Parallelised K-Means clustering algorithm using numpy, cuda and numba

Okechukwu Okeke
Udobang Joshua

Problem Statement

In this age of Big Data...

- Organizations often collect vast amounts of data without predefined categories or labels.

- Analyzing this data to uncover meaningful patterns and groupings can be challenging without prior knowledge of the underlying structure.

- **K-Means clustering comes to the rescue !!**

K-Means Algorithm

1. Initialization

- Randomly select k points as initial cluster centroids (or centers).

2. Assignment Step

- Assign each data point to the cluster with the closest centroid using a distance metric (commonly **Euclidean distance**).
- For a point x_i, it is assigned to cluster j if:

$$\text{Cluster}(x_i) = \underset{j}{\text{argmin}} \|x_i - \mu_j\|^2$$

where μ_j is the centroid of cluster j.

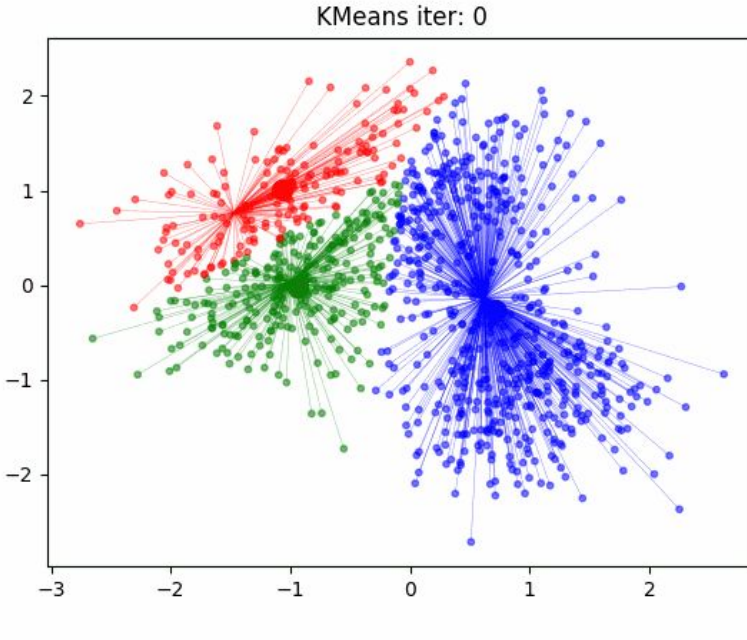
3. Update Step

- Recalculate the centroids by taking the mean of all points in each cluster

$$\mu_j = \frac{1}{n_j} \sum_{x_i \in C_j} x_i$$

4. Repeat

- Alternate between the assignment and update steps until the centroids converges or a maximum number of iterations is reached. Minimise the **Within-Cluster Sum of Squares (WCSS)**



$$WCSS = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

Applications of KMeans Algorithm

1. Image Compression

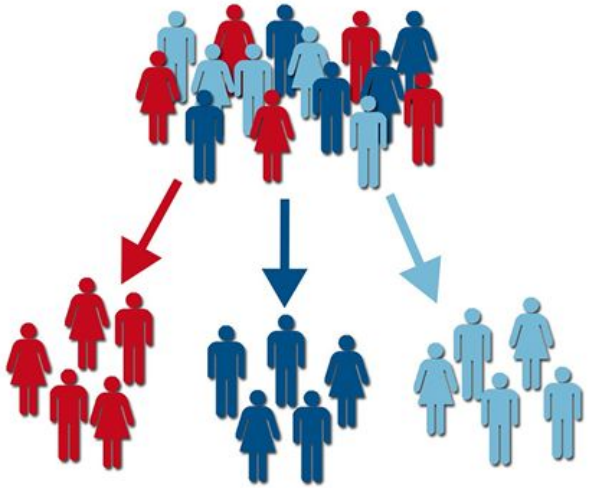
- K-Means reduces the number of unique colors in an image by clustering pixel colors into **K** groups. Each pixel is then replaced with the centroid of its cluster

  Example: Converting a high-resolution photo with millions of colors to an optimized image with only 16 or 32 colors.

2. Customer Segmentation in Marketing

- Group customers based on behaviors, demographics, or purchasing history.

  Example: Netflix uses similar clustering approaches to group users with similar viewing habits for personalized recommendations.



AND MANY MORE….

Challenges and Parallel Improvements

Iterative process scales poorly with dataset size.

- Computational cost increases quadratically with the number of data points.
- Updating centroids requires aggregating potentially millions of points per iteration.

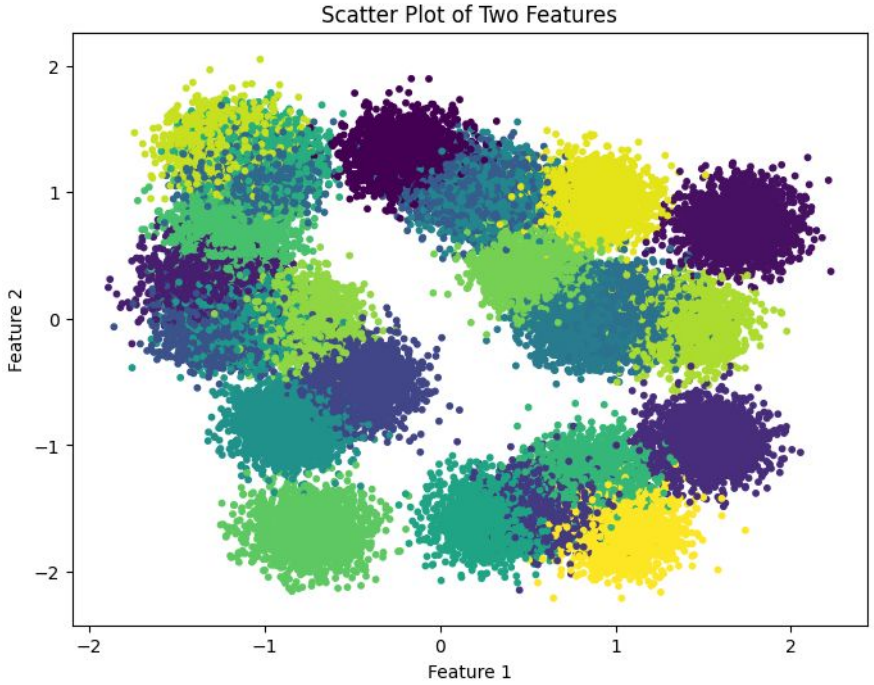
However, many parts of the computation exhibits data parallelism

- ❏ Compute distances for all data points simultaneously.
- ❏ Assign points to clusters independently.
- ❏ Calculate mean for each cluster in parallel.

Algorithm

1. **Initialization**
   Randomly select K cluster centers C from X.
2. **Memory Setup**
   Allocate GPU memory for X, C, L, and intermediate data.
3. **Define CUDA Kernels**
   - **ComputeDistances**: Compute $D[i][j] = \|x_i - c_j\|^2$ for all i and j.
   - **AssignClusters**: Assign $l_i = \text{argmin}_j \, D[i][j]$ for all i.
   - **UpdateCenters**: Update $c_j = \frac{1}{n_j} \sum_{i \mid l_i = j} x_i$.
4. **Iterative Process**
   a. Copy C to GPU.
   b. For $t = 1$ to max_iters:
   i. Launch **ComputeDistances** to compute distances.
   ii. Launch **AssignClusters** to update L.
   iii. Launch **UpdateCenters** to compute new C.
   iv. Copy updated C to host and check for convergence:
   - If $\max \|c_j^{\text{new}} - c_j^{\text{old}}\| < \epsilon$, break.
5. **Finalize**
   Copy C and L from GPU to host, then free GPU memory.
6. **Return**
   Return final C and L.

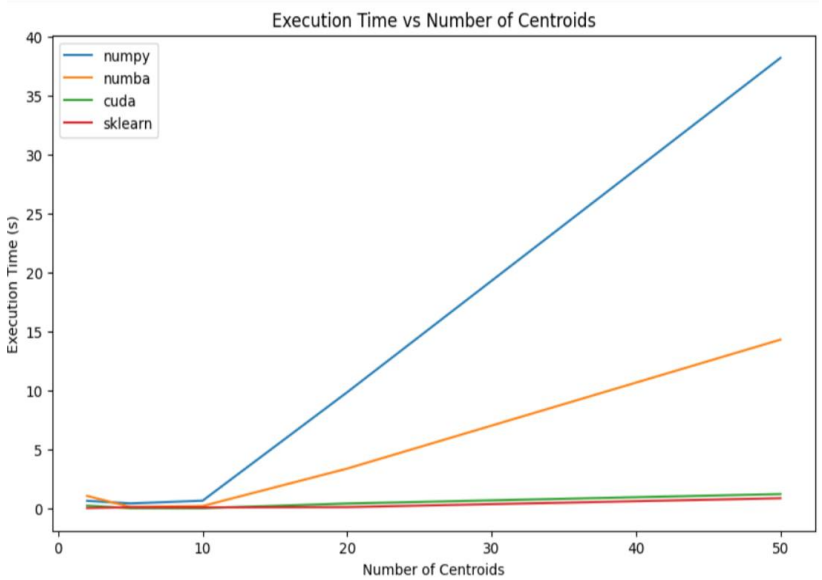
DataSet and CodeBlocks




```python
@cuda.jit
def compute_distances(data, centroids, distances):
    i, j = cuda.grid(2)
    if i < data.shape[0] and j < centroids.shape[0]:
        diff = 0
        for k in range(data.shape[1]):
            diff += (data[i, k] - centroids[j, k]) ** 2
        distances[i, j] = diff

def kmeans_cuda(data, n_centroids, max_iter=100):
    np.random.seed(42)
    centroids = data[np.random.choice(data.shape[0], n_centroids, replace=False)]
    data_device = cuda.to_device(data)
    labels = np.zeros(data.shape[0], dtype=np.int32)
    labels_device = cuda.to_device(labels)
    centroids_device = cuda.to_device(centroids)
    distances = np.zeros((data.shape[0], n_centroids), dtype=np.float32)
    distances_device = cuda.to_device(distances)

    threads_per_block = (16, 16)
    blocks_per_grid_x = int(np.ceil(data.shape[0] / threads_per_block[0]))
    blocks_per_grid_y = int(np.ceil(n_centroids / threads_per_block[1]))
    blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

    for _ in range(max_iter):
        compute_distances[blocks_per_grid, threads_per_block](data_device, centroids_device, distances_device)
        distances = distances_device.copy_to_host()
        labels = np.argmin(distances, axis=1)
        new_centroids = np.array([data[labels == i].mean(axis=0) if np.any(labels == i) else centroids[i] for i in range(n_centroids)])
        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids
        centroids_device = cuda.to_device(centroids)

    return labels, centroids
```

Results and Conclusion

- Parallelized K-Means clustering leverages on more compute power without compromising accuracy.
- Our CUDA implementation has similar performance with scikit-learn KMeans implementation **(sad, expected it to be faster)**.
- Our Numpy's execution time goes very steep, as number of clusters increases, compared to others **(as expected)**



THANK YOU FOR YOUR TIME