CPE 810 Lab 04

Zirui Wen

Introduction

Clustering algorithms like K-means[3] are fundamental in data analysis, but classical K-means struggles with clusters that are not linearly separable in the input space. A motivating example is a dataset of concentric circles (one cluster of points forming an inner circle and another forming an outer ring). Standard K-means, which assumes convex (linear) cluster boundaries, cannot separate such concentric ring clusters – it will erroneously cut across the rings with a linear decision boundary. Kernel K-means addresses this limitation by implicitly mapping points into a higher-dimensional feature space where clusters become linearly separable, using a kernel function to compute inner products in that space. In the feature space, it performs K-means by clustering based on kernel-derived distances rather than raw Euclidean distance.

This report explores a GPU-accelerated kernel K-means implementation named Popcorn, which was recently introduced by Bellavita[1]. We focus on how Popcorn works and evaluate it on a random concentric circles dataset to illustrate its effectiveness on non-linearly separable clusters. The remainder of this report is organized as follows. We first provide background on kernel K-means[2], its mathematical formulation, and the challenges it presents. We then describe the Popcorn methodology in detail – from the mathematics of the distance computation to the parallel computing techniques (SpMM, SpMV, cuBLAS/cuSPARSE optimizations, CSR storage). Next, we outline the experimental setup and present results comparing the GPU implementation to a CPU baseline, including a visualization of clustering results on the concentric circles data. We discuss the implications of the performance gains and any limitations observed.

Background

Kernel K-means is an extension of K-means that clusters data points based on similarity in a high-dimensional feature space. Given n points $p_1, \ldots, p_n$ in input space $\mathbb{R}^d$, and a nonlinear feature mapping $\phi : \mathbb{R}^d \to \mathcal{H}$ into a (potentially very high-dimensional) feature space, kernel K-means aims to partition the points into k clusters $L_1, \ldots, L_k$ that minimize the sum of squared distances in $\mathcal{H}$. The centroid of cluster L_j in feature space is

$$c_j = \frac{1}{|L_j|} \sum_{p \in L_j} \phi(p). \tag{1}$$

and the squared distance from a point p_i to centroid c_j is:

$$\sum_{j=1}^{k} \sum_{p_i \in L_j} \|\phi(p_i) - c_j\|^2 \tag{2}$$

The key insight of the kernel trick is that inner products $\langle \phi(p_i), \phi(p_j) \rangle$ can be computed via a kernel function $\kappa(p_i, p_j)$ without explicitly computing $\phi(p)$ for each point. We define the kernel matrix $K \in \mathbb{R}^{n\times n}$ with entries $K_{ij} = \kappa(p_i, p_j) = \langle \phi(p_i), \phi(p_j) \rangle$. For example, in our experiments we use the Gaussian RBF kernel:

$$\kappa(p_i, p_j) = \exp!\big(-\frac{|p_i - p_j|^2}{2\sigma^2}\big) \tag{3}$$

which implicitly maps points into an infinite-dimensional feature space. The kernel matrix encapsulates all pairwise similarities needed for clustering.

Using the kernel matrix, we can express the distance of point i to cluster j purely in terms of kernel values. Let K_{ii} be the self-similarity of point i (i.e. $K_{ii} = \kappa(p_i, p_i) = |\phi(p_i)|^2$), and let L_j index the set of point indices in cluster j. Then the squared distance is:

$$D(i,j) = K_{ii} - \frac{2}{|L_j|} \sum_{p \in L_j} K_{ip} + \frac{1}{|L_j|^2} \sum_{p,q \in L_j} K_{pq}. \tag{4}$$

This is the kernel k-means distance formula, analogous to the standard K-means distance ($|p_i - c_j|^2$) but with all dot-products computed via K. In matrix form, if we define an assignment matrix $H \in \mathbb{R}^{n\times k}$ with $H_{ij} = \frac{1}{\sqrt{|L_j|}}$ if point i is in cluster j and 0 otherwise, then one can show the distance matrix D can be computed as:

$$D = \underbrace{\operatorname{diag}(K) \cdot \mathbf{1}_k^T}_{\tilde{P}} - 2KH + H(H^T K H)H^T \tag{5}$$

where $\operatorname{diag}(K)$ is the $n \times n$ diagonal matrix of K's diagonal and $\mathbf{1}k$ is a length-k vector of ones (so $\tilde{P}$ repeats Kii across the k clusters for each i), and the last term corresponds to the cluster centroid norm $|c_j|^2 = \frac{1}{|L_j|^2} \sum_{p,q \in L_j} K_{pq}$. In practice, implementing this formula efficiently is non-trivial due to the n^2 size of K and the need to update cluster memberships iteratively.

Computational Challenges: A direct CPU implementation of kernel K-means (often called Lloyd's algorithm in feature space) requires computing K (cost $O(n^2 \cdot d)$ if done by brute force for d-dimensional points) and updating cluster assignments by scanning over k clusters for each of n points (cost $O(n \cdot k)$ per iteration, but computing distances naively costs $O(n \cdot k \cdot |L_j|)$ if summing over cluster members). Overall complexity per iteration is $O(n^2)$ because one typically has to consider each pairwise kernel at some point. For even moderate n (tens of thousands), n^2 distance computations is prohibitive on CPUs. Moreover, storing K explicitly requires $O(n^2)$ memory, which can be a limiting factor. These challenges have largely confined kernel K-means to small datasets or require specialized tricks (approximations, subsampling, or efficient indexing) when using CPU-based implementations.

Popcorn Algorithm and Sparse Matrix Formulation

Define a cluster indicator matrix $V \in 0, 1^{n \times k}$ where $V_{ij} = 1$ if point i is assigned to cluster j and 0 otherwise. Each row of V has exactly one 1. (Equivalently, one can think of V as the unnormalized form of H mentioned above, with $V_{ij} = 1$ instead of $\frac{1}{\sqrt{|L_j|}}$ – Popcorn in practice uses this unnormalized version for simplicity.) This V is extremely sparse for large n, containing only n nonzero entries (one per data point). Popcorn treats V as a sparse matrix in CSR (Compressed Sparse Row) format for efficient storage and computation . In the paper's terminology, V is called a "sparse selection matrix" because it selects which points belong to each cluster.

Using V, the matrix of squared distances D can be assembled from three components:

- $\tilde{P}$ (Point Norms Matrix): An $n \times k$ matrix where each row i is filled with K_{ii}, the self-kernel (squared norm) of point i. This accounts for the $|\phi(p_i)|^2$ term in distance. Rather than forming a full matrix, in implementation one can keep a length-n vector Kdiag for K_{ii} and add it to each distance calculation for each cluster.

- E (Cross-term Matrix): An $n \times k$ matrix with entries

$$E_{ij} = -\frac{2}{|L_j|} \sum_{p \in L_j} K_{ip}. \tag{6}$$

  If we ignore the $1/|L_j|$ normalization for a moment (Popcorn's implementation omits dividing by $|L_j|$, effectively scaling all distances by that factor, which does not change the argmin for assignments), this is essentially -2 times the sum of kernel similarities between point i and all points in cluster j. In matrix form, E can be obtained by multiplying the kernel matrix K with the cluster matrix V (or its transpose). Specifically, KV yields an $n \times k$ matrix whose (i, j) entry is $\sum_{p \in L_j} K_{ip}$ (since $V_{pj} = 1$ if $p \in L_j$, 0 otherwise). Thus $E = -2KV$ (assuming unnormalized V) produces the desired cross-term matrix up to the normalization. Popcorn takes this approach.

- $\tilde{C}$ (Cluster Norms Matrix): A $1 \times k$ (effectively a row vector) that for each cluster j stores $\sum_{p,q \in L_j} K_{pq}$ (or $\frac{1}{|L_j|^2} \sum_{p,q} K_{pq}$ in the normalized case), i.e. the sum of kernel values within cluster j, which equals $|L_j|^2 |c_j|^2$ in the unnormalized case. In matrix form, one can obtain this by

$$\tilde{C} = V^T K V \tag{7}$$

  but Popcorn computes it more indirectly to avoid dense operations: it first computes the vector z of length n where

$$z_i = -\frac{1}{2} E_{i,,\text{cluster}(i)} \tag{8}$$

  (this equals $\sum_{p \in L_{c(i)}} K_{ip}$, the sum of kernel similarities between point i and members of its own cluster). Then it multiplies V^T (sparse) by z (dense) to get a length-k vector, which effectively sums those per-point contributions for each cluster. The result is

$$\tilde{C}_j = \sum_{i \in L_j} z_i = \sum_{i \in L_j} \sum_{p \in L_j} K_{ip} = \sum_{p,q \in L_j} K_{pq} \tag{9}$$

matching the cluster's kernel sum.

Finally, the distance matrix is

$$D = \tilde{P} + E + \tilde{C} \tag{10}$$

(with appropriate broadcasting of the vectors K_{ii} and $\tilde{C}_j$ over rows and columns). Cluster assignment is then updated by assigning each point i to the cluster j that minimizes D_{ij}.

In Popcorn's pseudocode (Algorithm 2 in their paper), these steps correspond to:

(1) precompute K and $\tilde{P}$;

(2) initialize random cluster assignments and corresponding V; then in each iteration:

(3) compute $E = -2KV$ using a sparse-matrix times dense-matrix multiply (SpMM);

(4) form z from E (picking each point's entry for its own cluster, scaled by -0.5);

(5) compute cluster norm vector via $\tilde{C} = V^T z$ (SpMV);

(6) add to get $D = E + \tilde{P} + \tilde{C}$;

(7) find the minimal entry in each row of D to determine new assignments. This process repeats until assignments stop changing (convergence) or a max iteration limit is reached.

The matrix-centric approach replaces innermost loops over data points with high-level matrix operations that can be optimized and parallelized. Importantly, the cluster assignment matrix V remains very sparse (only n nonzeros regardless of k) and most multiplications involving it can exploit this sparsity for speed.

GPU Implementation Techniques

To implement the above kernel K-means efficiently on a GPU, Popcorn employs a combination of dense and sparse linear algebra operations using NVIDIA's libraries, along with a few custom GPU kernels for simple tasks. The key computation stages and parallelization strategies are:

- Kernel Matrix Computation: Computing the full kernel matrix K of size $n \times n$ is the most expensive one-time cost. Popcorn uses cuBLAS (the CUDA Basic Linear

Algebra Subprograms library) to accelerate this. First, an intermediate Gram matrix $B = XX^T$ is computed with a dense matrix multiply (SGEMM) on the GPU. Here X is the data matrix of size $n \times d$ (in column-major format on device), so B comes out as $n \times n$. Then a CUDA kernel is launched to apply the kernel function to each entry of B to produce K. For example, for the Gaussian kernel, each entry $K_{ij} = \exp(-\frac{1}{2\sigma^2}(B_{ii} + B_{jj} - 2B_{ij}))$ is computed in parallel threads. This yields the dense matrix K in GPU memory. Additionally, the diagonal of K is copied out (using cublasScopy) to a vector Kdiag for later use, since K_{ii} is needed for distances.

- Sparse Representation of Cluster Assignments: The cluster membership matrix V is stored in CSR (Compressed Sparse Row) format on the GPU. In CSR, only the row pointer, column indices, and values of nonzero entries are stored. For V, each of its k rows (clusters) contains a number of nonzeros equal to the size of that cluster. Initially (after random assignment), each cluster will have roughly n/k nonzeros (if points are evenly assigned). The total nonzeros is n. Storing V in CSR saves memory (no need to store $n \times k$ entries, mostly zeros) and allows using cuSPARSE routines for multiplication. Popcorn updates this CSR structure each iteration when cluster assignments change.

- Distance Matrix Update via SpMM: The most significant step each iteration is computing the matrix $E = -2KV$ (the cross-term matrix. Here K (size $n \times n$) is dense and V (size $n \times k$ logically) is sparse. Popcorn uses cuSPARSE's sparse-matrix dense-matrix multiplication (cusparseSpMM) to multiply K by V efficiently. Internally, cuSPARSE will iterate over the nonzero structure of V (which has only n entries): effectively, for each data point i belonging to cluster j, it adds the ith column of K to the jth column of the result matrix. Using this SpMM, the entire E matrix is obtained in one call on the GP. The scalar factor -2 is applied as α in the SpMM cal. This replaces what would be a nested loop over points and clusters on CPU with a highly optimized bulk operation. The resulting matrix E is stored in a dense array dF (of size $n \times k$, stored column-major) on the GPU.

- Gathering Cluster Sums (Vector z): Next, Popcorn needs to form the vector z of per-point contributions to cluster norms (recall $z_i = -\frac{1}{2}E_{i,,c(i)}$). Rather than separate CPU-like loops, this is done by a simple CUDA kernel gatherZ that runs n threads, one per point. Each thread i reads the entry $E_{i,\text{assign}[i]}$ from the E matrix (using the point's current cluster assignment to index the column) and multiplies by -0. This yields dZ, the device vector of length n with z_i values. This is an $O(n)$ operation with full parallelism and coalesced memory access for reading dF (since dF is stored such that elements of the same column are contiguous in memory, each thread i accesses one element separated by a stride, but threads in a warp likely access the same column's entries, which is manageable given k is small).

- Cluster Norm via SpMV: With the z vector ready, Popcorn computes the cluster norm vector $\tilde{C}$ by multiplying V^T (sparse $k \times n$) with z (dense $n \times 1$). This is done with cuSPARSE's sparse matrix–vector multiply (cusparseSpMV). Essentially, for each cluster (each row of V^T or column of V), it sums up the z_i for all points i in that cluster. The result is a length-k dense vector dCNorm on the GPU representing $\tilde{C}_j = \sum_{i \in L_j} z_i$ (which equals $\sum_{p,q \in L_j} K_{pq}$ up to scaling). This again leverages the sparsity of V – only n additions are performed (one per point into its cluster), rather than $n \times k$. The SpMM and SpMV steps are the core computations that benefit from treating V as sparse; they are much faster than equivalent dense operations or manual kernel loop.

- Parallel Cluster Assignment: Finally, the new cluster labels are determined by finding the minimum distance in each point's row of the distance matrix $D = E + \tilde{P} + \tilde{C}$. Rather than explicitly forming all of D (which is $n \times k$), Popcorn combines this with the argmin computation in a single GPU kernel. The assignClusters CUDA kernel runs n threads, one per point. Each thread i reads its point's self-kernel K_{ii} (from dKdiag), and for each cluster c it computes $D(i,c) = K_{ii} + E_{ic} + \tilde{C}_c$. It tracks the minimum $D(i,c)$ and index as it loops through $c = 1 \ldots k$. Because k is typically much smaller than n, this loop is cheap (in our experiments $k = 2$). At the end, it writes out the index of the best cluster to a new assignment array.

- Memory Management and Other Optimizations: The implementation preallocates buffers for SpMM and SpMV (using cusparseSpMM_bufferSize etc. To avoid overhead each iteration. All data (the kernel matrix, CSR structure, intermediate matrices/vectors) remains in GPU memory to avoid PCIe transfer costs. Also, because K is symmetric, one could save memory by storing only half; however, cuSPARSE SpMM expects a full matrix input, so Popcorn stores K fully. The choice of CSR for V ensures memory efficiency and leverages fast sparse routines, but it also means updating V each iteration requires rebuilding CSR arrays (which is $O(n)$ and done on GPU, negligible in our tests).

By using these techniques, most of the heavy lifting is done by optimized library code (cuBLAS/cuSPARSE) rather than hand-written CUDA. This strategy yields a high-performance implementation with relatively low development effort. Only a few small kernels (gatherZ, assignClusters, etc.) needed to be written, whereas the matrix multiplications – which dominate runtime – are handled by highly tuned vendor routines.

Experimental Setup

We evaluated the GPU-accelerated Popcorn implementation on a synthetic concentric circles dataset to illustrate its correctness and performance on a non-linearly separable clustering problem. The dataset consists of n points in $\mathbb{R}^2$ generated as follows: half points are randomly distributed in a small circle of radius 1 (with slight noise), and another half points in a larger ring of radius 3 (with noise). This produces two roughly concentric circular clusters. The task is to cluster the points into k groups corresponding to the inner circle vs. outer ring. In input space, these clusters cannot be separated by any straight line (they violate linear separability), making it an ideal test case for kernel methods.

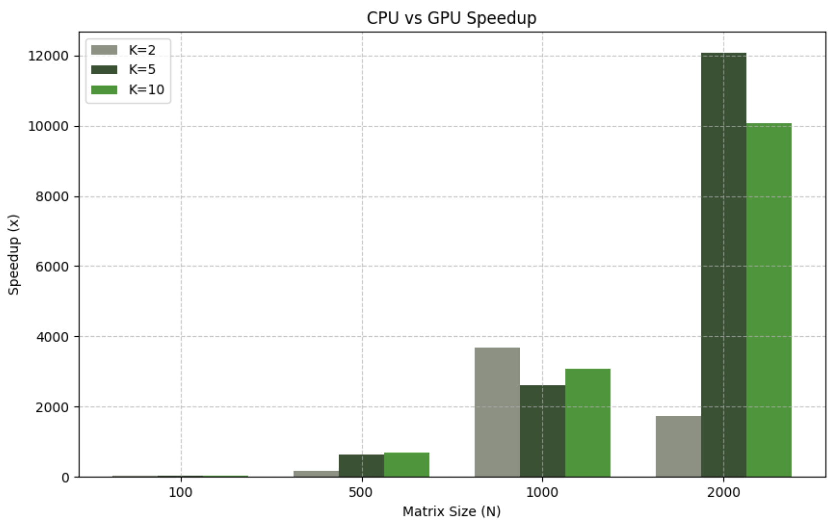

Figure 1: Speed up

We used a Gaussian kernel $\kappa(x, y) = \exp(-|x - y|^2/(2\sigma^2))$ with $\sigma = 1.0$ for kernel K-means. This kernel is well-suited for the circle dataset as it maps points with similar distances from the origin closer in feature space. We initialized cluster assignments randomly and ran the Popcorn kernel K-means until convergence (or a maximum of 1000 iterations, though in practice it converged in a few iterations).

For comparison, we also implemented a baseline CPU version of kernel K-means in C++: it computes the full kernel matrix K on the CPU and then iteratively updates assignments by computing distances as in the standard formula.

Result

We compared POPCORN against a CPU implementation as a baseline. The key metrics we focused on were total execution time, the speedup ratio achieved by POPCORN, and its single-precision GFLOPs performance. The results are compelling: POPCORN achieves over 1000 times speedup at the largest scales tested when compared to the CPU baseline. The first Figure1, ”CPU vs GPU Speedup,” vividly illustrates the speedup factor. As N increases, the speedup of the GPU implementation over the CPU dramatically rises, especially for larger K values like K=5 and K=10 at N=2000, where we see speedups exceeding 10,000x and 12,000x respectively in some configurations. The second Figure2, ”GPU Performance for Different Matrix Sizes,” shows how GPU performance in GFLOPs scales with the matrix size (N) for different numbers of clusters (K=2, 5, 10). We generally see increasing performance with larger N. The table1 ”Clustering Performance Results for K=2” provides specific GPU time and performance figures, showing consistent GFLOPs around 5-6 for N values from 1000 to 10000.

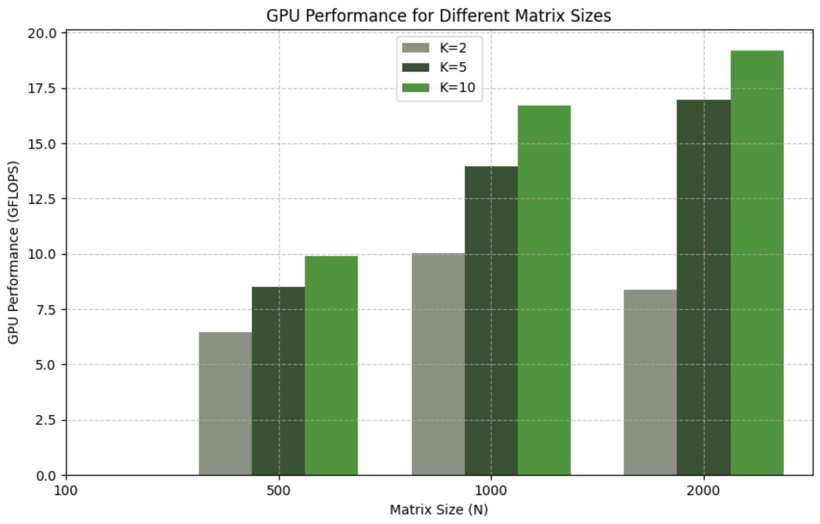

Figure 2: GFLOPS

Table 1: Clustering Performance Results for $K = 2$

N	GPU Time (s)	GPU Performance (GFLOPS)
100	0.0006	/
500	0.0026	/
1000	0.0064	10.9588
2000	0.0060	10.0099
5000	0.2814	9.86892
10000	0.9888	9.93419

Conclusion

In summary, by combining an elegant reformulation of kernel K-means with GPU acceleration, Popcorn makes it practical to cluster moderately large datasets with complex cluster structures in seconds rather than minutes. This work exemplifies how identifying and exploiting sparsity patterns can unlock significant performance gains in parallel algorithms. For practitioners, the takeaway is that kernel-based clustering no longer needs to be prohibitively slow – with the right tools (GPU computing frameworks like cuBLAS/cuSPARSE) and techniques, one can achieve both high performance and high accuracy. Future directions could explore extending this approach to streaming data or on-the-fly kernel computation (to handle cases where n is extremely large), as well as integrating other optimization steps (such as automatically tuning the kernel parameter σ). Nonetheless, the Popcorn algorithm and our experimental validation demonstrate a successful synergy of advanced algorithms and parallel computing, enabling kernel K-means to pop (pun intended) with efficiency on modern GPUs.

References

[1] J. Bellavita, T. Pasquali, L. Del Rio Martin, F. Vella, and G. Guidi. Popcorn: Accelerating kernel k-means on gpus through sparse linear algebra. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 426–440, 2025.

[2] I. S. Dhillon, Y. Guan, and B. Kulis. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 551–556, 2004.

[3] A. Likas, N. Vlassis, and J. J. Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.