.133 s | 58.419 s |
| Brute Force | 10.317 s | 12.639 s |
| nn-descent | 1.478 s (R: 0.860) | 15.698 s (R: 0.999) |

These results show the performance of different KNN algorithms on the GPU across various datasets and configurations. The next step involves comparing these GPU results with the CPU benchmarks to analyze the speedup achieved and to perform a detailed performance comparison.

Reference: https://docs.dgl.ai/en/0.9.x/api/python/knn_benchmark.html

### 3. How much is the speedup of the GPU over the CPU?

<table>
<tr>
<td><b>CPU</b></td>
<td>


Time taken with reference size 1000000: 201.8075 seconds.
</td>
</tr>
<tr>
<td><b>GPU</b></td>
<td>


Time taken with CUDA for n=1000000 = 945.879 milliseconds
</td>
</tr>
</table>

**Spead up of GPU over CPU = CPU time/GPU time=** 213.35

**4. How does this compare to the theoretical speedup?**

The theoretical speedup can be calculated using Amdahl's Law:

$$S_{overall} = \frac{1}{(1-f)+\frac{f}{S_{part}}}$$

Where:

- f is the fraction of the algorithm that can be parallelized.
- *S* is the speedup of the parallelizable portion of the algorithm (e.g., the speedup achieved by the GPU).

**5. If your speedup is below the theoretical one (this is mostly the case), how do you explain this, what could be changed to achieve a better one?**

1- **Memory Allocation**:

- In the **knn_kernel**, dynamically allocating memory (**min_distances** and **min_indices**) for each thread can be inefficient. Consider using shared memory (__**shared**__) for temporary storage if the array sizes are small and constant.

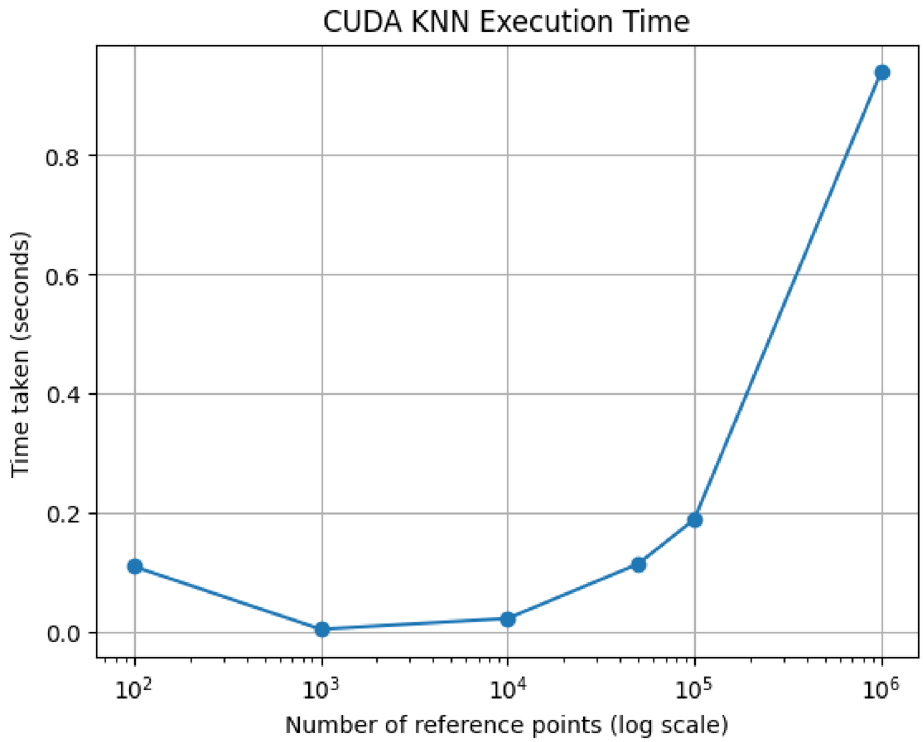
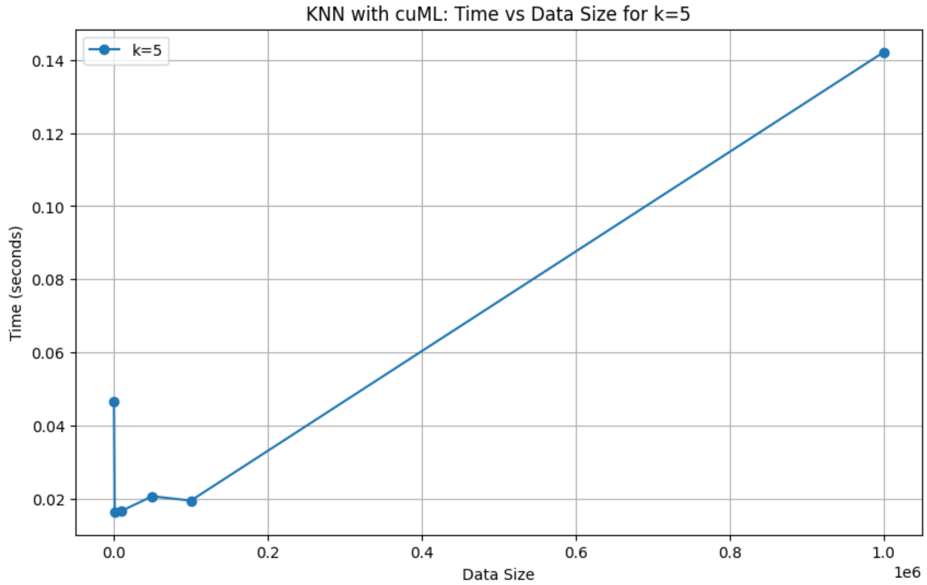
2- **Thread Divergence**:

- Conditional statements inside loops (**if (distance < min_distances[l])**) can lead to thread divergence, where threads in the same warp take different paths, reducing efficiency. Minimize such divergences if possible.

3- **Data Transfer Overhead**:

- If there is significant data transfer between the host and device, it can impact performance. Minimize data transfer by batching operations and overlapping computation with data transfer.

## 6. My GPU results vs KNN RAPIDS cuml results:

| | |
|---|---|
| **My GPU** | <br> <br>Time taken with CUDA for n=1000000 = 945.879 milliseconds |
| **RAPIDS cuml** | <br> <br>Time taken with reference size 1000000 and k=5: 0.1421 seconds |

**Spead up of RAPIDS cuml over my GPU= CPU time/GPU time=** 6.656