Supplementary Materials for "SparseKmeans: Efficient K-means Clustering For Sparse Data"

Khoi Nguyen Pham Dang*
He-Zhe Lin*
khoinguyen.phamdang@mbzuai.ac.ae
b07902028@csie.ntu.edu.tw
Mohamed bin Zayed University of Artificial Intelligence
Abu Dhabi, UAE

Chih-Jen Lin
National Taiwan University
Taipei, Taiwan
Mohamed bin Zayed University of Artificial Intelligence
Abu Dhabi, UAE
cjlin@csie.ntu.edu.tw

A Stopping Condition of K-means Algorithm

SparseKmeans follows the implementation in scikit-learn to stop the iteration when the new centroids are close enough to their corresponding old ones. Specifically, the procedure is terminated when

$$\sum_{j=1}^{K} \|\boldsymbol{c}_j - \hat{\boldsymbol{c}}_j\|^2 \leq \tau \times \frac{1}{n} \sum_{k=1}^{n} \sigma_k^2, \tag{A.I}$$

where $\boldsymbol{c}_j$'s and $\hat{\boldsymbol{c}}_j$'s are current and previous centroids, τ is a predefined parameter, and σ_k^2 is the variance of the k-th component in each $\boldsymbol{x}_i$, respectively.

In SparseKmeans, we store the current centroids and previous centroids as two matrices C and $\hat{C}$:

$$C = \begin{bmatrix} - & \boldsymbol{c}_1^T & - \\ - & \boldsymbol{c}_2^T & - \\ & \vdots & \\ - & \boldsymbol{c}_K^T & - \end{bmatrix} \text{ and } \hat{C} = \begin{bmatrix} - & \hat{\boldsymbol{c}}_1^T & - \\ - & \hat{\boldsymbol{c}}_2^T & - \\ & \vdots & \\ - & \hat{\boldsymbol{c}}_K^T & - \end{bmatrix}.$$

To calculate $\|\boldsymbol{c}_j - \hat{\boldsymbol{c}}_j\|^2$ in (A.I), we first take the subtraction between C and $\hat{C}$:

$$C - \hat{C} = \begin{bmatrix} \boldsymbol{c}_1^T - \hat{\boldsymbol{c}}_1^T \\ \boldsymbol{c}_2^T - \hat{\boldsymbol{c}}_2^T \\ \vdots \\ \boldsymbol{c}_K^T - \hat{\boldsymbol{c}}_K^T \end{bmatrix},$$

and then calculate the squared norms of $C - \hat{C}$

$$\begin{bmatrix} \|\boldsymbol{c}_1^T - \hat{\boldsymbol{c}}_1^T\|^2 \\ \|\boldsymbol{c}_2^T - \hat{\boldsymbol{c}}_2^T\|^2 \\ \vdots \\ \|\boldsymbol{c}_K^T - \hat{\boldsymbol{c}}_K^T\|^2 \end{bmatrix}. \tag{A.II}$$

Summing all elements in (A.II), we have the left-hand side of (A.I).

B K-means++ Initialization

The initialization of $\boldsymbol{c}_j$'s are often by choosing a subset of K samples $C \subseteq \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$. By default, scikit-learn uses K-means++ initialization [1] to determine C. Compared to randomly choosing K points as the initial centroids, K-means++ effectively reduces the number of iterations for convergence and minimizes the risk of poor clustering results.

B.1 The Procedure of K-means++

K-means++ sequentially selects the initial points in a more optimal way by prioritizing points that are far from the previously chosen centroids. The detailed procedure is given as follows.

(1) Uniformly choose $\boldsymbol{c}_1$ from $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ and let $C = \{\boldsymbol{c}_1\}$.
(2) Initialize the distance vector $\boldsymbol{d} \in \mathbb{R}^m$ with $d_i = \|\boldsymbol{x}_i - \boldsymbol{c}_1\|$. Through the whole procedure, d_i records the distance between $\boldsymbol{x}_i$ and the nearest point in C, i.e.,

$$d_i = \min_{\boldsymbol{c} \in C} \|\boldsymbol{x}_i - \boldsymbol{c}\|. \tag{B.I}$$

We define the "potential" with respect to C to be the summation over the squared distances d_i^2, denoted by $\Phi(C)$:

$$\Phi(C) = \sum_{i=1}^{m} d_i^2.$$

(3) For $j = 2, \ldots, K$, do the following to select $\boldsymbol{c}_j$.
   (a) Sample $L = 2 + \log_2 K$ candidate points from $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ using the distribution

$$\Pr(\boldsymbol{x}_i) = \frac{d_i^2}{\Phi(C)}. \tag{B.II}$$

   Let $S = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_L\}$ denote the sampled candidates. From (B.II), we see that a point that is further from C is more likely to be selected. Besides, according to (B.I), we have $d_i = 0$ for any $\boldsymbol{x}_i \in C$, so it is guaranteed that the candidate set S does not include any point in C.[1]
   (b) To find the "best" candidate to be $\boldsymbol{c}_j$, for each $\boldsymbol{s} \in S$, we first calculate the potential with respect to $C \cup \{\boldsymbol{s}\}$, and choose the one that yields the least potential, i.e.,

$$\boldsymbol{c}_j \in \arg\min_{\boldsymbol{s} \in S} \Phi(C \cup \{\boldsymbol{s}\}) \tag{B.III}$$

$$= \arg\min_{\boldsymbol{s} \in S} \sum_{i=1}^{m} \min_{\boldsymbol{c} \in C \cup \{\boldsymbol{s}\}} \|\boldsymbol{x}_i - \boldsymbol{c}\|^2 \tag{B.IV}$$

$$= \arg\min_{\boldsymbol{s} \in S} \sum_{i=1}^{m} \min\{d_i^2, \|\boldsymbol{x}_i - \boldsymbol{s}\|^2\}. \tag{B.V}$$

   (c) Update $C \leftarrow C \cup \{\boldsymbol{c}_j\}$ and

$$d_i \leftarrow \min\{d_i, \|\boldsymbol{x}_i - \boldsymbol{c}_j\|\}, i = 1, \ldots, m. \tag{B.VI}$$

*Both authors contributed equally to this research.

[1] However, there is an exception when the number of distinct points in $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is less than K. See Section B.2 for details.

B.2 The Concept of K-means++

Through the process of initializing the centroid $\boldsymbol{c}_j$'s, K-means++ ensures that the updated potential would be reduced after a candidate $\boldsymbol{x}_i$ is selected as a centroid. This can be seen from (B.VI): after the update, d_i becomes zero when $\boldsymbol{x}_i$ is chosen as a centroid, while $d_{i'}$ is non-increasing for all other $i' \neq i$. Therefore, as long as $d_{i'} > 0$ for some i', the updated potential keeps decreasing, even in the extreme case that only one $\boldsymbol{x}_{i'}$ with $d_{i'} > 0$.

In the case where $\Phi(C) = 0$, meaning that all the distances between points and their nearest centroids are zero, we are unable to select any of them as a candidate according to (B.II). In other words, every $\boldsymbol{x}_i$ is the same as some $\boldsymbol{c}_j \in C$, which indicates that the number of distinct points is less than K. This situation occurs if $m < K$, or $m \geq K$ but there are some duplicated points. As the procedure can no longer find K unique centroids, SparseKmeans chooses to reduce the predefined number of clusters to the number of distinct data points.

B.3 Implementation Details

The most time-consuming part in the above procedure is Step-3b, which requires $\|\boldsymbol{x}_i - \boldsymbol{s}\|^2$ for all $\boldsymbol{s} \in S$ and $\boldsymbol{x}_i$'s. Similar to (4), the pairwise squared distance matrix in $\mathbb{R}^{m \times L}$ is

$$\begin{bmatrix} \|\boldsymbol{x}_1\|^2 - 2\boldsymbol{x}_1^T\boldsymbol{s}_1 + \|\boldsymbol{s}_1\|^2 & \dots & \|\boldsymbol{x}_1\|^2 - 2\boldsymbol{x}_1^T\boldsymbol{s}_L + \|\boldsymbol{s}_L\|^2 \\ \|\boldsymbol{x}_2\|^2 - 2\boldsymbol{x}_2^T\boldsymbol{s}_1 + \|\boldsymbol{s}_1\|^2 & \dots & \|\boldsymbol{x}_2\|^2 - 2\boldsymbol{x}_2^T\boldsymbol{s}_L + \|\boldsymbol{s}_L\|^2 \\ \vdots & \ddots & \vdots \\ \|\boldsymbol{x}_m\|^2 - 2\boldsymbol{x}_m^T\boldsymbol{s}_1 + \|\boldsymbol{s}_1\|^2 & \dots & \|\boldsymbol{x}_m\|^2 - 2\boldsymbol{x}_m^T\boldsymbol{s}_L + \|\boldsymbol{s}_L\|^2 \end{bmatrix}, \tag{B.VII}$$

which needs the the following matrix product similar to (C.III)

$$XS^T = X \begin{bmatrix} | & | & & | \\ \boldsymbol{s}_1 & \boldsymbol{s}_2 & \cdots & \boldsymbol{s}_L \\ | & | & & | \end{bmatrix}. \tag{B.VIII}$$

However, unlike the centroid matrix C in (C.III) where the density greatly varies, S is always very sparse because each $\boldsymbol{s}$ is just a data point. Thus, we always apply a sparse-sparse product for calculating (B.VIII).

C Implementation Details of Lloyd's Cluster Assignment

In Section 2.1.1, we mentioned that we need to compute the following pairwise distance matrix in $\mathbb{R}^{m \times K}$

$$\begin{bmatrix} \|\boldsymbol{x}_1\|^2 - 2\boldsymbol{x}_1^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & \|\boldsymbol{x}_1\|^2 - 2\boldsymbol{x}_1^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \\ \|\boldsymbol{x}_2\|^2 - 2\boldsymbol{x}_2^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & \|\boldsymbol{x}_2\|^2 - 2\boldsymbol{x}_2^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \\ \vdots & \ddots & \vdots \\ \|\boldsymbol{x}_m\|^2 - 2\boldsymbol{x}_m^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & \|\boldsymbol{x}_m\|^2 - 2\boldsymbol{x}_m^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \end{bmatrix}.$$

In Lloyd's cluster assignment, for each i, all we need is to find

$$\ell_i \in \arg\min_{j=1,\dots,K} \|\boldsymbol{x}_i - \boldsymbol{c}_j\|^2,$$

i.e., finding the cluster j that yields the least $\|\boldsymbol{x}_i - \boldsymbol{c}_j\|$ among the i-th row of the above matrix. Therefore, we may omit calculating the shared $\|\boldsymbol{x}_i\|^2$ terms within each row; instead, we can only compute the following matrix

$$\begin{bmatrix} -2\boldsymbol{x}_1^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & -2\boldsymbol{x}_1^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \\ -2\boldsymbol{x}_2^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & -2\boldsymbol{x}_2^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \\ \vdots & \ddots & \vdots \\ -2\boldsymbol{x}_m^T\boldsymbol{c}_1 + \|\boldsymbol{c}_1\|^2 & \dots & -2\boldsymbol{x}_m^T\boldsymbol{c}_K + \|\boldsymbol{c}_K\|^2 \end{bmatrix}. \tag{C.I}$$

This only requires the norm of $\boldsymbol{c}_j$'s

$$\begin{bmatrix} \|\boldsymbol{c}_1\|^2 \\ \|\boldsymbol{c}_2\|^2 \\ \vdots \\ \|\boldsymbol{c}_K\|^2 \end{bmatrix}, \tag{C.II}$$

and the matrix multiplication

$$XC^T = \begin{bmatrix} - & \boldsymbol{x}_1^T & - \\ - & \boldsymbol{x}_2^T & - \\ & \vdots & \\ - & \boldsymbol{x}_m^T & - \end{bmatrix} \begin{bmatrix} | & | & & | \\ \boldsymbol{c}_1 & \boldsymbol{c}_2 & \cdots & \boldsymbol{c}_K \\ | & | & & | \end{bmatrix}. \tag{C.III}$$

D Details of Elkan's Cluster Assignment

D.1 The Maintenance of $u(\boldsymbol{x}_i)$ and $l(\boldsymbol{x}_i, \boldsymbol{c}_j)$

- For each $\boldsymbol{x}_i$ in cluster-ℓ_i, $u(\boldsymbol{x}_i)$ is an upper bound of $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$. Since the centroids are updated after the cluster assignment, we need to update $u(\boldsymbol{x}_i)$ correspondingly. Suppose $\hat{\boldsymbol{c}}_{\ell_i}$ is the centroid for cluster-ℓ_i before the centroid updates and $\hat{u}(\boldsymbol{x}_i)$ is an upper bound of $\|\boldsymbol{x}_i - \hat{\boldsymbol{c}}_{\ell_i}\|$. To update $u(\boldsymbol{x}_i)$ from $\hat{u}(\boldsymbol{x}_i)$, by the triangle inequality, we have

  $$\|\boldsymbol{x}_i - \hat{\boldsymbol{c}}_{\ell_i}\| + \|\hat{\boldsymbol{c}}_{\ell_i} - \boldsymbol{c}_{\ell_i}\| \geq \|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|.$$

  Therefore, after centroid updates, $u(\boldsymbol{x}_i)$ can be obtained via

  $$u(\boldsymbol{x}_i) \leftarrow \hat{u}(\boldsymbol{x}_i) + \|\boldsymbol{c}_{\ell_i} - \hat{\boldsymbol{c}}_{\ell_i}\|. \tag{D.I}$$

- For each $\boldsymbol{x}_i$ and any $j = 1, \dots, K$, $l(\boldsymbol{x}_i, \boldsymbol{c}_j)$ is a lower bound of $\|\boldsymbol{x}_i - \boldsymbol{c}_j\|$. Similar to the situation in $u(\boldsymbol{x}_i)$, since $\boldsymbol{c}_j$ is changed afer the centroid updates, we need to update $l(\boldsymbol{x}_i, \boldsymbol{c}_j)$ for all i. Suppose $\hat{\boldsymbol{c}}_j$ is the centroid of cluster-j before the centroid updates and $\hat{l}(\boldsymbol{x}_i, \hat{\boldsymbol{c}}_j)$ is a lower bound of $\|\boldsymbol{x}_i - \hat{\boldsymbol{c}}_j\|$. By the triangle inequality and the definition of $\hat{l}(\boldsymbol{x}_i, \hat{\boldsymbol{c}}_j)$, we have

  $$\|\boldsymbol{x}_i - \boldsymbol{c}_j\| \geq \|\boldsymbol{x}_i - \hat{\boldsymbol{c}}_j\| - \|\hat{\boldsymbol{c}}_j - \boldsymbol{c}_j\| \geq \hat{l}(\boldsymbol{x}_i, \hat{\boldsymbol{c}}_j) - \|\hat{\boldsymbol{c}}_j - \boldsymbol{c}_j\|.$$

  Therefore, after centroid updates, $l(\boldsymbol{x}_i, \boldsymbol{c}_j)$ can be updated by

  $$l(\boldsymbol{x}_i, \boldsymbol{c}_j) \leftarrow \max\left\{\hat{l}(\boldsymbol{x}_i, \hat{\boldsymbol{c}}_j) - \|\boldsymbol{c}_j - \hat{\boldsymbol{c}}_j\|, 0\right\}. \tag{D.II}$$

D.2 Elkan's K-means Assignment in the Orignal Work [2]

In the main paper, we give the sketch of Elkan's K-means assignment in the original work of [2]. Here we provide a detailed version in Algorithm D.1. In particular, there are two additional settings:

(1) (Line 3 to Line 4) To avoid the for-loop over j in Algorithm 1, Elkan [2] considers the following condition

$$u(\boldsymbol{x}) \leq \frac{1}{2} \min_{j: j \neq \ell_i} \|\boldsymbol{c}_j - \boldsymbol{c}_{\ell_i}\|. \tag{D.III}$$

If (D.III) holds, then condition (6) must hold for all $j \neq \ell_i$, which means $\boldsymbol{x}$ must remain in its original cluster.

Algorithm D.1: The original version of Elkan's cluster assignment in [2].

```
1  Compute D ∈ R^{K×K} with D_{j,j'} = ||c_j − c_{j'}||/2
2  for i = 1, ..., m do
3      if u(x) ≤ min_{j≠ℓ_i} D_{j,ℓ_i}/2 then
4          continue
5      bound_tight ← False
6      for j = 1, ..., K do
7          if j ≠ ℓ_i and u(x_i) > D_{j,ℓ_i}/2 and u(x_i) > l(x_i, c_j)
           then
                   // filter clusters that x_i must not belong to
8              if bound_tight is False then
9                  d_i ← ||x_i − c_{ℓ_i}||
10                 u(x_i) ← d_i, l(x_i, c_{ℓ_i}) ← d_i
11                 bound_tight ←True
12                 Go to Line 7
13             l(x_i, c_j) ← ||x_i − c_j||
14             if u(x_i) > ||x_i − c_j|| then
15                 ℓ_i ← j, u(x_i) ← ||x_i − c_j||
```

(2) To further reduce the number of distance calculations, we update $u(\boldsymbol{x}_i)$ to be the tightest upper bound $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ upon seeing the first $\boldsymbol{c}_j$ with $\|\boldsymbol{x}_i - \boldsymbol{c}_j\|$ may need to be calculated. Let us assume j^* is the first cluster satisfying neither (6) nor (7). Then the for-loop over j can be illustrated as follows:

$$\overbrace{1, \ldots, (j^* - 1),}^{bound\_tight=False} \underbrace{j^*}_{\text{Line 8 invoked}} \overbrace{, (j^* + 1), \ldots, K}^{bound\_tight=True}. \tag{D.IV}$$

We use a flag *bound_tight* to record if $u(\boldsymbol{x}_i) = \|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ or not. For $j = 1, \ldots, j^* - 1$, this flag remains its default value *False* because either condition (6) or (7) holds and $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ is not needed. Therefore, we do not update $u(\boldsymbol{x}_i)$ to be the tight bound $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$. When $j = j^*$, we need both $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ and $\|\boldsymbol{x}_i - \boldsymbol{c}_{j^*}\|$ to determine whether $\boldsymbol{x}_i$ should stay in cluster-ℓ_i or go to cluster-j^*. Under this situation, we update $u(\boldsymbol{x}_i)$ to $\|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ and go to Line 7 to recheck the conditions. Then, from $j = j^* + 1, \ldots, K$, we maintain $u(\boldsymbol{x}_i) = \|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ in Line 15 while ℓ_i may be changed. Since we use the smallest possible value of $u(\boldsymbol{x}_i)$, the number of distance calculations can be reduced.

E Additional Implementation Details for Updating Centroids

After assigning each sample to its nearest cluster ℓ_i, some clusters may end up empty, making the corresponding centroids undefined. To address this, scikit-learn selects the points with the largest distances $d_i = \|\boldsymbol{x}_i - \boldsymbol{c}_{\ell_i}\|$ and reassigns them to the empty clusters. This explains the need for maintaining the vector $\boldsymbol{d}$, as noted in Section 3.1. Nevertheless, this strategy introduces potential issues: if all points in a cluster are reassigned to fill other empty clusters, the original cluster itself becomes empty. In the worst case (though we have not constructed such an example), the algorithm may not be able to terminate.

Algorithm E.1: Algorithm for handling empty clusters.

```
   Input: The number of empty clusters E
1  filled_cluster ← 0
2  k ← 1
3  while filled_cluster < E do
4      x ← the point with the k-th largest d_i
5      if (the size of the cluster that x belongs to) > 1 then
6          Reassign x to an empty cluster
7          filled_cluster ← filled_cluster + 1
8      k ← k + 1
```

To avoid the potential risk, we propose Algorithm E.1. Similar to scikit-learn, our approach begins by reassigning points with the largest d_i to the empty clusters. However, if a point is the only remaining member of its original cluster, we skip it and move to the point with the next largest d_i. Since many of the clusters may contain only a single point, it is natural to ask that, after skipping these points for reassignment, whether Algorithm E.1 always provides enough points to fill the empty clusters. The following theorem shows that Algorithm E.1 guarantees enough points for filling the empty clusters.

Theorem E.1. *Assume the number of cluster is K and suppose there are E empty clusters after the assignment step. After doing Algorithm E.1, it is guaranteed that there is no empty cluster.*

Proof. Suppose there are $\bar{K}$ clusters containing only a single distinct point (possibly with duplicates). According to the K-means++ initialization procedure in Section B.2, the number of distinct points is at least K. Therefore, among the remaining $(K - E - \bar{K})$ clusters, there must be at least $(K - \bar{K})$ distinct points. Among these $(K - \bar{K})$ distinct points, at most $(K - E - \bar{K})$ points coincide with their cluster centroids, i.e., their corresponding $d_i = 0$. Consequently, there are at least

$$(K - \bar{K}) - (K - E - \bar{K}) = E$$

points with $d_i > 0$, so we are able to reassign E points to E empty clusters. □

F Data Sets

This section describes the preprocessing method applied to the four data sets across all experiments. For binary classification data set Url,[2] we directly use the data instances for clustering. Other data sets including Amazon-670K,[3] Wiki-500K[4] and Amazon-3M[5] are multi-label data sets, in which each instance can be associated with several labels. In the context of this work, to avoid confusion with clustering results, which is called as labels in the main paper, we refer to the original labels of these multi-label data sets as meta-labels. Instead of clustering on data instances, we preprocessed the data to obtain a set of representations for meta-labels. These representations are then normalized and used as input for the K-means

[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/url_combined_normalized.bz2

[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel/Amazon-670K_tfidf_train_ver2.svm.bz2

[4] https://drive.google.com/open?id=1bGEcCagh8zaDV0ZNGsgF0QtwjcAm0Afk

[5] https://drive.google.com/open?id=187vt5vAkGI2mS2WOMZ2Qv48YKSjNbQv4

clustering algorithms. To clarify, let z_i be the sparse representation of i^{th} instance, m be the total number of meta-labels and x_j be the representation of j^{th} meta-label. To obtain a meta-label representation, we compute the sum over all data instances corresponding to that meta-label as follows:

$$x_j = \frac{\bar{x}_j}{\|\bar{x}_j\|}. \tag{F.I}$$

in which

$$\bar{x}_j = \sum \{z_i \mid z_i \text{ is associated with } j^{th} \text{ meta-label}\}. \tag{F.II}$$

G A Comprehensive Study and Benchmark on the Major Operations of K-means Algorithm

In this section, we comprehensively study the optimization of the five operations mentioned in Section 3.2, including the optimized storage format for the matrices and the way to do the operation. We compare each setting in a real scenario by extracting matrices used in K-means clustering on the Wiki-500K data set. We run all experiments for 10 times on a machine with an Intel(R) Core(TM) i7-6900K CPU @ 3.20GHz and 128 GB RAM, using eight threads.

G.1 Operation (I): XC^T

In Section 2.1.1, we mentioned that the density of C may change across iterations. To find out a suitable setting under varying densities, we start with benchmarking XC^T using $X \in \mathbb{R}^{m \times n}$ from the Wiki-500K data set, where $m = 500,091$, $n = 2,381,304$, and the density of X is approximately 0.10%. For the centroid matrix, we use four matrices with different densities:

- $C_1 \in \mathbb{R}^{K \times n}$, where $K = 1,500$, $n = 2,381,304$, and the density of C is $\text{nnz}(C)/(Kn) \approx 2.9\%$.
- $C_2 \in \mathbb{R}^{K \times n}$, where $K = 1,000$, $n = 2,381,304$, and the density of C is $\text{nnz}(C)/(Kn) \approx 3.7\%$.
- $C_3 \in \mathbb{R}^{K \times n}$, where $K = 500$, $n = 2,381,304$, and the density of C is $\text{nnz}(C)/(Kn) \approx 5.7\%$.
- $C_4 \in \mathbb{R}^{K \times n}$, where $K = 100$, $n = 2,381,304$, and the density of C is $\text{nnz}(C)/(Kn) \approx 14.8\%$.

We compare the running time under different storage formats of C and the way to conduct the multiplication.

- We consider four possible storage formats for the matrix C based on a cross combination of dense/sparse and row/column-major settings. A sparse format stores only the non-zero entries in C, while a dense format stores all Kn entries regardless of value. In terms of layout, row-major (respectively, column-major) means the entries are stored in a row-wise (respectively, column-wise) order. This combination results in four formats: CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), FullR (dense row-major), and FullC (dense column-major).
- As for computing the product, we compare three settings implemented using the Python packages python-graphblas,[6] which provides a Python interface to GraphBLAS (GB), and sparse-dot-mkl,[7] which provides a Python wrapper for Intel Math Kernel Library (MKL). Since the result of XC^T is necessary for forming the typically dense matrix (C.I), it must be ultimately stored in a dense format. In GraphBLAS, this can be achieved in two ways, depending on when the conversion to dense is performed:
  - GB-NORMAL: Compute XC^T in GraphBLAS, store it in sparse format, and then convert it to dense. Although the conversion time is usually neglegible compared to the matrix multiplication, we still include it in the timing to ensure a fair comparison with other settings.
  - GB-ACCUM: Use the GraphBLAS accumulation operator to obtain a dense result directly. In this case, we first initialize a dense zero matrix $M \in \mathbb{R}^{m \times K}$, and the operator updates it as

    $$M \leftarrow M + XC^T,$$

    yielding the desired dense output.

  For MKL, we consider the following setting similar to GB-NORMAL:
  - MKL-NORMAL: XC^T is computed using MKL and assigned to a sparse matrix.

Table 1 shows the running time under each configuration. We first compare the three modes: GB-NORMAL, GB-ACCUM, and MKL-NORMAL.

- GB-NORMAL vs. GB-ACCUM: The performance of GraphBLAS heavily depends on the backend algorithm (the "kernel") it chooses. Therefore, we additionally list the kernel applied in each format combination in Table 1. The kernels can be roughly categorized into two types: `saxpy`-based kernels and `dot`-based kernels (details deferred to Appendix H). In Table 1, we see that for format combinations using the same kernel – such as CSR × CSR, CSR × FullC, CSC × CSR, CSC × CSC and CSC × FullC – the running times between GB-NORMAL and GB-ACCUM are almost the same. However, for other combinations, `saxpy`-based kernels are much faster than `dot`-based kernels. The explanation is in Appendix H. In particular, in the case of a higher density of C (e.g., C_3 and C_4), the `saxpy5` kernel used in GB-ACCUM achieves the most efficient performance.
- GB-ACCUM vs. MKL-NORMAL: In the products with a higher density of C such as XC_3^T and XC_4^T, Intel MKL achieves slightly better performance than GraphBLAS. However, the Python API occasionally encounters segmentation faults, which compromises the robustness of the implementation. This is one major reason for choosing GraphBLAS as the backend library for matrix operations.

With its efficiency and robustness, we select GB-ACCUM as our primary mode and further analyze the performance of GB-ACCUM under different format combinations in Appendix H. We summarize the analysis as follows.

- Computing XC^T in a "row-major × row-major" format achieve both time and memory access efficiency.
- As the density of C grows, "sparse × sparse" operations lead to more overheads compared to "sparse × dense" operations.

Based on the above analysis, we use GB-ACCUM to compute XC^T and heuristically set a density threshold 0.05 for C to dynamically switch the format of C:

- If the density of C from the previous iteration is below 0.05, we use "CSR × CSR."
- Otherwise, we use "CSR × FullR."

[6] https://github.com/python-graphblas/python-graphblas
[7] https://github.com/flatironinstitute/sparse_dot

Table 1: Running time (in seconds) of XC_1^T, XC_2^T, XC_3^T and XC_4^T. We terminate the timing if a setting takes more than one hour and mark "> 1 hr" in these cells. Cells marked with "-" indicate that a segmentation fault happens when using the MKL library. For each setting in GB-NORMAL and GB-ACCUM, we list the backend algorithm (i.e., kernel) applied in GraphBLAS.

Method	X	C^T	Kernel	Time for XC_1^T	Time for XC_2^T	Time for XC_3^T	Time for XC_4^T
GB-NORMAL	CSR	CSR	saxpy3	388.87	274.84	158.57	49.79
	CSR	CSC	saxpy3	385.19	281.44	161.33	56.83
	CSR	FullR	dot2	1,073.63	653.21	275.37	43.55
	CSR	FullC	dot2	3,117.31	2,036.03	831.29	144.62
	CSC	CSR	saxpy3	2,235.55	1,521.14	789.00	165.26
	CSC	CSC	saxpy3	2,222.29	1,520.76	793.10	164.95
	CSC	FullR	dot2	1,134.80	652.20	303.47	131.63
	CSC	FullC	bitmap saxpy	2,142.35	1,277.35	635.89	131.90
GB-ACCUM	CSR	CSR	saxpy3	389.12	274.73	158.97	49.77
	CSR	CSC	dot4	> 1hr	> 1hr	3,164.37	875.15
	CSR	FullR	saxpy5	462.05	286.92	134.40	20.15
	CSR	FullC	dot4	> 1hr	1,716.25	843.07	146.50
	CSC	CSR	saxpy3	2,246.83	1,541.31	789.32	166.31
	CSC	CSC	saxpy3	2,211.36	1,519.04	790.46	166.03
	CSC	FullR	saxpy5	667.90	305.87	154.66	132.35
	CSC	FullC	bitmap saxpy	2,083.11	1,250.90	637.95	134.45
MKL-NORMAL	CSR	CSR		444.24	318.43	199.40	73.86
	CSR	CSC		457.73	337.92	208.81	82.42
	CSR	FullR		-	-	115.02	18.37
	CSR	FullC		1,542.39	997.24	484.38	84.42
	CSC	CSR		-	-	-	-
	CSC	CSC		-	-	-	-
	CSC	FullR		> 1hr	2,989.16	1,452.36	288.27
	CSC	FullC		1,527.42	1,029.30	515.43	108.18

G.2 Operation (III): $\boldsymbol{x}_i^T \boldsymbol{c}_{\ell_i}$

Our new design of Elkan's method in Algorithm 2 requires calculating m inner products between each $\boldsymbol{x}_i$ and its closest centroid $\boldsymbol{c}_{\ell_i}$. To efficiently access $\boldsymbol{c}_{\ell_i}$ in the centroid matrix, C must be stored with a row-major format. Therefore, the remaining question is whether C should be in a sparse or dense format.

G.2.1 Theoretical Analysis. Let us start from analyzing the theoretical complexity of computing

$$\boldsymbol{x}_i^T \boldsymbol{c}_{\ell_i} = \sum_{k=1}^{n} x_{i,k} \cdot c_{\ell_i,k} = \sum_{k:x_{i,k}\neq 0} x_{i,k} \cdot c_{\ell_i,k}. \tag{G.I}$$

- If C is stored as a dense matrix, for each $x_{i,k} \neq 0$, we can directly access $c_{\ell_i,k}$ in constant time. Therefore, calculating (G.I) requires $\text{nnz}(\boldsymbol{x}_i)$ times memory accesses to $c_{\ell_i,k}$ and $\text{nnz}(\boldsymbol{x}_i)$ times of scalar products.
- If C is stored in a sparse format, then we need at most $\text{nnz}(\boldsymbol{c}_{\ell_i})$ times of element accesses to traverse all non-zero elements in $\boldsymbol{c}_{\ell_i}$, followed by $\text{nnz}(\boldsymbol{x}_i)$ times of scalar products.

According to the definition of $\boldsymbol{c}_{\ell_i}$ (2), $\text{nnz}(\boldsymbol{c}_{\ell_i}) \geq \text{nnz}(\boldsymbol{x}_i)$. As a result, using a dense format for C yields less memory accesses and better theoretical complexity.

G.2.2 Runtime Benchmark on C-code and GraphBLAS Python Interface. Although the theory suggests using a dense format for the centroid matrix C, we are not sure if the internal implementation in GraphBLAS behaves similarly as the above analysis, since GraphBLAS needs to handles various types of matrix operations. In order to guide the final decision in SparseKmeans, we conduct a detailed benchmark on randomly generated matrices.

Data Generation. Fixing $n = 1,000,000$ and $K = 100$, we randomly generate the matrix X (the sample size and matrix density are specified in Table 2) and the labels ℓ_i for each point. The centroid matrix C are computed according to the generated data matrix and label vector.

Backend for the Operation. We introduce the following backend engines to conduct $\boldsymbol{x}_i^T \boldsymbol{c}_{\ell_i}$ for $i = 1, \ldots, m$. Since this operation requires frequently accessing rows of X and C, we suppose X and C need to be stored as a row-major format.

- Handcrafted C-implementation: To verify the theoretical analysis, we consider the C implementation executing on a single thread .
- GraphBLAS: To acquire the m inner products at a time, we utilize the "mask operations" supported in GraphBLAS. For such operations, GraphBLAS selectively calculates certain entries of a

Table 2: Running time (in seconds) of calculating ten times of $x_i c_{\ell_i}^T$ for $i = 1, \ldots, m$ with C-implementation and GraphBLAS. We fix $n = 1,000,000$, $K = 100$ and vary m along with the density of X.

			C-code (1 thread)		GB (1 thread)		GB (8 threads)	
m	**Density of X**	**Density of C**	C: FullR	C: CSR	C: FullR	C: CSR	C: FullR	C: CSR
2,000	0.01%	0.20%	**0.01**	0.04	1.49	**0.07**	0.61	**0.07**
2,000	0.02%	0.40%	**0.02**	0.07	1.53	**0.14**	0.58	**0.07**
2,000	0.05%	0.99%	**0.06**	0.19	1.59	**0.37**	0.64	**0.10**
2,000	0.1%	1.98%	**0.13**	0.38	1.65	**0.79**	0.63	**0.19**
2,000	0.2%	3.92%	**0.26**	0.93	1.76	**1.68**	0.63	**0.38**
2,000	0.5%	9.49%	**0.63**	2.90	**2.13**	4.56	**0.79**	0.84
2,000	1%	18.06%	**1.31**	5.70	**2.81**	9.41	**0.99**	1.60
5,000	0.01%	0.50%	**0.03**	0.18	1.51	**0.22**	0.64	**0.09**
5,000	0.02%	0.99%	**0.07**	0.36	1.57	**0.46**	0.63	**0.10**
5,000	0.05%	2.47%	**0.18**	0.97	1.66	**1.30**	0.69	**0.24**
5,000	0.1%	4.87%	**0.36**	2.98	**1.85**	2.93	0.66	**0.52**
5,000	0.2%	9.51%	**0.66**	6.87	**2.18**	6.33	**0.77**	1.17
5,000	0.5%	22.06%	**1.58**	17.38	**3.12**	16.69	**0.98**	3.00
5,000	1%	39.19%	**3.12**	31.92	**4.70**	33.67	**1.53**	5.82
10,000	0.01%	1.00%	**0.07**	0.61	1.55	**0.57**	0.61	**0.13**
10,000	0.02%	1.98%	**0.15**	1.27	1.64	**1.21**	0.71	**0.19**
10,000	0.05%	4.88%	**0.36**	4.67	**1.85**	3.67	0.69	**0.67**
10,000	0.1%	9.51%	**0.70**	11.02	**2.20**	8.06	**0.76**	1.50
10,000	0.2%	18.11%	**1.33**	22.51	**2.87**	17.06	**0.88**	3.34
10,000	0.5%	39.25%	**3.17**	52.57	**4.76**	43.54	**1.40**	8.12
10,000	1%	63.02%	**6.19**	91.14	**6.44**	9.72[8]	**1.95**	3.10

matrix product by providing a boolean mask matrix. In SparseK-means, we compute the m inner products by applying a mask $M \in \{0,1\}^{m \times K}$ on the matrix product XC^T, where $M_{i,\ell_i} = 1$ for all i, and $M_{i,j} = 0$ for all $j \neq \ell_i$. To check the effect of parallel computing in GraphBLAS, we record the time using single and eight threads.

The running time for each setting is listed in Table 2. We summarize the results as follows:

- In the C-implementation, using a dense format for C is consistently faster than using a sparse format for C, which aligns with the theory. However, the conclusion does not hold in GraphBLAS.
- For a dense format of C, the C implementation consistently outperforms GraphBLAS. This is because our specialized C-implementation is designed to achieve the optimal theoretical complexity with minimal overheads, whereas GraphBLAS, as a general purpose library, incurs significant overheads. However, when C is stored as a sparse matrix, GraphBLAS can be faster than the naive C-implementation, showing its efficiency in doing sparse-sparse matrix operations.
- Comparing the GraphBLAS results, we observe that using a sparse format for C is faster than using a dense format for C when C's density is low (roughly less than 5%). Therefore, we may use the same threshold (5%) as Section G.1 to switch the format of C. If the density of C is less than 5%, we use a sparse format to conduct this operation.

[8]We comment that when the density of C is high (> 40%), GraphBLAS internally switches to dense format for C even if the input format is sparse.

G.3 Operation (IV): $X_{S_j,:}c_j$

G.3.1 Theoretical Analysis. Since this operation can be regarded as calculating $x_i^T c_j$ for $i \in S_j$, the theoretical analysis for the previous operation also suggests that using a dense c_j yields to better theoretical complexity.

G.3.2 GraphBLAS Runtime Benchmark. Similar to G.2, we utilize the mask operations in GraphBLAS on the matrix-vector product Xc_j to get $X_{S_j,:}c_j$. In Table 3, we compare the operation under the GraphBLAS implementation and the handcrafted C-implementation by varying the number of data points, the density of X and C, and the proportion of S_j (i.e., density of masks). The results show similar behavior to Table 2.

- Using a dense format for c_j with the C-implementation consistently outperforms other settings, which aligns with our theoretical analysis.
- Comparing the GraphBLAS results under dense/sparse formats for C, we find that sparse format shows its advantages when both X is very sparse and the size of S_j (i.e., the mask density) is low.

Table 3: Running time (in seconds) for benchmarking $X_{S_j,:}\boldsymbol{c}_j^T$ using both C-implementation and GraphBLAS. The experiment was conducted by randomly generating a mask for S_j $(j = 1, \ldots, K)$ with a specified density, repeated the calculation ten times, and recording the results. We set $n = 1,000,000$ while varying m and the density of X.

(a) $m = 1,000$

Density of X	Density of C	Density of mask	Dense C		Sparse C	
			C-code	GB	C-code	GB
0.1%	1.0%	0.1%	0.003	1.57	0.009	**0.09**
		1%	0.05	2.35	0.10	**0.31**
		10%	0.47	2.67	0.96	**1.90**
		20%	1.27	**4.46**	2.49	4.61
		50%	2.36	**5.56**	6.53	11.42
		100%	3.13	**6.96**	12.61	22.74
1%	9.5%	0.01%	0.002	**0.14**	0.01	0.71
		0.1%	0.04	**0.96**	0.12	1.44
		1%	0.41	**1.58**	1.27	2.26
		10%	2.25	**3.19**	9.88	3.61
10%	61.5%	0.01%	0.03	**0.19**	0.07	0.27
		0.1%	0.24	**1.19**	0.76	1.76
		1%	1.09	**2.88**	7.69	5.10
		10%	8.61	**13.32**	76.63	28.96

(b) $m = 5,000$

Density of X	Density of C	Density of mask	Dense C		Sparse C	
			C-code	GB	C-code	GB
0.01%	0.5%	0.1%	0.003	2.30	0.015	**0.12**
		1%	0.03	2.30	0.13	**0.25**
		10%	0.25	2.56	1.32	**1.56**
		20%	0.66	3.50	3.47	**3.13**
		50%	1.37	**5.04**	8.67	8.84
		100%	2.19	**5.81**	18.42	17.00
0.1%	4.8%	0.1%	0.03	1.83	0.14	**1.04**
		1%	0.27	1.83	1.44	**1.45**
		10%	1.87	**3.33**	13.55	3.36
		20%	3.65	**6.37**	37.00	7.86
		50%	5.99	**9.84**	91.25	17.22
		100%	9.10	**14.06**	17.89	31.33
1%	39.1%	0.1%	0.22	**2.07**	2.12	2.38
		1%	1.60	**3.04**	13.71	3.73
		10%	5.84	**7.99**	110.64	17.78

(c) $m = 10,000$

Density of X	Density of C	Density of mask	Dense C		Sparse C	
			C-code	GB	C-code	GB
0.01%	1.0%	0.1%	0.008	2.27	0.05	**0.15**
		1%	0.06	2.32	0.47	**0.46**
		10%	0.51	**2.80**	4.47	3.61
		20%	1.35	**4.63**	13.31	8.38
		50%	2.72	**6.10**	33.30	20.90
		100%	3.33	**7.17**	65.91	40.74
0.05%	4.8%	0.1%	0.03	2.26	0.25	**1.71**
		1%	0.27	2.48	2.45	**1.86**
		10%	1.92	3.25	23.63	**2.30**
		20%	4.30	6.94	67.19	**5.06**
		50%	6.81	**10.01**	163.32	10.80
		100%	9.19	**14.14**	327.31	19.63
0.1%	9.5%	0.1%	0.05	2.28	0.53	**2.08**
		1%	0.50	2.67	4.96	**2.24**
		10%	2.64	4.21	45.87	**3.55**
		20%	4.24	7.37	104.60	**6.97**
		50%	8.04	**11.51**	266.75	13.32
		100%	13.36	**21.98**	538.38	25.58
1%	30.0%	0.1%	0.44	**1.47**	6.04	2.47
		1%	2.30	**3.22**	36.74	5.40
		10%	11.23	**13.65**	336.35	34.69

G.4 Operation (V): BX

To benchmark the multiplication (9), we dump X from the Wiki-500K data set and prepare matrices B_1, B_2, B_3, B_4 with $K = 1500$, 1000, 500, and 100, respectively.

Table 4: Running time (in seconds) of $B_1X, \ldots, B_4X$.

Method	B	X	Time for B_1X	Time for B_2X	Time for B_3X	Time for B_4X
GB-NORMAL	CSR	CSR	3.92	3.88	4.05	4.77
	CSR	CSC	23.24	23.10	3.79	3.28
	CSC	CSR	3.94	3.88	4.05	4.79
	CSC	CSC	4.48	4.00	3.77	3.24
GB-ACCUM	CSR	CSR	5.97	5.24	4.81	4.97
	CSR	CSC	387.27	310.92	212.43	80.03
	CSC	CSR	5.98	5.27	4.82	4.99
	CSC	CSC	6.21	5.37	4.51	3.51
MKL-NORMAL	CSR	CSR	5.68	6.02	6.50	10.27
	CSR	CSC	29.85	30.13	30.59	34.28
	CSC	CSR	46.24	46.09	45.97	45.84
	CSC	CSC	2.62	2.53	2.44	2.30

As shown in (10), each column of B contains one non-zero entry, so the density of B is always $1/K$.

- $B_1 \in \mathbb{R}^{K\times m}$, where $K = 1500$, $m = 500,091$, and the density of B_1 is 0.0667%.
- $B_2 \in \mathbb{R}^{K\times m}$, where $K = 1000$, $m = 500,091$, and the density of B_2 is 0.1%.
- $B_3 \in \mathbb{R}^{K\times m}$, where $K = 500$, $m = 500,091$, and the density of B_3 is 0.2%.
- $B_4 \in \mathbb{R}^{K\times m}$, where $K = 100$, $m = 500,091$, and the density of B_4 is 1%.

For the data formats, since B is rather sparse, so we consider CSR and CSC. As for the way to compute BX, we also consider GB-NORMAL, GB-ACCUM and MKL-NORMAL as in Section G.1. From the benchmark results in Table 4, we see that the difference for this operation is minor compared to the results in Table 1. Therefore, for simplicity, we choose the configuration according to the storage format of the centroid matrix C. When C is stored in a sparse format, we use GB-NORMAL with "CSR × CSR;" otherwise, we use GB-ACCUM with "CSR × CSR."

G.5 Operation (II): CC^T

To benchmark the multiplication CC^T, we adopt the same C matrix settings as those used in Section G.1, where four matrices with varying densities are considered.

To evaluate the impact of storage formats, we examine C under the same four representations as before: CSR, CSC, FullR, and FullC. For each configuration, we compare the performance of GB-NORMAL, GB-ACCUM, and MKL-NORMAL to assess the efficiency of computing CC^T under different format combinations.

The detailed benchmark results are reported in Table 5. Based on the results, we use GB-NORMAL with "CSR × CSC" to compute CC^T.

H Analysis on XC^T in GB-ACCUM

In this section, we further investigate XC^T in GB-ACCUM under different storage-format combinations and compare their computational and memory access behavior. For simplicity, throughout this section we abuse the notation and analyze the product XC^T using the generic matrix-multiplication notation

$$C = A \times B \tag{H.I}$$

in GraphBLAS, where the matrices A, B, and C in (H.I) correspond to the data matrix X, the transposed centroid matrix C^T, and the resulting matrix XC^T in the context of the main paper, respectively. Therefore, in (H.I) we have

$$A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times k}, \text{ and } C = A \times B \in \mathbb{R}^{m\times k}.$$

Let $\mathrm{nnz}(A)$ and $\mathrm{nnz}(B)$ denote the number of non-zero entries in matrices A and B, respectively. We assume that the non-zero entries of both matrices are uniformly distributed across rows and columns. In the memory access analysis, we use a simple two-level memory model consisting of main memory and secondary memory. That is, when data needed by an operation are not already available in main memory, they must be brought from secondary memory into main memory. Thus, the memory access cost below counts the amount of data that needs to be moved from secondary memory to main memory during the multiplication.

Algorithm H.1: saxpy-based multiplication for CSR×CSR

```
for i = 1 to m do
    for A_id ≠ 0 in the i-th row of A do
        for B_dj ≠ 0 in the d-th row of B do
            C_ij += A_id · B_dj
```

H.1 CSR × CSR vs. CSR × CSC

The format combinations CSR × CSR and CSR × CSC lead to different multiplication algorithms in GraphBLAS. Algorithm H.1 describes the CSR × CSR multiplication. As illustrated in Figure 1, for each non-zero entry A_{id}, the algorithm traverses the corresponding d-th row of B. Each scalar A_{id} multiplies the non-zero entries B_{dj} in that row, and the results are accumulated into the i-th row of C. Since one scalar entry scales a row of another matrix, GraphBLAS refers to this as a `saxpy`-based algorithm.

Table 5: Running time (in seconds) of $C_1C_1^T$, $C_2C_2^T$, $C_3C_3^T$, and $C_4C_4^T$.

Method	C	C^T	Time for $C_1C_1^T$	Time for $C_2C_2^T$	Time for $C_3C_3^T$	Time for $C_4C_4^T$
GB-NORMAL	CSR	CSC	16.93	9.84	4.44	1.02
	CSC	CSR	17.80	9.06	4.00	0.70
	FullR	FullC	1,194.24	502.59	127.85	4.80
	FullC	FullR	610.96	272.92	73.07	9.12
GB-ACCUM	CSR	CSC	119.01	66.89	25.74	2.66
	CSC	CSR	17.29	8.97	4.06	0.66
	FullR	FullC	1,077.32	471.05	120.39	5.00
	FullC	FullR	612.66	272.73	73.26	9.29
MKL-NORMAL	CSR	CSR	19.62	12.83	6.22	1.46
	CSR	CSC	17.71	11.15	5.29	1.02
	CSC	CSR	28.19	14.14	4.97	0.25
	CSC	CSC	28.54	15.83	4.81	0.22

Algorithm H.2: dot-based multiplication for CSR×CSC

```
for i = 1 to m do
    for j = 1 to k do
        Find D = {d | A_id ≠ 0 and B_dj ≠ 0}
        for d ∈ D do
            C_ij += A_id · B_dj
```

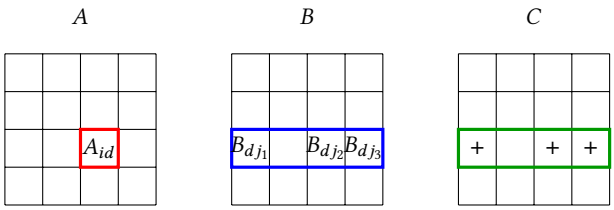

Figure 1: One saxpy3 update for CSR×CSR. A non-zero entry A_{id} multiplies the corresponding non-zero entries B_{dj} in the d-th row of B and accumulates them into the matching entries C_{ij} in the i-th row of C.

Algorithm H.2 describes the CSR × CSC multiplication. As illustrated in Figure 2, the algorithm computes each output entry C_{ij} sequentially by checking the non-zero entries in the i-th row of A and the j-th column of B. For every index d where both A_{id} and B_{dj} are non-zero, it multiplies the corresponding pair and accumulates the result into C_{ij}. Because each output entry is computed as an inner product, GraphBLAS refers to this as a `dot`-based algorithm.

To better understand the performance difference between these two algorithms under different format combinations, we compare the computational and memory access costs of both settings.

Computational Cost of CSR × CSR. For each non-zero entry A_{id}, the algorithm traverses the corresponding d-th row of B and updates the i-th row of C. The average number of non-zero entries per row of B is

$$\frac{\text{nnz}(B)}{n}.$$

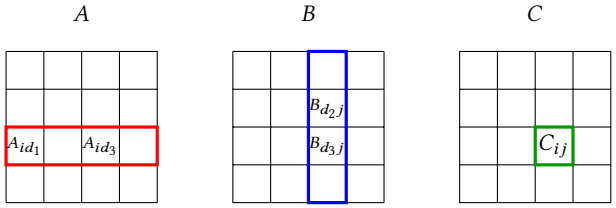

Figure 2: One inner dot-product update in sparse format. The algorithm checks indices d where both A_{id} and B_{dj} are non-zero, multiplies each corresponding pair $A_{id}B_{dj}$, and accumulates the result into C_{ij}.

Therefore, the computational cost is approximately

$$O\left(\frac{\text{nnz}(A)\cdot\text{nnz}(B)}{n}\right). \tag{H.II}$$

Memory access cost. For each non-zero entry A_{id}, the algorithm accesses the corresponding d-th row of B. Since these row accesses dominate the memory overhead, we omit the access costs of A and C in this rough estimate and focus on B. The average number of nonzeros per row of B is

$$\frac{\text{nnz}(B)}{n}.$$

Therefore, the total amount of data accessed from B is approximately

$$O\left(\frac{\text{nnz}(A)\cdot\text{nnz}(B)}{n}\right). \tag{H.III}$$

Computational cost of CSR × CSC. To compute each output entry C_{ij}, the dot-based algorithm first finds

$$D = \{d \mid A_{id} \neq 0 \text{ and } B_{dj} \neq 0\}.$$

This requires checking the non-zero entries in the i-th row of A and the j-th column of B, and then multiplying the matched pairs. The i-th row of A contains about $\text{nnz}(A)/m$ non-zero entries, while the

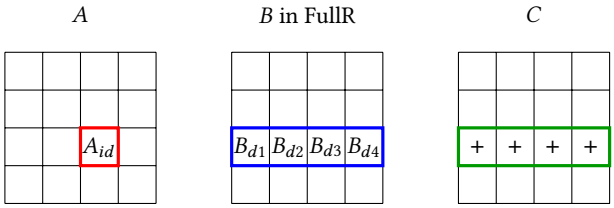

Figure 3: One saxpy5 update in CSR × FullR. A non-zero entry A_{id} multiplies the contiguous dense row $B_{d,:}$, so the accesses to B and the updates to C are row-wise and contiguous.

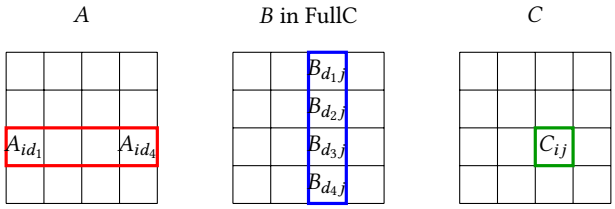

Figure 4: One dot-product update in CSR × FullC. The algorithm only accesses entries B_{dj} whose indices d correspond to nonzeros in the sparse row $A_{i,:}$.

j-th column of B contains about $\mathrm{nnz}(B)/k$ non-zero entries. Thus, the cost for computing one entry C_{ij} is

$$O\left(\frac{\mathrm{nnz}(A)}{m} + \frac{\mathrm{nnz}(B)}{k}\right).$$

Since there are mk output entries, the computational cost is

$$O\left(mk\left(\frac{\mathrm{nnz}(A)}{m} + \frac{\mathrm{nnz}(B)}{k}\right)\right) = O(k \cdot \mathrm{nnz}(A) + m \cdot \mathrm{nnz}(B)). \tag{H.IV}$$

Memory access cost of CSR × CSC. For each row of A, the dot-based algorithm computes all entries in the corresponding row of C by scanning the columns of B. Therefore, it accesses all non-zero entries of B once for each row of A. We assume that the current row of A and the corresponding row of C can remain in main memory during this process. However, since B is accessed as a whole for each row of A, the entries of B cannot all remain in main memory. Thus, the memory access cost is

$$O(m \cdot \mathrm{nnz}(B)). \tag{H.V}$$

Comparing (H.II) with (H.IV), and (H.III) with (H.V), CSR × CSC has higher computational and memory access costs than CSR × CSR. The higher computational and memory access costs of the dot-based method may explain why CSR × CSC is generally slower than the saxpy-based method in Table 1.

H.2 CSR × FullR vs. CSR × FullC

Computational cost of CSR × FullR. The computation of CSR × FullR is similar to CSR × CSR, except that B is stored in a dense row-major format. As illustrated in Figure 3, for each non-zero entry A_{id}, the algorithm multiplies it by the entire d-th dense row of B and accumulates the result into the i-th row of C. Thus, the computational cost is

$$O(\mathrm{nnz}(A) \cdot k). \tag{H.VI}$$

Memory access cost of CSR × FullR. Similar to the analysis of CSR × CSR, we focus on the memory accesses to B. For each non-zero entry A_{id}, the algorithm accesses the d-th dense row of B. Since B is stored in row-major format, the accessed row is contiguous in memory. Therefore, the memory access cost for B is

$$O(\mathrm{nnz}(A) \cdot k). \tag{H.VII}$$

Computational cost of CSR × FullC. For CSR × FullC, GraphBLAS uses a dot-based algorithm. As illustrated in Figure 4, to compute one output entry C_{ij}, the algorithm scans the non-zero entries in the i-th row of A and multiplies them by the corresponding entries in the j-th dense column of B. The cost of computing one output entry is therefore

$$O\big(\mathrm{nnz}(A_{i,:})\big).$$

Repeating this computation for all k columns of B gives the total computational cost

$$O\left(k\sum_{i=1}^{m}\mathrm{nnz}(A_{i,:})\right) = O(\mathrm{nnz}(A) \cdot k).$$

Memory access cost of CSR × FullC. For each output entry C_{ij}, the algorithm scans the non-zero entries in the i-th row of A and accesses the corresponding entries in the j-th dense column of B. If memory access cost is counted at the level of individual matrix entries, this gives a lower bound of $\mathrm{nnz}(A) \cdot k$ accesses. However, in practice data are loaded in blocks or pages, so accessing one entry B_{dj} may also load nearby entries in the same column of B. The actual memory access cost of CSR × FullC lies between this element-wise lower bound and the worst case of loading the dense column of B for each output entry. Thus,

$$O(\mathrm{nnz}(A) \cdot k) \le M_{\mathrm{CSR\times FullC}} \le O(mnk),$$

where $M_{\mathrm{CSR\times FullC}}$ denotes the memory cost of CSR × FullC.

The above bound clarifies why CSR × FullC may behave differently from CSR × FullR in practice. Although the two format combinations have the same computational complexity, CSR × FullC can incur a higher memory access cost.

In addition to its favorable memory access behavior, CSR × FullR can also make more effective use of vectorized instructions. As illustrated in Figure 3, it accesses the dense row $B_{d,:}$ contiguously and updates the output row of C in the same manner. This contiguous row-wise computation is cache-friendly and allows GraphBLAS to use vectorized instructions efficiently. In contrast, as illustrated in Figure 4, CSR × FullC computes each output entry C_{ij} as a dot product between the sparse row $A_{i,:}$ and the dense column $B_{:,j}$. For every non-zero entry A_{id} in $A_{i,:}$, the computation accesses the corresponding dense entry B_{dj}. Since the indices d are determined by the non-zero pattern of $A_{i,:}$, these accesses may be scattered across $B_{:,j}$ even though the column itself is stored densely. Moreover, each dot product produces only one output entry C_{ij}, resulting in a scalar-like output-update pattern with fewer opportunities for effective vectorization.

H.3 CSR × CSR vs. CSC × CSC

We next compare CSR × CSR and CSC × CSC that both use the saxpy-based algorithm described in Algorithm H.1. As discussed above, CSR × CSR iterates over a non-zero entry A_{id}, traverses the d-th row of B, and accumulates the result into the i-th row of C. For CSC × CSC, GraphBLAS uses the analogous column-oriented saxpy algorithm: it iterates over a non-zero entry B_{dj}, traverses the d-th column of A, and accumulates the result into the j-th column of C.

Therefore, under the same uniform non-zero assumption, the two settings have the same computational and memory access costs:

$$O\left(\frac{\text{nnz}(A)\text{nnz}(B)}{n}\right).$$

The runtime difference shown in Table 1 may come from the iteration order and the amount of data accessed in each inner-loop iteration.

For CSR × CSR, each inner-loop iteration over a row of A accesses, on average,

$$\frac{\text{nnz}(A)}{m} \cdot \frac{\text{nnz}(B)}{n} = \frac{\text{nnz}(A)\text{nnz}(B)}{mn} \tag{H.VIII}$$

entries from the inner traversal of B. For CSC × CSC, each inner-loop iteration over a column of B accesses, on average,

$$\frac{\text{nnz}(B)}{k} \cdot \frac{\text{nnz}(A)}{n} = \frac{\text{nnz}(A)\text{nnz}(B)}{nk} \tag{H.IX}$$

entries from the inner traversal of A. In the XC^T benchmark setting described in Section G.1,

$$\begin{aligned} A &= X \in \mathbb{R}^{500091\times 2381304}, \\ B &= C^T \in \mathbb{R}^{2381304\times K}, \\ K &\in \{100, 500, 1000, 1500\}. \end{aligned}$$

Comparing (H.VIII) and (H.IX), the amount of data accessed in each inner-loop iteration in CSC × CSC is roughly m/k times larger than that in CSR × CSR because $m \gg k$. This larger amount of data accessed per inner-loop iteration makes CSC × CSC less cache friendly and makes data reuse less effective. Consequently, although both format combinations are saxpy-based and have the same complexity, CSR × CSR is faster in practice.

H.4 CSR × CSR vs. CSR × FullR

The results in Appendix H.1 and H.2 suggest that we use "CSR × row-major" for the multiplication. To further decide whether the second operand B should be in dense or sparse, we first check their computational costs (H.II) and (H.VI), along with the memory costs (H.III) and (H.VII). Since $\text{nnz}(B)/n < k$ in general, it is expected that "CSR × CSR" achieves a better performance than "CSR × FullR." However, for B with higher density (which corresponds to the C_3 and C_4 in Table 1), "CSR × FullR" demonstrates more advantages than "CSR × CSR," regardless of using GB-NORMAL or GB-ACCUM. The reason might be that, as $\text{nnz}(B)$ grows, "sparse × sparse" operations lead to more overheads compared to "sparse × dense" operations.

H.5 Other Format Combinations

The remaining format combinations in Table 1 can be explained by how GraphBLAS internally handles each format.

For CSC × CSR, GraphBLAS first converts the right operand from CSR to CSC and then applies the same saxpy-based kernel as CSC × CSC. Thus, this case behaves like CSC × CSC after the conversion, with the additional overhead of converting the right operand.

For CSC × FullR, GraphBLAS converts the left operand from CSC to CSR and then applies the same saxpy-based kernel as CSR × FullR. Thus, this case behaves like CSR × FullR after the conversion, again with the additional conversion overhead. However, in Table 1, the running time for this setting becomes much larger when $K = 1500$. One possible reason is that this setting may put higher pressure on memory in our testing environment and trigger swapping between main memory and secondary memory. We leave a more detailed investigation of this issue as future work.

For CSC × FullC, GraphBLAS uses a column-oriented saxpy-based algorithm similar to CSC × CSC. The difference is that B is stored in dense column-major format. For each dense entry B_{dj}, the algorithm traverses the corresponding d-th column of A and accumulates the result into the j-th column of C. Thus, this case behaves like CSC × CSC, but with the right operand stored as a dense column-major matrix.

Acknowledgments

We thank Chen-Wei Lin and Ting-Xuan Wu for contributing Appendix H, which provides a detailed complexity analysis on the matrix products.

References

[1] David Arthur and Sergei Vassilvitskii. 2007. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 1027–1035.

[2] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning (ICML)*. 147–153.