Scalable and Efficient K-Means Data Clustering

Filemon Mateus
mateus@utah.edu

Abstract

This project investigates the performance characteristics of the K-MEANS clustering algorithm implemented in three distinct programming environments: PYTHON, C++, and CUDA. The project is aimed at determining the computational efficiency and scalability of each computing environment in handling the clustering of synthetic multidimensional data points with added Gaussian noise. PYTHON's implementation, using libraries such as SCIPY and SCIKITLEARN, focuses on accessibility and ease-of-use. The C++ version leverages low-level control and optimization capabilities through the STANDARD and EIGEN libraries to enhance computation speed. Finally, the CUDA implementation utilizes the parallel processing power of GPUs to reduce computation times significantly on large datasets. We evaluate the execution time, accuracy, and resource utilization of each implementation, providing a comprehensive comparison. The insights gained from this comparative analysis are crucial for professionals and researchers choosing a computing platform for large-scale machine learning tasks. This paper aims to serve as a guideline for selecting the appropriate programming environment based on specific requirements of computational tasks at hand.

1 Repository

— https://github.com/filemon-mateus/cs6140-final-proj

2 Introduction

K-MEANS clustering, a cornerstone method in the field of data mining, is extensively utilized for its efficiency, interpretability, and simplicity in classifying multidimensional datasets into distinct groups [1]. As the volume and complexity of data continue to grow, the demand for high-performance computing solutions has intensified, prompting a reevaluation of traditional approaches to data analysis. Herein, we delve into a comparative study of the K-MEANS algorithm as implemented in three different programming environments: PYTHON, C++, and CUDA. Each environment represents a unique approach to computational tasks, with PYTHON often hailed for its ease of use and integration, C++ recognized for its execution speed and memory control, and CUDA acclaimed for its capabilities in handling large-scale parallel computations. The primary objective of this study is to evaluate and contrast these environments in terms of execution efficiency, scalability, and practical usability in data clustering tasks. Through a detailed examination of the implementations and their performances on synthetic datasets, this paper seeks to provide valuable insights into the optimal choices of programming tools based on specific use-case scenarios in data analysis and machine learning. This introduction sets the stage for a comprehensive analysis of the K-MEANS algorithm across diverse computing platforms, highlighting the implications of each for the field of data mining.

3 Background

In the clustering problem instance, we are given a training set $X = \{x^{(1)}, \ldots, x^{(n)}\}$ of n observations, and want to group X into k cohesive "clusters." Here, each entry of X, say $x^{(i)}$, is in $\mathbb{R}^d$ as usual; but no labels $y^{(i)}$ are given, making this clustering problem instance effectively an unsupervised learning problem [2]. Solving the closed form algebraic representation of the K-MEANS algorithm via minimizing the intra-cluster variance among the data points is in actuality as hard as solving any other problem in NP [3]. To circumvent this computational intractability, we resort to an iterative approximation algorithm due to Lloyd (1957) which

decomposes the K-MEANS problem into two distinct, alternating steps: the assignment step (found in Step 4 of Algorithm 1) and the update step (found in Step 7 of Algorithm 1) [4].

Algorithm 1 K-MEANS($\{x^{(1)}, \ldots, x^{(n)}\}, k$)

1: Initialized **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^d$ randomly.
2: **repeat**
3: **for** $i \leftarrow 1$ **to** n **do**
4: $c^{(i)} \leftarrow \arg\min_j ||x^{(i)} - \mu_j||^2$. // Assigning Step
5: **end for**
6: **for** $j \leftarrow 1$ **to** k **do**
7: $\mu_j \leftarrow \frac{\sum_{i=1}^{n} \mathbb{1}\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{n} \mathbb{1}\{c^{(i)} = j\}}$ // Updating Step
8: **end for**
9: **until** CONVERGENCE

In the algorithm above, k (a parameter of the algorithm) is number of clusters we want to find; and the cluster centroid μ_j represent our current guesses for the positions of the centers of the clusters. To initialize the cluster centroids in Step 1 of Algorithm 1, we choose k independent training examples at random, and set the cluster centroids to be equal to the values of these k examples. The outer-loop of the algorithm repeatedly carries out the two steps discussed above: (1) "assigning" each training example $x^{(i)}$ to the closest cluster centroid μ_j, and (2) moving each cluster centroid μ_j to the mean of the points assigned to it.[1]

The convergence in Step 9 of Algorithm 1 is affected by the initial selection of the centroids in Step 1, which is often addressed by multiple random initializations or by sophisticated methods like the kmeans++ algorithm, which probabilistically chooses initial centroids to improve cluster quality. The computational complexity of K-MEANS is generally $O(nkdt)$, where n is number of points, k is number of clusters, d is the dimensionality of each data point, and t is the number of iterations required for convergence.

For the purpose of this paper, we note that parallelization of the K-MEANS algorithm enhances its performance significantly, making it feasible to handle vast and complex datasets that are typical in modern data-driven industries. The technique involves dividing the dataset into smaller chunks that can be processed concurrently across multiple computing cores or nodes. Each node computes the centroids of the clusters for its assigned data segment, and these partial results are then aggregated to update the global centroids. This process reduces the overall computational burden and leads to faster convergence times. GPU-based implementations, particularly using CUDA, exploit the massive parallelism of modern GPUs, allowing thousands of threads to operate in parallel to perform calculations and update centroids in a fraction of the time required by single-threaded solutions. This parallel approach not only accelerates the computation but also scales efficiently with increasing data sizes and cluster complexities as we will see the next sections.

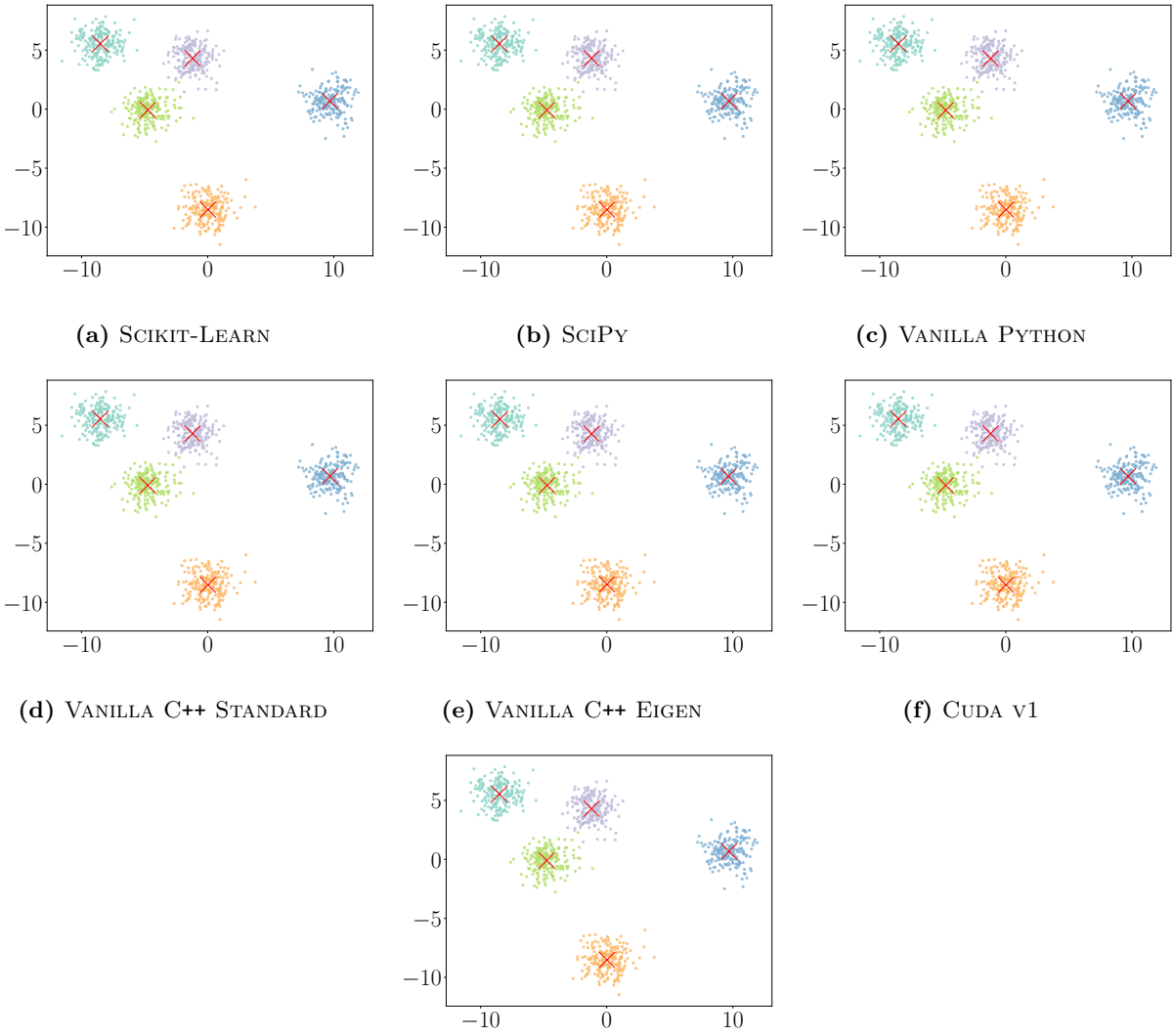
4 Dataset

In this project, we plan to generate synthetic datasets from well matured ML libraries such as `scikit-learn` using subroutines like `sklearn.datasets.make_blobs`, which allows for the creation of multi-dimensional datasets with controllable numbers of clusters (and centers) and samples. This method ensures controlled complexity and reproducibility, which are all essential for systematic benchmarking. Furthermore, these synthetic datasets enable precise evaluation of clustering algorithms across performance metrics such as speed, scalability, and accuracy since the true, ground-truth cluster assignments are known. Additionally, they allow for algorithm tuning and optimization in response to varied data characteristics such as cluster separation and density, all while avoiding the privacy concerns associated with real-world data. Finally, synthetic datasets not only provides flexibility in adjusting the complexity and size of the data but also enables the benchmarking of clustering performance across different programming paradigms (PYTHON, C++, CUDA) and dataset scales. In fact, this choice in data acquisition will allow a (possibly useful) controlled

[1] Outline weakly adapted from: https://cs229.stanford.edu/notes2020spring/cs229-notes7a.pdf.

environment to systematically assess the algorithm's performance across different cluster distributions and configurations; thereby, ensuring repeatability and comparability of results, crucial for a rigorous benchmark analysis.

5 Results



(a) Scikit-Learn **(b)** SciPy **(c)** Vanilla Python

(d) Vanilla C++ Standard **(e)** Vanilla C++ Eigen **(f)** Cuda v1

(g) Cuda v2

Figure 1. Different implementations of K-means algorithm on 1K.txt dataset.

6 Benchmark

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	**0.003521**	0.034318	0.535300	8.836155
SciPy	0.027732	0.166335	1.474478	14.327702
Vanilla Python	0.101650	1.221879	20.48854	191.42883
Vanilla C++ std	0.008865	0.032988	0.254328	2.671155
Vanilla C++ eigen	0.008202	0.034422	0.294001	3.425298
Cuda v1	0.009314	0.041698	0.090047	0.652350
Cuda v2	0.006274	**0.029613**	**0.053129**	**0.215813**

Table 1. Runtime Benchmark — this table shows the runtime performance measured in seconds of different K-means implementations across various data sizes, highlighting how GPU-accelerated versions (Cuda v1 and Cuda v2) dramatically reduce computation times, especially in larger datasets, compared to CPU-based implementations (Scikit-Learn, SciPy, Vanilla Python, Vanilla C++ standard, and Vanilla C++ eigen).

Pictorially, Table 1 translates to the following plot:

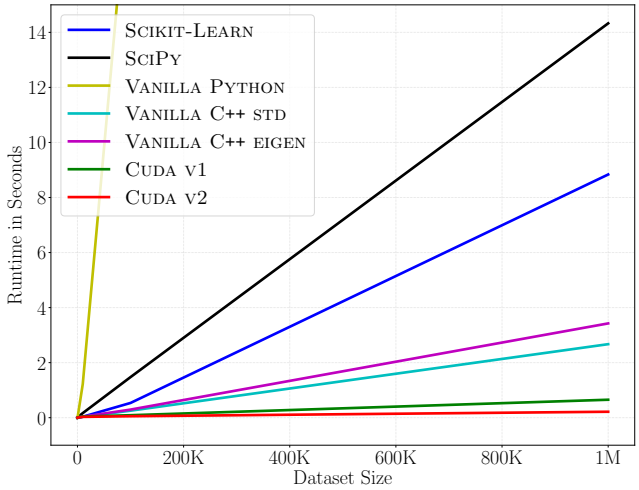


Figure 2. Runtime Benchmark.

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	0.756701	0.756096	0.749256	0.754255
SciPy	0.756701	0.752882	0.752448	0.752540
Vanilla Python	0.756701	0.759022	0.749302	0.752429
Vanilla C++ std	0.756701	0.752345	0.752440	0.745546
Vanilla C++ eigen	0.756701	0.756139	0.747363	0.755571
Cuda v1	0.756701	0.753208	0.755416	0.746047
Cuda v2	0.756701	0.753208	0.755416	0.746047

Table 2. Silhouette Score of the Mean Silhouette Coefficient of all Samples — this table presents the silhouette scores for each implementation across different dataset sizes, illustrating generally consistent clustering quality across all implementations with slight variation as the dataset size increases from left to right.

Pictorially, Table 2 translates to the following plot:

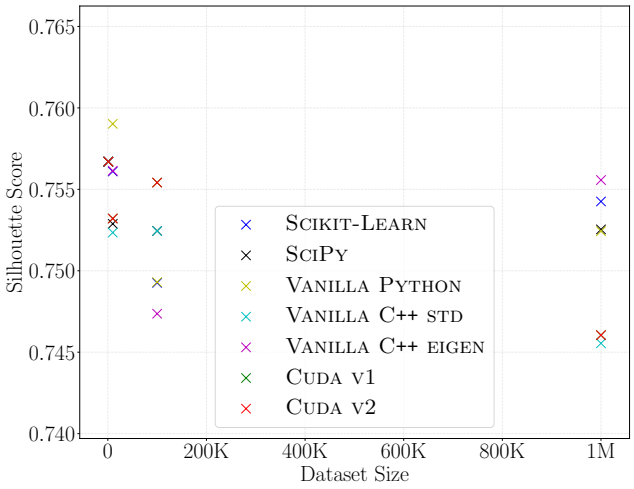


Figure 3. Silhouette Scores.

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	0.996065	0.993236	0.994860	0.994549
SciPy	0.996065	0.993236	0.994860	0.994549
Vanilla Python	0.996065	0.993236	0.994860	0.994549
Vanilla C++ std	0.996065	0.993236	0.994860	0.994549
Vanilla C++ eigen	0.996065	0.993236	0.994860	0.994549
Cuda v1	0.996065	0.993236	0.994860	0.994549
Cuda v2	0.996065	0.993236	0.994860	0.994549

Table 3. Adjusted Mutual Score (i.e. Accuracy Permutation Independent Score) — this table displays the adjusted mutual scores for each implementation, and confirms the accuracy and consistency of clustering results across different programming environments, indicating high similarity in cluster assignments regardless of the underlying computational approach.

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	1.390736	1.397050	1.410765	1.413165
SciPy	1.390736	1.397050	1.410765	1.413165
Vanilla Python	1.390706	1.397045	1.410765	1.413165
Vanilla C++ std	1.390706	1.397045	1.410765	1.413165
Vanilla C++ eigen	1.391263	1.397050	1.410765	1.413165
Cuda v1	1.390706	1.397045	1.410765	1.413165
Cuda v2	1.390706	1.397045	1.410765	1.413165

Table 4. k-means cost — this table provides the k-means cost for each implementation and dataset size, showing uniform costs across all methods, which implies consistent performance in terms of clustering efficiency.

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	3.940704	4.167821	4.924603	5.492129
SciPy	3.940704	4.167821	4.924603	5.492129
Vanilla Python	3.940704	4.167821	4.924530	5.491405
Vanilla C++ std	3.940705	4.167809	4.924526	5.491415
Vanilla C++ eigen	3.913538	4.163677	4.924756	5.491409
Cuda v1	3.940705	4.167809	4.924526	5.491415
Cuda v2	3.940705	4.167809	4.924526	5.491415

Table 5. k-center cost — listing the k-center costs for each K-means implementation, this table reveals identical costs across all environments, suggesting that the centroid positioning is effectively consistent among different implementations.

Algo/Lib	1K.txt	10K.txt	100K.txt	1M.txt
Scikit-Learn	1.234888	1.238547	1.250711	1.252577
SciPy	1.234888	1.238547	1.250711	1.252577
Vanilla Python	1.234840	1.238538	1.250713	1.252577
Vanilla C++ std	1.234840	1.238538	1.250713	1.252577
Vanilla C++ eigen	1.235289	1.238534	1.250712	1.252577
Cuda v1	1.234840	1.238538	1.250713	1.252577
Cuda v2	1.234840	1.238538	1.250713	1.252577

Table 6. k-medians cost — the results for k-medians cost from this table shows nearly the same for all implementations across various dataset sizes, highlighting a uniform distribution of medians in the clustering results produced by each implementation environment.

From the benchmarks, it is evident that all K-means implementations produce identically the same clusters with nearly identical adjusted mutual scores (see Table 3), silhouette scores (see Table 2), kmeans cost (see Table 4), kcenter cost (see Table 5), and kmedians cost (see Table 6). The contrast, however, rests on the runtime taken to produce these clusters, where we see a distinct difference in the runtime performance of the K-means implementations across various programming environments and datasets sizes.

Starting with Python implementations, Scikit-Learn demonstrates highly efficient runtime characteristics, especially noticeable in smaller datasets. For example, its runtime is merely **0.003521** seconds for the 1K.txt dataset, scaling up to **8.836155** seconds for the 1M.txt dataset. SciPy show less optimized performance compared to Scikit-Learn, starting from **0.027732** seconds and going up to **14.327702** seconds for the same dataset sizes, indicating a slightly higher computational overhead.

The Vanilla Python implementation, unsurprisingly, shows the least efficiency among the Python variants, with a dramatic increase in runtime as dataset size grows (from **0.101650** seconds to a staggering **191.423883** seconds), reflecting the lack for low-level optimizations and reliance on pure Python's slower execution speed.

In contrast, the C++ implementation offer significant runtime reductions. Both Vanilla C++ standard and Vanilla C++ eigen start at approximately **0.008** seconds for the 1K.txt dataset and remain under **4** seconds even for the 1M.txt dataset. This is indicative of the effectiveness of C++'s low-level optimizations and memory management capabilities, which provide additional computational efficiency for matrix operations and numerical solvers.

The Cuda implementations show exceptional runtime performance by exploiting GPU parallelism. Cuda v1 starts at **0.009341** seconds for the 1K.txt dataset and scales up to **0.0652350** seconds for the 1M.txt dataset. Cuda v2, an even more optimized version which uses a logarithmic tree-reduction on its shared memory (see Listing 7), begins **0.006274** and ends at **0.215813** seconds for the 1M.txt dataset, showcasing the substantial benefits of fine-tuned GPU computations in handling large-scale data efficiently.

These runtime performances are graphically represented in Figure 2, where we can visually assess the scalability and efficiency of each implementation as datasets size increases. This comparison highlights the substantial runtime performance gains that can be attained through more sophisticated programming environments and the use of specific computational techniques like GPU acceleration in CUDA. Such insights are crucial for choosing the right computing platform based on the specific needs of large-scale data mining tasks, where both data size and computational efficiency are critical factors.

7 Discussion

In this comparative study of K-MEANS implementations across various programming environments, the performance metrics clearly favor the CUDA V2 implementation on GPUS. This implementation not only excels in handling large datasets but also drastically reduces computational times when compared to more traditional CPU-based approaches like SCIKIT-LEARN and SCIPY.

Specifically, CUDA V2 outperforms SCIKIT-LEARN by up to **41 times** and surpasses SCIPY by up to **66 times** on the 1M.txt dataset. Such a stark difference in performance underscores the significant advantage of utilizing GPU-based parallel processing over CPU-based methods, particularly for task involving large volumes of data.

The superior performance of CUDA V2 can be attributed to its ability to leverage the massive parallelism of modern GPUs. This allows thousands of threads to operate simultaneously, significantly speeding up the computation and updating of centroids, which are crucial operations in the K-MEANS algorithm. This capability not only provides a boost in speed but also enhances scalability, making it a suitable choice for real-world applications that require handling of big data efficiently.

However, while CUDA V2 presents substantial benefits in terms of speed and scalability, it necessitates specific hardware and expertise in parallel programming, which may not be readily available in all settings. In contrast, SCIKIT-LEARN and SCIPY, while slower, offer greater accessibility and ease of use, with extensive support and documentation that can be advantageous for novice users with less technical expertise or computational resources.

In conclusion, the choice between these environments should be guided by the specific requirements of the task, including dataset size, computational resources, and available expertise. For large-scale applications where time efficiency is paramount, and appropriate hardware is available, CUDA V2 is clearly the superior contender. Conversely, for smaller projects or environments where simplicity and accessibility are more critical, SCIKIT-LEARN or SCIPY might be more appropriate despite their slower performance.

8 References

[1] A. Blum, J. Hopcroft, and R. Kannan, *Foundations of Data Science*. Cambridge: Cambridge University Press, 2020, ISBN: 9781108485067. DOI: DOI:. [Online]. Available: https://www.cambridge.org/core/books/foundations-of-data-science/6A43CE830DE83BED6CC5171E62B0AA9E.

[2] A. Ng, "The k-means clustering algorithm," 2012. [Online]. Available: https://cs229.stanford.edu/notes2020spring/cs229-notes7a.pdf.

[3] J. Phillips, *Mathematical Foundations for Data Analysis* (Springer Series in the Data Sciences). Springer International Publishing, 2021, ISBN: 9783030623401. [Online]. Available: https://books.google.com/books?id=AUDYzQEACAAJ.

[4] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, Second. Cambridge University Press, 2014, ISBN: 9781316147313 1316147312 9781139924801 113992480X 9781316147047 1316147045 1107077230 9781107077232. [Online]. Available: http://mmds.org.

9 Appendix

9.1 Scikit-Learn

```python
def main() -> None:
    args = parse_args()
    data = load_data(args.data_file)
    model = sklearn.cluster.KMeans(
        args.num_clusters,
        max_iter=args.max_iter,
        init='random',
        algorithm='full',
        tol=0,
        n_init=1
    )

    tic = time.perf_counter()
    for _ in range(args.num_trials):
        means = model.fit(data).cluster_centers_
    toc = time.perf_counter()

    dump_means(means)
    print(f'# runtime: {(toc-tic) / args.num_trials:.6f}')

if __name__ == '__main__':
    main()
```

Listing 1. kmeans-skl.py

9.2 SciPy

```python
def main() -> None:
    args = parse_args()
    data = load_data(args.data_file)
    model = scipy.cluster.vq.kmeans2

    tic = time.perf_counter()
    for _ in range(args.num_trials):
        means, _ = model(data, args.num_clusters, iter=args.max_iter, minit='points')
    toc = time.perf_counter()

    dump_means(means)
    print(f'# runtime: {(toc-tic) / args.num_trials:.6f}')

if __name__ == '__main__':
    main()
```

Listing 2. kmeans-sci.py

9.3 Vanilla Python

```python
def kmeans(data: np.ndarray, num_clusters: int, max_iter: int) -> np.ndarray:
    num_samples = data.shape[0]
    num_features = data.shape[1]
    initial_indices = np.random.choice(num_samples, num_clusters)
    means = data[initial_indices].T
```

```
6
7       data_repeat = np.stack([data] * num_clusters, axis=-1)
8       all_rows = np.arange(num_samples)
9       all_zero = np.zeros([1,1,2])
10
11      for _ in range(max_iter):
12          distances = np.sum(np.square(data_repeat - means), axis=1)
13          label = np.argmin(distances, axis=-1)
14          sparse = np.zeros([num_samples, num_clusters, num_features])
15          sparse[all_rows, label] = data
16          counts = (sparse != all_zero).sum(axis=0)
17          means = sparse.sum(axis=0).T / counts.clip(min=1).T
18
19      return means.T
```

Listing 3. kmeans-vnl.py

9.4 Vanilla C++ standard

```
1   DataFrame kmeans(const DataFrame &data, int num_clusters, int max_iter) {
2     static std::default_random_engine seeder(seed);
3     static std::mt19937_64 generator(seeder());
4     std::uniform_int_distribution<size_t> indices(0, data.size() - 1);
5
6     DataFrame means(num_clusters);
7     for (auto &cluster : means)
8       cluster = data[indices(generator)];
9
10    std::vector<size_t> labels(data.size());
11    for (size_t iteration = 0; iteration < max_iter; ++iteration) {
12      for (size_t point = 0; point < data.size(); ++point) {
13        float best_distance = std::numeric_limits<float>::max();
14        size_t best_cluster = 0;
15        for (size_t cluster = 0; cluster < num_clusters; ++cluster) {
16          const float distance = norm(data[point], means[cluster]);
17          if (distance < best_distance) {
18            best_distance = distance;
19            best_cluster = cluster;
20          }
21        }
22        labels[point] = best_cluster;
23      }
24
25      DataFrame new_means(num_clusters);
26      std::vector<size_t> counts(num_clusters, 0);
27      for (size_t point = 0; point < data.size(); ++point) {
28        const auto cluster = labels[point];
29        new_means[cluster].x += data[point].x;
30        new_means[cluster].y += data[point].y;
31        counts[cluster] += 1;
32      }
33
34      for (size_t cluster = 0; cluster < num_clusters; ++cluster) {
35        // turn 0/0 into 0/1 to avoid zero division
36        const auto count = std::max<size_t>(1, counts[cluster]);
37        means[cluster].x = new_means[cluster].x / count;
38        means[cluster].y = new_means[cluster].y / count;
39      }
```

```
40    }
41    return means;
42  }
```

Listing 4. kmeans-vnl.cc

9.5 Vanilla C++ eigen

```
1  Eigen::ArrayXXf kmeans(const Eigen::ArrayXXf &data, int num_clusters, int max_iter) {
2    static std::default_random_engine seeder(seed);
3    static std::mt19937_64 generator(seeder());
4    std::uniform_int_distribution<size_t> indices(0, data.size() - 1);
5
6    Eigen::ArrayX2f means(num_clusters, 2);
7    for (size_t cluster = 0; cluster < num_clusters; ++cluster)
8      means.row(cluster) = data(indices(generator));
9
10   const Eigen::ArrayXXf x_data = data.col(0).rowwise().replicate(num_clusters);
11   const Eigen::ArrayXXf y_data = data.col(1).rowwise().replicate(num_clusters);
12
13   for (size_t iteration = 0; iteration < max_iter; ++iteration) {
14     Eigen::ArrayXXf distances =
15       (x_data.rowwise() - means.col(0).transpose()).square() +
16       (y_data.rowwise() - means.col(1).transpose()).square();
17
18     Eigen::ArrayX2f sigma = Eigen::ArrayX2f::Zero(num_clusters, 2);
19     Eigen::ArrayXf counts = Eigen::ArrayXf::Ones(num_clusters);
20
21     for (size_t index = 0; index < data.rows(); ++index) {
22       Eigen::ArrayXf::Index argmin;
23       distances.row(index).minCoeff(&argmin);
24       sigma.row(argmin) += data.row(index).array();
25       counts(argmin) += 1;
26     }
27
28     means = sigma.colwise() / counts;
29   }
30   return means;
31 }
```

Listing 5. kmeans-eig.cc

9.6 Cuda v1

```
1  __global__ void assign_clusters(
2    const float* __restrict__ x_data,
3    const float* __restrict__ y_data,
4    int data_size,
5    const float* __restrict__ x_means,
6    const float* __restrict__ y_means,
7    float* __restrict__ x_new_sums,
8    float* __restrict__ y_new_sums,
9    int num_clusters,
10   int* __restrict__ counts
11 ) {
12   const int index = blockIdx.x * blockDim.x + threadIdx.x;
13   if (index ⩾ data_size) return;
```

```
14
15    const float x = x_data[index];
16    const float y = y_data[index];
17
18    float best_distance = FLT_MAX;
19    size_t best_cluster = 0;
20    for (size_t cluster = 0; cluster < num_clusters; ++cluster) {
21      const float distance = norm(x, y, x_means[cluster], y_means[cluster]);
22      if (distance < best_distance) {
23        best_distance = distance;
24        best_cluster = cluster;
25      }
26    }
27
28    atomicAdd(&x_new_sums[best_cluster], x);
29    atomicAdd(&y_new_sums[best_cluster], y);
30    atomicAdd(&counts[best_cluster], 1);
31  }
32
33  __global__ void compute_new_means(
34    float* __restrict__ x_means,
35    float* __restrict__ y_means,
36    const float* __restrict__ x_new_sums,
37    const float* __restrict__ y_new_sums,
38    const int* __restrict__ counts
39  ) {
40    const int cluster = threadIdx.x;
41    const int count = max(counts[cluster], 1);
42    x_means[cluster] = x_new_sums[cluster] / count;
43    y_means[cluster] = y_new_sums[cluster] / count;
44  }
```

Listing 6. kmeans-gpu1.cu

9.7 CUDA v2

```
1   __global__ void fine_reduce(
2     const float* __restrict__ x_data,
3     const float* __restrict__ y_data,
4     int data_size,
5     const float* __restrict__ x_means,
6     const float* __restrict__ y_means,
7     float* __restrict__ x_new_sums,
8     float* __restrict__ y_new_sums,
9     int num_clusters,
10    int* __restrict__ counts
11  ) {
12    extern __shared__ float shared_data[];
13
14    const int local_index = threadIdx.x;
15    const int global_index = blockIdx.x * blockDim.x + threadIdx.x;
16    if (global_index ⩾ data_size) return;
17
18    if (local_index < num_clusters) {
19      shared_data[local_index] = x_means[local_index];
20      shared_data[local_index + num_clusters] = y_means[local_index];
21    }
22
```

```
  __syncthreads();

  const float x_value = x_data[global_index];
  const float y_value = y_data[global_index];

  float best_distance = FLT_MAX;
  size_t best_cluster = 0;

  for (size_t cluster = 0; cluster < num_clusters; ++cluster) {
    const float distance = norm(
      x_value, y_value,
      shared_data[cluster],
      shared_data[cluster + num_clusters]
    );
    if (distance < best_distance) {
      best_distance = distance;
      best_cluster = cluster;
    }
  }

  __syncthreads();

  const int x = local_index;
  const int y = local_index + 1 * blockDim.x;
  const int z = local_index + 2 * blockDim.x;

  for (size_t cluster = 0; cluster < num_clusters; ++cluster) {
    shared_data[x] = (best_cluster == cluster) ? x_value : 0;
    shared_data[y] = (best_cluster == cluster) ? y_value : 0;
    shared_data[z] = (best_cluster == cluster) ? 1 : 0;
    __syncthreads();

    for (size_t stride = blockDim.x / 2; stride > 0; stride /= 2) {
      if (local_index < stride) {
        shared_data[x] += shared_data[x + stride];
        shared_data[y] += shared_data[y + stride];
        shared_data[z] += shared_data[z + stride];
      }
      __syncthreads();
    }

    if (local_index == 0) {
      const int cluster_index = blockIdx.x * num_clusters + cluster;
      x_new_sums[cluster_index] = shared_data[x];
      y_new_sums[cluster_index] = shared_data[y];
      counts[cluster_index] = shared_data[z];
    }
    __syncthreads();
  }
}

__global__ void coarse_reduce(
  float* __restrict__ x_means,
  float* __restrict__ y_means,
  float* __restrict__ x_new_sum,
  float* __restrict__ y_new_sum,
  int num_clusters,
  int* __restrict__ counts
) {
```

```
82    extern __shared__ float shared_data[];
83
84    const int index = threadIdx.x;
85    const int y_offset = blockDim.x;
86
87    shared_data[index] = x_new_sum[index];
88    shared_data[index + y_offset] = y_new_sum[index];
89    __syncthreads();
90
91    for (size_t stride = blockDim.x / 2; stride ⩾ num_clusters; stride /= 2) {
92      if (index < stride) {
93        shared_data[index] += shared_data[index + stride];
94        shared_data[index + y_offset] += shared_data[index + stride + y_offset];
95      }
96      __syncthreads();
97    }
98
99    if (index < num_clusters) {
100     const int count = max(counts[index], 1);
101     x_means[index] = x_new_sum[index] / count;
102     y_means[index] = y_new_sum[index] / count;
103     x_new_sum[index] = 0;
104     y_new_sum[index] = 0;
105     counts[index] = 0;
106   }
107 }
```

Listing 7. kmeans-gpu2.cu