Parallel algorithms for clustering large datasets on CPU-GPU heterogeneous architectures

Guanlin He

université
PARIS-SACLAY

Parallel algorithms for clustering large datasets on CPU-GPU heterogeneous architectures

Algorithmes parallèles de clustering *de grands ensembles de données pour architectures hétérogènes CPU-GPU*

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate school : Informatique et sciences du numérique
Référent : CentraleSupélec

Thèse préparée dans l'unité de recherche Laboratoire interdisciplinaire des sciences du numérique (Université Paris-Saclay, CNRS), sous la direction de Professeur Stéphane VIALLE et le co-encadrement de Professeur Marc BABOULIN

Thèse soutenue à Paris-Saclay, le 19 octobre 2022, par

Guanlin HE

Composition du jury

Céline HUDELOT Professeure, CentraleSupélec	Présidente
Pierre FORTIN Maître de conférences (HDR), Université de Lille	Rapporteur & Examinateur
Masha SOSONKINA Professeure, Old Dominion University	Rapporteur & Examinatrice
Sandrine MOUYSSET Maîtresse de conférences, Université de Toulouse 3	Examinatrice
Stéphane VIALLE Professeur, CentraleSupélec	Directeur de thèse
Marc BABOULIN Professeur, Université Paris-Saclay	Co-encadrant de thèse

THESE DE DOCTORAT
NNT : 2022UPASG062

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Algorithmes parallèles de *clustering* de grands ensembles de données pour architectures hétérogènes CPU-GPU.

Mots clés : *clustering* spectral, algorithme des k-moyennes, calcul haute performance, calcul sur GPU, optimisation de codes parallèles, évaluation de performances.

Résumé : *Clustering*, qui consiste à réaliser des groupements naturels de données, est une tâche fondamentale et difficile dans l'apprentissage automatique et l'exploration de données. De nombreuses méthodes de *clustering* ont été proposées dans le passé, parmi lesquelles le *clustering* en k-moyennes qui est une méthode couramment utilisée en raison de sa simplicité et de sa rapidité.

Le *clustering* spectral est une approche plus récente qui permet généralement d'obtenir une meilleure qualité de *clustering* que les k-moyennes. Cependant, les algorithmes classiques de *clustering* spectral souffrent d'un manque de passage à l'échelle en raison de leurs grandes complexités en nombre d'opérations et en espace mémoire nécessaires. Ce problème de passage à l'échelle peut être traité en appliquant des méthodes d'approximation ou en utilisant le calcul parallèle et distribué.

L'objectif de cette thèse est d'accélérer le *clustering* spectral et de le rendre applicable à de grands ensembles de données en combinant l'approximation basée sur des données représentatives avec le calcul parallèle sur processeurs CPU et GPU. En considérant différents scénarios, nous proposons plusieurs chaînes de traitement parallèle pour le *clustering* spectral à grande échelle. Nous concevons des algorithmes et des implémentations parallèles optimisés pour les modules de chaque chaîne proposée : un algorithme parallèle des k-moyennes sur CPU et GPU, un *clustering* spectral parallèle sur GPU avec un format de stockage creux, un filtrage parallèle sur GPU du bruit dans les données, etc. Nos expériences variées atteignent de grandes performances et valident le passage à l'échelle de chaque module et de nos chaînes complètes.

Title: Parallel algorithms for clustering large datasets on CPU-GPU heterogeneous architectures.

Keywords: spectral clustering, k-means algorithm, high performance computing, GPU computing, parallel code optimization, performance evaluation.

Abstract: Clustering, which aims at achieving natural groupings of data, is a fundamental and challenging task in machine learning and data mining. Numerous clustering methods have been proposed in the past, among which k-means is one of the most famous and commonly used methods due to its simplicity and efficiency.

Spectral clustering is a more recent approach that usually achieves higher clustering quality than k-means. However, classical algorithms of spectral clustering suffer from a lack of scalability due to their high complexities in terms of number of operations and memory space requirements. This scalability challenge can be addressed by applying approximation methods or by employing parallel and distributed computing.

The objective of this dissertation is to accelerate spectral clustering and make it scalable to large datasets by combining representatives-based approximation with parallel computing on CPU-GPU platforms. Considering different scenarios, we propose several parallel processing chains for large-scale spectral clustering. We design optimized parallel algorithms and implementations for each module of the proposed chains: parallel k-means on CPU and GPU, parallel spectral clustering on GPU using sparse storage format, parallel filtering of data noise on GPU, etc. Our various experiments reach high performance and validate the scalability of each module and the complete chains.

Acknowledgement

First of all, I would like to express my deep gratitude to my dissertation supervisor, Professor Stéphane Vialle. During nearly four years, from my application to the completion of this dissertation, Stéphane was always very supportive of me. Although Stéphane often had a hectic schedule, he ensured regular meetings and discussions with me and brought me help as soon as possible whenever I needed it. I would never have completed this dissertation without his guidance and help. Moreover, Stéphane introduced me to the fascinating field of High Performance Computing and taught me in many ways how to conduct scientific research. His rigorous and critical approach to research has greatly influenced me. I am also very grateful for his valuable advice on my future career. It was a wonderful time to be his student!

Secondly, I would like to thank my dissertation co-advisor, Professor Marc Baboulin. Marc was always responsive to my need and gave me much help. He provided a lot of useful advice on my dissertation. I have benefited greatly from his expertise in many areas including mathematics and academic writing. I really enjoyed working with Marc!

Also, many thanks to my homeland China and the China Scholarship Council for funding my study in France (No. 201807000143), and thanks to ParSys group, LISN laboratory, STIC doctoral school, CentraleSupélec, Université Paris-Saclay for providing me with a high-quality research platform and a comfortable working environment. Special thanks to the SAMI staff of the LISN laboratory for giving me a lot of IT services with kindness and patience.

Besides, thanks to my friends especially Baojie Li, Tianjiao Dai, Junjie Yang, Wenbo Zhou, Yifei Sun for their help and company. Thanks to my girlfriend Zengxian Tian for always being by my side and supporting me. Thanks to my parents and relatives for their continuous love and support.

Finally, thanks to all Jury members for evaluating my dissertation!

Contents

List of Figures

List of Tables

List of Algorithms

List of Code

List of Symbols

n	Number of data instances/points/objects
d	Number of dimensions/features/attributes
k_c	Number of clusters
k_{cmax}	Maximum number of clusters
k_r	Number of representatives
x_i	Data instance of index i
v_i	Vertex of index i
s_{ij}	Similarity between instances x_i and x_j
w_{ij}	Edge weight between vertices v_i and v_j
deg_i	Degree of vertex v_i
u_i	Eigenvector of index i
λ_i	Eigenvalue of index i
γ_i	Eigengap between λ_{i+1} and λ_i
ε	Neighborhood radius
σ	Scaling parameter of Gaussian similarity function
X	Dataset
G	Graph
V	Vertex set
E	Edge set
S	$n \times n$ similarity/Affinity/Adjacency matrix
D	$n \times n$ degree matrix
L	$n \times n$ graph Laplacian matrix
L_{sym}	$n \times n$ normalized symmetric L
L_{rw}	$n \times n$ normalized L related to a random walk
U	$n \times k_c$ eigenvector matrix
Σ	$k_c \times k_c$ eigenvalue matrix
nnz	Number of nonzero elements in S
$maxNnzRow$	Maximum number of nonzeros in a row of S
$hypoMaxNnzRow$	Hypothetical maximum number of nonzeros in a row of S
$spMaxNzPct$	Supposed maximum percent of nonzeros in S
$memUseRate$	Usage rate of GPU RAM for storing a chunk of S
$tholdNF$	Threshold for noise filtering

Introduction (English version)

Motivation

In today's era of explosive data growth, the need for High-Performance Data Analytics (HPDA) is becoming increasingly prominent. Classification and clustering are two fundamental tasks in data analysis. Both aim at dividing data instances into different groups through a learning process. The essential difference is that the former trains a predictive model on labeled data (a.k.a. supervised learning) while the latter involves only unlabeled data (a.k.a. unsupervised learning) [91]. We concern ourselves with clustering in this dissertation. Various clustering methods have been proposed in the literature, e.g. k-means [122], DBSCAN [53], and spectral clustering [136]. Although the time complexities of clustering algorithms vary considerably, clustering large-scale datasets is computationally expensive and even prohibitive! Moreover, some algorithms also have high memory space complexities.

Many scientific and technological endeavors have been made to address the scalability challenges of clustering algorithms. A major approach is to reduce their computational complexities by using approximation, which is the essence of many fast variants of traditional clustering algorithms. For example, an effective approximation method [203] is to perform a clustering algorithm on some representatives carefully extracted from the original dataset, and then approximate the global clustering solution by the clustering result of the representatives.

On the other hand, High Performance Computing (HPC) [49] can provide impressive speedups and scalability for computational tasks. It has developed rapidly in recent decades due to technological advances in computer hardware and software. For example, the advent of modern GPU architectures and CUDA programming has opened up the active field of general-purpose GPU computing and has led to a proliferation of GPU-accelerated applications. Thus, it seems very appealing to accelerate clustering algorithms by high performance computing. However, achieving it requires efficient algorithmic design and parallel programming with various optimizations on modern computer architectures.

In this dissertation, we mainly consider classical algorithms of spectral clustering, which usually produce higher quality clustering results than the k-means algorithm, but require cubic time and quadratic memory w.r.t. the number of data instances in the worst case. We are mainly motivated in accelerating spectral clustering and in making it scalable to large-scale datasets by combining the approximation approach with high performance computing on CPU-GPU platforms. Note that we basically regard a dataset containing a million instances and over ($n \geq 10^6$) as a "large-scale dataset", in view of: (1) the $\mathcal{O}(n^3)$ time complexity and $\mathcal{O}(n^2)$ space complexity of spectral clustering; (2) the limited computing and memory resources of single node CPU-GPU heterogeneous architectures. In fact,

most of the public clustering datasets we found[1] contain only thousands to hundreds of thousands of data instances, which we usually consider as small-scale or medium-scale datasets.

Dissertation overview

This dissertation consists of five chapters.

In Chapter 1, we introduce the background knowledge, related works, and the objectives of this dissertation. More specifically, we first introduce clustering by reviewing its notion, taxonomy of algorithms, and evaluation metrics. Then we expound the two clustering methods mainly involved in this dissertation: k-means clustering and spectral clustering, including their algorithms, advantages, disadvantages and existing approaches to improvement. Furthermore, related works on approximate spectral clustering and parallel spectral clustering are particularly reviewed. Finally, we introduce the possibility to combine representative-based approximate spectral clustering with parallel computing on CPU-GPU architectures.

In Chapter 2, we present optimized parallel implementations for k-means clustering on CPU and GPU, which can be used either independently or be used to extract representatives for spectral clustering. Particularly, we address the numerical accuracy issue caused by the propagation of rounding errors in the k-means algorithm. Experiments on large datasets demonstrate both numerical accuracy and parallelization efficiency of our k-means implementations.

In Chapter 3, we focus on the parallelization of spectral clustering on GPU. Essentially, several algorithms and associated optimized parallel implementations are proposed for matrix construction in Compressed Sparse Row (CSR) format on GPU. This can achieve significant performance acceleration while reducing substantial memory space requirements of spectral clustering. Then, we leverage NVIDIA's GPU-accelerated nvGRAPH library for remaining computations of spectral clustering. Finally, experimental results show the high performance and the scalability to large datasets of our spectral clustering implementation on GPU.

In Chapter 4, we address the noise sensitivity issue of spectral clustering by incorporating noise filtering into the spectral clustering implementation presented in Chapter 3 and exploiting some particular features of this implementation. Experiments show that our noise filtering implementation on GPU is efficient and significantly improves the performance of spectral clustering on noisy data.

In Chapter 5, we adopt the representative-based approximation method to further advance the scalability of spectral clustering. Several methods for representatives extraction are first investigated. Then, several processing chains on CPU-GPU platforms are proposed according to different scenarios. Finally, experimental results exhibit the validity and good scalability of our proposed chains.

[1]Clustering basic benchmark: http://cs.joensuu.fi/sipu/datasets/
UCI ML Repository: https://archive-beta.ics.uci.edu/ml/datasets

The benchmark datasets and testbed features are detailed in Appendix A and B, respectively. Our code is available at https://github.com/guanlin-he/clustering-release.

Introduction (French version)

Motivation

À l'époque de la croissance explosive des données, le besoin de l'analyse de données à haute performance (HPDA) se fait de plus en plus saillant. La classification et le *clustering* sont deux tâches fondamentales de l'analyse des données. Tous les deux visent à diviser les données en différents groupes par le biais d'un processus d'apprentissage. La différence essentielle réside dans le fait que la première tâche entraîne un modèle prédictif sur des données étiquetées (apprentissage supervisé), tandis que la seconde ne concerne que des données non étiquetées (apprentissage non supervisé) [91]. Nous nous intéressons au *clustering* dans cette thèse. Diverses méthodes de *clustering* ont été proposées dans la littérature, par exemple les k-moyennes [122], le DBSCAN [53], et le *clustering* spectral [136]. Bien que les complexités temporelles des algorithmes de *clustering* varient considérablement, le *clustering* d'ensembles de données à grande échelle est coûteux en termes de calcul, voire prohibitif ! De plus, certains algorithmes présentent également des complexités élevées en termes d'espace mémoire.

De nombreux efforts scientifiques et technologiques ont été faits pour relever les défis de passage à l'échelle des algorithmes de *clustering*. Une approche majeure consiste à réduire leurs complexités de calcul en utilisant l'approximation, qui est l'essence de nombreuses variantes rapides des algorithmes de *clustering* traditionnels. Par exemple, une méthode d'approximation efficace [203] consiste à exécuter un algorithme de *clustering* sur certains représentants soigneusement extraits de l'ensemble de données original, puis à approximer la solution de *clustering* globale par le résultat du *clustering* des représentants.

D'autre part, le calcul haute performance (HPC) [49] peut fournir des accélérations impressionnantes et un passage à l'échelle pour les tâches de calcul. Il s'est développé rapidement au cours des dernières décennies grâce aux progrès technologiques réalisés dans le matériel et les logiciels informatiques. Par exemple, l'avènement des architectures GPU modernes et de la programmation CUDA a ouvert le champ actif du calcul GPU à usage général et a conduit à une prolifération d'applications accélérées par le GPU. Ainsi, il semble très intéressant d'accélérer les algorithmes de *clustering* par le calcul haute performance. Cependant, pour y parvenir, il faut une conception algorithmique et programmation parallèle efficaces avec diverses optimisations sur les architectures informatiques modernes.

Dans cette thèse, nous considérons principalement les algorithmes classiques de *clustering* spectral, qui produisent généralement des résultats de *clustering* de meilleure qualité que l'algorithme des k-moyennes, mais nécessitent un temps cubique et une mémoire quadratique par rapport au nombre d'instances de données dans le pire des cas. Nous nous sommes principalement intéressés à l'accélération

du *clustering* spectral et à son passage à l'échelle en combinant l'approche d'approximation avec le calcul haute performance sur des plateformes CPU-GPU. Notez que nous considérons essentiellement un ensemble de données contenant un million d'instances et plus ($n \geq 10^6$) comme un « ensemble de données à grande échelle », compte tenu de : (1) la complexité temporelle $\mathcal{O}(n^3)$ et la complexité spatiale $\mathcal{O}(n^2)$ du *clustering* spectral ; (2) les ressources de calcul et de mémoire limitées des architectures hétérogènes CPU-GPU à nœud unique. En fait, la plupart des ensembles de données de clustering publics que nous avons trouvés[2] ne contiennent que des milliers à des centaines de milliers d'instances de données, que nous considérons généralement comme des ensembles de données à petite ou moyenne échelle.

Aperçu de la thèse

Cette thèse se compose de cinq chapitres.

Dans le chapitre 1, nous présentons les connaissances de base, les travaux connexes et les objectifs de cette thèse. Plus précisément, nous présentons d'abord le *clustering* en passant en revue sa notion, la taxonomie des algorithmes et les mesures d'évaluation. Ensuite, nous exposons les deux méthodes de *clustering* principalement impliquées dans cette thèse : le *clustering* en k-moyennes et le *clustering* spectral, y compris leurs algorithmes, leurs avantages, leurs inconvénients et les approches existantes pour les améliorer. En outre, les travaux connexes sur le *clustering* spectral approximatif et le *clustering* spectral parallèle sont particulièrement examinés. Enfin, nous présentons la possibilité de combiner le *clustering* spectral approximatif basé sur les représentants avec le calcul parallèle sur les architectures CPU-GPU.

Dans le chapitre 2, nous présentons des implémentations parallèles optimisées pour le *clustering* en k-moyennes sur CPU et GPU, qui peuvent être utilisées soit indépendamment, soit pour extraire des représentants pour le *clustering* spectral. En particulier, nous abordons le problème de la précision numérique causé par la propagation des erreurs d'arrondi dans l'algorithme des k-moyennes. Des expériences sur de grands ensembles de données démontrent à la fois la précision numérique et l'efficacité de la parallélisation de nos implémentations de k-moyennes.

Dans le chapitre 3, nous nous concentrons sur la parallélisation du *clustering* spectral sur GPU. Essentiellement, plusieurs algorithmes et implémentations parallèles optimisées associées sont proposés pour la construction de matrices au format CSR (Compressed Sparse Row) sur GPU. Cela permet d'obtenir une accélération significative des performances tout en réduisant considérablement l'espace mémoire nécessaire au *clustering* spectral. Ensuite, nous utilisons la bibliothèque nvGRAPH de NVIDIA qui accélère sur GPU les calculs restants du *clustering* spectral. Enfin, les résultats expérimentaux montrent la haute performance et l'extensibilité à de

[2]Clustering basic benchmark: http://cs.joensuu.fi/sipu/datasets/
UCI ML Repository: https://archive-beta.ics.uci.edu/ml/datasets

grands ensembles de données de notre implémentation du *clustering* spectral sur GPU.

Dans le chapitre 4, nous abordons le problème de la sensibilité au bruit du *clustering* spectral en incorporant le filtrage du bruit dans l'implémentation du *clustering* spectral présentée dans le chapitre 3 et en exploitant certaines caractéristiques particulières de cette implémentation. Les expériences montrent que notre implémentation du filtrage du bruit sur GPU est efficace et améliore significativement les performances du *clustering* spectral sur des données bruyantes.

Dans le chapitre 5, nous adoptons la méthode d'approximation basée sur les représentants pour faire progresser le passage à l'échelle du *clustering* spectral. Plusieurs méthodes d'extraction des représentants sont d'abord étudiées. Ensuite, plusieurs chaînes de traitement sur des plateformes CPU-GPU sont proposées en fonction de différents scénarios. Enfin, les résultats expérimentaux démontrent la validité et le passage à l'échelle des chaînes proposées.

Les ensembles de données de benchmark et les caractéristiques du banc d'essai sont détaillés en annexe A et B, respectivement. Notre code est disponible sur https://github.com/guanlin-he/clustering-release.

1 - Related Works and Objectives

1.1 . Clustering

Data clustering [91, 73, 168, 54], also known as cluster analysis, refers to an automatic process that discovers the natural groupings (i.e. clusters) of a set of unlabeled data points, instances, or objects. As one of the most important and challenging tasks in data analysis and unsupervised machine learning, clustering has been actively studied for decades with interdisciplinary endeavor [91]. It has a wide range of applications, such as market and customer segmentation [52, 98], image and video segmentation [21, 103], network analysis [64], recommender systems [1], document clustering [93], bibliometrics analysis [191], bioinformatics analysis [72, 216], disease analysis [217], crime analysis [2].

Generally, the clustering process seeks to maximize intra-cluster similarity and minimize inter-cluster similarity. However, neither the notion of a "cluster", nor the measure of "similarity", nor the realization way are precisely defined. Moreover, there are various data distributions and the data to be clustered can be low or high dimensional, small or large scale, noiseless or noise-laden, quantitative or categorical, static or dynamic, homogeneous or heterogeneous, etc. Due to these facts, a large number and variety of clustering algorithms have been proposed in the past.

1.1.1 . Algorithms and taxonomy

Clustering has been continuously surveyed and resurveyed over time, e.g. [102] in 1990, [92] in 1999, [91] in 2010, [73] in 2011, [4] in 2014, [201] in 2015, [168] in 2017, [62] in 2020, [54] in 2022. The number of published clustering algorithms is overwhelming and continues to grow. Early clustering algorithms can usually be categorized into partitional clustering and hierarchical clustering. Later some new categories of clustering algorithms have emerged, such as density-based clustering, grid-based clustering, distribution-based clustering, graph-based clustering, and deep learning-based clustering. Some clustering algorithms combine the methods of at least two categories. Here we call them cross-category clustering.

Partitional clustering aims at dividing n data instances into k_c partitions (clusters) such that instances within the same cluster are as close to each other as possible and instances in different clusters are as far apart from each other as possible [73]. As hinted, partitional algorithms are usually based on distances and lead to spherical partitions as clusters. Each partition/cluster is characterized by its cluster center (also called *centroid*). Hence partitional algorithms are also known as centroid-based algorithms. Dedicated surveys on partitional clustering include References [160, 31, 108]. Representative algorithms include the k-means algorithm (1967) [122] and its variants, e.g. k-medoids (1987) [101], CLARA

(1986) [100], CLARANS (1994) [137], k-modes (1997) [86], k-prototypes (1997) [85], X-means [151] (2000), k-means++ (2006) [13].

Hierarchical clustering forms a cluster hierarchy (nested clusters) either in an agglomerative or divisive manner [91]. Agglomerative hierarchical clustering uses a bottom-up approach, which starts with each data instance as a cluster and then recursively merges similar clusters into larger clusters. In contrast, divisive hierarchical clustering uses a top-down approach, which starts with all data instances as one cluster and then recursively splits each cluster into smaller clusters. The merging or splitting operations in hierarchical clustering can be based on distances, densities, *links*, model probabilities, etc. However, once a merging or splitting operation has been completed, it usually cannot be undone even if it was a wrong decision, which is a drawback of hierarchical clustering [73]. Dedicated surveys on hierarchical clustering include References [131, 160, 138, 132]. Most hierarchical clustering algorithms are agglomerative. Representatives include SAHN (1973) [175], AGNES (1990) [102], BIRCH (1996) [213], CURE (1998) [67], and ROCK (2000) [68]. Divisive hierarchical clustering algorithms include DIANA (1990) [102] and DHCC (2012) [200].

Density-based clustering defines clusters as high-density regions separated by low-density regions [91], where the density is computed as the number of points within the neighborhood of a given radius. In contrast to partitional clustering which typically cannot find non-spherical clusters, density-based clustering can find clusters of arbitrary shapes. Moreover, density-based clustering is usually good at filtering noise and outliers which have low densities. Dedicated surveys on density-based clustering include References [105, 29, 20]. Representative algorithms include DBSCAN (1996) [53], DENCLUE (1998) [84], OPTICS (1999) [12], DENCLUE 2.0 (2007) [83], HDBSCAN (2013) [30], and density peaks clustering (2014) [162].

Distribution-based clustering, a.k.a. model-based clustering or probabilistic clustering, models the data with a mixture of distributions and defines clusters as the instances that are most likely to belong to the same distribution [4]. The user needs to assume a model, e.g. Gaussian mixture model. The parameters of the model are usually initialized randomly and need to be optimized iteratively to better fit the dataset. However, the model may converge to a local optimum or suffer from overfitting. Dedicated surveys on distribution-based clustering include References [26, 123]. Representative algorithms include EM (1977) [43, 206] and DBCLASD (1998) [202].

Grid-based clustering divides the data space into a finite number of cells to form a grid structure in which clustering is then performed according to the density of each cell [73]. The number of cells is usually significantly smaller than the number of data instances. A major advantage of grid-based clustering is that it greatly reduces the computational complexity, because it clusters the neighborhood around each cell instead of directly clustering all data instances [4]. Dedicated surveys on grid-based clustering include Reference [36]. Representative algorithms

are STING (1997) [193], WaveCluster (1998) [172], and CLIQUE (1998) [5].

Graph-based clustering models the data as a graph and converts the data clustering problem into the graph partitioning/clustering problem. The data-to-graph modeling process is crucial to both the quality and the scalability of clustering. The graph partitioning/clustering process groups vertices of the graph into clusters in such a way that many edges exist within each cluster and relatively few edges exist between clusters [169]. Dedicated surveys include Reference [169] on graph clustering, and References [188, 133] on spectral clustering. Representative algorithms include spectral clustering (2001) [136], PIC (2010) [118], and Chameleon 2.0 (2019) [17].

Deep learning-based clustering, a.k.a. deep clustering, uses prevalent deep learning methods to tackle the clustering problem. Essentially, it transforms the data into more clustering-friendly representations by employing deep neural networks, e.g. Autoencoder (AE), Variational Autoencoder (VAE), Generative Adversarial Network (GAN) [125]. Dedicated surveys on deep learning-based clustering include References [125, 6]. Representative algorithms include DEC (2016) [199], VaDE (2016) [95], simultaneous deep learning and clustering (2017) [205], and DEPICT (2017) [63].

Cross-category clustering algorithms include the bisecting k-means (2000) [176] (partitional and divisive hierarchical), Chameleon (1999) [99] (hierarchical and graph-based), BHC (2005) [80] (distribution-based and hierarchical), HDBSCAN (2013) [30] (hierarchical and density-based), graph clustering based on deep learning (2014) [182] (graph-based and deep learning-based). Other well-known clustering algorithms that cannot be categorized in the above way include Support Vector Clustering (2001) [19] and Affinity Propagation (2007) [61].

Besides, clustering algorithms can also be classified into hard clustering (i.e. each data instance is assigned to a single cluster) and soft clustering (i.e. each data instance can belong to multiple clusters, a.k.a. fuzzy clustering) [91]. Dedicated surveys on soft clustering include References [128, 57].

1.1.2 . Evaluation

The quality or accuracy of the results of a clustering algorithm can be evaluated by various metrics. This evaluation procedure is also known as *cluster validity*. Dedicated surveys on this topic include References [189, 161]. According to whether the evaluation relies on the *ground truth clustering* (i.e. authentic cluster labels), the metrics can be divided into external and internal. Let us denote C_e as the clustering to be evaluated and C_g as the ground truth clustering.

External metrics assume C_g is known and measure how well C_e matches C_g. They include RI, ARI, MI, AMI, NMI, purity [161], V-measure [163], and Fowlkes-Mallows index [60]. Here we only explain RI-based and MI-based ones which are very commonly used in the literature and will also be used in this dissertation.

- **Rand Index (RI), Adjusted Rand Index (ARI).** RI [159] measures the

similarity between C_e and C_g, or it can be regarded as the percentage of correct decisions made by the algorithm. Assume a $positive$ decision represents that a pair of instances are grouped into the same cluster, otherwise it is a $negative$ decision. Assume a $true$ decision represents that a pair of instances are grouped in the same cluster or separated in different clusters like in C_g, otherwise it is a $false$ decision. Then RI can be calculated using the following formula:

$$RI(C_e, C_g) = \frac{TP + TN}{TP + TN + FP + FN} \tag{1.1}$$

where TP is the number of $true\ positive$ decisions, TN is the number of $true\ negative$ decisions, FP is the number of $false\ positive$ decisions, FN is the number of $false\ negative$ decisions. The range of RI is $[0, 1]$. A higher RI often indicates a better C_e. However, RI does not guarantee that a random C_e will get a score close to 0. Moreover, RI is often close to 1 even if C_e differs significantly from C_g !

To avoid such problems, the ARI metric [87] is defined based on RI but adjusted against chance. It is calculated as follows:

$$ARI(C_e, C_g) = \frac{RI(C_e, C_g) - E[RI(C_e, C_g)]}{max(RI(C_e, C_g)) - E[RI(C_e, C_g)]} \tag{1.2}$$

where $E[RI]$ denotes the expected[1] RI, and $max(RI)$ denotes the maximum RI (i.e. 1 for $C_e = C_g$). The range of ARI is $[-1, 1]$. A higher ARI indicates a better C_e, and a random C_e has an ARI close to 0.

- **Mutual Information (MI), Normalized Mutual Information (NMI), Adjusted Mutual Information (AMI).** MI [39] measures the statistical information shared between C_e and C_g. NMI is normalized MI, with two slightly different formulations: NMI_1 [178] and NMI_2 [8]. AMI [187] is an adjusted version of MI. They are defined as follows, respectively.

$$MI(C_e, C_g) = \sum_{i=1}^{|C_e|} \sum_{j=1}^{|C_g|} P(i, j) \log(\frac{P(i, j)}{P(i)P(j)}) \tag{1.3}$$

$$NMI_1(C_e, C_g) = \frac{MI(C_e, C_g)}{\sqrt{H(C_e)H(C_g)}} \tag{1.4}$$

$$NMI_2(C_e, C_g) = \frac{MI(C_e, C_g)}{(H(C_e) + H(C_g))/2} \tag{1.5}$$

[1] In probability theory, the expected value (a.k.a. expectation, mean, average) is the weighted average of all possible outcomes of a random variable. Source: https://en.wikipedia.org/wiki/Expected_value

$$AMI(C_e, C_g) = \frac{MI(C_e, C_g) - E[MI(C_e, C_g)]}{\sqrt{H(C_e)H(C_g)} - E[MI(C_e, C_g)]} \tag{1.6}$$

where $|C_e|$ and $|C_g|$ are the number of clusters in C_e and C_g respectively, $P(i)$ is the probability that a randomly chosen instance belongs to cluster i in C_e, $P(j)$ is the probability that a randomly chosen instance belongs to cluster j in C_g, $P(i,j)$ is the probability that a randomly chosen instance belongs to both cluster i in C_e and cluster j in C_g, $E[MI]$ is the expected value of MI, $H(C_e)$ is the entropy of C_e (i.e. $H(C_e) = -\sum_{i=1}^{|C_e|} P(i)\log(P(i))$ [8], where $P(i)$ is the probability that a randomly chosen instance belongs to cluster i in C_e), $H(C_g)$ is the entropy of C_g. The range of NMI is $[0, 1]$ (for both NMI_1 and NMI_2), where 1 means that C_e and C_g are identical and 0 means that C_e and C_g are independent. Similarly, AMI equals 1 when C_e and C_g are identical. However, AMI equals 0 means that the mutual information between C_e and C_g equals the expected mutual information. Similar to ARI, the AMI of a random C_e is close to 0 (i.e. AMI is adjusted against chance), however, MI and NMI are not.

Internal metrics do not require the knowledge of C_g and therefore the evaluation is performed based on the clustering itself. They include Silhouette Coefficient [164], Calinski-Harabasz Index [28], and Davies-Bouldin Index [42]. We do not explain them more because they will not be used in this dissertation.

In practice, the *scikit-learn*[2] [150] provides fast and easy-to-use APIs for most of the metrics (external and internal) mentioned above. In our experiments, we use *scikit-learn* (version 0.22.2.post1 and version 1.1.1) to compute the scores of ARI, AMI, and NMI (more precisely NMI_2, which is the default option in the versions of *scikit-learn* that we use).

1.2 . k-means clustering

1.2.1 . Classical algorithm

The k-means algorithm is one of the most well-known and widely used clustering algorithms. The essential method used in the k-means algorithm was proposed independently by several people over time: Hugo Steinhaus in 1956 [177], Stuart Lloyd in 1957 (although not published as a journal article until 1982) [121], Geoffrey H. Ball in 1965 [16], James MacQueen in 1967 [122]. So the k-means algorithm is also referred to as *Lloyd's algorithm*, although the term "k-means" was first used by James MacQueen.

As stated in Section 1.1.1, the k-means algorithm belongs to partitional clustering. Given a dataset $X = \{x_1, ..., x_n\}$ where each instance x_i have d dimensions

[2] https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation

and the desired number of clusters is k_c ($k_c \leq n$), the k-means algorithm partitions X into k_c clusters $C = \{C_1, ..., C_{k_c}\}$ with the objective of minimizing the sum of within-cluster variances ϕ (i.e. the sum of squared Euclidean distances between each instance and the centroid of the cluster to which it belongs). The mathematical formulation is:

$$\arg\min_{C} \sum_{i=1}^{k_c} \sum_{x \in C_i} \| x - \mu_i \|^2 = \arg\min_{C} \phi \tag{1.7}$$

where μ_i is the centroid of C_i, i.e. the mean of instances in C_i. In fact, finding the optimal solution to the problem 1.7 is NP-hard even for $k_c = 2$ [7]. The k-means algorithm adopts an iterative refinement approach, as briefly summarized in Algorithm 1. It first chooses k_c instances uniformly at random from the dataset X as initial cluster centroids (i.e. seeding step). Then the algorithm iterates two steps which we call: *ComputeAssign* and *Update*, until reaching the stopping criterion.

- The *ComputeAssign* step computes the Euclidean distance between each instance and each centroid. For each instance, the algorithm compares the distances[3] related to different centroids and assign the instance to the nearest centroid. In addition, the algorithm tracks the number of instances that have different cluster labels over two consecutive iterations.
- The *Update* step calculates the means of all instances that are assigned to the same centroid and updates the centroids.
- The stopping criterion can be a maximum number of iterations, or a relatively stable result, i.e., when the proportion of data instances that change labels is below a predefined tolerance.

The time per iteration of the k-means algorithm is $\mathcal{O}(n \cdot k_c \cdot d)$. The number of iterations varies with the nature of data, the initial positioning of centroids and the chosen stopping criterion.

The k-means algorithm is simple and efficient, but it has several drawbacks: (1) It usually forms spherical or convex clusters even if they do not really exist [91]; (2) It is sensitive to the initialization of centroids and may converge to local optima if the initial centroids are not properly chosen [92], actually it can yield arbitrarily bad clusterings with random initialization [13]; (3) It suffers from the *curse of dimensionality* because the Euclidean distance metric will lose sensitivity in high-dimensional space [89, 46]; (4) It requires the knowledge of k_c.

[3] In practice, we can compute and compare **squared Euclidean distances** to avoid square root operations.

Algorithm 1: k-means algorithm [122]

Input:
(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$
(2) Number of clusters k_c

Output: Cluster labels of n data instances

1 Seeding step;
2 **repeat**
3 | *ComputeAssign* step ;
4 | *Update* step ;
5 **until** *stopping criterion satisfied*;

1.2.2 . Better seeding with k-means++

There are many works that address the drawbacks of k-means, e.g. methods for finding better initial centroids [209, 13, 3], methods for choosing the number of clusters [71, 154, 126]. In this dissertation, we are particularly interested in the k-means++ algorithm [13], which was proposed by Arthur and Vassilvitskii in 2006 and has become probably the most well-known and widely used method for improving the seeding step of k-means.

Algorithm 2: k-means++ algorithm [13]

Input:
(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$
(2) Number of clusters k_c

Output: Cluster labels of n data instances

1 Seeding step:
1.1 | choose the first centroid uniformly at random from X;
1.2 | choose an instance x from X with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$ as the next centroid, where $D(x)$ is the shortest distance from an instance x to the previously chosen centroids;
1.3 | repeat Step 1.2 until k_c centroids have been chosen.
2 Proceed as with the standard k-means algorithm.

Unlike k-means which simply uses random sampling, k-means++ uses sequential adaptive sampling in the way that the chosen initial centroids are likely to be well scattered. Algorithm 2 describes the steps of k-means++. It chooses k_c initial centroids one by one. The first centroid is chosen uniformly at random from the dataset X. Then from the second centroid to the k_c-th centroid, each one is chosen from the dataset X with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$, where $D(x)$ denotes the shortest distance between a data instance x and the centroids that have been chosen previously (i.e. the distance between x and the nearest centroid). Therefore, an instance farther from the previously chosen centroids has a higher chance

of being chosen as the next centroid. This probabilistic sampling way is also referred to as D^2 *weighting*. The steps after seeding are identical with the k-means algorithm.

Thanks to careful seeding, the k-means++ algorithm can improve, often significantly, both the speed and the accuracy of k-means. In fact, Arthur and Vassilvitskii [13] proved that the sum of within-cluster variances ϕ satisfies: $E[\phi] \leq 8(\ln k_c + 2)\phi_{OPT}$, which means k-means++ is $\mathcal{O}(\log k_c)$-competitive with the optimal clustering.

1.3 . Spectral clustering

Spectral clustering is a more recent clustering method than k-means. Essentially, it embeds data into the sub-eigenspace of graph Laplacian (where the cluster-properties in the data is enhanced), and then finds the clusters in the embedded representation (often by k-means). Based on spectral graph theory (see Section 1.3.1), spectral clustering has several fundamental advantages over k-means (see Section 1.3.2). However, it also has several disadvantages which need to be addressed (see Section 1.3.3).

Let us start by briefly reviewing the history of spectral clustering. Broadly speaking, spectral clustering refers to methods that cluster data instances using eigenvectors of matrices derived from the data. It is closely related to spectral graph partitioning, for which the interesting links between spectral and cluster properties of graphs were first discovered in 1973 by Donath and Hoffman [47], and Fiedler [58]. Since then, many works have deepened the study on spectral partitioning and clustering, e.g. References [156, 25, 69, 66]. From 1990s to early 2000s, a number of algorithms [171, 38, 153, 173, 136] for spectral clustering have been proposed, but they differ in three ways [194, 136]: which matrix to compute, which eigenvectors to use, how to derive clusters from the chosen eigenvectors. The version proposed by Shi and Malik [173] in 2000 and the version proposed by Ng, Jordan, and Weiss [136] in 2001 have gained the most popularity over the last two decades. They are usually considered as the classical algorithms for spectral clustering. Meanwhile, spectral clustering has been actively studied in many non-standard settings. Particularly, Ulrike von Luxburg [188] provided in 2007 a very nice and comprehensive tutorial on spectral clustering, covering the related graph theory, algorithms, perspectives, history, practical details, etc. This Section 1.3 is mainly based on the tutorial [188].

1.3.1 . Theoretical basis and algorithms

Given a set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$ and the number of clusters k_c, the goal of clustering is to divide the n instances into k_c clusters according to pairwise similarities, such that instances inside the same clusters are similar and instances in different clusters are dissimilar. This can be interpreted, in spectral graph theory, as splitting a graph into k_c partitions such that the graph

cut is minimized and balanced.

Specifically, let $G = (V, E)$ represent an undirected weighted graph with a vertex set $V = \{v_1, ..., v_n\}$ and an edge set $E = \{(i, j, w_{ij})\}_{i,j \in 1,...,n}$. Each vertex v_i in the graph G corresponds to the instance x_i in the given dataset. The edge (i, j, w_{ij}) (representing the connection between vertices v_i and v_j with weight w_{ij}) corresponds to the similarity s_{ij} between the data instances x_i and x_j. Note that $w_{ij} \geq 0$ and $s_{ij} \geq 0$. In case of no connection/edge between v_i and v_j, we have $s_{ij} = w_{ij} = 0$. Therefore, the graph G can be represented algebraically by the similarity matrix S (a.k.a. affinity matrix or adjacency matrix or kernel matrix in the literature) defined by

$$S = [s_{ij}]_{i,j=1,...,n}, \text{ with } s_{ij} = \begin{cases} w_{ij}, & \text{if } i \neq j \text{ and } (i, j, w_{ij}) \in E \\ 0, & \text{otherwise.} \end{cases} \tag{1.8}$$

Note that by definition $s_{ij} = 0$ if $i = j$, i.e. the diagonal elements of the similarity matrix are always 0. Since the graph is undirected, we have $s_{ij} = s_{ji}$ and S is symmetric.

The first step of spectral clustering is to construct the similarity graph and generate the corresponding similarity matrix. Two things are worth noting as they can essentially affect the final clustering result. **(1) How to measure the distance or similarity between two instances?** There are a number of metrics, such as Euclidean distance, Gaussian similarity, and cosine similarity. The choice of metric should depend on the domain the data comes from and no general advice can be given [188]. The most commonly used metric seems to be the Gaussian similarity function (see Eq. 1.9), where the Euclidean distance is embedded, the parameter σ controls the width of neighborhood and the similarity is bound to (0, 1]. However, the cosine similarity metric (see Eq. 1.10) appears to be more effective for data in high-dimensional space [89]. **(2) How to construct the similarity graph?** There are several common ways, such as *full connection*, ε*-neighborhood* and k*-nearest neighbor* [188]. The first way generates a dense matrix. The last two ways yield typically a sparse similarity matrix by setting the similarity s_{ij} to 0 if the distance between instances x_i and x_j is greater than a threshold (ε) or x_j is not among the nearest neighbors of x_i, respectively. However, the k*-nearest neighbor* seems more computationally expensive as it requires sorting operations.

$$\text{Gaussian similarity metric: } s_{ij} = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right) \tag{1.9}$$

$$\text{Cosine similarity metric: } s_{ij} = \frac{x_i \cdot x_j}{\| x_i \| \| x_j \|} \tag{1.10}$$

A vertex may have connections with other multiple vertices. The degree of a vertex v_i is defined as $deg_i := \sum_{j=1}^{n} w_{ij} = \sum_{j=1}^{n} s_{ij}$, and the degree matrix

D of graph G is defined as a diagonal matrix with the degrees $deg_1, ..., deg_n$ on the diagonal. The (unnormalized) graph Laplacian is then defined as $L := D - S$ and can be further normalized as the symmetric matrix $L_{sym} := D^{-1/2}LD^{-1/2}$ or the non symmetric matrix $L_{rw} := D^{-1}L$ which is closely related to a random walk [188]. It can be proved[188] that L, L_{sym} and L_{rw} are all positive semi-definite and have n non-negative real eigenvalues with the smallest one being 0. Particularly, the graph Laplacian does not depends on the diagonal elements of the similarity matrix and its eigenvalues and eigenvectors (together called *eigenpairs*) are associated with many properties of the graph [188].

As mentioned before, clustering on a dataset X corresponds to partitioning a graph G into k_c partitions by finding a minimum balanced cut. *Ratio cut* and *normalized cut* are the two most common ways to measure the balanced cut, however minimizing *ratio cut* or *normalized cut* is an NP-hard optimization problem. Fortunately, if we relax one of its constraints, the problem can be approximately solved through the smallest k_c eigenvectors (associated with the smallest k_c eigenvalues) of the unnormalized graph Laplacian (standard eigenproblem) or of the normalized graph Laplacian (generalized eigenproblem) [188, 135]. In the unnormalized case we have then

$$L\,U \;=\; U\,\Sigma, \tag{1.11}$$

where U is the $n \times k_c$ matrix composed of the k_c eigenvectors $u_1, ..., u_{k_c}$ as columns, and Σ is the diagonal $k_c \times k_c$ matrix with the k_c eigenvalues $\lambda_1, ..., \lambda_{k_c}$ on the diagonal. In order to achieve good clustering in broader cases, it is argued and advocated [188] to use normalized instead of unnormalized graph Laplacian, and in the two normalized cases to use L_{rw} instead of L_{sym}. Obviously, choosing a Laplacian matrix and its properties impacts the choice of solvers that can be used to calculate its eigenvectors (e.g. choosing L_{rw} will not allow the use of the dense symmetric eigensolver `syevdx` in the cuSOLVER library [143]). Solving an eigenvalue problem has a time complexity of $\mathcal{O}(n^3)$ in general [203]. The computed k_c eigenvectors can be considered as the embedded representation of the original n data instances in the k_c-dimensional eigenspace of graph Laplacian. Or, each row of U can be regarded as the embedded representation in $\mathbb{R}^{k_c}$ of the original data instance in $\mathbb{R}^d$ with the same row number.

The computed k_c eigenvectors are the continuous solution to the above relaxed problem. To finally get the clustering, it needs to be transformed into a discrete solution. This is usually undertaken by the k-means algorithm. We just need to apply the k-means on the embedded representation by considering each row of the matrix U as a k_c-dimensional point, which therefore allows to find k_c clusters of original n data instances. In addition, to further improve the clustering result, it is customary to scale each row of matrix U to unit length before performing the k-means.

As summarized in Algorithm 3 and illustrated in Figure 1.1, spectral clustering involves several data transformation steps. A similarity matrix is first computed

Algorithm 3: Spectral clustering algorithm [188]

Input:

(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$

(2) Number of clusters k_c

Output: Cluster labels of n data instances

1 Construct the similarity graph and generate the similarity matrix S;
2 Derive the graph Laplacian (L, L_{sym}, or L_{rw});
3 Compute the smallest k_c eigenvectors of graph Laplacian which form the columns of the matrix U;
4 Normalize each row of matrix U to have unit length;
5 Perform the k-means on the points defined by the rows of U.

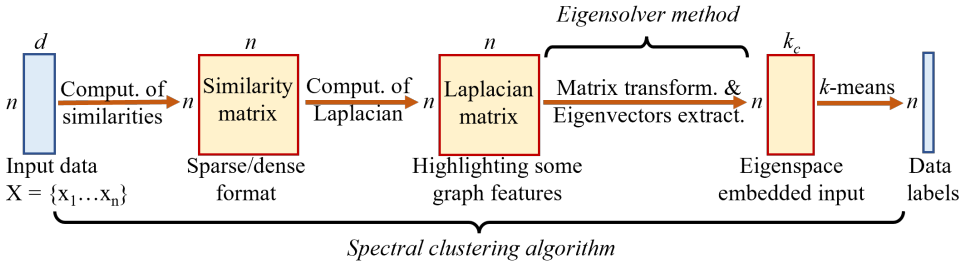


Figure 1.1: Main computation steps in spectral clustering

to model the connectivity of similarity graph based on the input data. Then a Laplacian matrix is deduced, highlighting some information about the graph topology and the desired clustering. Eigenvectors are extracted, transcribing the information from the Laplacian matrix and allowing to form a $n \times k_c$ matrix where the n input data are encoded in the eigenspace of the first k_c eigenvectors. In this space, a simple k-means can then group the input data into k_c clusters. Therefore, spectral clustering may also be regarded as the combination of a heavy *preprocessing* step (including main computations) and a classical k-means step.

1.3.2 . Advantages

Spectral clustering has several important advantages:

1. It does not make strong assumptions on the form of clusters [188]. Contrary to k-means and k-means++ clustering which can only form convex clusters, spectral clustering can also discover non-convex or linearly non-separable clusters and is more likely to find the global minimum owing to the *embedding* step. Examples are shown in Figure 1.2.

2. As the embedding step projects data from $\mathbb{R}^d$ to $\mathbb{R}^{k_c}$, it can play a role of dimensionality reduction for high-dimensional data that has d dimensions and k_c clusters with $d > k_c$, which will benefit the following k-means step.

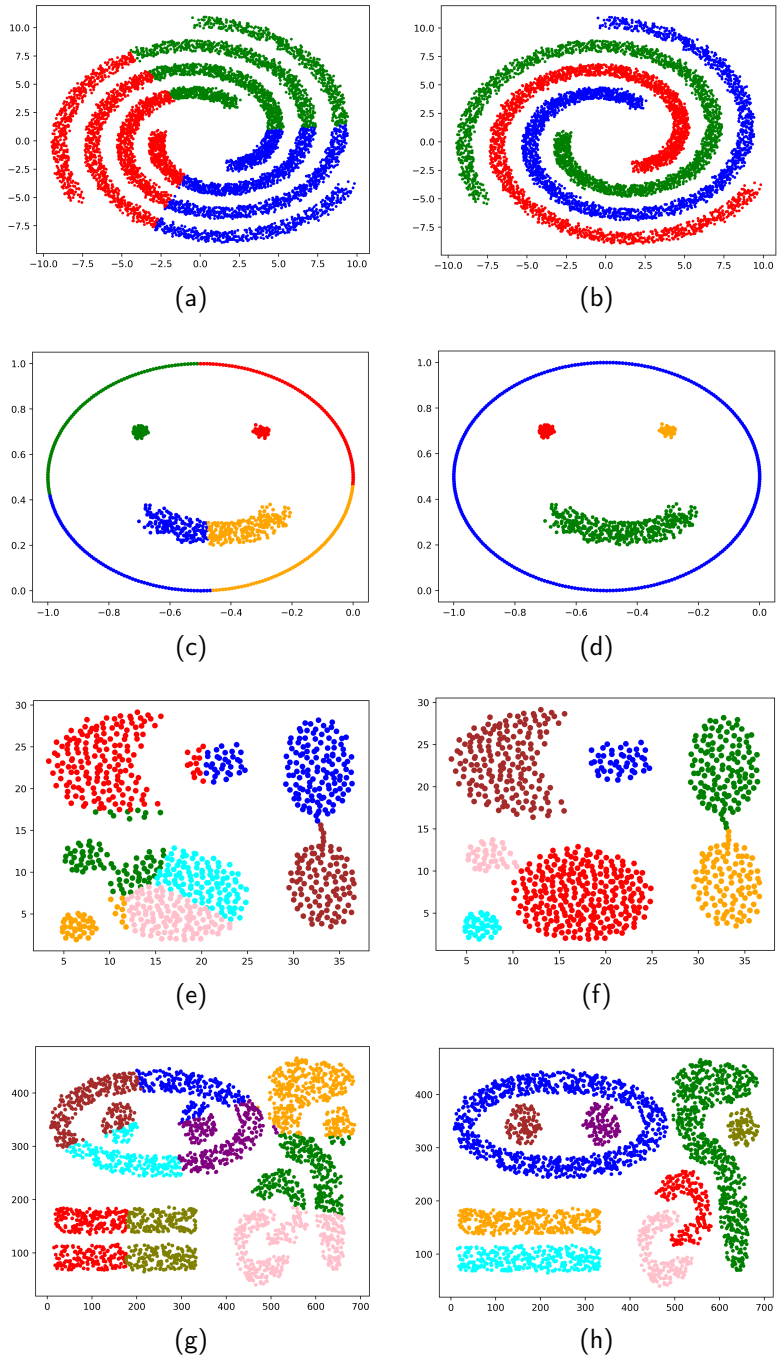


Figure 1.2: k-means++ (left: a, c, e, g) vs. spectral clustering (right: b, d, f, h) on four datasets (Spirals, Smile2, Aggregation, Complex9)

3. Spectral clustering algorithms have the potential to be efficiently implemented on HPC platforms because they require substantial linear algebra

computations which can be processed using existing libraries.

4. When k_c is unknown, the eigenvalues and eigenvectors calculated in spectral clustering can be exploited to estimate the natural k_c [188, 210, 198].

1.3.3 . Drawbacks and approaches for improvement

Although spectral clustering is attractive with the advantages mentioned in Section 1.3.2, its classical algorithms have three serious disadvantages: (1) scalability challenge; (2) importance and difficulty in tuning several parameters; (3) sensitivity to noise and outliers. In the following we explain each of them and summarize the existing approaches to address them. This dissertation will mainly focus on solving the scalability challenge of spectral clustering, while additionally we will address the noise sensitivity problem by proposing a noise filtering algorithm in Chapter 4.

Scalability challenge

The time complexity[4] of classical spectral clustering algorithms is $\mathcal{O}(n^3)$ [203], mainly due to the construction of the similarity matrix ($\mathcal{O}(n^2d)$) and the calculation of eigenvectors ($\mathcal{O}(n^3)$ when using direct methods). Moreover, storing the similarity matrix and the graph Laplacian matrix requires $\mathcal{O}(n^2)$ memory space. Therefore, the high complexities in terms of number of operations and memory space requirements lead to a great challenge when processing large datasets with spectral clustering.

We surveyed existing methods for addressing this scalability issue and broadly classify them into the following two classes.

- **Reduce the time and space complexities using approximation, a.k.a. Approximate Spectral Clustering.** The basic idea is to first solve the clustering problem for a relatively small subset of data and then extrapolate the solution to the entire dataset by approximation. By doing this, one can avoid the expensive construction and eigendecomposition of the original $n \times n$ matrices. Popular approximation methods include Nyström-based [59], representative-based [203], landmark-based [34], and EFM-based [79] methods. We will introduce them in more detail in Section 1.4.

- **Accelerate spectral clustering using parallel and distributed computing, a.k.a. Parallel Spectral Clustering.** Modern parallel and distributed architectures provide powerful computing capabilities. With efficient parallel implementations on these architectures, algorithms can often be accelerated considerably, e.g. by tens to hundreds of times or even more. We will review the related works on parallel spectral clustering in Section 1.5.

[4] People in different domains may have different understandings of the term "time complexity". In this dissertation, we consider it as the number of operations required by an algorithm.

Importance and difficulty in tuning several parameters

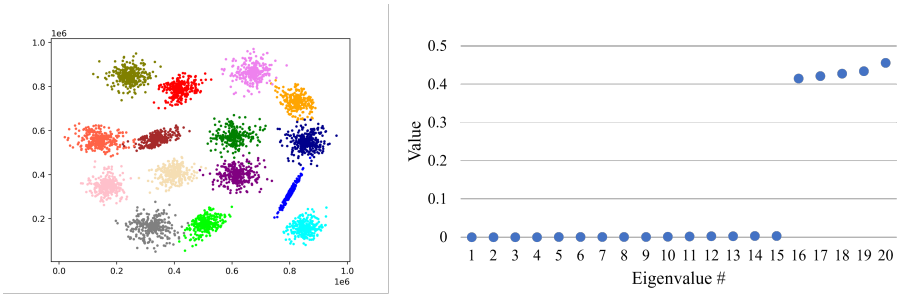
The proper functioning of classical spectral clustering algorithms relies on the appropriate setting of several parameters[5], such as the number of clusters k_c, and connectivity parameters ε, k, σ for similarity graph construction (see Section 1.3.1). These parameters are usually not easy to tune.

The number of clusters k_c is required as input in spectral clustering. However, for datasets for which we know little about their distribution or characteristics, it would be difficult to know k_c in advance. There are many methods that can be used to automatically determine k_c for spectral clustering. They can be roughly divided into two categories: (1) generic methods that can be used for any clustering algorithm, e.g. based on evaluating cluster validity indexes (e.g. gap heuristic [183], data depth difference [149]), based on deep learning [51]; (2) methods dedicated to spectral clustering, e.g. based on analyzing eigenvalues [188, 148], based on analyzing eigenvectors [210, 198]. In case of generic methods, a wide variety of cluster validity indices for determining k_c have been implemented in several R packages such as `cclust` [45], `clusterSim` [190], `NbClust` [32].

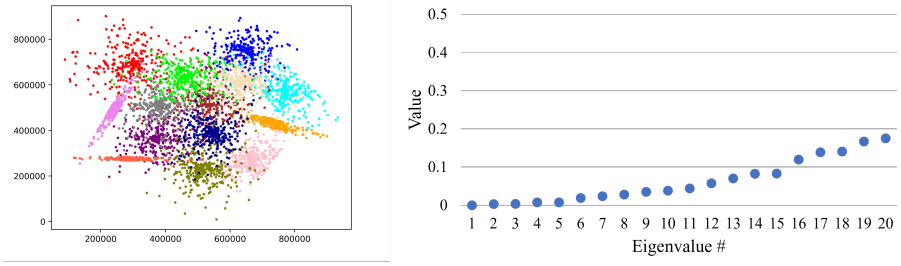
Here we introduce a simple and interesting method based on analyzing eigenvalues [188]. Considering the eigenvalues of a matrix in ascending order ($\lambda_1 \leq ... \leq \lambda_n$), the eigengap (also called spectral gap) is defined as $\gamma_k = |\lambda_k - \lambda_{k+1}|$. Uniquely for spectral clustering, there exists an eigengap heuristic for determining k_c: for a dataset with k_c distinctly separated clusters, the smallest k_c eigenvalues $\lambda_1, ..., \lambda_{k_c}$ of its associated graph Laplacian matrix (L, L_{sym} or L_{rw}, see Section 1.3.1) are close to 0, but the $(k_c + 1)$-th eigenvalue λ_{k_c+1} is distinctly larger than 0. An example of such eigengap heuristic is shown in Figure 1.3 (a). However, the eigengap heuristic is less effective when the clusters are not well separated, e.g. more noisy or overlapping clusters in Figure 1.3 (b), as in that case all eigengaps tend to be approximately the same and it would be more difficult to detect the number of clusters.

In 2007, Von Luxburg [188] summarized some theoretical results and gave some rules of thumb regarding choosing the values for **connectivity parameters** (e.g. k for *k-nearest neighbor* graph, ε for *ε-neighborhood* graph, σ for Gaussian similarity function). Essentially, it is suggested that the connectivity parameters should be chosen such that the constructed similarity graph is connected, or is composed of only few connected components. To guarantee such connectivity in the limit of sample size $n \to \infty$, k should be chosen in the order of $\log(n)$, ε should be chosen as $(\log(n)/n)^d$. However, these theoretical results may not work on a finite sample. Another way to achieve a safely connected ε-neighborhood graph is to choose ε as the longest edge length in a minimal spanning tree of the fully connected graph. However, ε will be chosen too large if the data contains outliers or contains several tight and significantly separated clusters. For the Gaussian similarity function,

[5] In fact, this is also a common issue for many machine learning algorithms.



(a) On the S1 set with 15 clusters



(b) On the S4 set with 15 clusters

Figure 1.3: Success and failure of the eigengap heuristic on S-sets

the rules of thumb are to choose σ in the order of the average of k-th nearest neighbor distance where k is chosen in the order of $\log(n)$, or to choose $\sigma = \varepsilon$ where ε is determined by the minimal spanning tree heuristic. Nonetheless, the above suggestions and rules of thumb might not work at all depending on the data distribution [188]. In 2004, Zelnik-Manor and Perona [210] proposed to compute a local scaling parameter σ_i for each data instance x_i instead of using a single σ for all instances. The similarity between a pair of instances can then be formulated as $s_{ij} = exp(-\frac{dist^2(x_i,x_j)}{\sigma_i \sigma_j})$. The parameter σ_i can be chosen as the distance between instance x_i and its k-th nearest neighbor x_k, i.e. $\sigma_i = dist(x_i, x_k)$. The interest of local scaling is that it can capture the respective neighborhood information of data instances and therefore the resulting spectral clustering can handle data of multiple scales/densities. However, local scaling can be computationally expensive and introduces another parameter k. Other related works on the tuning of σ include [129, 130].

Sensitivity to noise and outliers

Classical algorithms of spectral clustering often fail to achieve satisfactory clustering on noisy data, mainly because the *block structure* of the similarity matrix is destroyed by noise [117]. Besides, Hennig et al. [81] strongly recommended that outliers should be detected and removed before performing eigendecomposition in spectral clustering, because outliers would introduce spurious information into

eigenvalues and eigenvectors. Note that the two terms "noise" and "outlier" are often used imprecisely and interchangeably in the literature.

Many methods have been proposed for noise or outlier robust spectral clustering. In 2007, Li et al. [117] used a data warping model to map data into a new space where each cluster becomes more compact and different clusters (including the noise cluster formed by noise points) become well separated. Then spectral clustering is applied in the new space. In 2015, Hennig et al. [81] suggested that the easiest way to eliminate outliers is by removing all points whose corresponding vertex degrees are under a small threshold w.r.t. the average degree. In 2017, Ina et al. [89] discovered mathematically and confirmed experimentally that outliers can form a cluster during the process of spectral clustering if the outlier cluster is counted into the number of clusters k_c. Essentially, this is because similarities between outliers have a low variance. After eigendecomposition of the graph Laplacian, all outliers will have similar coefficients of the first k_c eigenvectors, and thus tend to form a cluster. Their experiments further indicated that the more outliers the data contain, the more stable the outlier cluster formation. Based on the idea that the similarity graph can be decomposed into clean data and sparse corruptions, Bojchevski et al. [24] proposed to jointly learn the latent corruptions and the spectral embedding of clean data to improve spectral clustering performance on noisy data. Other related works include References [15, 195].

1.4 . Approximate spectral clustering

Following the indication in Section 1.3.3, we introduce some influential methods for (non-parallel) approximate spectral clustering.

Nyström-based approximation

The Nyström method was initially proposed by E. J. Nyström in 1928 [146] for finding numerical approximations to integral equations. It can also be used to efficiently generate a low-rank approximation of a matrix from a sampled subset of matrix columns [107].

In 2004, Fowlkes et al. [59] applied the Nyström method to spectral clustering. Essentially, they randomly choose m samples from the dataset, compute the similarities between the m samples and all n data instances to form a narrow strip of the full similarity matrix, then use the Nyström method to approximate the full similarity matrix S and the leading eigenvectors of $D^{-1/2}SD^{-1/2}$. The time and memory space complexities are thus substantially reduced to $\mathcal{O}(n \cdot m^2) + \mathcal{O}(m^3)$ and $\mathcal{O}(n \cdot m)$, respectively [111]. However, the work of Fowlkes et al. [59] has several drawbacks according to Reference [203]: (1) the random sampling does not incorporate any information about the similarity matrix; (2) the work does not provide theoretical guarantees of performance (although an empirical quantitative analysis of performance is provided); (3) the working memory requirements can

still be high, e.g. 6GB for data of size $n = 10^5$, 17GB for data of size $n = 10^6$; (4) if the dataset is unbalanced, small clusters may be missed and numerical stability problems may occur.

In 2010, Zhang and Kwok [212] analyzed how the choice of landmark points (i.e. samples) affect the approximation quality of the Nyström method, and based on their error analysis, they proposed to use the k-means clustering centers as the landmark points. In 2011, Li et al. [111] further reduced the time and memory space complexities of Nyström-based spectral clustering by directly computing a rank-k approximation of $D^{-1/2}SD^{-1/2}$ and avoiding storing the sampled similarity matrix. In 2012, Kumar, Mohri, and Talwalkar [107] analyzed a variety of fixed and adaptive sampling techniques for the Nyström method and found experimentally that the k-means algorithm was the state-of-the-art adaptive sampling technique, producing the most accurate approximations in nearly all settings while taking about the same time as other adaptive techniques. Nevertheless, the k-means-based sampling is more expensive than random sampling and will be time-consuming if the data size or the sample size is large. In 2013, Choromanska et al. [37] applied the Nyström approximation to the normalized graph Laplacian matrix L_{sym} for fast spectral clustering and provided performance guarantees through theoretical analysis.

Representative-based approximation

In 2009, Yan, Huang, and Jordan [203] proposed a general framework[6] for fast approximate spectral clustering, in which a preprocessor first reduces the data to a relatively small number of representative points (data reduction/preprocessing step), then spectral clustering is performed on the representatives, and finally the original data points are assigned cluster memberships based on the representatives. Through a theoretical analysis that establishes a quantitative relationship between the distortion in the input and the mis-clustering rate in the output, the authors argued that the goal of a preprocessor should be to minimize distortion so that the effect of data reduction on spectral clustering is minimized. The authors provided two examples of such preprocessors: the first is the classical k-means algorithm, the second is the random projection tree. They showed experimentally that their fast approximate spectral clustering algorithms can achieve significant speedups with little degradation in clustering accuracy compared to classical spectral clustering algorithms, and their algorithms run several times faster than Nyström-based approximate spectral clustering, with comparable accuracy and significantly smaller memory footprint.

[6] The authors admitted that their approach is not fundamentally new since using preprocessing techniques to overcome computational bottlenecks is a tradition in the data mining community.

Landmark-based approximation

Inspired by the works on sparse coding [110] and large graph construction [120], Chen and Cai [34] proposed in 2011 a scalable spectral clustering method, called Landmark-based Spectral Clustering (LSC). It first generates p ($p \ll n$) landmark points (i.e. representatives) using random sampling or the k-means algorithm, and builds a $p \times n$ sparse representation matrix Z to represent the original n data points as the linear combinations of the p landmarks. Then it performs the eigendecomposition on the landmark-based representation ZZ^T to get the first k_c eigenvectors (denoted by $A \in \mathbb{R}^{p \times k_c}$) and finally derives the first k_c eigenvectors of $Z^T Z$ (i.e. $n \times n$ similarity matrix) from A. Their experiments showed that compared to original spectral clustering, LSC can reduce significantly the running time with comparable and sometimes even better clustering accuracy.

EFM-based approximation

In 2018, He et al. [79] proposed a fast spectral clustering method via Explicit Feature Mapping (EFM), named FastESC, which reduces the complexity of classical spectral clustering algorithms in a different manner than Nyström-based methods. FastESC first employs random Fourier features to explicitly represent data in *kernel space* (i.e. EFM). Let Y denote the kernel mapped data with dimension d_m ($d_m \ll n$), $K = Y^T Y$ denote the $n \times n$ kernel matrix, $J = YY^T$ denote the $d_m \times d_m$ matrix. Then, similarly to LSC, FastESC builds J and solves the eigenvalue problem of J (instead of building K and running eigendecomposition on K), and finally approximate eigenvectors of K by those from J using a correlation equation. Their experiments showed that, with a large enough d_m, FastESC can achieve similar clustering accuracy to Nyström methods while running two times faster.

Summary

Finally, we point out that there are still many other methods that we have not covered here, e.g. References [174, 18, 167, 192, 152, 119, 186, 115, 113, 35, 196]. In 2020, Tremblay and Loukas [185] reviewed existing sampling-based methods for approximate spectral clustering, focusing particularly on their approximation guarantees. Interestingly, they concluded that: "the most scalable methods are only intuitively motivated or loosely controlled, whereas those that come with end-to-end guarantees rely on strong assumptions or enable a limited gain of computation time."

1.5 . Parallel spectral clustering

From the hardware side, there are various parallel and distributed architectures, such as multi-core CPU architectures, many-core GPU architectures, CPU-GPU

heterogeneous architectures, multi-CPU architectures, multi-GPU architectures, FPGA architectures, computer clusters, and supercomputers. From the software side, various parallel programming models/languages are available, such as POSIX threads, OpenMP, CUDA, OpenACC, MPI, MapReduce, and Julia.

In this dissertation, we will parallelize spectral clustering on GPU architectures and CPU-GPU heterogeneous architectures (see objectives in Section 1.6) using OpenMP and CUDA programming models. Parallelization of spectral clustering on other architectures or using other programming models can be found in, e.g. References [33, 96, 204, 88]. A survey on parallelization of various clustering methods can be found in Reference [41].

1.5.1 . Strengths and challenges of CPU vs. GPU

Modern CPU architectures

Modern CPU architectures feature several levels of parallelism: a CPU can have multiple processors, each processor has multiple physical cores (dozens at most), each physical core has several Arithmetic Logic Units (ALUs). Moreover, each physical core can typically be virtualized as two logical cores which share the vector units and the cache memory, so the operating system can run simultaneously two threads on each physical core. The parallelism among vector units is usually referred to as *SIMD (Singe Instruction Multiple Data)*, which can be realized by explicit vectorization using intrinsics [90], or by the compiler's *auto-vectorization*. However, in order to enable auto-vectorization, we need to be careful that the code is written in a way that facilitates the compiler to achieve vectorization, and the pertinent compilation flags are specified (e.g. `-O3`/`-Ofast`, `-march=native`). The parallelism among multiple cores is realized by multithreading using, e.g. POSIX threads [27], OpenMP [147], or by multiprocessing using, e.g. MPI [124].

Besides the hierarchy of computing resources, modern CPU architectures also have a memory hierarchy: registers $\rightarrow$ L1 cache $\rightarrow$ L2 cache $\rightarrow$ L3 cache $\rightarrow$ RAM $\rightarrow$ hard disk. They are listed in descending order of memory access speed, and in ascending order of memory size. Each core has and accesses its own registers and L1 cache (on-chip memory), while multiple cores per processor share the L2 cache (on-chip memory) and L3 cache (when it exists). For best performance, programs should maximize cache utilization and minimize cache misses. All cores of all processors share RAM and hard disk (a.k.a. shared memory model), but there is a problem called *NUMA effect (Non-Uniform Memory Access)*: for each processor, the time to access data in the RAM attached to another processor is much longer than the time to access data in the RAM attached to itself. Therefore, the relative location of data and threads has an impact on program performance.

As can be seen, developing efficient code on modern CPUs is not easy. It requires special care and various optimizations.

Modern GPU architectures

A GPU (short for Graphics Processing Unit) is a type of computer hardware specialized in processing graphics and images, acting as a coprocessor or device to a CPU. Modern GPUs[7] feature massively parallel architectures and are especially suited to large fine-grained parallel problems. Therefore, modern GPUs can also be used for *GPGPU* (short for General-purpose computing on GPUs) and can be regarded as *accelerators* for CPUs. The microarchitecture of NVIDIA GPUs have been continuously evolving over time, but some fundamental features remain.

An NVIDIA GPU consists of a number of *Stream Multiprocessors (SMs)*. Each SM usually has 32 or 64 processing units called *CUDA cores*. Basically, a GPU SM can be compared to a CPU core, and a CUDA core can be compared to a CPU ALU. A modern GPU usually has tens of SMs, resulting in thousands of CUDA cores. Recent NVIDIA GPUs also have *Tensor cores*, which are hardware implementations of matrix operators and enable mixed-precision computing. In terms of memory, GPUs have a similar hierarchy to CPUs. Each SM has its own registers, shared memory, and cache (shared memory is a programmable L1 cache). They all have much faster access speed and much smaller size than GPU global memory which is shared by all SMs. Nevertheless, GPU global memory usually have several times higher peak bandwidth and several times smaller size than CPU RAM. The CPU and GPU can communicate across a standard PCIe or a fast NVLink, but the bandwidth is still much lower than RAM access speed. Therefore, programs should minimize data transfers between CPU and GPU. Besides, GPUs also have other special-purpose memory, including local memory, constant memory, and texture memory.

GPGPU programming interfaces include CUDA, OpenACC, OpenCL, OpenMP, and Julia. The most influential one is CUDA (short for Compute Unified Device Architecture) [141]. NVIDIA has been actively developing, extending, and enriching the CUDA framework since its initial release in 2007. CUDA can support running hundreds of thousands of threads on modern GPUs. These threads are organized into grids, blocks, and warps in descending order. A warp typically consists of 32 consecutive threads running on 32 consecutive CUDA cores of a SM, which is called the *SIMT (Single Instruction Multiple Threads)* execution model. A block usually consists of multiple warps of threads, but is typically limited to 1024 threads. Similarly, a grid usually consists of multiple blocks of threads. Both blocks and grids can have one to three dimensions (X, Y, Z). Depending on GPU architectures, there are different characteristics of hardware resources and different constraints on thread programming. A scheduler is responsible for scheduling warps and blocks to run on SMs. Threads within the same block can communicate with each other through the shared memory of that block, while threads from different blocks can

[7]We mainly refer to NVIDIA GPUs as NVIDIA is the leading GPU company and provides a rich development environment for GPU programming.

only communicate through GPU global memory.

As can be seen, CUDA programming on GPU is complicated. It is an art to take full advantage of the GPU while respecting various constraints. The CUDA C++ Best Practices Guide [140] provides a nice tutorial to help GPGPU developers achieve the best performance from NVIDIA GPUs. It presents various parallelization and optimization techniques, such as memory optimizations, execution configuration optimizations, and instruction optimizations.

CPU vs. GPU

To summarize, the CPU can run a few dozen heavy threads in parallel, while the GPU can run thousands of light threads in parallel and achieve a higher overall instruction rate and memory bandwidth. Thus, the GPU is specialized for large fine-grained parallel computations. Due to the high computation cost of constructing the similarity matrix and computing the eigenvectors, it appears more interesting to exploit the massively parallel nature of the GPU. However, the GPU has limited global memory (at most tens of GB). How to store the memory-demanding similarity matrix and the graph Laplacian matrix on the GPU remains an important concern.

1.5.2 . GPU-accelerated spectral clustering

This section introduces the related works on GPU-accelerated spectral clustering and graph partitioning, and the existing works on the parallel implementation related to the three constituent steps of spectral clustering (i.e. similarity graph construction, partial eigendecomposition, and final k-means clustering).

GPU-accelerated spectral clustering and graph partitioning

The first paper that we found on this topic [215] was published in 2008, shortly after CUDA came out. It parallelizes spectral clustering algorithm on multi-core CPU and on GPU. However, their GPU implementation cannot scale to large datasets because dense matrices are constructed and stored on GPU. In fact, their benchmark datasets contain only thousands of instances.

Then, an example of video segmentation through spectral clustering in pixel level has been implemented on a cluster of GPUs [179], but unfortunately the authors introduced too briefly their parallelization details and did not give performance analysis of their parallel implementation.

Another work [97] proposes a parallel implementation for spectral clustering on CPU-GPU hybrid platforms. It constructs a sparse representation of the similarity graph, but it assumes the neighborhood information is given beforehand by an edge list, which facilitates the construction process. Their benchmark datasets are of medium size, with n at most in the order of 10^5. Besides, speedup limitations are reported for the eigen-decomposition step.

NVIDIA has developed efficient implementations of spectral graph partitioning on the GPU [135, 56, 55], and released the products in the nvGRAPH library [139] and RAPIDS cuGraph library [181]. However, since these works are oriented to graph analytics, they typically assume the edge list or the adjacency list of a graph is available, thus do not consider the graph construction process which would take $\mathcal{O}(n^2 d)$ arithmetical operations in the general sense of spectral clustering.

Similarity graph/matrix construction on GPU

To the best of our knowledge, most existing works on graph construction [48, 70, 10, 11] target k-*nearest neighbor* graph. We found a single work [97] on the construction of ε-*neighborhood* graph in sparse format on the GPU. It constructs sparse similarity matrix in Coordinate (COO) format but on the assumption that the neighborhood information is given by an edge list. However, in data clustering, it is generally assumed that the neighborhood information is not available in advance. Consequently, similarity matrix construction becomes harder especially in sparse format (see Section 3.3).

Interestingly, in the literature we identified a close connection between similarity graph/matrix construction and another field called *similarity search* [9] or *similarity query* [157]. However, again most studies in the field concern k-*nearest neighbor* search [158, 114, 112, 214, 65].

Partial eigendecomposition on GPU

Based on the work [180] that Nicolas Sylvestre did during his master internship under our direction, we briefly summarize three well-known methods for the calculation of the first few eigenpairs of a matrix. They include new matrix transformations to facilitate the eigenvectors extraction and are not specific to spectral clustering.

- **Arnoldi's** method [166]: it takes any input matrix (like L, L_{sym} or L_{rw}, see Section 1.3.1) and transforms it into an Hessenberg matrix, then calls an eigensolver (usually based on the *QR* algorithm). This is a generic but computationally expensive method.

- **Lanczos** method [166]: similar to Arnoldi's method but requires a real and symmetric (or Hermitian) input matrix (like L or L_{sym}) which is transformed into a tridiagonal matrix, before calling an eigensolver (like *QR*). It is considered as an efficient method but it suffers from numerical instabilities and cannot handle eigenvalues with multiplicity (which often happens in spectral clustering) [135].

- **LOBPCG** method [104]: requires a symmetric input matrix (like L or L_{sym}) or a pair of matrices with one symmetric and one symmetric positive definite (like (L, D)), then starts extracting the smallest k_c eigenpairs. The

LOBPCG method performs some transformations of the matrices and calls other eigensolvers on smaller internal submatrices. LOBPCG is more recent (released in 2000) than the previous two methods. Compared to Lanczos method, LOBPCG can handle eigenvalues with multiplicity and is more stable numerically. [135].

Implementations of these methods exist in different libraries. They require input matrices in dense or sparse format and are sometimes improved to be more robust to numerical instabilities. Mainly interested in GPU-accelerated implementations for large sparse matrices, we have surveyed the sparse eigensolvers of several GPU-accelerated libraries including cuSOLVER[8], nvGRAPH[9], cuGraph[10], MAGMA[11], AmgX[12], and ViennaCL[13]. Table 1.1 summarizes our survey results from different aspects.

The cuSOLVER library [143] is a GPU-accelerated library from NVIDIA providing LAPACK-like features (decompositions and linear system solutions) for both dense and sparse matrices. Based on the cuBLAS and cuSPARSE libraries, the cuSOLVER library contains several dense eigensolvers (e.g. `gesvd`, `syevd`, `syevdx`, `syevj`) and one sparse eigensolver (`csreigvsi`). As will be explained in Section 3.2.2, the `syevdx` eigensolver is the most appropriate for the computation of the smallest k_c eigenvectors regarding a symmetric Laplacian matrix stored in dense format. The sole sparse eigensolver `csreigvsi` is dedicated to sparse matrices defined in CSR storage format. However, it solves the simple eigenvalue problem by shift-inverse power method which requires an initial guess of eigenvalue and calculates only one eigenpair at a time. Thus it appears unsuitable for our need to automatically find the first few eigenpairs.

The nvGRAPH library [139] is dedicated to graph analytics with a set of graph algorithms optimized for the GPU. It was first released in 2017 with NVIDIA CUDA 8.0. The library contains three eigensolver-embedded (specifically Lanczos solver and LOBPCG solver) algorithms for spectral graph partitioning, which can satisfy our need. We will show in Section 3.4 the usage of these algorithms. However, since the last release in November 2019 with CUDA 10.2, NVIDIA does not actively develop the nvGRAPH product any more although the legacy version is still available[14]. We found in practice that the nvGRAPH library is backed by the cuBLAS, cuSPARSE and cuSOLVER libraries, because with more recent versions of CUDA we always encounter compilation warnings indicating that older versions of the backing libraries needed by nvGRAPH may conflict with more recent versions of those libraries. In place of nvGRAPH, NVIDIA has been actively developing the

[8] https://docs.nvidia.com/cuda/cusolver/index.html
[9] https://docs.nvidia.com/pdf/nvGRAPH_Library.pdf
[10] https://github.com/rapidsai/cugraph
[11] https://icl.utk.edu/magma/index.html
[12] https://github.com/NVIDIA/AMGX
[13] http://viennacl.sourceforge.net
[14] https://github.com/rapidsai/nvgraph

Table 1.1: Investigation of some GPU-accelerated libraries with eigensolvers

	cuSOLVER	**nvGRAPH**	**cuGraph**	**MAGMA**	**AmgX**	**ViennaCL**
Application field	Linear algebra	Graph analytics	Graph analytics	Linear algebra	Solver	Linear algebra
Source	NVIDIA	NVIDIA	NVIDIA	Public domain	NVIDIA	Public domain
Last release until Feb. 2022	CUDA 11.6 in Jan. 2022	CUDA 10.2 in Nov. 2019	RAPIDS 22.02 in Feb. 2022	MAGMA 2.5.4 in Oct. 2020	AMGX v2.2.0 in Apr. 2021	ViennaCL 1.7.1 in Jan. 2016
Oriented architectures	Single GPU, single node multi-GPU	Single GPU	Single GPU, multi-GPU	Multicore + multi-GPU hybrid systems	Single GPU, multi-GPU	Many-core architectures (GPUs, MIC), multi-core CPUs
Supported languages	CUDA	CUDA	Python (encouraged), C++	CUDA, HIP	C	CUDA, OpenCL, OpenMP
Eigensolvers	Dense solvers: QR, Jacobi, ... Sparse solver: shift-inverse power iteration	Lanczos, LOBPCG	Lanczos	Dense solvers: QR, divide-and-conquer, ... Sparse solver: LOBPCG	Power iteration, subspace iteration, Arnoldi, Lanczos, LOBPCG, ...	Power iteration, Lanczos
Our tests on eigensolvers	Yes	Yes	Yes	Yes	Not yet	Not yet

cuGraph library for a few years. It is very similar to the nvGRAPH library as it contains most nvGRAPH algorithms (including only two graph partitioning algorithms). However, the nvGRAPH is used in the CUDA environment while the cuGraph, as part of RAPIDS [181], is mainly used through Python interfaces with CUDA source code hidden behind. Despite this fact, we have built with efforts the cuGraph library (version associated with CUDA 11.5) from source[15] on our machine, and we succeeded in using the C++/CUDA API of cuGraph's graph partitioning algorithms. However, according to our experiments we found that the LOBPCG-eigensolver-embedded algorithm that exists in nvGRAPH seems to be missing in cuGraph, which is adverse for our use. So we conclude that the nvGRAPH library fits better our need than the current cuGraph library.

The MAGMA library [184] is a public domain linear algebra library optimized for "multi-core + multi-GPU" hybrid architectures. It contains a variety of dense eigensolvers and one sparse eigensolver. We did not try the dense eigensolvers of MAGMA because the cuSOLVER library already satisfies our need and we focus more on the eigenvalue problem of large sparse matrices. The sole sparse eigensolver of MAGMA is a GPU implementation of the LOBPCG method. We tried it (with MAGMA 2.5.4 installed) to calculate the eigenvectors of the graph Laplacian matrix, but unfortunately we were blocked by a "floating point exception" error.

The Algebraic Multigrid Solver (AmgX) library [134] is a GPU-accelerated core solver library from NVIDIA that accelerates computationally intense linear solver portion of simulations. It possesses multiple eigensolvers such as power iteration solver, subspace iteration solver, Arnoldi solver, Lanczos solver, LOBPCG solver, etc. The ViennaCL library [165] is an open-source linear algebra library designed for many-core architectures (GPUs, MIC) and multi-core CPUs. It includes eigensolvers based on power iteration and Lanczos methods.

We have yet to test the eigensolvers of AmgX and ViennaCL libraries for spectral clustering but it would be interesting as future work. In this dissertation we will mainly rely on the sparse eigensolvers embedded in nvGRAPH's graph partitioning algorithms.

Final parallel k-means

Recall that the k-means algorithm has a time complexity of $\mathcal{O}(n \cdot k_c \cdot d \cdot nbIters)$ (see Section 1.2.1), which is usually much smaller than the time complexities of similarity matrix construction and partial eigendecomposition. Nonetheless, it can still be time-consuming if any element of $\mathcal{O}(n \cdot k_c \cdot d \cdot nbIters)$ becomes very large. This can be addressed by designing efficient parallel implementations.

There are a large number of works on the parallelization of the k-means algorithm on different platforms. We mainly surveyed those published in recent years

[15] https://github.com/rapidsai/cugraph/blob/branch-22.04/SOURCEBUILD.md

and meanwhile related to CPU and GPU. For example, in 2015, Bhimani, Leeser, and Mi [22] parallelized k-means on three different platforms: shared memory using OpenMP, distributed memory using MPI, and CPU-GPU heterogeneous platform using CUDA. However, their OpenMP implementation only parallelizes the *ComputeAssign* step, leaving the *Update* step sequential; their CUDA implementation involves many transfers between CPU and GPU during the k-means iterations, which is generally recommended to be avoided in best practice. In 2017, Böhm, Perdacher, and Plant [23] proposed a highly optimized parallel implementation of k-means on multi-core CPU, named Multi-core K-Means (MKM), which is multi-threaded with OpenMP and explicitly vectorized using intrinsics. In 2019, Cuomo et al. [40] proposed a GPU-accelerated implementation of the k-means algorithm aimed at clustering large datasets. However, their implementation performs the *ComputeAssign* step on GPU while conducting the *Update* step on CPU, causing many data transfers between CPU and GPU at each iteration (similar to [22]). Other CPU and GPU implementations for k-means can be found in, e.g. References [109, 106].

1.6 . Objectives

Motivated by k-means-based approximate spectral clustering [203] and emerging heterogeneous computing [197], we propose a completely parallel processing chain for large-scale approximate spectral clustering on CPU-GPU heterogeneous architectures, as shown in Figure 1.4. The purpose of this dissertation is the efficient parallelization of this complete chain.

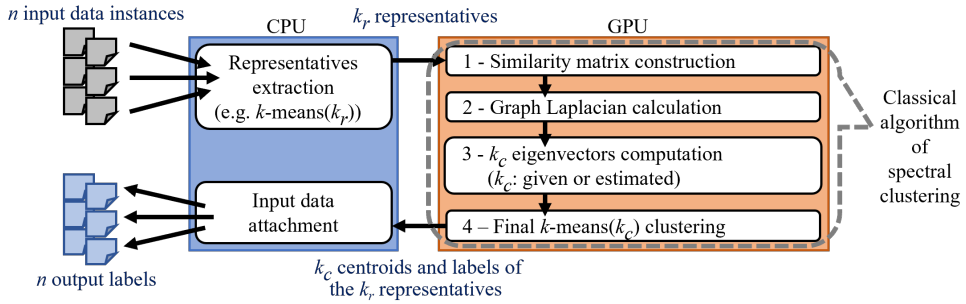


Figure 1.4: Data flow of a CPU-GPU parallel processing chain for large-scale approximate spectral clustering

- The first step of the data flow (upper left part) allows to reduce significantly the volume of data for subsequent intensive computations, extracting k_r representatives from n data instances. Each instance is then attached to its nearest representative. The k-means algorithm appears an interesting method to achieve this goal, with limited impact on final clustering quality

[203]. However, determining an appropriate k_r on unknown data requires some experiments.

- Then the k_r representatives are transferred from CPU to GPU and the spectral clustering algorithm is performed on GPU on these representatives to find the k_c clusters (right part). Typically, we have $k_c \ll k_r \ll n$. The eigenvector computations can be performed using existing GPU libraries, e.g. cuSOLVER [143], nvGRAPH [139].
- The clustering result for the k_r representatives is then sent back to CPU, and finally we set the cluster labels of n data instances according to the attachment relationships in the first step (bottom left part).

Note that the input dataset may require more memory space than the GPU RAM and the extraction of representatives consumes even more memory, this should be done on the CPU rather than the GPU. Moreover, as shown in Step 3 in Figure 1.4, adopting representatives approach does not prevent the use of heuristic methods for k_c estimation (e.g. based on eigenvalue or eigenvector analysis [188, 198, 210]).

2 - Parallel and Accurate k-means Clustering

2.1 . Introduction

In this chapter, we focus on designing two optimized parallel implementations of the k-means algorithm on CPU and GPU, respectively. They can either be used independently for large-scale k-means clustering, or they can serve as two important steps of our CPU-GPU processing chain for large-scale approximate spectral clustering (see Figure 1.4). Specifically, in the second scenario, the CPU implementation can be used for the preprocessing step which extracts representatives while the GPU implementation can be used for the last step of the classical spectral clustering algorithm.

As described in Section 1.2.1, the k-means algorithm (Algorithm 1) adopts an iterative strategy, and each iteration consists of the *ComputeAssign* step and the *Update* step. The *ComputeAssign* step exhibits a natural parallelism, leading to a relatively straightforward parallel implementation both on CPU and GPU. However, the *Update* step appears more difficult to be efficiently parallelized and is a source of rounding error accumulation due to reduction operations. This accumulation of rounding errors is trivial when processing small datasets or using double precision arithmetic for floating-point numbers, however it can become nontrivial and spoil the clustering accuracy when processing large datasets using single precision arithmetic. To our knowledge, there is no specific study on this numerical accuracy issue in existing related works on parallel k-means (described in Section 1.5.2). We will particularly address this issue in Section 2.2 and design optimized parallel k-means implementations in Sections 2.3 and 2.4. Finally, we will evaluate the numerical accuracy and performance of our implementations through experimental campaigns on synthetic and real-world large datasets in Section 2.5.

The work presented in this chapter has been first published as a workshop paper of Euro-Par 2020 [76] (initial version) and then published as a journal article in CCPE [75] (extended version). Our CPU code and GPU code for parallel k-means implementation are available at https://gitlab-research.centralesupelec.fr/Stephane.Vialle/cpu-gpu-kmeans.

2.2 . Numerical accuracy issue

In the *Update* step of the k-means algorithm, we need to calculate the sum of data instances in each cluster and then divide the sum by the number of instances in the cluster. For both CPU and GPU implementations of the *Update* step, we encountered the accumulation of rounding errors when a large number of instances are added together one by one naively in single precision (*32-bit arithmetic*). Essentially, rounding errors (a.k.a. round-off errors) are caused by the finite

representation capacity of floating-point numbers and are particularly significant when adding two numbers of different magnitudes (see [94] for more explanation). The accumulation/effect of rounding errors in the *Update* step led to an issue of numerical accuracy and finally spoiled the clustering quality. On the other hand, using double precision (*64-bit arithmetic*) can significantly reduce the accumulation of rounding errors and reach a satisfying level of numerical accuracy because double precision has a higher representation capacity of floating-point numbers. However, the computational cost using double precision is typically higher than using single precision [14].

We intend to preserve the performance of single precision computations while minimizing the effect of rounding errors. Some special summation methods exist, but require many extra calculations, e.g. summation methods requiring data sorting, or Kahan's compensated summation with extra additions (see [82], Chapter 4). We designed a simple and effective two-level summation/reduction method for the *Update* step, as shown in Figure 2.1. The idea is to split data instances into a certain number of packages of similar size, first calculate the sum per cluster within each package (1st level summation), and then compute the sum per cluster of all packages (2nd level summation). By choosing a sufficient number of packages, we can avoid adding floating-point numbers of significantly different magnitudes and achieve a satisfactory numerical accuracy.

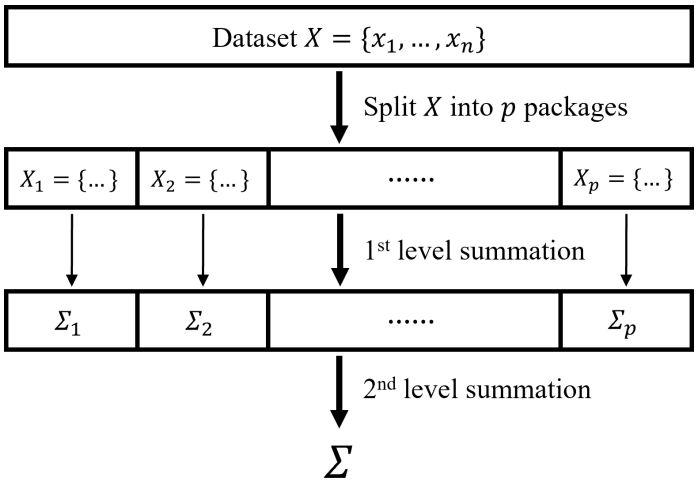


Figure 2.1: Two-level summation method for the *Update* step

In fact, our experiments in Section 2.5.2 show that satisfying numerical accuracy can be achieved in case of using up to 5×10^5 instances per package on the Syn4D-50M dataset ($n = 5 \times 10^7$, 100 packages). Hence, our two-level update scheme can theoretically scale up to at least 5×10^5 packages with 5×10^5 instances per package while having the guarantee of numerical accuracy, which leads to a huge dataset with 250 billion instances ($n = 2.5 \times 10^{11}$) requiring at least $2.5 \times 10^{11} \times 4 \div 10^{12} = 1$ TB memory (far beyond both our GPU RAM

and CPU RAM). Therefore, the two-level update scheme is sufficient for our need, while using more summation levels would not improve accuracy further but would complicate parallel implementation.

Although our two-level summation method looks simple, parallelizing it efficiently on multi-core CPU and GPU requires efforts and special care. Fortunately, the implementation of the 2nd level summation can rely on OpenMP *reduction* mechanism on multicore CPU, and can rely on CUDA `atomicAdd` operations on GPU (which have become faster on modern GPUs). Unfortunately, these efficient implementations do not allow to control the reduction scheme (e.g. to ensure the pairwise summation [82]).

2.3 . Parallel and accurate k-means on the CPU

We parallelize the k-means algorithm on CPU using OpenMP multithreading and auto-vectorization. The random selection of initial centroids is implemented with the `rand_r` function (which is a thread-safe version of the `rand` function). The parallelization of the *ComputeAssign* step and the *Update* step is described in the following two sections.

2.3.1 . Parallelization of the *ComputeAssign* step

Figure 2.2 shows our multithreading approach for the *ComputeAssign* step. We parallelize distance computations among multiple threads (say n_t threads). In other words, each thread calculates the distances between $\frac{n}{n_t}$ consecutive instances and k_c centroids.

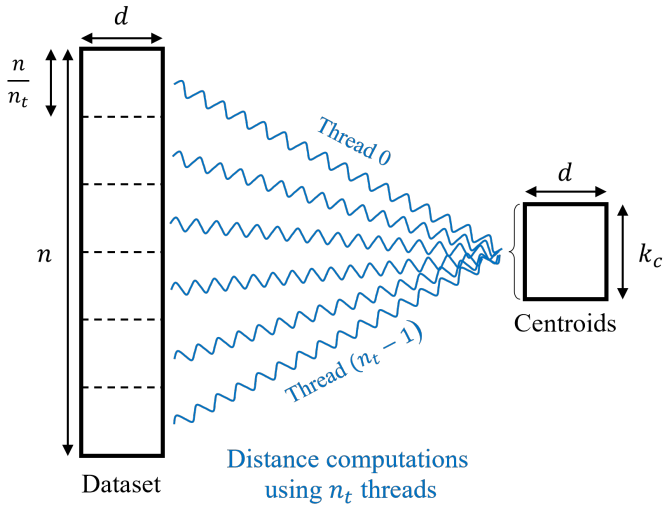


Figure 2.2: Multithreading for the *ComputeAssign* step

Listing 2.1 presents our CPU code for the *ComputeAssign* step. We use the `#pragma omp for` directive of OpenMP to parallelize distance computations

(lines 3-12). Furthermore, with the `-Ofast`[1] and `-march=skylake-avx512` compilation flags[2] and the number of dimensions defined as a constant, we optimize GCC code generation for our dual-skylake CPU, enable AVX units usage and auto-vectorization mechanisms to vectorize the distance calculation of each instance-centroid pair across all dimensions (lines 11-12). In practice we only need to calculate the square of the Euclidean distance instead of the distance itself. Then the nearest centroid for each instance can be found and recorded (lines 14-16). The cluster label of each instance is updated according to its nearest centroid, and the changes of labels are counted into the private variable `track` of each thread (lines 19-22). Finally, the `reduction` directive of OpenMP sums the private `track` of all threads (line 3). Note that we avoid storing and accessing an $n \times n$ distance matrix by integrating distance computation and instance assignment in the *ComputeAssign* step.

```
1  #define d ... // Nb of dimensions is a constant
2  #pragma omp parallel {
3    #pragma omp for reduction(+: track)
4    for (int i = 0; i < n; i++) {
5      int min = 0;
6      T_real sqDist, minSqDist = FLT_MAX; // T_real: float or double
7      for (int k = 0; k < kc; k++) {
8        sqDist = 0.0f;
9        // Calculate the squared distance between instance i and
10       // centroid k across d dimensions
11       for (int j = 0; j < d; j ++)
12         sqDist += (data[i*d+j]-cent[k][j])*(data[i*d+j]-cent[k][j]);
13       // Find the nearest centroid to instance i
14       bool a = (sqDist < minSqDist);
15       min = (a ? k : min);
16       minSqDist = (a ? sqDist : minSqDist);
17     }
18     // Change the label if necessary and count this change into track
19     if (labels[i] != min) {
20       labels[i] = min;
21       track++;
22     }
23   }
24 }
```

Listing 2.1: CPU implementation for the *ComputeAssign* step

2.3.2 . Parallelization of the *Update* step

Recall that we use a two-level summation method in the *Update* step (see Section 2.2). Figure 2.3 presents our multithreading approach for the summation process. Consider splitting n instances into p packages of similar size, we parallelize the processing of p packages among n_t threads, i.e. each thread processes $\frac{p}{n_t}$ packages. The local summation results are first computed within each package

[1]The `-Ofast` flag of GCC introduces strong optimizations in floating-point computations (like the `-ffast-math` flag) but they are supported by our code.

[2]Or we use the `-march=native` flag when compiling the code on the target machine.

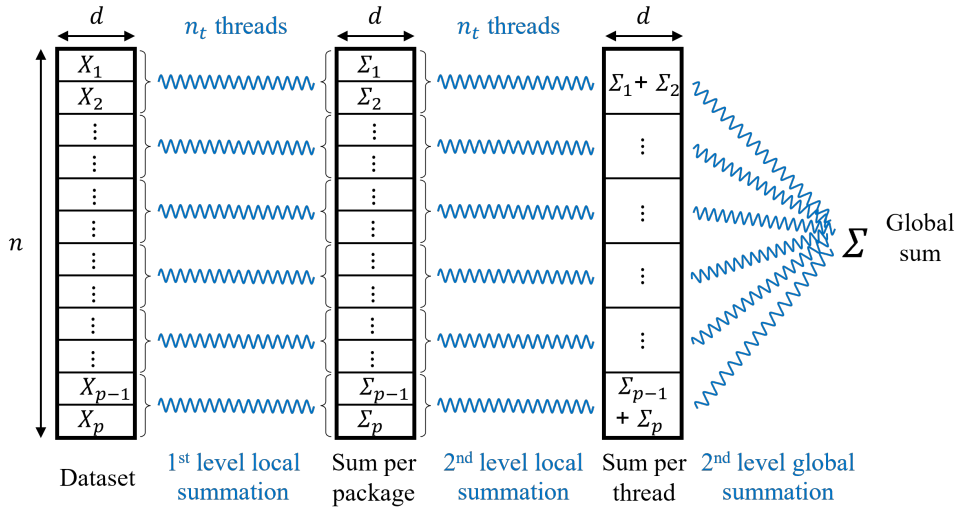


Figure 2.3: Multithreading for the two-level summation in the *Update* step

(1st level), then the global summation results are computed with the local results of all packages (2nd level).

Listing 2.2 displays our CPU code for the *Update* step. The starting index (i.e. offset) and length of each package is first computed (lines 9-10, 15-16). Then we use the `#pragma omp for` directive to parallelize the processing of packages among multiple threads (lines 12-13). For each package of instances, a thread counts the number of instances assigned to each cluster into the thread private `count[]` array, and accumulates the values of instances related to each cluster into the thread private `pkg[]` array (lines 14, 17-22). This is the 1st level summation. Then a thread accumulates the `pkg[]` results of the $\frac{p}{n_t}$ packages (that it is responsible for) into the thread private `cent[]` array (lines 23-25). This is the 2nd level local summation. Finally, the `reduction` directive of OpenMP sums the thread private `count[]` and `cent[]` results of all threads into the global `count[]` and `cent[]` array, respectively (lines 12, 26-27). This is the 2nd level global summation. With the global sum of each cluster, we then calculate the new centroids by averaging, which is parallelized using the `#pragma omp for` directive (lines 30-33). Note that the inner loops (lines 20-21, 24-25, 32-33) are compliant with the main requirements of auto-vectorization, i.e. accessing contiguous array indices and avoiding divergences, engaged with `-O3` or `-Ofast` compilation flag (cooperating with `-march=skylake-avx512` or `-march=native` flag on our dual-skylake CPU).

```
#define d ...        // Nb of dimensions
#define p ...        // Nb of packages
int count[kc];       // Storing the nb of instances in each cluster
T_real pkg[kc][d];   // Storing the (local) sum per cluster in a package
T_real cent[kc][d]; // Storing centroids

#pragma omp parallel {
  ...  // Declare variables, reset count[] and cent[] to zeros
  q = n / p;  // Quotient
  r = n % p;  // Remainder
  // Sum the contributions to each cluster
  #pragma omp for private(pkg) reduction(+: count, cent)
  for (int a = 0; a < p; a++) {  // Process by package
    ...                                  // Reset pkg[] to zeros
    ofs = (a < r ? ((q + 1) * a) : (q * a + r));  // Offset
    len = (a < r ? (q + 1) : q);                  // Length
    for (int i = ofs; i < ofs + len; i++){// 1st level local summation
      int k = labels[i];                  // - Count nb of instances in
      count[k]++;                         //   OpenMP reduction array
      for (int j = 0; j < d; j++)         // - Reduction in thread
        pkg[k][j] += data[i*d + j];       //   private array
    }
    for (int k = 0; k < kc; k++)          // 2nd level local summation
      for (int j = 0; j < d; j++)         // - Reduction in local
        cent[k][j] += pkg[k][j];         //   OpenMP reduction array
  }  // 2nd level global summation
     // - Final reduction by OpenMP in global cent[] array

  // Final averaging to get new centroids
  #pragma omp for
  for (int k = 0; k < kc; k++)    // Process by cluster
    for (int j = 0; j < d; j++)
      cent[k][j] /= count[k];     // - Update global cent[] array
}
```

Listing 2.2: CPU implementation for the *Update* step

2.4 . Parallel and accurate k-means on the GPU

2.4.1 . Global approach

We parallelize the k-means algorithm on GPU using CUDA. Specifically, the data instances to be clustered are first transferred from CPU to GPU, then a series of CUDA kernels and library functions are launched from CPU to perform k-means clustering on GPU, finally the cluster labels are transferred to CPU.

Data transfers between CPU and GPU are minimized. They mainly occur at the beginning and end of our program. In order to check the stopping criterion, we also need to transfer the quantity of instances that change labels (i.e. *track* in listings) from GPU to CPU at each iteration, but the price of this single integer transfer is negligible. Moreover, we use pinned memory for faster transfers.

For coalesced access to GPU global memory, we use the $d \times n$ transposed data matrix (SoA, i.e. Structure of Array) instead of the $n \times d$ data matrix (AoS, i.e. Array of Structure), as shown in Figure 2.4. Note that when the $n \times d$ matrix of data instances is loaded into CPU RAM, it can be directly stored into a

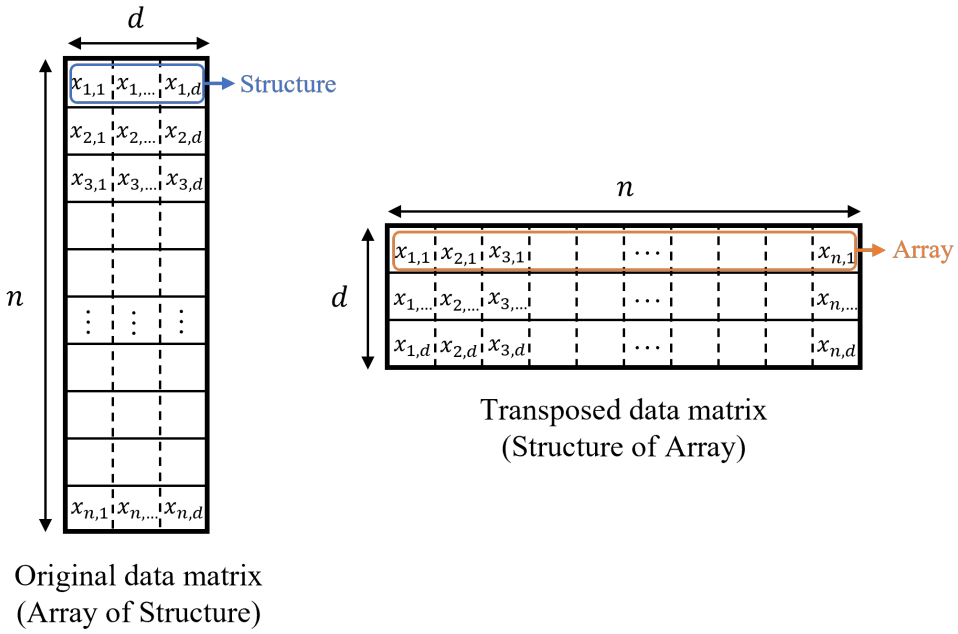


Figure 2.4: Array of Structure (AoS) vs. Structure of Array (SoA)

transposed $d \times n$ matrix and then transferred to GPU RAM. Besides, we use the centroid matrix in the *ComputeAssign* step, but the transposed centroid matrix in the *Update* step. Thus we need to transpose the transposed centroid matrix to get the centroid matrix on GPU at each iteration, but the overhead is trivial since the two matrices are typically small and the efficient *geam* function of cuBLAS library is employed to perform the transposition.

The random selection of initial centroids is implemented on GPU using the cuRand library [142]. The parallelization details of the *ComputeAssign* and *Update* steps are described in the following two sections.

2.4.2 . Parallelization of the *ComputeAssign* step

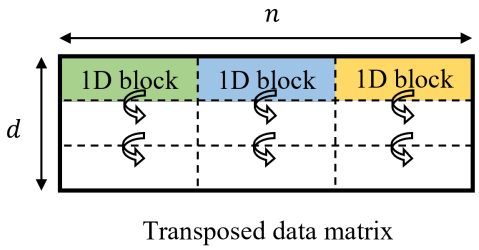


Figure 2.5: Grid and block configuration for the `ComputeAssign` kernel

As presented in Figure 2.5, we create a 1D grid containing 1D blocks of threads. The transposed matrix of data instances is used for the coalescence of memory access. Each block accesses some instances by going through d dimensions, and

computes the distances between these instances and k_c centroids.

```
#define BSXN ...  // Block size in x axis related to n
__global__ void ComputeAssign (T_real *GPU_dataT, T_real *GPU_cent,
                               int *GPU_labels,
                               unsigned long long int *AdrGPU_track)
{
  int idx = blockIdx.x * BSXN + threadIdx.x;
  __shared__ unsigned long long int shTrack[BSXN];
  shTrack[threadIdx.x] = 0;

  if (idx < n) {
    int min = 0;
    T_real diff, sqDist, minSqDist;
    for (int k = 0; k < kc; k++) {
      sqDist = 0.0f;
      // Calculate the squared distance between instance idx and
      // centroid k across d dimensions
      for (int j = 0; j < d; j++) {
        diff = GPU_dataT[j*n + idx] - GPU_cent[k*d + j];
        sqDist += (diff*diff);
      }
      // Find the nearest centroid to instance idx
      if (sqDist < minSqDist || k == 0) {
        minSqDist = sqDist;
        min = k;
      }
    }
    // Change the label if necessary
    if (GPU_labels[idx] != min) {
      shTrack[threadIdx.x] = 1;
      GPU_label[idx] = min;
    }
  }

  // Count the changes of label into GPU_track: two-part reduction
  // 1 - Parallel reduction of shared array shTrack[*] into shTrack[0]
  if (BSXN > 512) {
    __syncthreads();
    if(threadIdx.x<512) shTrack[threadIdx.x]+=shTrack[threadIdx.x+512];
    else  return;  // kill useless threads
  }
  ...  // # of remaining threads per block: 512 --> 256 --> 128 --> 64
  if (BSXN > 32) {
    __syncthreads();
    if(threadIdx.x<32) shTrack[threadIdx.x]+=shTrack[threadIdx.x+32];
    else  return;  // kill useless threads
  }
  if (BSXN > 16) {
    __syncwarp();  // avoid races between threads within the same warp
    if(threadIdx.x<16) shTrack[threadIdx.x]+=shTrack[threadIdx.x+16];
    else  return;  // kill useless threads
  }
  ...  // # of remaining threads per block: 16 --> 8 --> 4 --> 2
  if (BSXN > 1) {
    __syncwarp();  // avoid races between threads within the same warp
    if(threadIdx.x<1) shTrack[threadIdx.x]+=shTrack[threadIdx.x+1];
    else  return;  // kill useless threads
  }               // only thread 0 survives
  // 2 - Final reduction into a global array
  if (shTrack[0] > 0)  atomicAdd(AdrGPU_track, shTrack[0]);
}
```

Listing 2.3: `ComputeAssign` kernel for the *ComputeAssign* step

Listing 2.3 shows our GPU code for the *ComputeAssign* step. Each block has `BSXN` threads and computes the distances between `BSXN` instances and k_c centroids (lines 11-19). Again, we compute practically the square of distance. Then the nearest centroid for each instance can be found and recorded (lines 21-24). The cluster label will be changed if necessary, and the change will be marked with 1 in the shared 1D block array `shTrack[]` (lines 27-30). Finally, we count the changes of labels by a two-part reduction. The first part reduction (lines 36-57) sums rapidly the values of `shTrack[]` into the first element `shTrack[0]` in shared memory, then the second part reduction (line 59) accumulates the sum into the global variable `GPU_track` by only one `atomicAdd` operation.

Our reduction is based on the classical method recommended by NVIDIA[3], but we kill the useless threads step by step by `return` instructions so that only the first thread survives at the end, which reduces working warps in the first part reduction and eliminates the check of thread index at the end of the second part reduction. Moreover, our nonzero check of the sum avoids many unnecessary `atomicAdd` operations when no change of label occurs in a block, especially in the last iterations of k-means.

2.4.3 . Parallelization of the *Update* step

Based on our two-level summation method (see Section 2.2), we implement the *Update* step on GPU by two substeps: the first substep *Update_S1* computes the sum of instances related to each cluster within each package and the number of instances in each cluster across all packages, then the second substep *Update_S2* computes new centroids. Our GPU implementation for the *Update* step exploits dynamic parallelism (i.e. CUDA threads launching child grids), multiple streams and shared memory to optimize performance (see CUDA C++ Programming Guide [141]). As illustrated in Figure 2.6, child grids launched on different streams run concurrently. In our case, this allows to maximize the utilization of GPU hardware resources independently of the number and size of packages. Thus, the number of packages is determined only based on the effect of rounding errors.

A GPU implementation usually consists of host code and device code. By using dynamic parallelism, the host code is simplified to two parent kernel launches, as shown in Listing 2.4. Specifically, we launch the first parent kernel `Update_S1_Parent` with a grid of n_{s1} threads to complete the first substep *Update_S1*, where each thread creates its own working stream (i.e. n_{s1} streams in total) and launches its child grids on the stream. Similarly, we launch the second parent kernel `Update_S2_Parent` with a grid of n_{s2} threads to complete the second substep *Update_S2*, where each thread creates its own working stream (i.e. n_{s2} streams in total) and launches its child grids on the stream.

[3]Mark Harris, NVIDIA Developer Technology. Optimizing Parallel Reduction in CUDA. Source: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

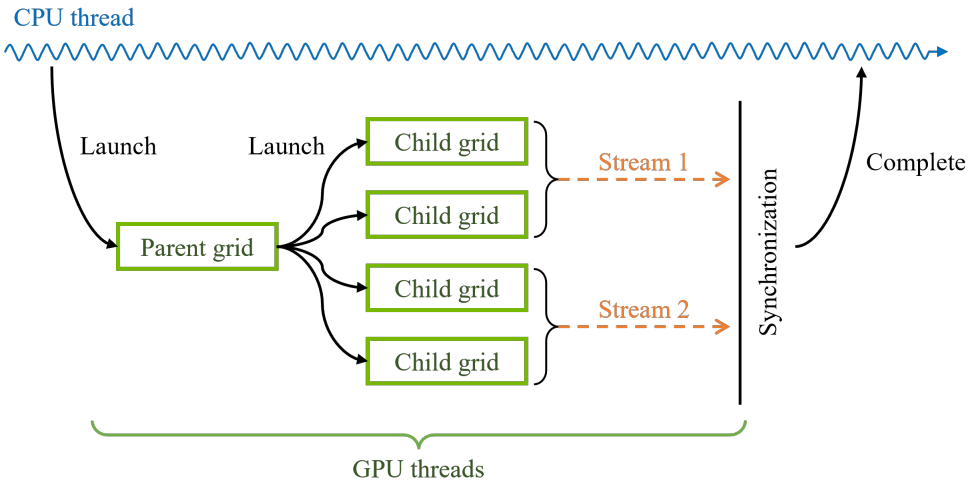


Figure 2.6: Combined use of dynamic parallelism and multiple streams

```
cudaMemset(...);  // Reset GPU_count, GPU_pkg to zeros
// ns1: nb of threads (or streams) used in the Update_S1_Parent kernel
// ns2: nb of threads (or streams) used in the Update_S2_Parent kernel
Update_S1_Parent<<<1,ns1>>>(GPU_labels, GPU_pkg, GPU_dataT, GPU_count);
Update_S2_Parent<<<1,ns2>>>(GPU_pkg, GPU_centT, GPU_count);
```

Listing 2.4: Host code of GPU implementation for the *Update* step

```
#define p ...  // Nb of packages
// Parent kernel of Update_S1
__global__ void Update_S1_Parent (...) {
  int tid = threadIdx.x;
  if (tid < p) {
    ...                          // Declare variables and stream
    cudaStreamCreateWithFlags(&s, cudaStreamDefault);
    q = n / p;                   // Quotient
    r = n % p;                   // Remainder
    np = (p - 1) / nS1 + 1;      // Nb of packages for each stream
    Db.x = BSXP;                 // BSXP: Block X-size for package
    Db.y = BSYD;                 // BSYD: Block Y-size for dim
    Dg.y = (d - 1) / BSYD + 1;
    for (int i = 0; i < np; i++) {
      pid = i * nS1 + tid;       // Package ID
      if (pid < p) {
        ofs = (pid < r ? ((q + 1) * pid) : (q * pid + r));  // Offset
        len = (pid < r ? (q + 1) : q);                      // Length
        Dg.x = (len - 1) / BSXP + 1;
        // Launch a child kernel on a stream to process a package
        Update_S1_Child<<<Dg,Db,0,s>>>(pid, ofs, len, GPU_labels,
                                       GPU_pkg, GPU_dataT, GPU_count);
      }
    }
    cudaStreamDestroy(s);
  }
}
```

Listing 2.5: `Update_S1_Parent` kernel for the *Update* step

Listing 2.5 exhibits the device code of the `Update_S1_Parent` kernel. Each thread creates its own stream (created on line 7), and processes several packages by launching a 2D child grid for each package (lines 10-23). Each 2D child grid is composed of 2D blocks of threads (as shown in Figure 2.7), resulting in about $\frac{n}{p} \times d$ working threads where $\frac{n}{p}$ is about the number of instances per package. The `cudaStreamDestroy` (line 25) ensures that this stream will not be reused to launch other threads, while the parent thread will not end until all its child threads have finished. As expected, the combined use of dynamic parallelism and multiple streams proved to be efficient in our case (see Figure 2.10 in Section 2.5.2).

```
1  // Child kernel of Update_S1
2  __global__ void Update_S1_Child (int pid, int ofs, int len,
3                                   int *GPU_labels, T_real *GPU_pkg,
4                                   T_real *GPU_dataT, int *GPU_count)
5  {
6    __shared__ T_real shTabV[BSYD][BSXP]; // Table of instance values
7    __shared__ int shTabL[BSXP];          // Table of labels (cluster Id)
8    // Index initialization
9    int baseRow = blockIdx.y * BSYD;      // Base row of the block
10   int row = baseRow + threadIdx.y;      // Row of child thread
11   int baseCol = blockIdx.x * BSXP + ofs;// Base column of the block
12   int col = baseCol + threadIdx.x;      // Column of child thread
13   int cltIdx = threadIdx.y * BSXP + threadIdx.x;  // 1D cluster index
14   // Load the values and cluster labels of instances into sh mem tables
15   if (col < (ofs + len) && row < d) {
16     shTabV[threadIdx.y][threadIdx.x] = GPU_dataT[row*n + col];
17     if (threadIdx.y == 0) shTabL[threadIdx.x] = GPU_labelss[col];
18   }
19   __syncthreads();  // Wait for all data loaded into the sh mem
20
21   // Compute partial evolution of centroid related to 'cltIdx'
22   if (cltIdx < kc) {
23     #define BlND (d<BSYD ? d:BSYD) // BlND: nb of dims stored by block
24     T_real Sv[BlND];               // Sum of values in BlND dimensions
25     for (int j = 0; j < BlND; j++)  Sv[j] = 0.0f;  // Init Sv to zeros
26     int count = 0;                 // Init the counter of instances
27     // - Accumulate contributions to cluster number 'cltIdx'
28     for (int x = 0; x < BSXP && (baseCol + x) < (ofs + len); x++) {
29       if (shTabL[x] == cltIdx) {
30         count++;
31         for (int y = 0; y < BSYD && (baseRow + y) < d; y++)
32           Sv[y] += shTabV[y][x];
33       }
34     }
35     // - Save the contrib. of block into global contrib. of the package
36     if (count != 0) {
37       if (blockIdx.y == 0) atomicAdd(&GPU_count[cltIdx], count);
38       int BlND_max = (blockIdx.y == d/BSYD ? d%BSYD : BSYD);
39       // BlND_max: nb of dims managed by a block
40       for (int j = 0; j < BlND_max; j++)
41         atomicAdd(&GPU_pkg[(baseRow+j)*kc*p + kc*pid + cltIdx], Sv[j]);
42     }
43   }
44 }
```

Listing 2.6: `Update_S1_Child` kernel for the *Update* step

Listing 2.6 exhibits the device code of the `Update_S1_Child` kernel. Each

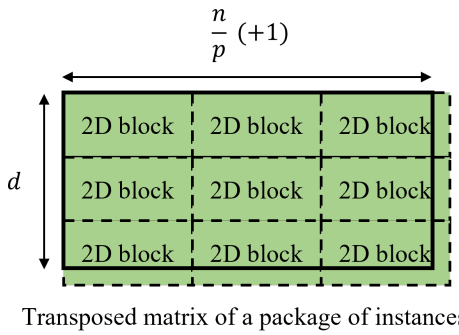


Figure 2.7: Grid and block configuration for `Update_S1_Child` kernel

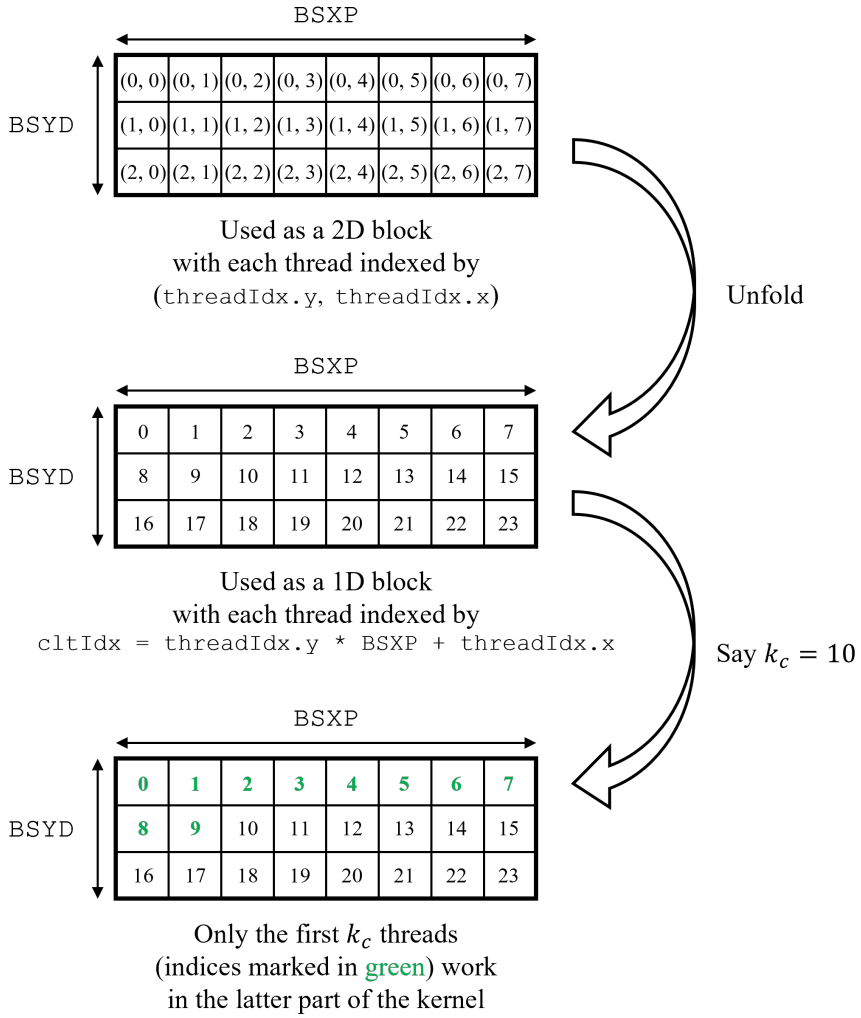


Figure 2.8: Use of each block in the `Update_S1_Child` kernel

2D block first loads some dimension values and cluster labels of some instances into fast shared memory (lines 6-18). Then the first k_c threads in each 2D block

are used as a 1D block (as illustrated in Figure 2.8) to perform local reductions in shared memory (lines 22-43). Specifically, they sum the dimension values of instances per cluster (line 32), and perform `atomicAdd` operations only one time per cluster (lines 37-41) instead of one time per loaded instance. Therefore, the number of expensive `atomicAdd` operations in global memory is significantly reduced. Nevertheless, this kernel has several limitations. A number of expensive `atomicAdd` operations in global memory are still performed to avoid conflicts between blocks. Some losses of coalescence occur when each thread accesses its own array in local memory (lines 25 and 32). Only k_c threads per block work after line 22. The number of clusters that can be processed is currently limited to 1024 which is the maximum size of a block (lines 13 and 22).

The second substep *Update_S2* is implemented using a strategy similar to that used for *Update_S1*. Each thread of the parent grid processes several packages, and creates child grids on its own stream. Each child grid is in charge of updating the $k_c \times d$ centroid values with the contribution of its package. So, it contains $k_c \times d$ working threads, each one executing only few operations and one `atomicAdd` operation (shared memory is not used in *Update_S2*). Again, using dynamic parallelism and multiple streams has allowed to speed up the execution.

2.5 . Experimental results

Experimentally, we evaluate our parallel k-means implementations on one synthetic and two real-world datasets, and we compare the performance of our code with some existing parallel k-means implementations.

2.5.1 . Testbed and compilation settings

The testbed is our *john3* server consisting of two Intel Xeon Silver 4114 processors as CPU and a NVIDIA GeForce RTX 2080 Ti as GPU. More information about the testbed is provided in Appendix B. The CPU code is compiled with gcc (with `-fopenmp`, `-Ofast`, `-march=skylake-avx512`, `-funroll-loops` flags) to have thread parallelization using OpenMP, auto-vectorization using AVX-512 instructions and various optimizations. The GPU code is compiled with nvcc in CUDA. Particularly, to use dynamic parallelism in CUDA, we need to adopt the *separate compilation mode*: generating and embedding relocatable device code into the host object, before calling the device linker.

2.5.2 . Experiments on a synthetic dataset

We first experiment on a synthetic 4D dataset called **Syn4D-50M** which contains 50 million instances uniformly distributed in 4 convex clusters (12.5 million instances in each cluster). Each cluster has a radius of 9 and the centroids are supposed to be (40, 40, 60, 60), (40, 60, 60, 40), (60, 40, 40, 60) and (60, 60, 40, 40), respectively. However, due to the intrinsic errors of generating pseudo-random numbers and the rounding errors of floating-point numbers, it appears the calcu-

lated centroids could have a deviation of order 10^{-4} from the ideal ones. Note that the dataset is created in the way that the k-means algorithm would not be sensitive to the initialization of centroids and would not be trapped in local optima. More information about the dataset is available in Appendix A.

Generally, we select k_c initial centroids uniformly at random from n data instances with `rand_r` function on CPU and cuRAND library on GPU. This one-time random selection step usually takes little time. Since the number of iterations can vary with the selected initial centroids, we are more interested in the elapsed time per iteration than the overall time. However, for the sake of comparison, we intend to achieve the same number of iterations on CPU and GPU by setting the same initial centroids. We execute the algorithm until all cluster labels of data instances remain unchanged (tolerance = 0, see Section 1.2.1). The most important results in our tables are highlighted in boldface.

Numerical accuracy & performance on CPU

In Table 2.1, we evaluate the k-means clustering on CPU in terms of *numerical error* and average time per iteration by varying the number of threads, the arithmetic precision and the number of packages. The numerical error is defined as the average absolute error of the final calculated centroids with respect to the ideal theoretical ones. It derives from the accumulation of rounding errors during summation of a large number of instance coordinates (see explanation in Section 2.2).

The column "Full iter." represents one k-means iteration mainly consisting of the *ComputeAssign* step and the *Update* step. We observe that using a certain number of packages in the *Update* step reduces the numerical error in single precision and consequently decreases the number of iterations from 7 to 5. In our case, using 100 packages is enough for achieving the same level of numerical accuracy as in double precision. Moreover, using single precision instead of double precision decreases the elapsed time. Parallelization with 20 CPU threads (distributed over 20 physical cores including AVX units) has been found to be the most efficient compared to other numbers of threads.

Numerical accuracy & performance on GPU

We give in Table 2.2 the accuracy and performance results of k-means clustering on GPU. The numerical error is decreased by our two-level summation method using multiple packages. The first level is performed within each package. It is implemented using a local reduction in the shared memory of each block of threads, and with a minimal number of `atomicAdd` operations in global memory. The second level sums the contributions of all packages using `atomicAdd` operations. Due to the expensive `atomicAdd` operations and other limitations (see Listing 2.6 and explanations in Section 2.4.3), the *Update* step appears the most time-consuming

Table 2.1: CPU k-means on the Syn4D-50M dataset with $(n, d, k_c) = (50M, 4, 4)$

Threads	Precision	Nb of packages	Numerical error	Init time (ms)	Average time per iteration (ms)			Nb of iters.	Overall time (ms)
					ComputeAssign	*Update*	Full iter.		
1 thread	double	1	0.000741	0.002	242.21	182.16	424.37	5	2121.85
	single	1	3.009794	0.003	153.22	149.36	302.58	7	2118.06
		10	0.244048	0.002	155.83	151.22	307.05	5	1535.25
		100	**0.000745**	0.002	150.34	151.33	**301.67**	5	1508.35
		1000	**0.000745**	0.003	154.52	154.59	**309.11**	5	1545.55
20 threads (20 physical cores)	double	1[a]	0.000741	0.083	51.23	192.37[a]	243.60	5	1218.08
	single	1[a]	3.009794	0.099	34.34	152.71[a]	187.05	7	1309.45
		10[b]	0.244048	0.091	34.24	24.43[b]	58.67	5	293.44
		100	**0.000745**	0.100	32.95	19.44	**52.39**	5	261.95
		1000	**0.000746**	0.126	32.80	19.43	**52.23**	5	261.28
40 threads (40 logical cores)	double	1[a]	0.000741	0.207	60.18	226.55[a]	286.72	5	1433.81
	single	1[a]	3.009794	0.174	39.50	165.86[a]	205.36	7	1437.69
		10[b]	0.244048	0.144	35.84	31.95[b]	67.79	5	339.09
		100	**0.000745**	0.155	35.31	27.62	**62.93**	5	314.81
		1000	**0.000747**	0.175	31.20	21.07	**52.28**	5	261.58

[a] 1 package $\longrightarrow$ 1 task during main computations $\longrightarrow$ only 1 working thread
[b] 10 packages $\longrightarrow$ 10 tasks during main computations $\longrightarrow$ only 10 working threads

Table 2.2: GPU k-means on the Syn4D-50M dataset with $(n, d, k_c) = (50M, 4, 4)$

Precision	Nb of packages	Numerical error	Overhead time (ms)		Init time (ms)	Average time per iteration (ms)			Nb of iters.	Overall time (ms)
			Transfer	Transpose		*ComputeAssign*	*Update*	Full iter.		
double	1	0.000741	81.13	0.14	2.65	8.98	34.49	43.47	5	301.27
single	1	0.000992	81.15	0.15	2.64	1.96	12.94	14.90	5	158.44
	10	0.000760	81.13	0.12	2.75	1.96	12.04	14.00	5	154.00
	100	**0.000739**	81.18	0.19	2.74	1.97	12.72	**14.69**	5	157.56
	1000	**0.000741**	81.11	0.29	2.65	1.98	13.47	**15.45**	5	161.30

step on GPU while the *ComputeAssign* step represents a small proportion of the running time.

In our GPU implementation, we experimentally optimize the configuration of grids and blocks of threads. Figure 2.9 shows an example of how the block size on x-axis (`BSXP` in listings) affects the performance of the *Update* step when the block size on y-axis (`BSYD`) is set to 4 (the number of dimensions of the Syn4D-50M dataset) in our 2D-blocks.

The random initialization of centroids and most of data transfers are performed only one time, hence their impact on the whole runtime decreases with the number of iterations. The elapsed time for regular transpositions of the centroid matrix appears negligible.

Figure 2.10 demonstrates the impact of GPU optimization on the running

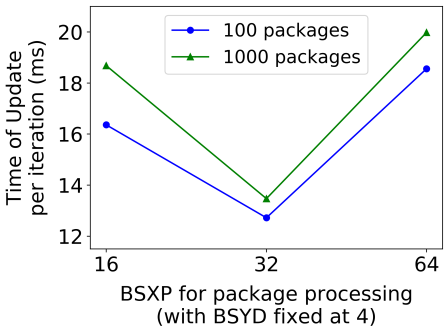


Figure 2.9: Impact of block size on the performance of the *Update* step with the Syn4D-50M dataset (using single precision)

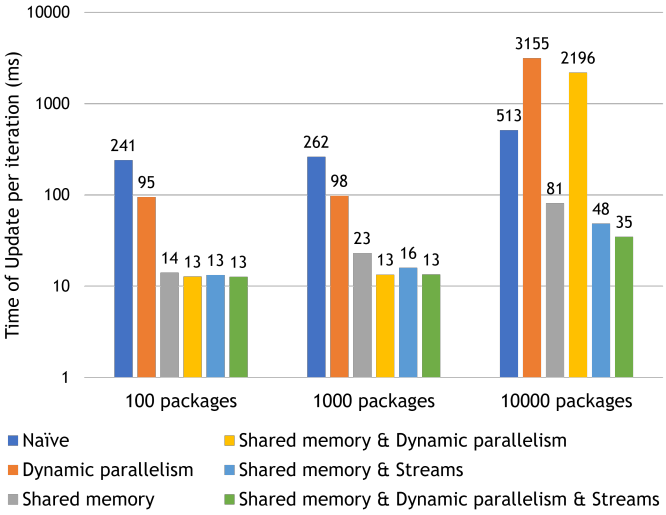


Figure 2.10: Impact of GPU code optimizations on the performance of the *Update* step with the Syn4D-50M dataset (using single precision)

time of the *Update* step. Compared to our naïve implementation with many `atomicAdd` operations, using shared memory reduces significantly the execution time for different number of packages. The dynamic parallelism also improves the performance in the case of 100 packages and 1000 packages but it degrades the performance for 10000 packages. This is because the GPU hardware resources are not fully concurrently exploited when there are a large number of small packages to be processed on the default stream. Therefore, introducing multiple streams could contribute to the concurrent use of hardware resources and consequently reduce the execution time, which is clearly demonstrated in the case of 10000 packages. We use 16 streams and 32 streams for the first and second substeps of the *Update* step, respectively, to minimize the execution time. The combined use of shared memory, dynamic parallelism and multiple streams always achieves optimal performance.

Performance comparison: GPU vs. CPU

Figure 2.11 displays the speedup of the two steps of k-means iterations and of the resulting full iterations. Here we consider 20 CPU threads instead of 40 threads since the former achieves the best performance on the Syn4D-50M dataset. The speedup of the k-means iteration is summarized as follows:

- Compared with the CPU mono-thread auto-vectorized implementation, the best speedup obtained on CPU is almost $\times 6$ running 20 threads with auto-vectorization.
- Compared with the CPU mono-thread auto-vectorized implementation, the best speedup obtained on GPU is about $\times 20$.
- Finally, our GPU implementation appears about $\times 3.5$ faster than our best parallel CPU implementation (running 20 threads with auto-vectorization).

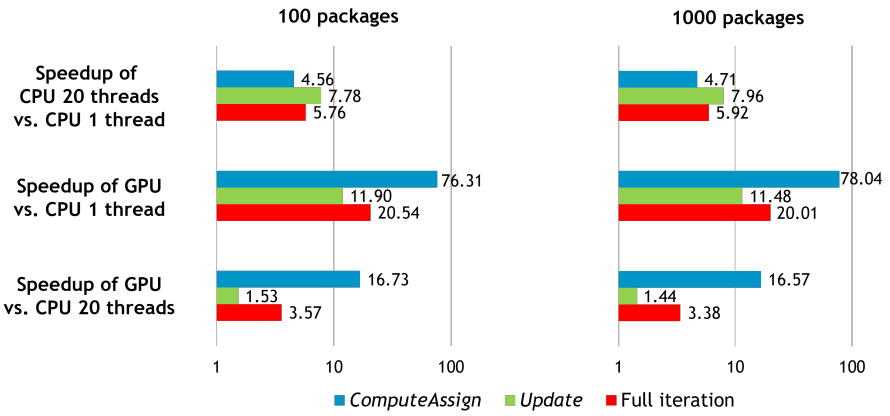


Figure 2.11: Speedup of k-means steps and iterations with the Syn4D-50M dataset (using $k_c = 4$, single precision)

In fact, the *ComputeAssign* step on GPU is over $\times 16$ faster than the best parallel CPU version, while the *Update* step on GPU is only about $\times 1.5$ faster. Thus, it seems that the *ComputeAssign* step is more suited to the GPU architecture than the *Update* step.

2.5.3 . Experiments on real-world datasets

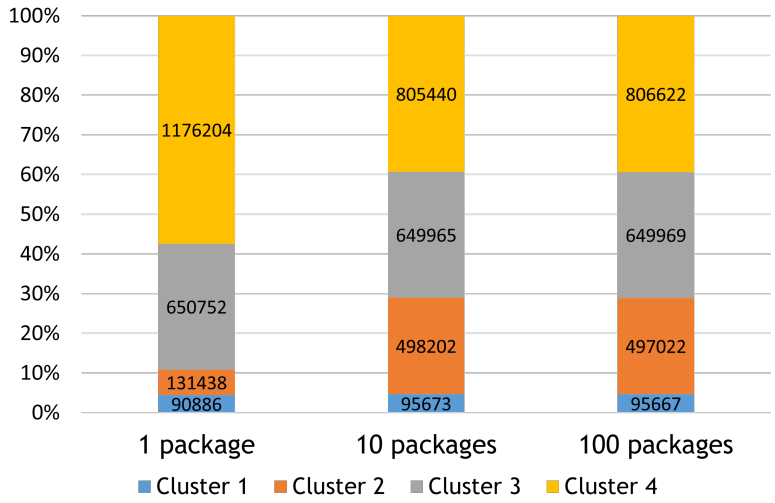
In the following we evaluate our parallel k-means implementation on two real-world datasets:

- Household power consumption dataset (HPO for short): $n = 2\,049\,280$, $d = 7$
- US census 1990 dataset (USC for short): $n = 2\,458\,285$, $d = 68$

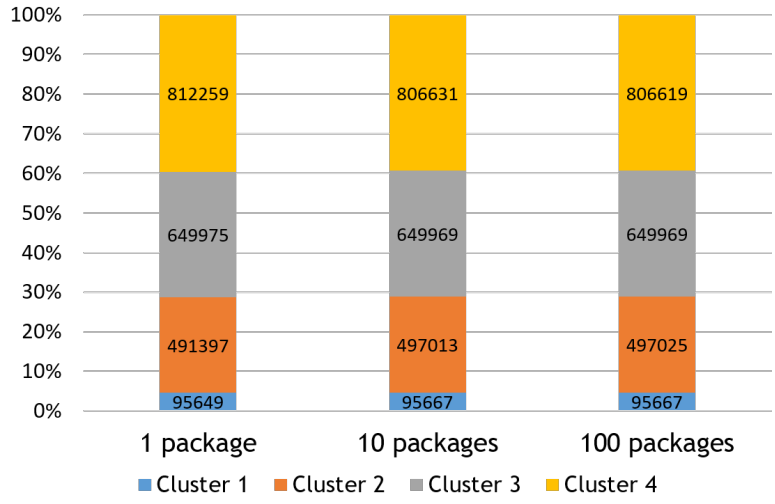
They are described in more detail in Appendix A. As their ground truth clusterings are unavailable, we impose k_c to specific values in the subsequent evaluation.

Numerical accuracy

To reveal the effect of rounding errors and the improvement of numerical accuracy with the use of packages in the *Update* step, we observe the changes of the number of instances assigned to each cluster.



(a) On CPU using 20 threads



(b) On GPU

Figure 2.12: Changes in cluster size with the use of packages in the *Update* step on the HPO dataset (using $k_c = 4$, single precision)

Household power consumption (HPO) dataset: Figure 2.12 displays the changes in cluster size with the number of packages on the HPO dataset by imposing $k_c = 4$. The following results emerge:

- On CPU, the distribution of instances in each cluster is evidently different between the use of 1 package and multiple packages. Note that the use of 1 package means in fact no use of package, or the entire dataset is regarded as 1 package. Hence, we infer that the effect of rounding errors arises in the *Update* step when calculating the sum of instances directly without the use of multiple packages, and this negative effect impairs significantly the clustering accuracy.
- On GPU, the effect of rounding errors with 1 packages appears less evident owing to the local reductions using shared memory in our implementation. Nevertheless, it can be seen from the specific numbers in the chart that, using multiple packages still procures some improvement in clustering accuracy.
- With 100 packages, the sizes of the 4 clusters are almost identical on CPU and GPU.

US census 1990 (USC) dataset: Similarly we checked the number of instances assigned to each cluster on the USC dataset. In this case we observed few differences when using one package and multiple packages for all values of k_c. We reckon it is because the values in the USC are all integers. Thus there is little accumulation of rounding errors in the *Update* step even with only 1 package. Despite this fact, this dataset is suitable for evaluating the computational performance of our k-means implementation.

Performance of each step

Tables 2.3 and 2.4 present the performance of our k-means implementation using 100 packages on CPU and GPU respectively, for the above two datasets. We set the tolerance $= 10^{-4}$ as the stopping criterion of k-means iterations. For each benchmark, we set the same initial centroids for k-means on CPU and on GPU, thus reasonably resulting in an identical number of iterations. We observe that:

- The execution time of the *ComputeAssign* step is always more significant than the time of the *Update* step on CPU, but not on GPU.
- For k-means on CPU, parallelization running 20 threads (on 20 physical cores including AVX units) was found to be the most efficient for the HPO dataset, while running 40 threads (on 40 logical cores including AVX units) achieves the best performance for the USC dataset.

- The *ComputeAssign* step on GPU is always considerably faster than on CPU, while the *Update* step on GPU can be either faster or slower than on CPU (using multithreading) depending on the test cases.

  As there are tens of iterations, the elapsed time of data transfers between CPU and GPU is insignificant compared to the whole runtime of k-means on GPU.

Finally, note that the elapsed time for selecting initial centroids randomly is negligible and not shown in the tables.

Table 2.3: CPU k-means on two real-world datasets (using single precision, 100 packages)

Dataset	k_c	Nb of threads	Average time per iteration (ms)			Nb of iters.
			ComputeAssign	*Update*	Full iter.	
HPO $(n, d) = (2\,049\,280, 7)$	4	1	9.51	8.62	18.13	29
		20	**1.54**	**1.26**	**2.80**	**29**
		40	1.59	1.43	3.02	29
USC $(n, d) = (2\,458\,285, 68)$	16	1	278.02	54.79	332.81	39
		20	34.88	**13.55**	48.43	39
		40	**21.29**	19.86	**41.15**	**39**
	64	1	1099.71	53.74	1153.45	35
		20	121.91	**17.64**	139.55	35
		40	**70.74**	19.88	**90.62**	**35**
	256	1	4501.31	57.31	4558.62	82
		20	329.89	**10.59**	340.48	82
		40	**274.61**	15.17	**289.78**	**82**

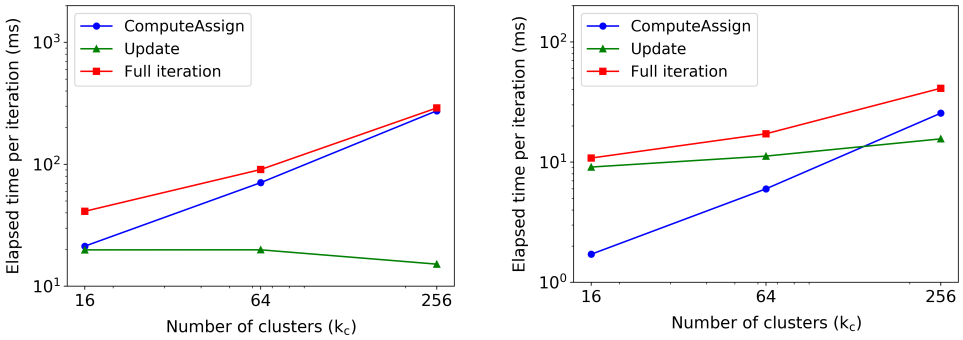
Table 2.4: GPU k-means on two real-world datasets (using single precision, 100 packages)

Dataset	k_c	Transfer time (ms)	Average time per iteration (ms)			Nb of iters.
			ComputeAssign	*Update*	Full iter.	
HPO $(n, d) = (2\,049\,280, 7)$	4	5.41	0.17	2.71	2.88	29
USC $(n, d) = (2\,458\,285, 68)$	16	55.91	1.71	9.09	10.80	39
	64	55.89	5.99	11.21	17.20	35
	256	55.88	25.53	15.60	41.13	82

Impact of k_c on performance

It can be seen more intuitively in Figure 2.13 that, when augmenting the number of clusters on the US census benchmark, the time of the *ComputeAssign* step grows approximately linearly both on CPU using 40 threads and on GPU, which is normal

because the *ComputeAssign* step calculates $n \times k_c$ distances at each iteration. The time of the *Update* step on GPU increases quite slowly with k_c and appears not very sensitive to k_c. This is reasonable because the main calculation is composed of $n \times d$ additions independently of k_c, but is organized in k_c reductions. However, the time of the *Update* step on CPU using 40 threads slowly decreases when k_c becomes larger (parallelization on larger loops).



(a) On CPU using 40 threads

(b) On GPU

Figure 2.13: Average time per iteration of k-means steps on the USC dataset (using single precision, 100 packages)

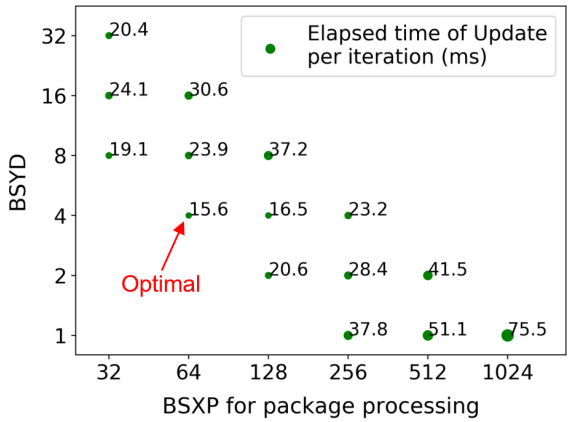


Figure 2.14: Impact of block size on the performance of *Update* with the USC dataset (using $k_c = 256$, single precision, 100 packages)

Impact of block size on performance

Again, we experimentally optimize the block size for all CUDA kernels since it can have a significant impact on the performance. An example of this impact is given

in Figure 2.14. Note that the block size (BSXP×BSYD) cannot exceed 1024 and meanwhile it should be no less than the number of clusters. The optimal block size in that case is $\texttt{BSXP} = 64$ and $\texttt{BSYD} = 4$ on our GPU (GeForce RTX 2080 Ti).

Performance comparison: GPU vs. CPU

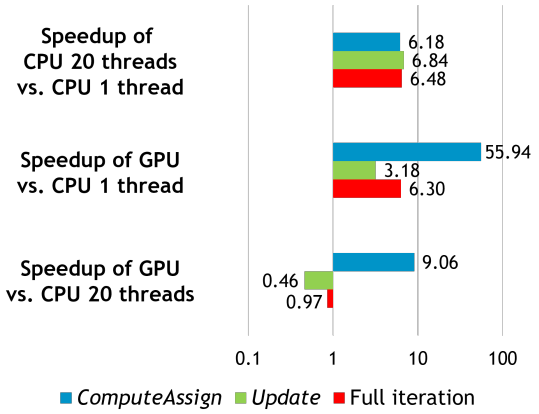


Figure 2.15: Speedup of k-means steps and iterations with the HPO dataset (using $k_c = 4$, single precision, 100 packages)

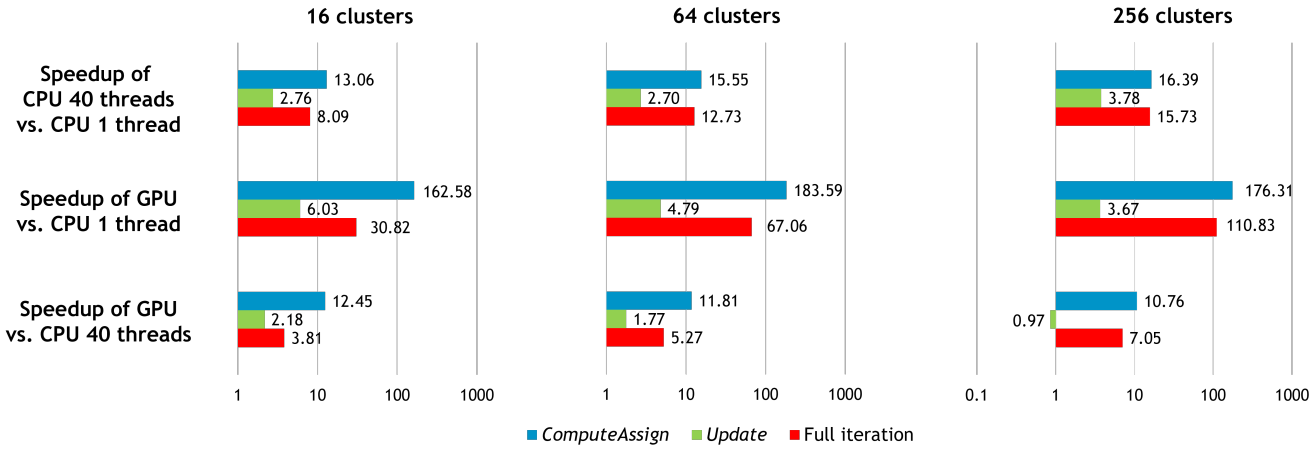


Figure 2.16: Speedup of k-means steps and iterations with the USC dataset (using single precision, 100 packages)

Figure 2.15 and Figure 2.16 present the speedup of our k-means steps and iterations on the two real-world datasets, respectively. Globally, the *ComputeAssign* step on GPU is from $\times 9$ up to $\times 12.5$ faster than on CPU running optimal number of threads with auto-vectorization (20 threads for the HPO dataset, 40 threads

for the USC dataset), while the *Update* step on GPU is from $\times 2$ slower to $\times 2$ faster than on CPU running optimal number of threads with auto-vectorization. The resulting full iteration on GPU is comparable or up to $\times 7$ faster than the full iteration on CPU running optimal number of threads with auto-vectorization, depending on the benchmark dataset and the desired number of clusters k_c. When increasing k_c on the USC dataset, the two steps as well as the full iteration on CPU using 40 threads obtain increasing speedups (compared to the CPU mono-thread implementation). Similarly, the acceleration effect of k-means iterations on GPU is greater with a larger k_c.

Despite our efforts, the *Update* step remains difficult to be further accelerated both on GPU and CPU. Significant differences in terms of speedup on GPU vs. CPU can be observed between the *ComputeAssign* and the *Update* steps. On the one hand, the *ComputeAssign* step has much more computations and natural parallelism than the *Update* step. On the other hand, the GPU kernel of the *Update* step suffers from three losses of performance (see Section 2.4.3): many `atomicAdd` operations, some loss of coalescence, and only k_c threads per block doing the summation.

Similar behaviour on synthetic & real-world datasets

According to the performance achieved on the Syn4D-50M dataset (see Figure 2.11) and on the two real-world datasets (see Figure 2.15 and Figure 2.16), we conclude that in all cases:

- Our k-means implementation on CPU running optimal number of threads with auto-vectorization (20 or 40 threads depending on the benchmark datasets) is significantly faster (from $\times 6$ up to $\times 16$) than our CPU mono-thread auto-vectorized implementation.
- Our GPU implementation generally outperforms our multithreaded auto-vectorized CPU implementation (up to $\times 7$ for the average time of k-means iterations).
- The obtained speedups come mainly from the *ComputeAssign* step.

2.5.4 . Comparison with others

As shown in Table 2.5 and Table 2.6, we compare the performance of our k-means implementations with five other parallel k-means implementations developed between 2016 and 2021.

Benchmarking approach

We chose to impose the same initial centroids for the same benchmark dataset in the comparative experiments on our CPU & GPU testbed. The performance results

in Table 2.5 represent the average time per iteration over the first 10 iterations before satisfying the criterion of convergence, while the results in Table 2.6 are the average time per iteration over all iterations until convergence. This is why some results of our GPU implementation in Table 2.5 are mildly different from those corresponding results in Table 2.4 and Table 2.6. Finally, considering the fluctuations of elapsed time, every time measurement above is the average of 5 runs.

Comparison with the MKM of Böhm, et al [23] on CPU

The MKM code is multithreaded with OpenMP (like ours) and explicitly vectorized with AVX 1 / AVX 2 intrinsic operations (while our code relies on auto-vectorization with the `-Ofast` and `-march=skylake-avx512` compilation flags). According to the paper [23], the MKM code compiled by gcc 4.7 for a corei7-avx architecture worked regardless of the number of data dimensions. However, when it was recompiled on our dual-Skylake CPU by gcc 9.3, it only worked for a number of dimensions that was a multiple of 4 (even when adjusting the options of compilation). Moreover, the MKM code computes only in double precision, while our code can work in single or double precision. Therefore, the comparison on CPU was done in double precision in Table 2.5. Since the MKM code was designed for a corei7-avx architecture but is now run on our dual-skylake CPU, we measure both the performance with `-march=corei7-avx` flag and the performance with `-march=native` flag for the MKM code, and we present the best performance in the table (other flags lead to less performance). Moreover, we also tested replacing `-O3` with `-Ofast` for the MKM code compilation, but this did not achieve higher performance.

Based on successful runs on some benchmarks on our dual-CPU, our multithreaded auto-vectorized implementation run on 20 physical cores was sometimes $\times 1.3$ slower and sometimes $\times 3.6$ faster, and run on 2×20 logical cores was sometimes $\times 1.2$ to $\times 2$ slower and sometimes $\times 2.5$ faster, depending on the benchmark. It is certain that before performing new tests, it would be necessary to solve the problems encountered by the MKM code at runtime on our architecture, for certain problem sizes.

Comparison with the cuda-kmeans of Kruliš, et al [106] on GPU

The cuda-kmeans code offers several algorithms and two data layouts (SoA & AoS) to choose from, but some algorithms did not accept certain numbers of points, numbers of dimensions or numbers of clusters of our benchmarks. For example, the use of one of the fastest algorithms (named *cuda_best*) requires the number of clusters to be a multiple of the *shmK* constant, and the number of dimensions to be a multiple of the *shmDim* constant. So in order to compare our code with *cuda_best*, we intervened in the cuda-kmeans code to tune the *shmK* and *shmDim*

constants (while our code does not require this kind of tunning). Additionally, the SoA layout was adopted because it was experimentally more efficient.

Since the time of data transfers is not included in the native measurements of cuda-kmeans code, neither it is counted in our average measured time per iteration in Table 2.5. The experiments on RTX 2080Ti shows that our GPU code appears sometimes slower and sometimes faster, depending on the benchmark, as previously with our CPU code.

Comparison with the KMeans of RAPIDS framework [181] on GPU

Developped by a community and *incubated* by NVIDIA, RAPIDS provides a suite of GPU-accelerated libraries and APIs (including the KMeans API in the cuML library) exploitable via user-friendly Python interfaces. As the KMeans API embeds both data transfers and program execution, the performance comparison of our code against the API in Table 2.5 also considers both transfers and execution time.

Although the KMeans API is supposed to be highly optimized and fast, it turns out that our GPU code appears $\times 1.4$ to $\times 9.3$ faster than the KMeans of RAPIDS v0.19 on RTX 2080Ti. We guess this significant difference is mainly induced by the wrapper overhead of Python interface (as our tests do not last long) and by the youth of RAPIDS (v0.19 in April 2021).

Comparison with the k-means of Yu, et al [208] on 1 node of Sunway TaihuLight supercomputer

The SW26010 manycore processor offers 260 cores and has a significantly different design from other multicore and manycore processors [208]. This processor appeared in 2016 as part of the Sunway TaihuLight supercomputer which was at that time ranked #1 in the Top500 list from 2016 to 2018.

The performances in Table 2.6 show that our k-means implementation in single precision on GeForce GTX 1080 (also appeared in 2016) is $\times 1.6$ to $\times 2.8$ faster than the single-node implementation for the SW26010 processor, considering the average execution time per iteration. As expected, our implementation is even faster on the more recent RTX 2080Ti GPU device (launched in 2018).

Comparison with the k-means of Li, et al [116] on an FPGA board

The Xilinx ZC706 FPGA board with an xc7z045ffg900-2 FPGA was available in 2015. The performances in Table 2.6 show that our k-means implementation in single precision, run on a GeForce GTX 1080 (appeared in 2016, just one year after the FPGA) is $\times 4.8$ faster than the FPGA implementation, regarding the average execution time per iteration. As previously, our implementation is even faster on a more recent GPU device appeared in 2018.

Main results of the comparisons

To summarize, our comparative experiments on real-life datasets show that:

- Our implementation running on a GPU that appeared in 2016 is more efficient than implementations on a FPGA and on a manycore appeared in 2015 and 2016, respectively (see Table 2.6).
- Compared to state-of-the-art implementations (Böhm, et al [23] and Kruliš, et al [106]) run on our classic CPU and GPU, our implementations are sometimes faster and sometimes slower, depending on the benchmark (see Table 2.5). However, our more generic source code has not required adaptations to run the different benchmarks.
- Moreover, in order to guarantee the numerical accuracy in case of rounding error accumulations with large-scale datasets, our code supports to split the numerous summations of the centroid updating without losing significant performances. All experiments of our implementations in Tables 2.5 and 2.6 have been done with 100 packages split.

Table 2.5: Performance comparison with recent parallel k-means implementations on our CPU & GPU testbed

Our testbed	**Precision**	**Authors or API**	**Language**	**Measured time**	**Average measured time (ms) per iteration over 10 iters**				
					HPO $(n, k_c) = (2\,049\,280, 4)$		**USC** $(n, d) = (2\,458\,285, 68)$		
					$d = 4$	$d = 7$	$k_c = 16$	$k_c = 64$	$k_c = 256$
Intel Xeon 4114 20 physical cores	Double	Böhm, et al [23]	C++ & Intrinsics & OpenMP	Execution	13.39	Segfault	**56.32**	**152.37**	**450.70**
		Ours (100 pkgs)	C & OpenMP		**3.75**	**5.74**	73.99	202.14	580.27
Intel Xeon 4114 40 logical cores	Double	Böhm, et al [23]	C++ & Intrinsics & OpenMP	Execution	11.64	Segfault	**41.16**	**88.33**	**328.31**
		Ours (100 pkgs)	C & OpenMP		**4.58**	**7.05**	82.89	139.31	406.33
Nvidia GeForce RTX 2080 Ti	Single	Kruliš, et al [106]	C++ & CUDA	Execution	1.73	**1.81**	14.57	19.49	**36.67**
		Ours (100 pkgs)	C & CUDA		**1.12**	2.88	**12.09**	**19.28**	44.66
		KMeans in RAPIDS [181] cuML	Python & CUDA	Transfers + Execution	13.60	15.21	26.15	35.82	120.43
		Ours (100 pkgs)	C & CUDA		**1.46**	**3.42**	**17.68**	**24.87**	**50.25**

Table 2.6: Performance comparison with parallel k-means implementations on other architectures

Testbed	Precision	Authors	Measured time	Average execution time (ms) per iteration over all iterations	
				HPO $(n, d, k_c) = (2\,049\,280, 4, 4)$	**USC** $(n, d, k_c) = (2\,458\,285, 68, 64)$
1 node of Sunway TaihuLight supercomputer with 1 SW26010 260-core manycore (2016)	N/A	Yu, et al [208]	Execution	2.84	$\approx$110
Xilinx ZC706 FPGA board with an xc7z045ffg900-2 FPGA (2015)	Single	Li, et al [116]	Execution	8.50	N/A
Nvidia GeForce GTX 1080 (**2016**)	Single	Ours (100 pkgs)	Execution	**1.76**	**38.97**
Nvidia GeForce RTX 2080Ti (2018)				1.11	17.20

2.6 . Summary

In this chapter, we have proposed parallel implementations on CPU and GPU for the k-means clustering algorithm, which can be used independently or serve as two components in our computational chain for spectral clustering when processing large amount of data (see Section 1.6). Through a two-level summation method with package processing, we have addressed the numerical accuracy issue in the phase of updating cluster centroids due to the effect of rounding errors. To our knowledge, we are the first to consider and address the numerical accuracy issue in the k-means algorithm.

Our CPU implementation relies on thread parallelization using OpenMP and on auto-vectorization using AVX-512 instructions. Our CUDA implementation on GPU employs dynamic parallelism, multiple streams and shared memory to achieve optimal performance. Experiments on synthetic and real-world datasets demonstrate both numerical accuracy and parallelization efficiency of our k-means implementations on CPU and GPU.

3 - Scalable Data Formats and Algorithms for Spectral Clustering

3.1 . Introduction

As explained in Section 1.3.3, traditional spectral clustering algorithms have high time complexity due to $\mathcal{O}(n^2d)$ complexity for similarity matrix construction and $\mathcal{O}(n^3)$ complexity for eigen-decomposition. The storage of similarity matrix and Laplacian matrix in dense format requires $\mathcal{O}(n^2)$ memory space. Together, the huge calculation cost and the huge memory space requirements constitute the barrier to large-scale spectral clustering.

In this chapter, we address the scalability issue of spectral clustering on the GPU. Most importantly, we propose in Section 3.3 three optimized GPU algorithms for constructing similarity graph and matrix in Compressed Sparse Row (CSR) format. This can achieve significant performance improvement compared to sequential algorithm, and meanwhile reduce substantial memory space requirements on the GPU compared to using dense data format. Then, in Section 3.4 we leverage the Spectral Clustering API of NVIDIA's GPU-accelerated nvGRAPH library for subsequent computations including Laplacian matrix calculation, eigen-decomposition, and final k-means clustering (see Figure 1.1). Although called "Spectral Clustering API" by nvGRAPH, its function is essentially equivalent to spectral graph partitioning in the field of graph analytics, and it requires the CSR-format similarity graph to be provided as input.

Finally, extensive experiments in Section 3.6 demonstrate the high performance and scalability of our GPU implementation for spectral clustering.

The work presented in this chapter has been first published as a conference paper of NPC 2021 [78] (initial version) and then submitted as a journal article to IJPP [77] (extended version).

3.2 . Spectral clustering using dense data format

In this section, we start from parallelizing spectral clustering algorithm (Algorithm 3 in Section 1.3.1) on the GPU using dense data format. Globally, the similarity matrix and Laplacian matrix construction steps are parallelized using optimized CUDA kernels (see Section 3.2.1), then the eigen-decomposition step is implemented by leveraging the cuSOLVER library (see Section 3.2.2), finally the normalization and k-means steps are also parallelized using optimized CUDA kernels (see Section 3.2.3). The host code is presented in Listings 3.1 and the CUDA kernels are shown in Listings 3.2, 3.3 and 3.4. They are explained below and can provide some basis for understanding more complicated CUDA kernels of

Section 3.3.

3.2.1 . Similarity matrix and Laplacian matrix construction

As mentioned in Section 1.3.1, the similarity graph can be constructed in several ways (e.g. *full connection*, ε*-neighborhood*, k*-nearest neighbor*), resulting in either dense or sparse similarity matrix S. However, in this section we just consider the simple case of storing all matrices in dense format.

We first launch a 2D grid with 2D blocks of threads (Listing 3.1 lines 10-16) to compute the similarity matrix S and the diagonal degree matrix D. Basically, each thread calculates one element of similarity matrix and stores the value into the global memory array (Listing 3.2 lines 14-54). Moreover, the similarity value is also stored into a shared memory array of block size, so that a classic parallel reduction within the shared memory array is performed and the per-block contribution to a degree is accumulated with an `atomicAdd` operation (Listing 3.3).

We highlight several points regarding the above kernel:

- We choose to construct *full connected* graph and ε*-neighborhood-like* graph (see definition in Section 3.3.1) instead of k*-nearest neighbor* graph, because the last one requires expensive sorting operations and usually need to be symmetrized.
- The kernel is generic in the sense that it supports multiple similarity metrics and thresholds, such as uniform similarity with threshold for squared distance (Listing 3.2 lines 16-25), Gaussian similarity with threshold for squared distance (Listing 3.2 lines 27-32) or with threshold for similarity (Listing 3.2 lines 34-39), cosine similarity with threshold for similarity (Listing 3.2 lines 41-51), and the kernel can also be extended to support other metrics.
- We ensure most of the global memory accesses are coalesced (e.g. Listings 3.2, lines 19 & 44).
- We use the `__expf()` function instead of the `expf()` function for Gaussian similarity computation (line 20) because the former maps directly to the hardware level, thus it is faster (but provides lower accuracy) than the latter [140].
- For Gaussian similarity, if the threshold for squared distance is infinitely great or the threshold for similarity is equal to 0, then the similarity graph will be full connected and the similarity matrix will be dense theoretically, however, practically tiny similarity values that result from distant instance pairs may be stored as 0 due to the underflow of floating-point numbers.
- We pay special attention to the implementation of the cosine similarity with threshold checking (Listing 3.2 lines 47-49), since different implementation ways can lead to varied performance and precision, the selected way can limit the propagation of rounding errors and meanwhile provide good performance.

Some aspects related to scaling and memory space are also worth mentioning. (1) It is possible that the value of `Dg.y` exceeds the upper limit (65535 for RTX 3090) if the number of data instances n is too large, but in this case the storage of similarity matrix would also exceeds the GPU memory. (2) It is also possible that the element index of similarity matrix exceeds the integer limit (about 2 billion) if n^2 exceeds the integer limit, in this case it would be necessary to extend the index representation capacity by using, e.g., `size_t`. (3) We only need to allocate an array of size n to store the diagonal elements of degree matrix.

Next we launch likewise a 2D grid with 2D blocks of threads (Listing 3.1 lines 17-18) to compute the normalized Laplacian matrix $L_{sym} = D^{-1/2}SD^{-1/2}$ (Listing 3.4). In practice we use the same array to store S and L_{sym} so as to save GPU memory. Note that we choose to compute the symmetric matrix L_{sym} instead of the non-symmetric normalized Laplacian matrix $L_{rw} = D^{-1}S$, because the eigensolver in the cuSOLVER library that we want to use afterwards requires the input matrix to be symmetric.

3.2.2 . Eigen-decomposition using cuSOLVER library

To compute the first k_c eigenvectors (associated with the smallest k_c eigenvalues) of L_{sym}, we leverage the dense symmetric eigensolver `syevdx` of NVIDIA's GPU-accelerated cuSOLVER library [143] (Listing 3.1 lines 21-36). It can compute all or a selection of the eigenvalues and eigenvectors of a symmetric (or Hermitian) matrix, solving the standard symmetric eigenvalue problem through the QR algorithm [155]. For output, the first k_c eigenvectors will be stored in the input matrix L_{sym} by rewriting its first $k_c \times n$ elements.

There are also other dense symmetric eigensolvers in the cuSOLVER library, but they either compute all eigenpairs (e.g. `syevd`, `sygvd`, `syevj`, `sygvj`), or require the matrix D to be an array of size $n \times n$ when targeting the generalized eigenvalue problem $LU = DU\Sigma$ (e.g. `sygvd`, `sygvdx`, `sygvj`). Therefore we choose the `syevdx` solver rather than others.

3.2.3 . Normalization and final k-means(++) clustering

After obtaining the first k_c eigenvectors stored in the form of $k_c \times n$, we regard them as the transposed matrix U^T and launch a CUDA kernel to normalize each column of U^T (i.e. each row of U) to unit length. We use 1D grid with 1D blocks of threads (Listing 3.1 lines 39-42) for the kernel so that each thread normalizes one column of U^T by going through all k_c elements of that column. The kernel is too simple so we do not show its content in the listings.

Finally, we apply our GPU implementation of k-means (see Section 2.4) or k-means++ (see Appendix G Listings G.1 & G.2) on the points represented by the normalized rows of U and obtain the cluster labels of data instances (Listing 3.1 lines 45-47).

```
// Declaration, memory allocation, initialization
float *GPU_sim, *GPU_deg, *GPU_lap, *GPU_eigVects;
cudaMalloc((void**) &GPU_sim, (sizeof(float)*n)*n);
cudaMalloc((void**) &GPU_deg, sizeof(float)*n);
cudaMemset(GPU_deg, 0, sizeof(float)*n);
GPU_lap = GPU_sim;       // Use the same memory
GPU_eigVects = GPU_lap;  // Use the same memory

// Launch CUDA kernels to compute similarity, degree, Laplacian matrices
dim3 Dg, Db;
Db.x = BSX;   Db.y = BSY;
Dg.x = n/Db.x + (n%Db.x > 0 ? 1 : 0);
Dg.y = n/Db.y + (n%Db.y > 0 ? 1 : 0);
size_t shMemSize = (sizeof(float)*Db.y)*Db.x;
constructSimDegMat<<<Dg,Db,shMemSize>>>(...,                // input
                                        GPU_sim, GPU_deg);  // output
computeLapMat<<<Dg,Db>>>(GPU_sim, GPU_deg, ..., // input
                         GPU_lap);              // output

// Configure eigensolver parameters
cusolverEigMode_t  jobz  = CUSOLVER_EIG_MODE_VECTOR;
cusolverEigRange_t range = CUSOLVER_EIG_RANGE_I;
int il = 1;   int iu = kc;  // Selection from 1st to kc-th eigenpairs
int lwork = 0;              // Size of work
// Calculate the sizes needed for working space (lwork)
CUSOLVERDN_SYEVDX_BUFFERSIZE(jobz, range, il, iu,  // input
                             GPU_lap, ...,         // input
                             GPU_eigVals, &lwork); // output
// Allocate memory for working space (d_work)
cudaMalloc((void**)&d_work, sizeof(float)*lwork);
// Compute the first kc eigenpairs of Laplacian & store them in GPU_lap
CUSOLVERDN_SYEVDX(jobz, range, il, iu,  // input
                  d_work, lwork, ...,   // input
                  GPU_lap,              // input & output
                  GPU_eigVals);         // output
cudaFree(d_work); // Free working space

// Launch a CUDA kernel to normalize the eigenvector matrix
Db.x = BSX;   Db.y = 1;
Dg.x = n/Db.x + (n%Db.x > 0 ? 1 : 0);   Dg.y = 1;
normalizeEigenvectorMatrix<<<Dg,Db>>>(GPU_lap, ...,   // input
                                       GPU_eigVects);  // output

// Apply k-means++ on points spanned in the first kc-dim eigenspace
d = kc;  // Set the # of dims equal to the # of clusters
kmeanspp(GPU_eigVects, ...,  // input
         GPU_labels, ...);   // output

... // Memory deallocation
```

Listing 3.1: Host code of GPU implementation for spectral clustering in dense data format

```
// Starting address for dynamic allocation of shared memory
extern __shared__ float shBuff[];

__global__ void constructSimDegMat (...) {
  // 2D blocks, 2D grid
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int tidPerBlock = threadIdx.y*blockDim.x + threadIdx.x;
  // Declaration & initialization of a shared memory array
  float *shTabSim = shBuff;
  shTabSim[tidPerBlock] = 0.0f;

  // Calculation of similarity matrix S (1 elt/thread)
  if (row < n && col < n) {
    // Uniform similarity with threshold for squared distance
    #ifdef UNI_SIM_WITH_SQDIST_THOLD
      float diff, sqDist = 0.0f;
      for (int j = 0; j < d; j++) {
        diff = GPU_dataT[j*n + row] - GPU_dataT[j*n + col];
        sqDist += diff*diff;
      }
      if (sqDist < tholdSqDist && row != col)
        shTabSim[tidPerBlock] = 1.0f;
      GPU_sim[row*n + col] = shTabSim[tidPerBlock];
    #endif
    // Gaussian similarity with threshold for squared distance
    #ifdef GAUSS_SIM_WITH_SQDIST_THOLD
      ... // Calculate squared distances (sqDist)
      if (sqDist < tholdSqDist && row != col)
       shTabSim[tidPerBlock]=__expf((-1.0f)*sqDist/(2.0f*sigma*sigma));
      GPU_sim[row*n + col] = shTabSim[tidPerBlock];
    #endif
    // Gaussian similarity with threshold for similarity
    #ifdef GAUSS_SIM_WITH_THOLD
      ... // Calculate squared distances (sqDist)
      float sim = __expf((-1.0f)*sqDist/(2.0f*sigma*sigma));
      if (sim > tholdSim && row != col) shTabSim[tidPerBlock] = sim;
      GPU_sim[row*n + col] = shTabSim[tidPerBlock];
    #endif
    // Cosine similarity with threshold for similarity
    #ifdef COS_SIM_WITH_THOLD
      float elm1, elm2, dot = 0.0f, sq1 = 0.0f, sq2 = 0.0f;
      for (int j = 0; j < d; j++) {
        elm1 = GPU_dataT[j*n + row];  elm2 = GPU_dataT[j*n + col];
        dot += elm1*elm2;  sq1 += elm1*elm1;  sq2 += elm2*elm2;
      }
      float sqSim = (dot*dot)/(sq1*sq2);
      if (sqSim > tholdSim*tholdSim && row != col)
        shTabSim[tidPerBlock] = SQRT(sqSim);
      GPU_sim[row*n + col] = shTabSim[tidPerBlock];
    #endif
    // Other metrics
    #ifdef ... #endif
  }

  ... // Part 2
}
```

Listing 3.2: `constructSimDegMat` kernel (part 1)

```cpp
__global__ void constructSimDegMat (...) {
  ... // Part 1

  // Calculation of degree array: two-part reduction
  // 1 - Parallel reduction of the shared memory array
  //     shTabSim[tidPerBlock] into shTabSim[threadIdx.y*blockDim.x]
  //     (i.e. from the 2D view: shTabSim[*][*] --> shTabSim[*][0])
  if (BSX > 512) {
    __syncthreads();
    if (threadIdx.x < 512)
      shTabSim[threadIdx.y*blockDim.x + threadIdx.x]
        += shTabSim[threadIdx.y*blockDim.x + threadIdx.x + 512];
    else  return;  // kill useless warps
  }
  ...  // # of remaining threads per block: 512 --> 256 --> 128 --> 64
  if (BSX > 32) {
    __syncthreads();
    if (threadIdx.x < 32)
      shTabSim[threadIdx.y*blockDim.x + threadIdx.x]
        += shTabSim[threadIdx.y*blockDim.x + threadIdx.x + 32];
    else  return;  // kill useless warps
  }                // only the 1st warp survives
  if (BSX > 16) {
    __syncwarp();  // avoid races between threads within the same warp
    if (threadIdx.x < 16)
      shTabSim[threadIdx.y*blockDim.x + threadIdx.x]
        += shTabSim[threadIdx.y*blockDim.x + threadIdx.x + 16];
  }
  ...  // # of working threads in the 1st warp: 8 --> 4 --> 2
  if (BSX > 1) {
    __syncwarp();  // avoid races between threads within the same warp
    if (threadIdx.x < 1)
      shTabSim[threadIdx.y*blockDim.x + threadIdx.x]
        += shTabSim[threadIdx.y*blockDim.x + threadIdx.x + 1];
  }
  // 2 - Final reduction into the global array
  if (threadIdx.x == 0 && row < n) {
    if (shTabSim[threadIdx.y*blockDim.x] > 0.0f)
      atomicAdd(&GPU_deg[row], shTabSim[threadIdx.y*blockDim.x]);
  }
}
```

Listing 3.3: `constructSimDegMat` kernel (part 2)

```cpp
__global__ void computeLapMat (...) {
  // 2D blocks, 2D grid
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < n && col < n) {
    float deg  = GPU_deg[col];
    float degL = RSQRT(GPU_deg[row]); // 1.0/sqrt()
    float degR = RSQRT(deg);
    float sim  = GPU_sim[row*n + col];
    float lap;
    if (row != col)  lap = - sim;
    else             lap = deg - sim;
    GPU_lap[row*n + col] = degL * lap * degR;
  }
}
```

Listing 3.4: `computeLapMat` kernel

3.3 . Construction of the similarity matrix in sparse format

The previous section handles the high time complexity of spectral clustering through GPU computing, however it uses dense data format. As the number of data instances n grows over the order of 10^4, it will become impossible to store the dense-format square matrices with limited GPU memory.

In this section we propose efficient GPU algorithms for constructing the similarity matrix in CSR format, which play an important role in handling the scalability challenge of spectral clustering in terms of both calculation cost and memory requirements.

3.3.1 . Sparsification and choice of a storage format

The similarity matrix associated with ε-*neighborhood* graph or k-*nearest neighbor* graph generally has a sparse pattern, i.e. containing numerous zeros. Even for the similarity matrix associated with *fully connected* graph, we observe that usually a significant portion of elements are very close to zero. By setting a small threshold for similarity (e.g. 0.01) and regarding those below-threshold similarities as zeros, we are likely to obtain a sparse similarity matrix. We think this sparsification way is reasonable since it resembles the way of ε-*neighborhood* graph. The difference is that the former sets below-threshold similarities to zeros and while the latter sets similarities associated with over-threshold distances to zeros. For simplicity, we call both of the related graphs as ε-*neighborhood-like* graph in this dissertation. Storing the similarity matrix in a sparse format can save typically most of the memory needed for dense format storage, thus increase significantly the scale of datasets able to be processed on the GPU.

There are various sparse formats for storing a sparse matrix. Here we do not try to enumerate all but introduce several commonly used sparse formats in many linear algebra libraries: Coordinate format (COO), Compressed Sparse Row format (CSR), Compressed Sparse Column format (CSC), and Ellpack format. Note that we use nnz to represent the total number of nonzero elemnts in a matrix.

Coordinate format (COO)

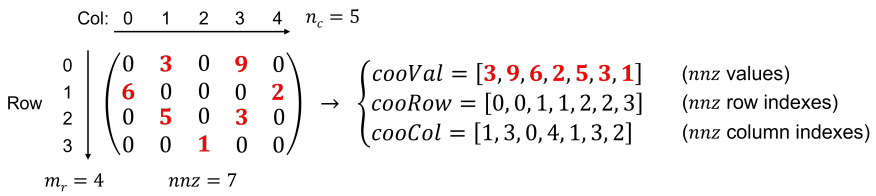


Figure 3.1: An example of COO format storing an $m_r \times n_c$ matrix

The COO format of a sparse matrix consists of three arrays. We call them `cooVal[]`, `cooRow[]`, `cooCol[]`. Figure 3.1 gives a COO example with an

$m_r \times n_c$ matrix. The three arrays `cooVal[]`, `cooRow[]` and `cooCol[]` store the values, row indexes, and column indexes of all nonzero matrix elements in row-major format, respectively. Clearly, the memory requirement for COO format is $3 \times nnz$.

In graph analytics, the COO representation of similarity graph corresponds to an edge list. Let (v_i, v_j, w_{ij}) denote a directed edge from vertex v_i to vertex v_j with weight w_{ij}. The edge list is composed of a list of couples (v_i, v_j) or triples (v_i, v_j, w_{ij}) for all edges. Note that undirected edges are represented in both directions.

Compressed Sparse Row format (CSR)

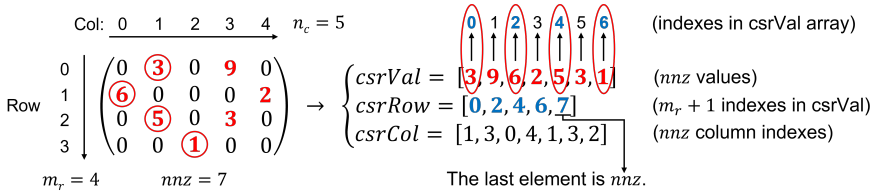


Figure 3.2: An example of CSR format storing an $m_r \times n_c$ matrix

The CSR format of a sparse matrix consists of three arrays. We call them `csrVal[]`, `csrCol[]`, `csrRow[]`. Figure 3.2 gives a CSR example with the $m_r \times n_c$ matrix. Similar to COO format, `csrVal[]` and `csrCol[]` store the values and column indexes of all nonzero matrix elements in row-major format, respectively. However, unlike COO format, `csrRow[]` considers the first nonzero element in each row of the matrix (i.e. the circled red numbers in the figure) and holds their indexes that count in `csrVal[]` (i.e. the blue numbers circled by red ellipses), and in the end contains the total number of nonzero elements of the matrix. In other words, `csrRow[]` considers the number of nonzeros (in row-major order) before the first nonzero element of each row and stores it in row-major order. Therefore, the memory requirement for CSR format is $2 \times nnz + m_r + 1$ (see annotations on right side of Figure 3.2).

In graph analytics, the CSR representation of similarity graph corresponds to an adjacency list, where for each vertex v_i, all its neighbours (e.g. $v_j, ..., v_k$) and optionally the corresponding edge weights (e.g. $w_j, ..., w_k$) are recorded in one row for that vertex.

Compressed Sparse Column format (CSC)

The CSC format of a sparse matrix consists of three arrays. We call them `cscVal[]`, `cscCol[]`, `cscRow[]`. Figure 3.3 gives a CSC example with the $m_r \times n_c$ matrix. The CSC format is similar to the CSR format except that the latter stores

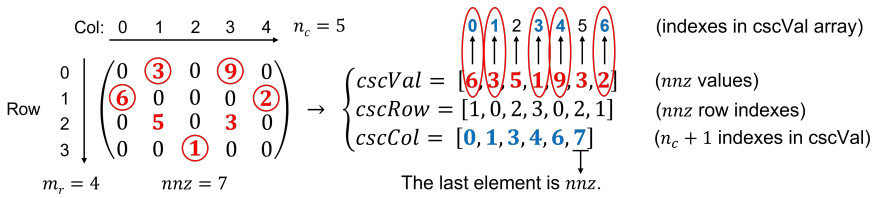


Figure 3.3: An example of CSC format storing an $m_r \times n_c$ matrix

the matrix in row-major format while the former stores it in column-major format. Specifically, `cscVal[]` and `cscRow[]` store the values and row indexes of all nonzero matrix elements in column-major format, respectively. `cscCol[]` considers the first nonzero element in each column of the matrix (i.e. the circled red numbers in the figure) and holds their indexes that count in `cscVal[]` (i.e. the blue numbers circled by red ellipses), and in the end contains the total number of nonzero elements of the matrix. In other words, `cscCol[]` considers the number of nonzeros (in column-major order) before the first nonzero element of each column and stores it in column-major order. Therefore, the memory requirement for CSC format is $2 \times nnz + n_c + 1$.

Ellpack format

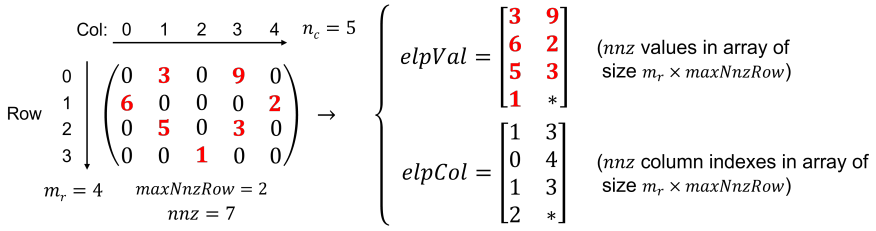


Figure 3.4: An example of Ellpack format storing an $m_r \times n_c$ matrix

The Ellpack format of a sparse matrix consists of two arrays. We call them `elpVal[]` and `elpCol[]`. Figure 3.4 gives an Ellpack example with the $m_r \times n_c$ matrix. Let $maxNnzRow$ denote the maximum number of nonzero elements in a row, i.e. the number of nonzeros in the densest row. For each row, a segment of $maxNnzRow$ size is reserved respectively in `elpVal[]` and `elpCol[]` for storing the values and column indexes of nonzero elements of that row in row-major order. If a row has fewer nonzeros than $maxNnzRow$, then the superfluous space will be wasted, as marked '*' in the figure. Therefore, the memory requirement for Ellpack format is $2 \times m_r \times maxNnzRow$.

Comparison and our choice

Table 3.1: Comparison of four sparse matrix formats

	COO	CSR	CSC	Ellpack

Arrays	`cooVal`
	`cooRow`
	`cooCol`
		`csrVal`
		`csrRow`
		`csrCol`
			`cscVal`
			`cscRow`
			`cscCol`
				`elpVal`
				`elpCol`

Storage order	Row-major	Row-major	Column-major	Row-major
Memory requirement	$3 \times nnz$	$2 \times nnz + m_r + 1$	$2 \times nnz + n_c + 1$	$2 \times m_r \times maxNnzRow$
Advantages	(1) Natural and easy to understand; (2) Support fast conversions from/to other sparse formats; (3) Suited to all sparsity patterns.	(1) Generally less memory consumption than COO and Ellpack; (2) Support efficient row slicing; (3) Support efficient matrix-vector computations; (4) Suited to all sparsity patterns.	(1) Generally less memory consumption than COO and Ellpack; (2) Support efficient column slicing; (3) Support efficient matrix-vector computations; (4) Suited to all sparsity patterns.	(1) Only 2 arrays; (2) Efficient row slicing; (3) Suited to regular sparsity pattern.
Drawbacks	(1) Generally more memory consumption than CSR and CSC due to redundancy of information; (2) Does not support efficient slicing.	Does not support efficient column slicing.	Does not support efficient row slicing.	(1) Generally more memory consumption than CSR and CSC due to imbalance in the number of nonzeros per row; (2) Unsuited for power law sparsity pattern.

Table 3.1 compares the four sparse formats. Among them, we prefer the CSR format for storing sparse similarity matrix. Because it is well suited to both regular and irregular (e.g. power law distribution) sparsity patterns [55] and usually requires less memory than COO and Ellpack formats. Moreover, the CSR format is efficient for matrix-vector computations[1]. Additionally, we will also show in Chapter 4 that the CSR format can support efficient row slicing (while COO and CSC cannot). With these advantages, the CSR format has been widely used and supported in most libraries. Finally, we intend to use the spectral graph partitioning algorithms for the nvGRAPH library and they support only the CSR format for graph representation.

[1]According to the SciPy API reference for scipy.sparse.csr_matrix

3.3.2 . Difficulties

Recall that we want to address the memory space bottleneck of large-scale spectral clustering by storing the similarity matrix in CSR format. Hence it makes no sense to first construct the similarity matrix using a dense format storage and then transform it from dense to CSR format. It seems that the construction of similarity matrix should be directly performed in CSR format. However, several restrictions make it difficult to be efficiently implemented in parallel especially on the GPU. First, the total number of nonzero elements is unknown, so we cannot allocate memory for `csrVal[]` and `csrCol[]`. Moreover, the number of nonzeros per row is unknown, thus we cannot know in advance in which segment of `csrVal[]` and `csrCol[]` we should store the value and column index of each nonzero entry, respectively. Besides, although GPU threads can compute similarities and check nonzeros in parallel, they are unable to store nonzeros (values and column indexes) at the right places of `csrVal[]` and `csrCol[]`, since each thread does not know the number of nonzeros ahead of it. In contrast, the Ellpack format, which we use for intermediate storage, will cause us fewer problems (see Section 3.3.4).

We point out that in this dissertation only *ε-neighborhood-like* graph construction is considered for generating sparse similarity matrix. The *k-nearest neighbor* graph does not have the first two issues stated above, but it requires expensive sorting operations. We will also show in Chapter 4 that the *ε-neighborhood-like* graph is more informative than *k-nearest neighbor* graph in terms of filtering noise.

In the following we propose 3 different algorithms and their associated GPU implementations for the parallel construction of *ε-neighborhood-like* similarity graph and matrix in CSR format, always avoiding storing the full similarity matrix in dense format.

3.3.3 . Algo CSR-1: straightforward CSR

Algorithm

Algorithm 4 describes the construction of the CSR format similarity matrix in the straightforward way. It is mainly composed of two full passes across all the elements of similarity matrix. The first pass (`1stPass` kernel) is dedicated to count the number of nonzeros per row into `nnzPerRow[]`, so that we can get the total number of nonzeros (nnz) and allocate exact size of memory for `csrVal[]` and `csrCol[]`. Moreover, `csrRow[]` can be derived from `nnzPerRow[]` with an exclusive scan, which allows to know the location of nonzeros related to each row in `csrVal[]` and `csrCol[]`. With all these information, the second pass (`2ndPass` kernel) can then parallelly store the nonzeros into `csrVal[]` and `csrCol[]` after re-finding them. Additionally, we find from `nnzPerRow[]` the minimum and maximum number of nonzeros in a row, which will be used for filtering noise in Chapter 4.

Algorithm 4: Straightforward construction of the CSR format similarity matrix (Algo CSR-1)

Inputs:

(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$

(2) Similarity metric and connectivity parameters, e.g. σ, threshold

Outputs:

(1) Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`

(2) Number of nonzeros per row: `nnzPerRow[]`

(3) Minimum and maximum number of nonzeros in a row: $minNnzRow$, $maxNnzRow$

(4) Total number of nonzeros: nnz

1 Conduct the **first pass** (`1stPass` kernel) that:
- computes similarities and find nonzeros satisfying threshold;
- counts the number of nonzeros per row and stores in `nnzPerRow[]`.

2 Find $minNnzRow$ and $maxNnzRow$ from `nnzPerRow[]`.

3 Perform an exclusive scan on `nnzPerRow[]` to obtain `csrRow[]`.

4 Get nnz from `csrRow[]` and allocate memory for `csrVal[]` and `csrCol[]`.

5 Conduct the **second pass** (`2ndPass` kernel) that:
- recomputes similarities and find nonzeros satisfying threshold;
- stores the values and column indexes of nonzeros into `csrVal[]` and `csrCol[]`, respectively.

GPU implementation

Our GPU implementation for Algo CSR-1 is detailed in Appendix C. Essentially, it consists in two optimized CUDA kernels: the `1stPass` kernel and the `2ndPass` kernel.

3.3.4 . Algo CSR-2: Ellpack-to-CSR

Algorithm

Algorithm 5 describes the construction of the CSR format similarity matrix based on an Ellpack-to-CSR approach. The basic idea is to try to first store the similarity matrix in Ellpack format and then convert it into CSR format. So we need to make a hypothesis for the maximum number of nonzeros in a row ($hypoMaxNnzRow$), and allocate two temporary arrays of Ellpack format (`csrValMax[]` and `csrColMax[]`) with the size of $n \times hypoMaxNnzRow$ (n is the number of instances).

The algorithm is primarily composed of a single full pass across all the elements in similarity matrix, and if necessary a supplementary pass across a part of similarity matrix. The full pass (`fullPass` kernel) undertakes multiple tasks:

Algorithm 5: Construction of the CSR format similarity matrix based on an Ellpack-to-CSR approach (Algo CSR-2)

Input:

(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$

(2) Similarity metric and connectivity parameters, e.g. σ, threshold

(3) Hypothetical maximum number of nonzeros in a row: $hypoMaxNnzRow$

Output:

(1) Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`

(2) Number of nonzeros per row: `nnzPerRow[]`

(3) Minimum and maximum number of nonzeros in a row: $minNnzRow$, $maxNnzRow$

(4) Total number of nonzeros: nnz

1 Allocate two arrays `csrValMax[]` and `csrColMax[]` with each of $n \times hypoMaxNnzRow$ size for storing the values and column indexes of nonzeros in Ellpack format, respectively.

2 Conduct a **full pass** (`fullPass` kernel) across the similarity matrix that:
- computes similarities and find nonzeros satisfying threshold;
- counts the number of nonzeros per row and stores in `nnzPerRow[]`;
- accumulates the values and column indexes of nonzeros for each row into `csrValMax[]` and `csrColMax[]` until the number of nonzeros in a row reaches $hypoMaxNnzRow$;
- records restart indexes that may be used for a supplementary pass for storing remaining nonzeros.

3 Find $minNnzRow$ and $maxNnzRow$ from `nnzPerRow[]`.

4 Perform an exclusive scan on `nnzPerRow[]` to obtain `csrRow[]`.

5 Get nnz from `csrRow[]` and allocate memory for `csrVal[]` and `csrCol[]`.

6 Launch the `ellpackToCSR` kernel to fill `csrVal[]` and `csrCol[]` with accumulated nonzeros stored in `csrValMax[]` and `csrColMax[]`.

7 **if** $maxNnzRow > hypoMaxNnzRow$ **then**

8 Conduct a **supplementary pass** (`supPass` kernel) across a part of similarity matrix that:
- recomputes similarities from restart indexes and find remaining nonzeros satisfying threshold;
- complete `csrVal[]` and `csrCol[]` by storing the values and column indexes of remaining nonzeros.

9 **end**

1. it computes all similarities and counts the number of nonzeros per row into `nnzPerRow[]`;

2. it stores as many nonzeros as possible in the Ellpack arrays;

3. it records the restart places in each row for the possible supplementary pass in case that the hypothesis ($hypoMaxNnzRow$) is too small.

With `nnzPerRow[]`, we can easily get the real maximal number of nonzeros in a row ($maxNnzRow$), the `csrRow[]` array, and the total number of nonzeros (nnz) which allows to allocate memory for `csrVal[]` and `csrCol[]`.

If our hypothesis is large enough (i.e. $maxNnzRow <= hypoMaxNnzRow$), indicating the constructed Ellpack arrays contain the information of all nonzeros, then it just remains to fill the CSR arrays (`csrVal[]` and `csrCol[]`) by an Ellpack-to-CSR copy (`ellpackToCSR` kernel). However, if our hypothesis is too small (i.e. $maxNnzRow > hypoMaxNnzRow$), indicating the constructed Ellpack arrays miss some nonzeros, then besides the Ellpack-to-CSR copy we also need to conduct a supplementary pass (`supPass` kernel) to find the missing nonzeros and store them at the right places in `csrVal[]` and `csrCol[]`. Note that the supplementary pass does not traverse all elements of similarity matrix, but only starts the work from the restart indexes recorded by the first pass.

Additionally, we also find the minimum number of nonzeros in a row ($minNnzRow$), which will be used, together with $maxNnzRow$, for filtering noise in Chapter 4.

GPU implementation

Our GPU implementation for Algo CSR-2 is detailed in Appendix D. Essentially, it consists in three optimized CUDA kernels: the `fullPass` kernel, the `ellpackToCSR` kernel, and the `supPass` kernel.

3.3.5 . Algo CSR-3: chunkwise dense-to-CSR

Algorithm

Algorithm 6 describes the construction of the CSR format similarity matrix based on a chunkwise dense-to-CSR approach. As mentioned in Section 3.3.2, it makes no sense to first construct the similarity matrix with dense format storage and then transform it from dense to CSR format, since for datasets with large number of instances (n) it would be impossible to store the $n \times n$ similarity matrix in dense format with limited GPU memory. However, it is feasible to construct only a chunk of similarity matrix in dense format at a time so that we can convert each part into CSR format and finally merge the CSR results of all parts to obtain the CSR representation of the whole similarity matrix. We consider partitioning the similarity matrix horizontally into chunks of similar size. The horizontal partitioning can facilitate merging the CSR results of different chunks since the CSR format is stored in row-major order. The number of chunks can be determined automatically in the way that only one chunk can fit into the available GPU memory or the percent

of free GPU memory that we want to use. However, the total number of nonzeros is still unknown in advance. We need to assume the maximum percentage of nonzeros in the matrix so that we can allocate memory for CSR arrays. Additionally, we derive the number of nonzeros per row from `csrRow[]`, then find the minimum and maximum number of nonzeros in a row, which will be used for filtering noise in Chapter 4.

Algorithm 6: Construction of the CSR format similarity matrix based on a chunkwise dense-to-CSR approach (Algo CSR-3)

Input:

(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$
(2) Similarity metric and connectivity parameters, e.g. σ, threshold
(3) Supposed maximum percentage of nonzeros: $spMaxNzPct$
(4) Usage rate of free GPU RAM for storing a chunk of similarity matrix: $memUseRate$

Output:

(1) Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`
(2) Number of nonzeros per row: `nnzPerRow[]`
(3) Minimum and maximum number of nonzeros in a row: $minNnzRow$, $maxNnzRow$
(4) Total number of nonzeros: nnz

1 Allocate memory for `csrVal[]` and `csrCol[]` according to $spMaxNzPct$.
2 Consider the horizontal partitioning of similarity matrix into chunks of similar size, and calculate the desired amount of memory (based on the size of free GPU RAM and $memUseRate$) used for storing a chunk of similarity matrix.
3 Determine automatically the number of chunks in the way that only one chunk can fit into the allocated GPU memory.
4 For each chunk of the similarity matrix:
  - launch the `chkPass` kernel to compute the similarity elements and store them in dense format;
  - perform the *denseToCSR* step to transform the matrix chunk from dense to CSR format and accumulate the number of nonzeros into nnz.
5 Perform the *mergeCSR* step to merge the CSR results of all chunks and obtain the CSR format of the whole similarity matrix.
6 Derive `nnzPerRow[]`, $minNnzRow$ and $maxNnzRow$ from `csrRow[]`.

GPU implementation

Our GPU implementation for Algo CSR-3 is detailed in Appendix E. Essentially, it consists in one optimized CUDA kernel: the `chkPass` kernel, and the use of some functions of NVIDIA's cuSPARSE library.

3.3.6 . Comparison of the three algorithms

Table 3.2: Comparison of our three GPU algorithms for constructing the similarity matrix in CSR format

	Algo CSR-1	**Algo CSR-2**	**Algo CSR-3**
Method feature	Straightforward	Ellpack-to-CSR	Chunkwise dense-to-CSR
Additional input	No	$hypoMaxNnzRow$ ($hypo$)	$spMaxNzPct$ (spp), $memUseRate$
# of computed similarities	$2n^2$	n^2 to $2n^2$	n^2
Supported sparsity pattern	All	All but regular sparsity patterns are preferred	All
GPU implementation	`1stPass` kernel + `2ndPass` kernel + Thrust `exclusive_scan`	`fullPass` kernel + `ellpackToCSR` kernel + `supPass` kernel + Thrust `exclusive_scan`	`chkPass` kernel + cuSPARSE APIs + Thrust `transform`
Size of arrays stored in GPU RAM	Input data arr.: $n \cdot d$ CSR arr.: $2 \cdot nnz + n + 1$ `nnzPerRow` arr.: $n + 1$	Input data arr.: $n \cdot d$ CSR arr.: $2 \cdot nnz + n + 1$ `nnzPerRow` arr.: $n + 1$ Ellpack arr.: $2n \cdot hypo$ Restart. idx arr.: $2n$	Input data arr.: $n \cdot d$ CSR arr.: $2n^2 \cdot spp + n + 1$ Chunk of matrix: $n \cdot nbRowPerChunk$ cuSPARSE workspace
Max required shared memory per block (in bytes)	$sizeof(float) \cdot Db.y \cdot Db.x$ + $sizeof(int) \cdot Db.y \cdot (Db.x+1)$	$sizeof(float) \cdot Db.y \cdot (Db.x + hypo)$ + $sizeof(int) \cdot Db.y \cdot (Db.x + hypo + 3)$	Unknown (due to the use of cuSPARSE)

Table 3.2 compares in many aspects our three algorithms for constructing the similarity matrix in CSR format on the GPU. Each algorithm has its own advantages and drawbacks compared to other two algorithms. Most importantly, Algo CSR-1 needs the most similarity computations but requires the least amount of GPU global memory and no extra parameter, while Algo CSR-3 needs the fewest similarity computations but may require most of the GPU global memory, and surely requires more than $2n^2$ accesses to global memory and two extra parameters. Algo CSR-2 can be regarded as a trade-off algorithm between the two previous algorithms, but

it requires the most efforts to be efficiently implemented. Besides, although it can support all kinds of sparsity patterns like the other two algorithms, it prefers regular sparsity patterns that are favorable to Ellpack format. Finally, it can require too much shared memory per block if the $hypoMaxNnzRow$ or the block y dimension is great.

Based on the above analysis and our experimental results in Section 3.6.4, we give some advice on how to choose among the three algorithms in practice:

- If the dataset has many dimensions (large d), which means it is expensive to compute each similarity based on the values of all dimensions, then we recommend using Algo CSR-3 or Algo CSR-2 as they compute much fewer similarities than Algo CSR-1.
- If the dataset has a huge number of instances (large n), then we suggest using Algo CSR-2, because it computes much fewer similarities than Algo CSR-1 and meanwhile requires much fewer accesses to global memory than Algo CSR-3.
- If the user does not want to tune any extra parameters, or if the user wants to acquire some initial knowledge of the similarity matrix (e.g. $minNnzRow$, $maxNnzRow$, nnz, sparsity) before running any faster algorithms, then Algo CSR-1 is the very choice.

We point out that we have considered whether it would be possible to exploit the symmetry property of similarity matrix to halve the similarity computations. Unfortunately, none of the above algorithms seem suitable to utilize the symmetry due to the complicacy of CSR format. Besides, we have also considered whether it would be easier and faster to first construct the similarity matrix in COO format and then convert it into CSR format. However, we found that similar restrictions and difficulties (see Section 3.3.2) would exist when using COO format. Moreover it would require an extra COO-to-CSR conversion and also more memory space for storing both COO and CSR results. Nevertheless, all our three algorithms above can be readily generalized to COO-format similarity matrix construction if necessary.

3.4 . Spectral graph partitioning using nvGRAPH

The previous section presents our optimized GPU algorithms for constructing CSR format similarity graph and matrix, which is an important step for scaling up spectral clustering. In this section, we concentrate on the graph partitioning step of spectral clustering. This step takes the similarity matrix as input graph and aims at partitioning the graph into balanced subgraphs (equivalent to data clusters) with minimum cut. This is known to be an NP-hard problem, but a relaxed and approximated solution is to compute the first few eigenvectors of the

graph Laplacian matrix and extract the partitioning information from the calculated eigenvectors.

In Section 1.5.2 we have investigated some eigensolver methods and GPU-accelerated libraries with eigensolvers. We are particularly interested in using NVIDIA's nvGRAPH library for spectral graph partitioning on the GPU. With the CSR format similarity matrix constructed in Section 3.3, the remaining steps of spectral clustering can be completed on the GPU by calling the "Spectral Clustering API" of the nvGRAPH library. The API supports two graph partitioning algorithms based on balanced cut minimization with embedded eigensolvers.

- **Minimization of the balanced cut with Lanczos method**. The balanced cut refers to the volume of inter-cluster connections relative to the size of clusters. The algorithm constructs the Laplacian matrix and then calls the Lanczos solver to calculate the smallest eigenpairs.

- **Minimization of the balanced cut with LOBPCG method**. Similar to the first algorithm, but it utilizes the LOBPCG eigensolver to handle the constructed Laplacian matrix.

Compared to Lanczos method, LOBPCG can handle eigenvalues with multiplicity [135] which often happens in spectral clustering. Moreover, the NVIDIA implementation of LOBPCG is able to restart the computation when it encounters numerical instabilities. Thus nvGRAPH's LOBPCG-embedded algorithm has appeared to be the most reliable on our benchmarks.

We emphasize that despite its name called by nvGRAPH, the API does not take care of similarity graph/matrix construction. It actually takes the similarity graph in CSR topology (equivalent to similarity matrix in CSR format) as input graph and performs spectral graph partitioning which includes several steps like Laplacian matrix computation, eigen-decomposition, and final k-means clustering (see Section 1.3.1 and Figure 1.1). Note that the nvGRAPH documentation [139] does not report which type of Laplacian matrix is constructed in the above algorithms. Besides, the API has also a modularity maximization algorithm for graph partitioning, which constructs a *modularity matrix* and finds its largest eigenpairs (while the balanced cut minimization algorithms construct the Laplacian matrix and find its smallest eigenpairs).

Besides, the API also offers a function for measuring the clustering quality with three supported metrics: modularity, edge cut, and ratio cut. The modularity metric tells how good the clustering is versus random assignments. The edge cut metric counts the total number of edges across clusters. The ratio cut metric accumulates for all clusters the ratio of the number of edges going outside of a cluster to the number of vertices inside the cluster. For the first metric, higher is better, while for the last two metrics, lower is better.

Listing 3.5 shows the usage of the API. Before invoking the `nvgraphSpectralClustering` function, we should first conduct some

preparation steps in sequence (lines 2-22): initialize the nvGRAPH library, create a graph descriptor, upload graph data in CSR format, and specify the parameters. The tolerance and the maximal number of iterations should be given appropriate values for both eigensolver and final k-means. They can affect the clustering quality and elapsed time. With all settings done, we call the `nvgraphSpectralClustering` function which partitions the similarity graph using spectral technique and returns cluster assignments of all vertices as well as the k_c smallest or largest eigenpairs (lines 25-26). Finally we can call the `nvgraphAnalyzeClustering` function to measure clustering quality (lines 29-31).

```
1  #include <nvgraph.h>
2  ...  // Omitted declarations
3  nvgraphHandle_t nvgHandle;    // Declare a handle of nvGRAPH library
4  nvgraphGraphDescr_t descrG;   // Declare a graph descriptor
5  nvgraphCreate(&nvgHandle);    // Initialize nvGRAPH library
6  nvgraphCreateGraphDescr(nvgHandle, &descrG); // Create graph descriptor
7
8  // Upload CSR-topology similarity graph
9  nvgraphCSRTopology32I_st CSR_input = {n, nnz, GPU_csrRow, GPU_csrCol};
10 nvgraphSetGraphStructure(nvgHandle, descrG, (void*)&CSR_input, ...);
11 nvgraphAllocateEdgeData(nvgHandle, descrG, ...);
12 nvgraphSetEdgeData(nvgHandle, descrG, (void*)GPU_csrVal, ...);
13
14 // Initialize parameters
15 struct SpectralClusteringParameter SC_params;
16   SC_params.n_clusters  = kc;          // Nb of clusters
17   SC_params.n_eig_vects = kc;          // Nb of eigenvectors
18   SC_params.algorithm = NVGRAPH_BALANCED_CUT_LOBPCG;
19   SC_params.evs_tolerance = 0.0001;   // Tolerance for eigensolver
20   SC_params.evs_max_iter  = 4000;     // Max nb of eigensolver iter.
21   SC_params.kmean_tolerance = 0.0001; // Tolerance for k-means
22   SC_params.kmean_max_iter  = 200;    // Max nb of k-means iterations
23
24 // Perform spectral graph partitioning
25 nvgraphSpectralClustering(nvgHandle,descrG,...,&SC_params,     //input
26                           GPU_labels,GPU_eigVals,GPU_eigVects);//output
27
28 // Measure clustering quality according to a metric (e.g. modularity)
29 nvgraphAnalyzeClustering(nvgHandle, descrG, ...,         // input
30                          GPU_labels, NVGRAPH_MODULARITY, // input
31                          &md_score);                     // output
```

Listing 3.5: Usage of the nvGRAPH spectral graph partitioning API

We point out that the API also has some limits: (1) it does not support directed graphs; (2) it supports only the CSR format for graph representation; (3) the supported maximum number of edges equals the maximum value for `int` type, which is about 2 billion in case of using 32 bits for `int`; (4) it only scales to single GPU. The first two limits have little effect on our current work, but the last two limits really prohibit us from advancing spectral clustering to even larger scale. The same limits exist for the corresponding APIs in cuGraph library.

3.5 . Tuning of parameters

Spectral clustering has the potential to produce high-quality clustering results. However, as summarized in Table 3.3, multiple parameters are introduced in the algorithms and implementations for spectral clustering. These parameters may affect the clustering quality or the algorithm performance and thus need to be tuned properly.

Some related works on the tuning or auto-tuning of spectral clustering parameters have been mentioned in Section 1.3.3. In this section, we describe our implementation for the auto-tuning of the number of clusters k_c based on the eigengap heuristic, and give some suggestions on the tuning of other parameters based on our practical experience.

Table 3.3: Parameters related to spectral clustering

Parameter	**Origin**	**Impact on clustering quality**	**Impact on algo perf.**	**Tuning difficulty**
# of clusters (k_c)	Spectral clustering	**High**	Low	**High**
Threshold for sparsification	Construction of the similarity matrix in CSR format	**High**	Medium	**High**
k for *k-nearest neighbor* graph		**High**	**High**	**High**
σ for Gaussian similarity		**High**	Low	**High**
$hypoMaxNnzRow$ in Algo CSR-2		No	**High**	Medium
$spMaxNzPct$ in Algo CSR-3		No	Low	Medium
$memUseRate$ in Algo CSR-3		No	Medium	Low
Tolerance	Iterative eigensolver	**High**	Medium	Medium
Max # of iter.		**High**	Medium	Low
Tolerance	Final k-means or k-means++	**High**	Medium	Low
Max # of iter.		**High**	Medium	Low

3.5.1 . Auto-tuning of the number of clusters

We have implemented the eigengap heuristic [188] (introduced in Section 1.3.3) for k_c auto-tuning as follows.

1. Define a maximal number of clusters denoted by k_{cmax} ($k_{cmax} > k_c$).

2. Leverage the nvGRAPH library (see Section 3.4) to calculate the k_{cmax} smallest eigenvalues and eigenvectors of graph Laplacian. Note that in this case the k-means step embedded in nvGRAPH's partitioning algorithms is no longer necessary. Thus we set the maximum number of k-means iterations to 1 (if this value is ≤ 0, nvGRAPH will use the default value 200).

3. Automatically determine k_c based on the eigengap heuristic.

Finally we run our own GPU implementation of the k-means++ algorithm to obtain k_c clusters.

3.5.2 . Tuning of the parameters for similarity matrix construction

Despite the related works presented in Section 1.3.3, the auto-tuning of connectivity parameters (e.g. k for k-*nearest neighbor* graph, ε for ε-*neighborhood* graph, σ for Gaussian similarity function) is difficult and remains an open issue. In this dissertation we manually tune connectivity parameters. Nevertheless, we suggest that performing min-max feature scaling (as formulated in Equation 3.1) before spectral clustering can facilitate the tuning of the parameters ε and σ, because it helps knowing in what range to choose the parameters.

Let X^l be a numeric feature containing n values $x_1^l, ..., x_n^l$ (column l of data matrix X), the min-max scaling transforms each value x_i^l of X^l as follows:

$$x_i^l \longleftarrow \frac{x_i^l - x_{min}^l}{x_{max}^l - x_{min}^l} \tag{3.1}$$

where x_{max}^l and x_{min}^l are the maximum value and minimum value of X^l, respectively. It can be found that the min-max scaling actually rescales the values of every feature/dimension to the same range $[0, 1]$.

We have introduced other parameters in the algorithms that have been proposed previously for constructing the similarity matrix in CSR format, including $hypoMaxNnzRow$ in Algo CSR-2, $spMaxNzPct$ and $memUseRate$ in Algo CSR-3. These three parameters have no effect on clustering quality, but have different degrees of impact on algorithm performance.

$hypoMaxNnzRow$

We suggest that $hypoMaxNnzRow$ in Algo CSR-2, which represents the hypothetical maximum number of nonzeros in a row of the similarity matrix, should be neither too small nor too large. As demonstrated experimentally in Section 3.6.4, a value of a few tens to a few thousands for $hypoMaxNnzRow$ would be acceptable in terms of performance.

$spMaxNzPct$

The cuSPARSE library [144] targets matrices with zero elements representing over 95% of the total entries (i.e. matrices with sparsity $> 95\%$). We consider it as a reference and suggest that $spMaxNzPct$ in Algo CSR-3, which represents the supposed maximum percent of nonzero elements in the similarity matrix, should be $< 5\%$. In fact, our experiments in Section 3.6.4 show that the constructed similarity matrices typically have $> 99\%$ sparsity, and larger similarity matrices are usually more sparse. Moreover, too much memory would be required if $spMaxNzPct$ is not small enough for large matrices. Thus, for datasets with a large number of instances, a small value under 1% for $spMaxNzPct$ is generally preferable.

$memUseRate$

The $memUseRate$ parameter in Algo CSR-3 refers to the usage rate of free GPU RAM for storing a chunk of similarity matrix in dense format. If $memUseRate$ is too small (e.g. under 10%), the chunk size would be too small; if $memUseRate$ is too large (e.g. over 90%), other steps of the algorithm may lack RAM. Our experiments in Section 3.6.4 show that a value in the range [50%, 80%] for $memUseRate$ is generally a nice choice, while a value in the range [10%, 50%] or [80%, 90%] also works well for some datasets.

3.5.3 . Tuning of the parameters for eigensolvers and k-means

Since we use nvGRAPH's LOBPCG-embedded algorithm which contains the LOBPCG eigensolver and the k-means algorithm (see Section 3.4), we need to specify the approximation tolerance and the maximum number of iterations for both the eigensolver and the k-means. It is declared in the nvGRAPH documentation [139] that:

- The smaller the tolerance, the better the approximation.
- For the tolerance related to eigensolvers, the default value is 0.001 while values between 0.01 and 0.0001 are generally acceptable, however setting a value of less than 0.0001 may result in divergence due to numerical roundoff.
- For the tolerance related to the k-means, the default value is 0.01 while values between 0.01 and 0.001 are usually acceptable.
- For the maximum number of iterations, the default value is 4000 for eigensolvers and 200 for the k-means.

Experimentally, we found that the tolerance for the LOBPCG eigensolver can have a significant impact on the clustering quality and therefore needs to tuned with special care, while it is generally good to use the default values for the other parameters.

3.6 . Experimental results

In this section, we experiment and evaluate our GPU implementation for spectral clustering. Specifically, Section 3.6.1 introduces the experimental framework including hardware and software, Section 3.6.2 describes the benchmark datasets and parameter settings, Section 3.6.3 presents the results of spectral clustering using dense data format, Section 3.6.4 shows the performance of the CSR format for the similarity graph/matrix construction, Section 3.6.5 shows the performance of nvGRAPH's LOBPCG-embedded algorithm for graph partitioning, and finally Section 3.6.6 gives the global performance of spectral clustering using CSR format.

3.6.1 . Experimental framework

Apart from the GPU algorithms and implementations for CSR graph/matrix construction, we have also developed a well optimized CPU implementation related to Algo CSR-1 as a baseline for performance comparison. It is parallelized with OpenMP for multi-threaded execution and has been designed to facilitate auto-vectorization with gcc for AVX units. To differentiate each implementation, we call them "CPU CSR-1", "GPU CSR-1", "GPU CSR-2" and "GPU CSR-3" in this section.

We take advantage of the LOBPCG-embedded algorithm from nvGRAPH for the graph partitioning step on GPU, as explained in Section 3.4. For performance comparison, we tried to find a CPU implementation of the LOBPCG eigensolver that could be used through C/C++ interface for spectral clustering[2], but unfortunately we could not find one until April 2022.

All experiments are performed on our *john3* server which consists of two Intel Xeon Silver 4114 processors as CPU and a NVIDIA GeForce RTX 3090 as GPU. The CPU code is compiled by gcc 9.3.0 with `-Ofast -funroll-loops -march=native` optimization flag and `-fopenmp` flag. The GPU code is compiled by nvcc of CUDA Toolkit 11.5[3] with `-gpu-architecture=sm_86`. More information about our testbed *john3* can be found in Appendix B. Computations are in single precision.

3.6.2 . Datasets and parameter settings

Table 4.1 summarizes the datasets and algorithmic parameter settings used in our experiments. The datasets can be divided into two groups according to their use cases:

[2]The Spectral Clustering API of *scikit-learn* [150] has a LOBPCG eigensolver, but the usage interface is in Python.

[3]A compilation warning reports that "libcusolver.so.10, needed by /usr/lib/gcc/x86_64-linux-gnu/../libnvgraph.so, may conflict with libcusolver.so.11" since the latest nvGRAPH library comes from CUDA 10.2 (see Section 1.5.2). Nonetheless, there is no runtime error and the results are correct.

Table 3.4: Datasets and parameter settings of our benchmarks

Dataset	**(n, d, k_c)**	**Similarity metric**	**Threshold**	**Supposed max. % of nonzeros for CSR-3**	**Tolerance for eigen-solver**
Jain	(373, 2, 2)	Gaussian($\sigma = 0.03$)	0 (sim.)	N/A	N/A
Aggregation	(788, 2, 7)	Gaussian($\sigma = 0.02$)	0.02 (sq. dist.)	N/A	N/A
S1	(5K, 2, 15)	Gaussian($\sigma = 0.03$)	0 (sim.)	N/A	N/A
Spirals	(7.5K, 2, 3)	Gaussian($\sigma = 0.01$)	0 (sim.)	N/A	N/A
MNIST-60K	(60K, 784, 10)	Cosine	0.8 (sim.)	1%	0.005
MNIST-120K	(120K, 784, 10)	Cosine	0.8 (sim.)	1%	0.005
MNIST-240K	(240K, 784, 10)	Cosine	0.8 | 0.84*(sim.)	1%	0.005
Syn4D-1M	(1M, 4, 4)	Gaussian($\sigma = 0.01$)	0.0008 (sq. dist.)	0.01%	0.001
Syn4D-5M	(5M, 4, 4)	Gaussian($\sigma = 0.01$)	0.0004 (sq. dist.)	0.001%	0.0001

* To obtain acceptable clustering quality, the threshold 0.8 is good for MNIST-60K and MNIST-120K, while a higher threshold (0.84) is required for MNIST-240K.

- **2D small datasets**: Jain, Aggregation, S1 and Spirals. Each of them contains only hundreds or thousands of points. They are used in Section 3.6.3 for spectral clustering using dense data format.
- **Large datasets**: MNIST-based sets (MNIST-60K, MNIST-120K and MNIST-240K) and 4D synthetic sets (Syn4D-1M and Syn4D-5M). The former sets contain tens to hundreds of thousands of instances in 784 dimensions, while the latter sets contain millions of instances in 4 dimensions. They are used in Sections 3.6.4, 3.6.5 and 3.6.6 for evaluating the performance of spectral clustering using CSR format.

More information about the benchmark datasets can be found in Appendix A.

In the beginning we perform feature scaling on the 2D sets and the 4D synthetic sets to transform every dimension into [0, 1], which facilitates the tuning of σ and thresholds. The feature scaling is efficiently implemented on GPU and its time overhead is negligible compared to the computations of spectral clustering.

We adopt the cosine similarity metric for the MNIST-based datasets because it is more effective than the Gaussian similarity for high-dimensional data (as introduced in [89] and as we verified empirically). In contrast, we use Gaussian similarity for other low-dimensional datasets.

We impose a lower bound threshold on the similarity value or an upper bound threshold on the squared distance to construct the *ε-neighborhood-like* graph and associated sparse similarity matrix (see Section 3.3.1). For comparison, we set the same threshold (0.8) on the similarity for all MNIST-based datasets. Using this threshold can result in good enough clustering quality for MNIST-60K and MNIST-120K, but a higher threshold (0.84, marked with *) is needed for MNIST-

240K to achieve similar clustering quality. For Syn4D-1M and Syn4D-5M, we set a threshold (0.0008 and 0.0004 respectively) on the squared distance.

Regarding GPU CSR-3, we suppose that the maximum percentage of nonzeros in the associated similarity matrix (in contrast to sparsity) is 1% for the MNIST-based datasets, 0.01% for Syn4D-1M and 0.001% for Syn4D-5M, based on some quick preliminary experiments. After allocating enough memory for the CSR arrays, we query the amount of remaining free GPU RAM via `cudaMemGetInfo` and allocate 80% of it for storing a chunk of similarity matrix.

For nvGRAPH's LOBPCG-embedded algorithm, several parameters need to be specified (see Listing 3.5). We simply set the maximal number of iterations to the nvGRAPH default value, i.e. 4000 for the LOBPCG eigensolver and 200 for the final k-means. However, we found that the approximation tolerance for the eigensolver needs to be tuned with care because it has a significant impact on the clustering quality and the execution time. Based on some preliminary experiments, we set it to 0.005 for the MNIST-based datasets, 0.001 for Syn4D-1M and 0.0001 for Syn4D-5M. Besides, the tolerance for the k-means algorithm is set to 0.0001 for all benchmarks.

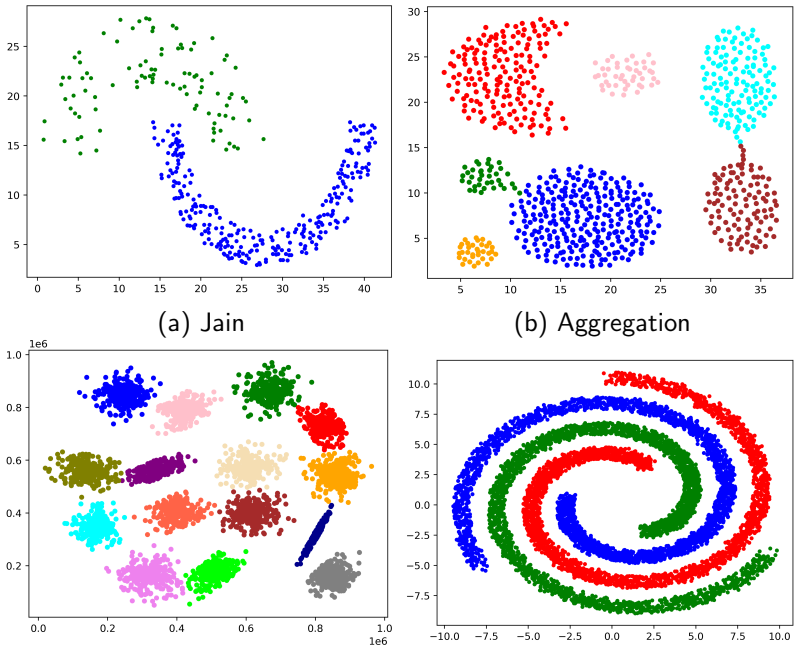
3.6.3 . Performance of spectral clustering using dense data format

Figure 3.5 and Table 3.5 show respectively the result and quality of spectral clustering using dense data format. As expected, spectral clustering can well discover different shapes of clusters (including non-convex ones) and yield satisfying clustering on the four shape sets. Table 3.6 presents the elapsed time breakdown in milliseconds. Since the datasets are small, we simply set the block size of all related kernels to (BSX, BSY)=(128, 2). The time of eigensolver includes the time to initialize the cuSolverDN library. It turns out that the eigensolver dominates the running time, while the matrix construction and the normalization consume little time. Other parts including CPU-GPU data transfers take some time but not much. The total time of spectral clustering is less than 1 second for all the four datasets.

Additionally, we find that the results of the `syevdx` eigensolver (i.e. eigenvalues and eigenvectors) are not exactly the same for each run with the same Laplacian matrix and input parameters. This sometimes leads to unstable clustering when using the k-means algorithm in the final step. However, using the k-means++ algorithm can usually handle this instability.

Table 3.5: Quality of spectral clustering on GPU using dense format

Dataset	ARI	AMI	NMI
Jain	1.000	1.000	1.000
Aggregation	0.987	0.982	0.982
S1	0.989	0.989	0.989
Spirals	1.000	1.000	1.000



(a) Jain (b) Aggregation

(c) S1 (d) Spirals

Figure 3.5: Spectral clustering on GPU using dense data format

Table 3.6: Time (ms) of spectral clustering on GPU using dense format

Dataset	Data transfers	Sim. Lap. constr.	Eigensolver syevdx	Norm.	Final k-means++* <# of iters>	Total
Jain	0.78	0.01	92.36	0.01	0.57 <2>	93.73
Aggregation	0.84	0.02	110.82	0.01	1.10 <3>	112.79
S1	2.91	0.02	441.71	0.01	5.84 <2>	450.49
Spirals	11.98	0.02	896.92	0.02	5.69 <2>	914.63

* The GPU implementation for the seeding step of k-means++ is displayed in Appendix G Listings G.1 & G.2.

3.6.4 . Performance of CSR format similarity matrix construction

Table 3.7 shows the characteristics of the similarity matrices constructed by any of our algorithms with the previously specified settings. It turns out that all the similarity matrices are extremely sparse (with sparsity greater than 99%) although they contain tens or hundreds of millions of nonzeros.

Tuning of the grid and block configuration

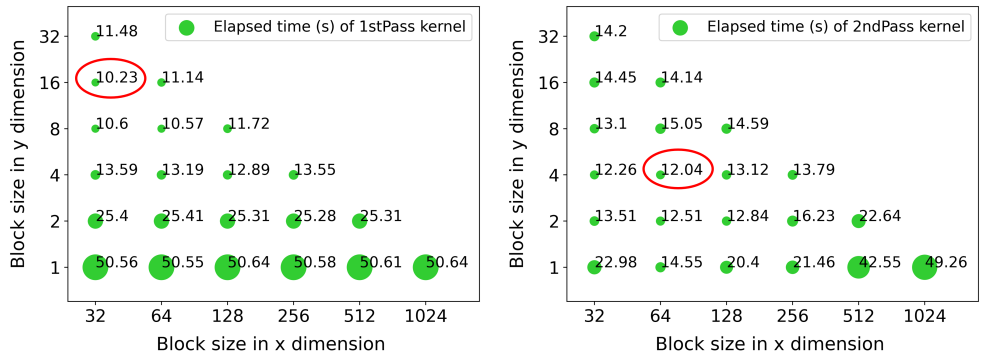
The configuration (i.e. dimension and size) of the grid and blocks of threads for a CUDA kernel can have a significant impact on the kernel performance. One of

Table 3.7: Characteristics of the constructed sparse similarity matrices

Dataset	Max nnz in a row	Avg. nnz per row	Total nnz	Sparsity (% of 0)
MNIST-60K	2196	251	15.1M	99.581%
MNIST-120K	3310	299	35.9M	99.751%
MNIST-240K	5552	478	114.8M	99.801%
MNIST-240K*	3520	199	47.8M	99.917%
Syn4D-1M	54	23	23.4M	99.998%
Syn4D-5M	64	29	149.9M	99.999%

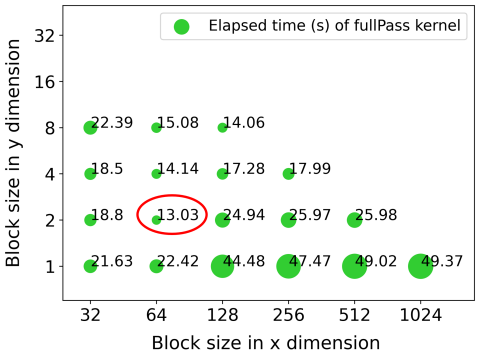
the suggestions [140] is that a grid should have sufficient number of blocks so that all multiprocessors of the GPU are kept busy, and meanwhile each multiprocessor should have multiple active blocks and sufficient number of active warps so as to hide latencies and keep the hardware busy. Although these suggestions are provided, it requires experiments to determine the optimal grid and block configuration of each kernel and for each dataset. Instead of creating only 1D grids with 1D blocks for all our CUDA kernels [78], now we choose to create 2D grids with 2D blocks for the `1stPass` kernel of GPU CSR-1 and the `chkPass` kernel of GPU CSR-3, and create 1D grids with 2D blocks for the other kernels (see details in Sections 3.3.3, 3.3.4 and 3.3.5).

We illustrate in Figures 3.6, 3.7 and 3.8 the impact of block size on kernel performance. Essentially, block sizes close to the optimal ones are often sub-optimal choices, while block sizes far from the optimal ones may lead to about $\times 2$ to $\times 5$ lower kernel performance.

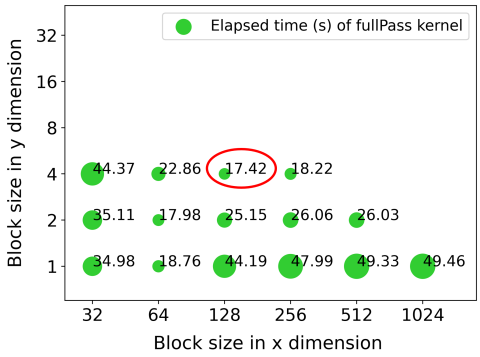


(a) `1stPass` kernel (optimal: x=32, y=16) (b) `2ndPass` kernel (optimal: x=64, y=4)

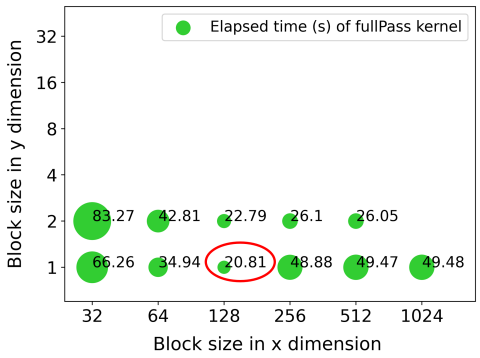
Figure 3.6: Block size impact on the kernels of GPU CSR-1 with MNIST-120K



(a) `fullPass` kernel with $hypo = 512$ (optimal: x=64, y=2)

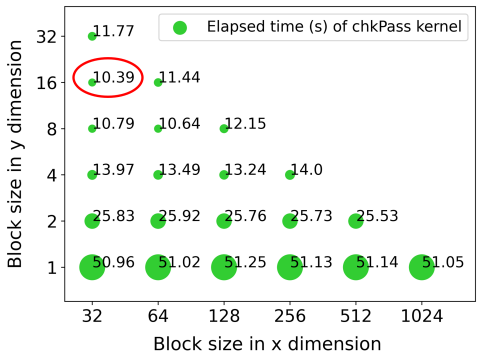


(b) `fullPass` kernel with $hypo = 1024$ (optimal: x=128, y=4)



(c) `fullPass` kernel with $hypo = 2048$ (optimal: x=128, y=1)

Figure 3.7: Block size impact on the `fullPass` kernel of GPU CSR-2 with MNIST-120K



(a) `chkPass` kernel (optimal: x=32, y=16)

Figure 3.8: Block size impact on the kernel of GPU CSR-3 with MNIST-120K

Table 3.8: Optimal block size configuration of our CUDA kernels for the CSR format similarity matrix construction

Dataset	GPU CSR-1 (BSX, BSY)		GPU CSR-2 (BSX, BSY) <hypo>		GPU CSR-3 (BSX, BSY)
	`1stPass` kernel	`2ndPass` kernel	`fullPass` kernel	`supPass` kernel	`chkPass` kernel
MNIST-60K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (N / A) <2048>	(32, 16)
MNIST-120K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
MNIST-240K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 4) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
MNIST-240K*	(32, 16)	(64, 4)	(64, 2) <512> (256, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
Syn4D-1M	(32, 8)	(32, 4)	(32, 4) <16> (32, 4) <32> (32, 4) <54>	(128, 1) <16> (128, 1) <32> (N / A) <54>	(32, 16)
Syn4D-5M	(32, 8)	(32, 4)	(32, 4) <16> (32, 4) <32> (32, 4) <64>	(128, 1) <16> (128, 1) <32> (N / A) <64>	(32, 16)

[1] BSX and BSY represents the block size in x and y dimensions, respectively.
[2] (N / A), short for Not Applicable or Not Available, represents that with the given hypo, the `supPass` kernel consumes little time regardless of the block size, or the `supPass` kernel is not involved in the computation.

Table 3.8 shows the optimal block size in x and y dimensions that we found experimentally for each kernel and each benchmark. It can be observed from the table that:

- Although the optimal block sizes are different for different kernels, they are similar for datasets of the same category (which is quite user-friendly), and all have at least 128 threads per block.
- As expected, we need to reduce the size of block y dimension for the `fullPass` kernel when $hypoMaxNnzRow$ (abbr. hypo) becomes large leading to proportionally more shared memory consumption per block (see Section 3.3.5 for explanation).
- Due to the $hypoMaxNnzRow$ parameter, it requires more efforts to find the optimal block size configuration for the kernels of GPU CSR-2 than for the other two CSR algorithms. However, as will be presented later, these efforts will be rewarded with superior performance in many benchmarks.

- Particularly, the optimal block size for the `chkPass` kernel of GPU CSR-3 is constant (BSX=32, BSY=16) for all benchmarks.

Tuning of the $hypo$ parameter for GPU CSR-2

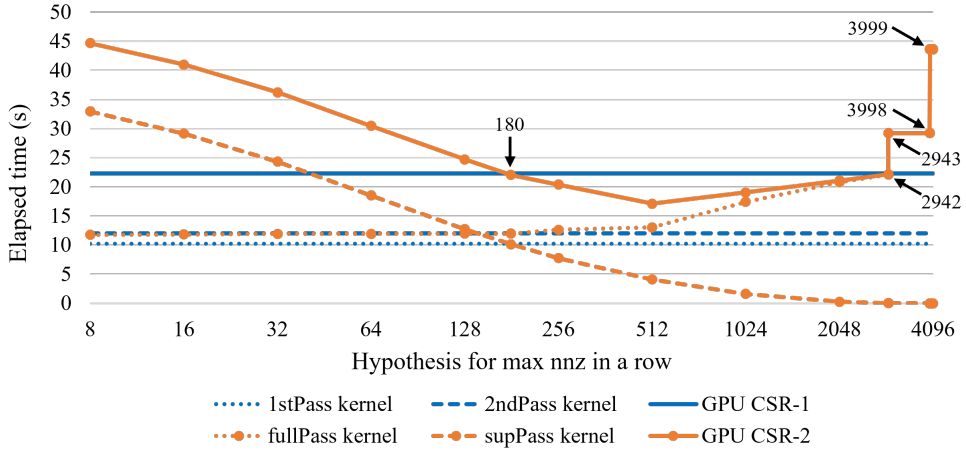
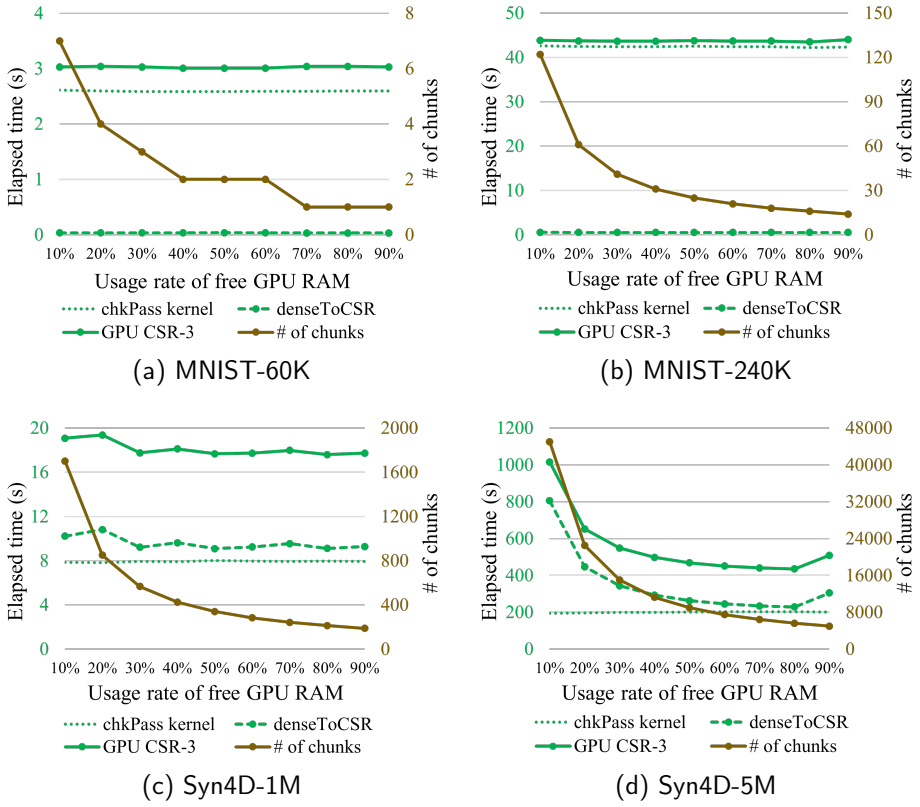


Figure 3.9: Performance comparison of GPU CSR-1 vs. GPU CSR-2 on MNIST-120K

As shown in Figure 3.9, we consider the MNIST-120K benchmark as an example for studying the performance of each kernel of GPU CSR 1 and 2, especially the impact of $hypoMaxNnzRow$ on the performance of GPU CSR-2. Note that the $hypoMaxNnzRow$ is irrelevant to the `1stPass` and `2ndPass` kernels of GPU CSR-1. The optimal block sizes for the `fullPass` and `supPass` kernels of GPU CSR-2 vary gradually with $hypoMaxNnzRow$, so the performance presented in the figure for each value of $hypoMaxNnzRow$ is obtained with the corresponding optimal block size. Besides, the `ellpackToCSR` kernel of GPU CSR-2 consumes so little time compared to the other kernels that we have omitted it in the figure.

The `fullPass` kernel of GPU CSR-2 always computes the n^2 similarities regardless of the $hypoMaxNnzRow$ parameter. The computation of similarities is the most time-consuming part of the kernel, so its execution time should remain constant as seen in Figure 3.9 up to a hypothesis of 512. However, this kernel allocates shared memory in a quantity proportional to the hypothesis. For hypotheses higher than 512, the amount of shared memory required by each block limits the number of blocks residing simultaneously in a Stream Multiprocessor, and causes an increase in the computation time, as shown in Figure 3.9. In particular, we observe two sudden increases of time when the hypothesis grows from 2942 to 2943 and from 3998 to 3999. On the contrary, the `supPass` kernel recomputes on each row only the similarities beyond the hypothesis and its execution time decreases when the hypothesis increases. Finally, there is a range of hypothesis values for

which the total time of GPU CSR-2 is lower than GPU CSR-1 ([180-2942] on our measurements in Figure 3.9). GPU CSR-2 can therefore run faster but requires some tests to identify the interesting hypothesis range.



(a) MNIST-60K (b) MNIST-240K

(c) Syn4D-1M (d) Syn4D-5M

Figure 3.10: Performance evolution of GPU CSR-3

Tuning of the dense matrix chunk size for GPU CSR-3

As shown in Figure 3.10, we investigate the performance breakdown of GPU CSR-3 on four representative benchmarks, and the impact from the usage rate of free GPU RAM allocated for a chunk of similarity matrix (see explanation in Section 3.6.2). Note that we consider only the range from 10% to 90% for the free memory usage rate because it is meaningless to use a too small rate and it is necessary to leave some memory for other uses (e.g. the workspace needed by cuSPARSE functions). Recall that GPU CSR-3 consists of some initialization steps, a loop of the `chkPass` kernel and the *denseToCSR* step, and finally the *mergeCSR* step (see Section 3.3.5). For all benchmarks, the initialization steps take only 0.34 second and the *mergeCSR* step costs little time compared to the global time, so they are not presented in the figure.

It is normal to see that the number of chunks decreases with more memory usage rate, while the number of chunks increases with datasets of large size n. For the MNIST-based datasets, the performance of GPU CSR-3 (including its `chkPass` kernel and *denseToCSR* step) is very stable when the memory usage rate varies. The elapsed time is dominated by the `chkPass` kernel while the *denseToCSR* step consumes little time. For Syn4D-1M and Syn4D-5M, the kernel performance is also very stable as the memory usage rate changes. However, the execution time of the *denseToCSR* step becomes significant and less stable on Syn4D-1M and makes the time of GPU CSR-3 less stable. This instability becomes more severe on Syn4D-5M. Specifically, the performance deteriorates when the memory usage rate decreases under 30% for Syn4D-1M, and under or over 80% for Syn4D-5M. This performance deterioration in case of decreasing memory usage rate is due to the growing initialization overhead in the *denseToCSR* step (implemented by cuSPARSE functions), since the number of chunks grows rapidly and the *denseToCSR* step is performed once for each chunk. In any case, a free memory usage rate of 80% seems to be the best choice, while a rate between 50% and 80% is also a fairly good choice.

GPU vs. CPU performance of CSR matrix construction

Table 3.9: Performance of the similarity matrix construction in CSR format

Dataset	CPU CSR-1 (s)			GPU CSR-1 (s)	GPU CSR-2 (s)			GPU CSR-3 (s)
	1 thr.	20 thr.	40 thr.		<1st hypo>	<2nd hypo>	<3rd hypo>	
MNIST-60K	1815	146	**74**	5.53	3.91 <512>	4.49 <1024>	5.42 <2048>	**3.04**
MNIST-120K	7427	590	**301**	22.27	17.12 <512>	19.05 <1024>	21.09 <2048>	**11.09**
MNIST-240K	28293	2502	**1266**	91.47	77.78 <512>	83.07 <1024>	85.29 <2048>	**43.84**
MNIST-240K*	29520	2650	**1244**	91.05	62.71 <512>	59.03 <1024>	72.52 <2048>	**43.56**
Syn4D-1M	2845	194	**151**	14.01	7.71 < 16>	5.71 < 32>	**5.66** < 54>	17.62
Syn4D-5M	too long	5772	**3928**	363.69	242.69 < 16>	176.06 < 32>	**145.89** < 64>	435.76

Table 3.9 compares the performance of CPU and GPU algorithms on different datasets. Each result on GPU is the average time of 5 consecutive runs, while each result on CPU is the rounded time of a single run as it takes much longer. Since our CPU has 20 physical cores (40 logical cores), we measured the performance of CPU CSR-1 running 1, 20 and 40 threads. In particular, the time of CPU CSR-1 using 1 thread is too long (over 20 hours) on the Syn4D-5M dataset so we did not get the final time. Figure 3.11 visualizes the speedup of GPU algorithms versus the best performance of CPU CSR-1 (using 40 threads and auto-vectorization).

Globally, it can be seen that multi-threading accelerates significantly CPU CSR-1, however, it is still much slower than any of the GPU algorithms. Compared

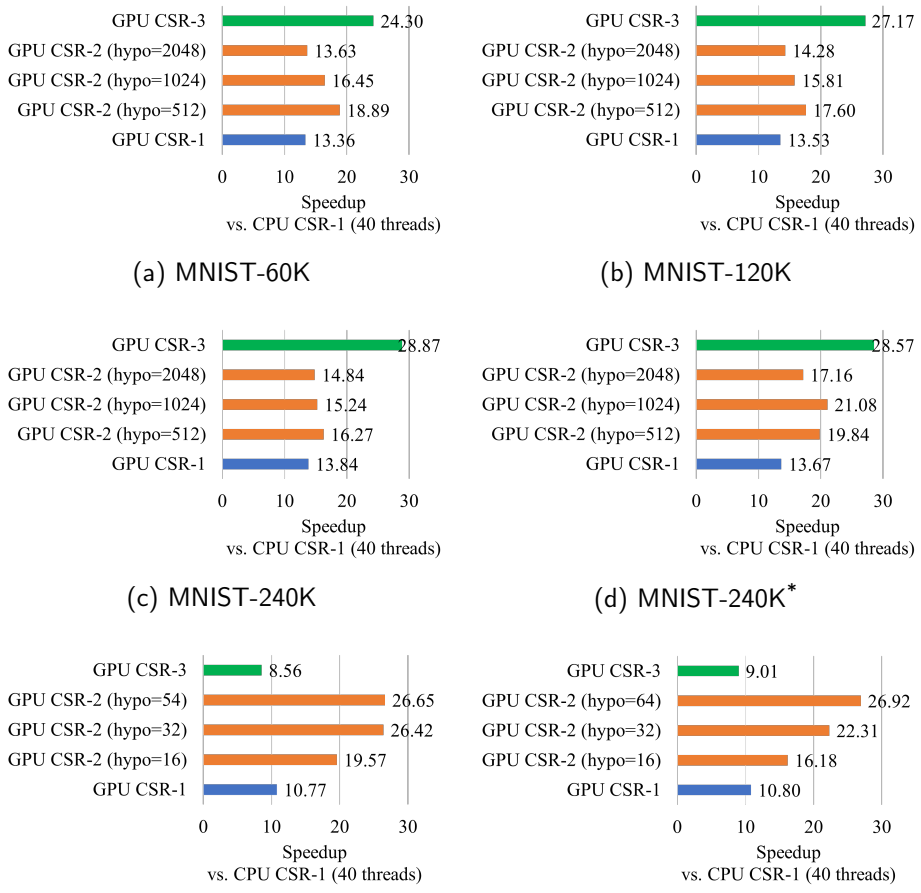
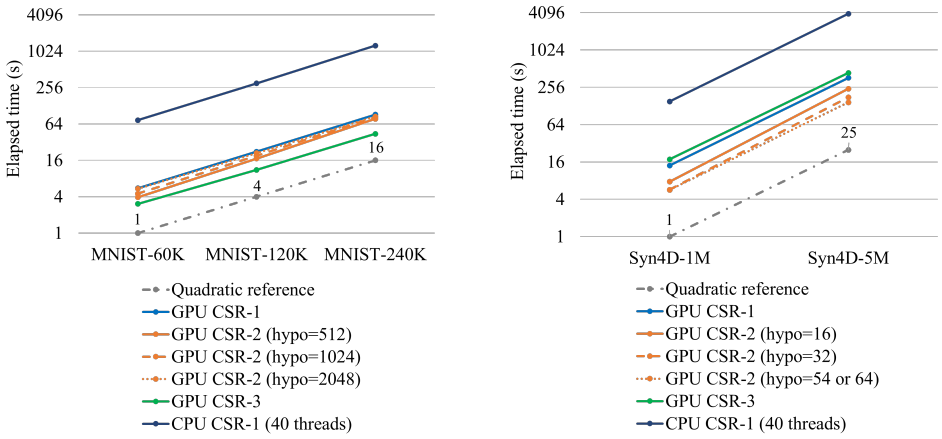


Figure 3.11: Speedup of the CSR format similarity matrix construction on GPU vs. CPU CSR-1 using 40 threads

to the best performance of CPU CSR-1, GPU CSR-1 is $\times 10.8$ to $\times 13.8$ faster, depending on $hypoMaxNnzRow$ GPU CSR-2 can be $\times 13.6$ to $\times 26.9$ faster, and GPU CSR-3 is $\times 8.6$ to $\times 28.9$ faster.

With the chosen values for $hypoMaxNnzRow$, GPU CSR-2 can outperform GPU CSR-1 and this superiority is especially significant on MNIST-240K*, Syn4D-1M and Syn4D-5M benchmarks. This is because the gain from reducing similarity computations surpasses the cost of recording restart indexes for GPU CSR-2 (see Section 3.3.4).

Compared to the other GPU algorithms, GPU CSR-3 is around $\times 2$ faster on the MNIST-based datasets but is significantly slower on Syn4D-1M and Syn4D-5M. In fact, GPU CSR-3 computes relatively much fewer similarities (n^2 instead of $1\times$ to $2 \times n^2$), but requires many extra global memory accesses (n^2 writes and n^2 reads). On the MNIST-based datasets which have numerous dimensions leading to long



(a) (b)

Figure 3.12: Scalability of the similarity matrix construction in CSR format

similarity computations, GPU CSR-3 achieves a speedup. However, on Syn4D-1M and Syn4D-5M datasets which have only 4 dimensions and million-scale n, it reaches less performance than the other GPU algorithms.

With logarithmic scales for both axes t and n, Figure 3.12 shows straight lines with slopes close to 2, meaning that the elapsed time varies quadratically with n for all the CPU and GPU algorithms. Hence all the algorithms follow the $\mathcal{O}(n^2 d)$ time complexity of similarity matrix construction, although in CSR format. They are all scalable to large datasets, but our GPU algorithms on a GeForce RTX 3090 are considerably faster than the parallelized and auto-vectorized CPU algorithm on a dual Xeon Silver 4114.

3.6.5 . Performance of nvGRAPH's LOBPCG-embedded algorithm

Table 3.10: Clustering quality and elapsed time of nvGRAPH's LOBPCG-embedded graph partitioning algorithm (based on 10 runs)

Dataset	Clustering quality			Time of nvGRAPH (s)		
	ARI	AMI	NMI	Min.	Max.	Average
MNIST-60K	0.44	0.66	0.66	2.30	3.34	**2.88**
MNIST-120K	0.50	0.67	0.67	3.48	4.59	**3.95**
MNIST-240K*	0.56	0.73	0.73	4.41	5.90	**5.01**
Syn4D-1M	1.00	1.00	1.00	3.63	5.18	**4.08**
Syn4D-5M	1.00	1.00	1.00	17.67	19.15	**18.25**

After obtaining the CSR format similarity matrix, we leverage the LOBPCG-embedded graph partitioning algorithm of the nvGRAPH library to fulfill the remaining steps of spectral clustering on the GPU (see Section 3.4). Table 3.10

presents the elapsed time of the nvGRAPH algorithm and the final clustering quality measured by three commonly used metrics: Adjusted Rand Index (ARI), Adjusted Mutual Information (AMI), Normalized Mutual Information (NMI) (introduced in Section 1.1.2). All three metrics return a score less or equal to 1, and a score closer to 1 indicates a better clustering. The results in the table are based on 10 runs.

For the MNIST-based datasets which are gray-scale digit images of $28 \times 28 = 784$ pixels, the ARI, AMI, and NMI scores achieved by our spectral clustering implementation are around 0.5, 0.7, 0.7, respectively. Although kind of far from 1, they are normal results that can be achieved by traditional spectral clustering algorithms on high-dimensional image datasets. In fact, the NMI score is close to that obtained in [207] by traditional spectral clustering algorithm and is better than that obtained by the k-means algorithm. The clustering quality on Syn4D-1M and Syn4D-5M is perfect since they are formed by convex clusters and are easy to be correctly clustered by spectral clustering.

For all benchmarks we observed a certain degree of performance fluctuations, but globally we are satisfied with the performance of nvGRAPH's LOBPCG-embedded algorithm. Although the theoretical time complexity of eigenvector computation is $\mathcal{O}(n^3)$ in the worst case, our experiments exhibit a low time complexity close to $\mathcal{O}(log(n))$ on the MNIST-based datasets and close to $\mathcal{O}(n)$ on Syn4D-1M and Syn4D-5M datasets. We infer there are two reasons for this good performance. First, the constructed similarity matrices are extremely sparse and the numerous matrix-vector multiplications of the LOBPCG eigensolver are efficiently performed in CSR format. Second, the LOBPCG solver adopts an iterative and approximate method instead of expensive direct methods. Note that the time of initializing the nvGRAPH library takes about 0.7 to 1 second with CUDA 11.5, and it is not included in the performance measurements.

Additionally, our experiments in Section 5.3.2 show that NVIDIA's LOBPCG-embedded graph partitioning algorithm (on GPU) runs significantly faster than that of *scikit-learn* (on CPU) when the number of instances to be processed is large enough, e.g. a speedup from $\times 8$ to $\times 28$ when processing 10^4 instances.

3.6.6 . Global performance of spectral clustering using CSR format

Figure 3.13 presents the global performance of spectral clustering on the GPU, consisting in the best performance of CSR-format similarity matrix construction (achieved by one of the three algorithms) and the performance of nvGRAPH's LOBPCG-embedded algorithm. The similarity matrix construction appears to be the most time-consuming part of spectral clustering especially on MNIST-120K, MNIST-240K* and Syn4D-5M, mainly due to its $\mathcal{O}(n^2 d)$ time complexity. The elapsed time consumed by the LOBPCG-embedded algorithm appears to take the

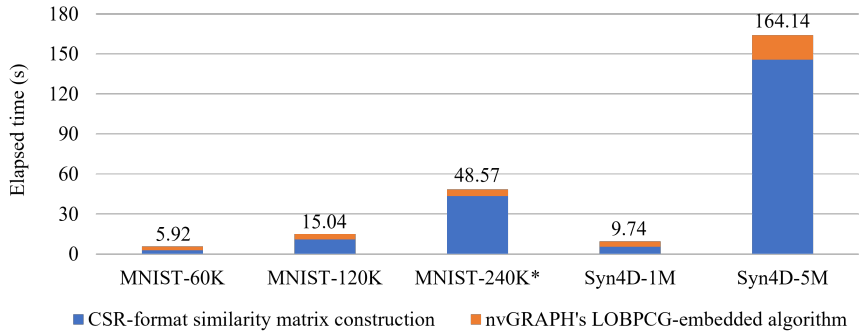


Figure 3.13: Performance of spectral clustering on GPU using CSR format

second place[4]. Data transfers between the CPU and the GPU are performed with pinned memory to achieve higher bandwidth and they occur only at the beginning and end of the program. Their elapsed time is negligible (less than 0.2 s) for all benchmarks and is therefore not included in the figure.

Globally, with our optimized algorithms for CSR graph/matrix generation and nvGRAPH's graph partitioning algorithm, we obtain a parallel implementation of spectral clustering that is able to process large datasets entirely on the GPU in just a few seconds to a few minutes.

3.7 . Summary

In this chapter we have addressed the scalability of spectral clustering on GPU architectures. We have proposed three different algorithms and optimized parallel implementations for the construction of the sparse similarity graph/matrix in CSR format. Storing this matrix in CSR format enables us to save a large amount of memory space compared to the dense format storage, which is crucial on the GPU since it usually provides much less memory than the CPU. Furthermore, our GPU implementations of these algorithms are deeply optimized by applying various high-level and low-level good practices of CUDA programming (e.g. coalesced access to global memory, full exploitation of the shared memory, maximization of hardware utilization, minimization of warp divergence, use of fast arithmetic instructions, etc).

Moreover, our matrix generation in CSR format is ideally suited to the graph partitioning algorithms provided by nvGRAPH which require the input graph to be in CSR format. These algorithms possess built-in and adapted eigensolvers (including Lanczos and LOBPCG) and a k-means implementation, so they can be leveraged to accomplish the remaining steps of spectral clustering. We particularly favor the LOBPCG-embedded algorithm because the LOBPCG solver can handle

[4]However, experiments in Section 5.3.2 show that the LOBPCG-embedded algorithm takes more time than the similarity matrix construction on other benchmark datasets.

eigenvalues with multiplicity which often occur in spectral clustering, and it is numerically more stable then the Lanczos solver.

With our algorithms for CSR graph/matrix construction and nvGRAPH's eigensolver-embedded partitioning algorithm, we have obtained a parallelized end-to-end spectral clustering implementation on a single GPU. Finally, experiments show that our GPU implementation on a GeForce RTX 3090 succeeds in scaling up to millions of data instances.

4 - Parallel and Efficient Noise Filtering for Spectral Clustering

4.1 . Introduction

As stated in Section 1.3.3, spectral clustering is sensitive to noise points (including outliers) which are widely present in many datasets. Their existence can destroy the block structure of the similarity matrix [117] and thus have a significant negative impact on the clustering quality.

To address this problem, we propose two simple and effective noise filtering approaches in Sections 4.2 and 4.3, which exploit the ready-made similarity matrix in CSR format constructed by the algorithms of Section 3.3. One filtering approach is based on the number of nonzeros per row, and the other is based on vertex degree. We point out that although we independently devised both approaches on our own, we found afterwards that our second approach is actually equivalent to the idea suggested by Hennig et al. [81]. However, we provide a strategy to help find the optimal filtering threshold. Moreover, our filtering implementation is parallel and works on the CSR format of similarity matrix. In Section 4.4, we give an algorithm for noise robust spectral clustering which exploits either of the two noise filtering approaches, and we present our efficient GPU implementation for this algorithm. Finally, experiments in Section 4.5 demonstrate the effectiveness and efficiency of our noise filtering implementation.

4.2 . Noise filtering based on nnz per row

According to the background introduced in Section 1.3.1, the nonzeros in row i of the similarity matrix (associated with ε-*neighborhood* graph) can be regarded as the similarities between instance i and other instances within its neighborhood of radius ε. So the number of nonzeros per row (`nnzPerRow[]`) can be regarded as the number of ε-*neighborhood* neighbors per instance. We assume that noise instances usually have much fewer ε-*neighborhood* neighbors than non-noise instances. Therefore, with an appropriate threshold on `nnzPerRow[]`, it is feasible to separate noise instances from non-noise instances.

This filtering approach should be effective if the neighborhood radius ε is not too large. Otherwise noise instances may have as many neighbors as non-noise instances and the filtering approach will lose its effectiveness. Obviously this filtering approach is inapplicable to the similarity matrix associated with k-*nearest neighbor* graph where every instance has k neighbors.

Interestingly, we found that our noise filtering approach based on nnz per row

has a connection with a famous density-based clustering method called DBSCAN[1]. With two predefined parameters: ε (neighborhood radius) and *MinPts* (a minimum number of points in ε-neighborhood), DBSCAN [53] finds each point that has at least *MinPts* neighbors in its ε-neighborhood as a *core point*, and forms clusters based on the connectivity of *core points* and the reachability to *non-core points*. The points that do not belong to any cluster are identified as noise. Thus each noise point has fewer than *MinPts* neighbors in its ε-neighborhood, but note that a point with fewer than *MinPts* neighbors in its ε-neighborhood is not necessarily a noise point because it may be a *border point* of a cluster. Nevertheless, our noise filtering approach based on nnz per row assumes that most border points of a cluster still have more neighbors than noise points.

4.3 . Noise filtering based on vertex degree

Recall that a data instance corresponds to a graph vertex, and the similarities between different instances correspond to the connections/edges between different vertices. Each similarity value equals an edge weight. As defined in Reference [188], the degree of a vertex is the weight sum of all the edges connected to the vertex, i.e. the sum of similarities in the corresponding row of the similarity matrix[2]. We assume that noise vertices usually have much fewer edges and probably smaller edge weights than non-noise vertices, which means noise vertices generally have much smaller degrees than non-noise vertices. Therefore, given an appropriate threshold on the degrees of vertices (`deg[]`), we can separate noise vertices from non-noise vertices.

Unlike the previous filtering approach which is based on `nnzPerRow[]` and only applicable to ε-*neighborhood* similarity graph, this filtering approach based on `deg[]` is suitable to both ε-*neighborhood* and k-*nearest neighbor* graphs. In case of ε-*neighborhood* graph, this filtering approach should be less sensitive to a large value of ε than the previous approach because farther vertex neighbors should have smaller edge weights and thus less impact on the vertex degree.

4.4 . Noise robust spectral clustering on GPU

Algorithm

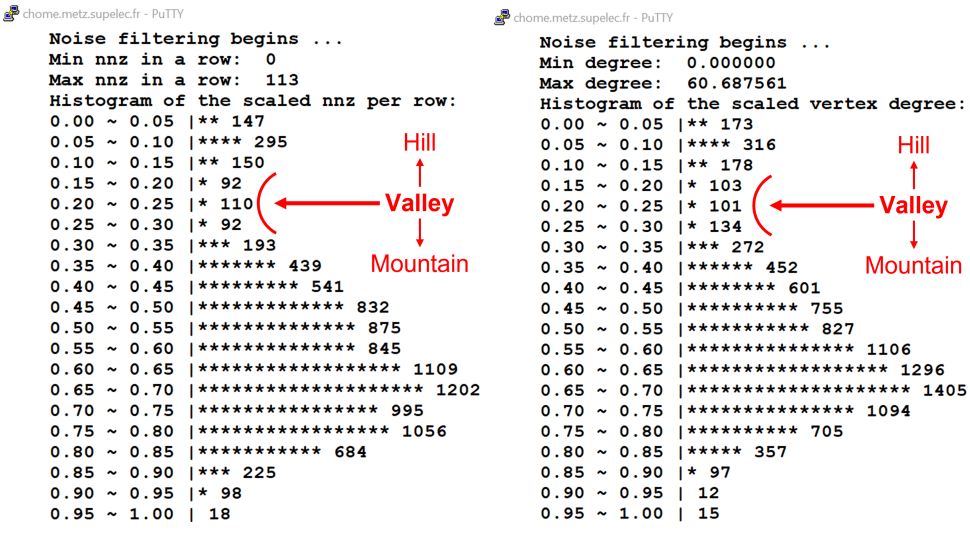
Algorithm 7 describes our noise robust spectral clustering that incorporates noise filtering either based on nnz per row (see Section 4.2) or based on vertex degree (see Section 4.3). First we construct the similarity matrix in CSR format via the

[1]Also, an interesting relationship between spectral clustering and DBSCAN have been discovered in References [127, 170]

[2]Note that this definition given by Reference [188] is different from the general definition in graph theory [44], where the degree of a vertex is generally defined as the number of edges connected to the vertex. Nevertheless, we use the former in this dissertation.

algorithms proposed in Section 3.3. Then we get the number of nonzeros per row (`nnzPerRow[]`) or compute the degrees of vertices (`deg[]`) depending on the employed approach for noise filtering. As the values of `nnzPerRow[]` or `deg[]` can vary considerably with different datasets, we transform them into the bounded range [0, 1] by min-max scaling (introduced in Section 3.5.2) and reveal their distribution in [0, 1] by a histogram.

Two examples of the histogram are shown in Figure 4.1. It is generally reasonable to assume that noise instances are small in quantity compared to non-noise instances and they have relatively few neighbors or small degrees. Hence it is likely to find a distinct boundary between noise and non-noise instances in the histogram. For example, in Figure 4.1 (a), the "hill" in the small upper area of the histogram are actually formed by noise instances as they have small values of scaled nnz per row, while the "mountain" in the large middle and lower area of the histogram are formed by non-noise instances with relatively high values of scaled nnz per row. Thus the optimal threshold for filtering noise are usually located at the "valley" area between the "hill" and the "mountain". Such feature also exists in the histogram of scaled degrees of vertices in Figure 4.1 (b). We developed a method that tries to automatically estimate the optimal threshold for noise filtering (denoted by $etholdNF$) based on the feature of the histograms, however our current auto-estimation method is not mature and generic enough to handle various cases and remains to be improved in future work. Nonetheless, we can easily choose a good threshold based on the observation of the histogram in interactive mode.



(a) Histogram of scaled nnz per row (b) Histogram of scaled degrees of vertices

Figure 4.1: Histogram examples obtained on the Cluto_t7 dataset

Let us denote by $tholdNF$ the noise filtering threshold that is finally used on the scaled `nnzPerRow[]` or `deg[]`. Based on $tholdNF$, we can readily

identify noise instances and label them with "-1". Then we remove all similarity elements (in both row and column directions) that are related to noise instances from the CSR format similarity matrix and we obtain the noise-free similarity matrix in CSR format. We regard all noise instances as one cluster and assume the defined number of clusters (k_c) has counted the noise cluster. So, with the obtained noise-free similarity matrix in CSR format and the number of noise-free clusters $k_c - 1$, we can continue the spectral clustering process (e.g. spectral graph partitioning using nvGRAPH in Section 3.4) to find the clusters labels (from 0 to $k_c - 2$) of non-noise instances. Note that the indexes of non-noise instances associated with the noise-free similarity matrix differ from their original indexes associated with the original $n \times n$ similarity matrix. Therefore, we need to finally recover the labels of non-noise instances that are associated with the original instance indexes.

Algorithm 7: Noise robust spectral clustering

Input:
(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$
(2) Number of clusters k_c (including 1 noise cluster)
(3) Parameters needed for similarity matrix construction

Output: Cluster labels of n data instances with the "-1" label for noise

1 Construct the similarity matrix in CSR format by one of the algorithms proposed in Section 3.3.
2 **if** *filter noise based on nnz per row* **then**
3 Get the number of nonzeros per row (`nnzPerRow[]`).
4 Scale the elements of `nnzPerRow[]` into the range [0, 1] and compute the histogram of scaled `nnzPerRow[]`.
5 **else if** *filter noise based on vertex degree* **then**
6 Compute degrees of vertices (`deg[]`), i.e. sum of elements in each row of the similarity matrix (see Section 1.3.1).
7 Scale the elements of `deg[]` into the range [0, 1] and compute the histogram of scaled `deg[]`.
8 Automatically estimate the optimal threshold for noise filtering ($etholdNF$) based on the histogram (ONGOING work).
9 **if** *automatic mode* **then**
10 Use $etholdNF$ as the final threshold for filtering noise ($tholdNF$).
11 **else if** *interactive mode* **then**
12 Print the histogram and $etholdNF$, let the user determine the final threshold for filtering noise ($tholdNF$).
13 Identify noise based on $tholdNF$ and set their cluster labels to "-1".
14 Remove noise-related elements from the similarity matrix in CSR format.
15 Perform the subsequent steps of spectral clustering on the noise-free similarity matrix and find $k_c - 1$ noise-free clusters.
16 Find the labels of non-noise instances indexed in the original dataset.

GPU implementation

Our GPU implementation for Algorithm 7 is detailed in Appendix F. Essentially, the noise filtering part consists in some CUDA kernels and the use of some Thrust APIs, while other parts use the implementations presented in Chapter 3.

4.5 . Experimental results

In this section, we experimentally evaluate the performance of our noise robust spectral clustering algorithm on GPU.

4.5.1 . Datasets and parameter settings

The benchmark datasets are four noisy 2D datasets: Compound, Cure_t2, Cluto_t8, Cluto_t7. Their features and associated algorithmic parameter settings are presented in Table 4.1. Note that noise points are regarded together as one noise cluster in each of the four datasets. The experiments are performed on our *john3* server which consists of two Intel Xeon Silver 4114 processors as CPU and a NVIDIA GeForce RTX 3090 as GPU. More information about the benchmark datasets and our testbed *john3* can be found in Appendix A and B, respectively. Computations are in single precision.

In the beginning we perform min-max feature scaling on each dataset to facilitates the tuning of σ and similarity threshold. Then, the Gaussian similarity metric with $\sigma = 0.02$ is used for all datasets, and a lower bound threshold is imposed on the similarity value to construct the ε-*neighborhood-like* graph and associated sparse similarity matrix (see explanation in Section 3.3.1). Besides, except that the tolerance for eigensolver is set to 0.001 for all datasets, other parameters of nvGRAPH's LOBPCG-embedded algorithm are set to the same values as in Section 3.6.2.

Table 4.1: Datasets and parameter settings

Dataset	**(n, d, k_c)**	**Similarity metric**	**Threshold**	**Tolerance for eigensolver**
Compound	(399, 2, 6)	Gaussian($\sigma = 0.02$)	0.02 (sim.)	0.001
Cure_t2	(4.2K, 2, 7)	Gaussian($\sigma = 0.02$)	0.1 (sim.)	0.001
Cluto_t8	(8K, 2, 9)	Gaussian($\sigma = 0.02$)	0.2 (sim.)	0.001
Cluto_t7	(10K, 2, 10)	Gaussian($\sigma = 0.02$)	0.2 (sim.)	0.001

4.5.2 . Effect of noise filtering

Cluto_t7

Let us first take the Cluto_t7 dataset as an example. As displayed in Figure 4.2 (a), this dataset contains 9 closely distributed shape clusters surrounded by 1 noise cluster. Thus the number of clusters $k_c = 9 + 1 = 10$. Such dataset with

close clusters and many noise points is a great challenge for classical algorithms of spectral clustering despite the tuning of parameters, as shown in Figure 4.2 (b). In contrast, our noise robust spectral clustering algorithm based on either nnz per row or vertex degree can successfully identify noise and distinguish different shape clusters (Figure 4.2 (e) and (f)) on condition that an appropriate threshold $tholdNF$ for noise filtering is chosen. The choice of $tholdNF$ can be aided by observing the histogram of scaled nnz per row or scaled degrees (Figure 4.2 (c) and (d)). Good choices of $tholdNF$ are often located at the valley between the "hill" related to noise points and the "mountain" related to core points. However, since some shape clusters are very close to each other in Cluto_t7, a little higher value is set for $tholdNF$ to filter out more border points as noise and reduce the connections between extremely adjacent clusters, otherwise these border points can hinder spectral clustering from distinguishing some adjacent clusters (Figure 4.3 (a) and (b)). Therefore, the tuning of $tholdNF$ is delicate on Cluto_t7.

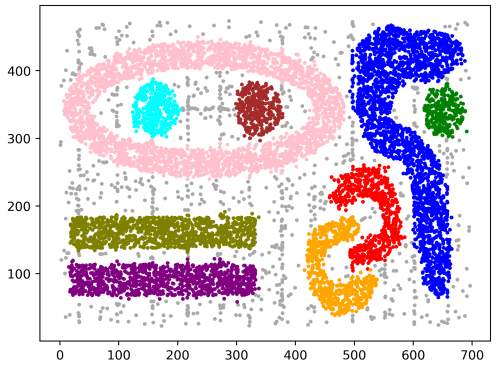
Similarly, the results of spectral clustering on the Cluto_t8, Cure_t2, and Compound datasets are presented in Figures 4.4, 4.5, and 4.6, respectively.

Cluto_t8

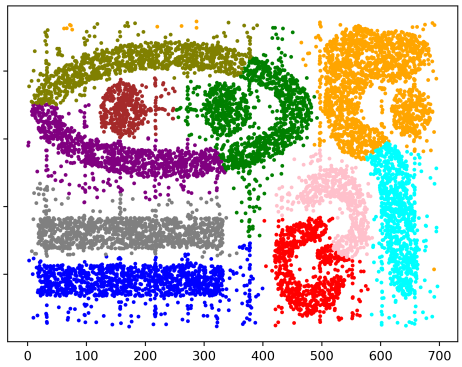
In the Cluto_t8 dataset (Figure 4.4), the vertical cluster and the inverted Y-shaped cluster on the right are too close to be distinguished by spectral clustering, so are the inverted Y-shaped cluster and the lower pie-shaped cluster. Unlike the case of Cluto_t7, using a higher $tholdNF$ cannot solve this issue, because the vertical cluster has a significantly lower density than the other clusters and it can be easily destroyed when our algorithm tries to filters out noise between the clusters mentioned above using a higher $tholdNF$. Nonetheless, spectral clustering with noise filtering produces a much better result than without noise filtering.

Cure_t2

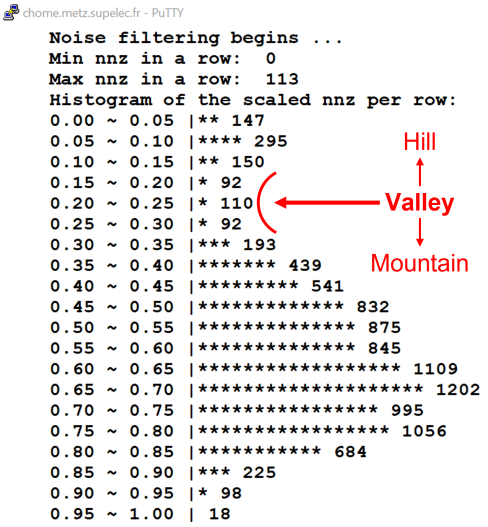
For the Cure_t2 dataset (Figure 4.5), the ground truth clustering regards the point distributions in the upper area as two oval clusters plus one strip-shaped cluster. However, they are actually connected and spectral clustering cannot identify them as three clusters correctly. Hence, we consider them as a whole cluster instead when imposing the number of clusters k_c for spectral clustering. Moreover, as the single large oval cluster appears to have the lowest density compared to other shape clusters, the histograms using 20 bins can no longer expose the valleys for choosing $tholdNF$. Instead we draw the histograms using 50 bins but show only the first half parts containing the valleys in Figure 4.5 (c) and (d). By choosing a small threshold $tholdNF = 0.02$, our algorithm can successfully obtain satisfying clusterings as shown in Figure 4.5 (e) and (f) (except that several points near clusters are not identified as noise), while spectral clustering without noise filtering cannot work well, as shown in Figure 4.5 (b).



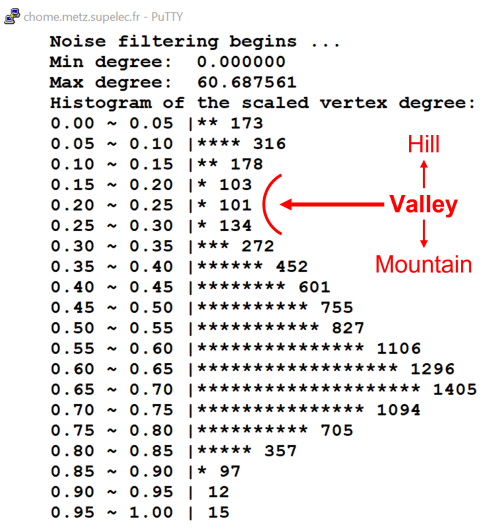
(a) Ground truth clustering ($k_c = 10$, noise cluster in gray)



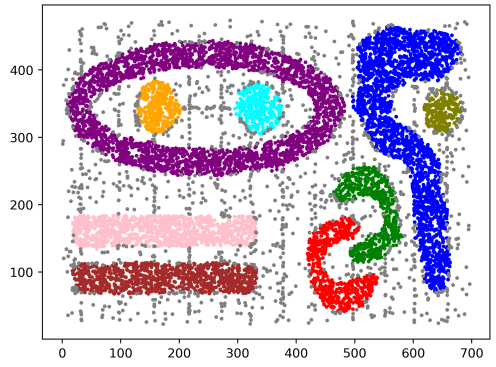
(b) SC ($k_c = 10$, without filtering noise)



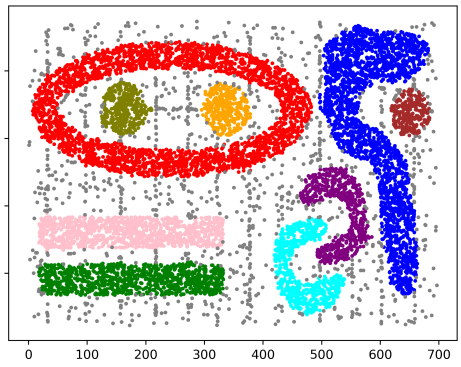
(c) Histogram of scaled nnz per row



(d) Histogram of scaled degrees of vertices

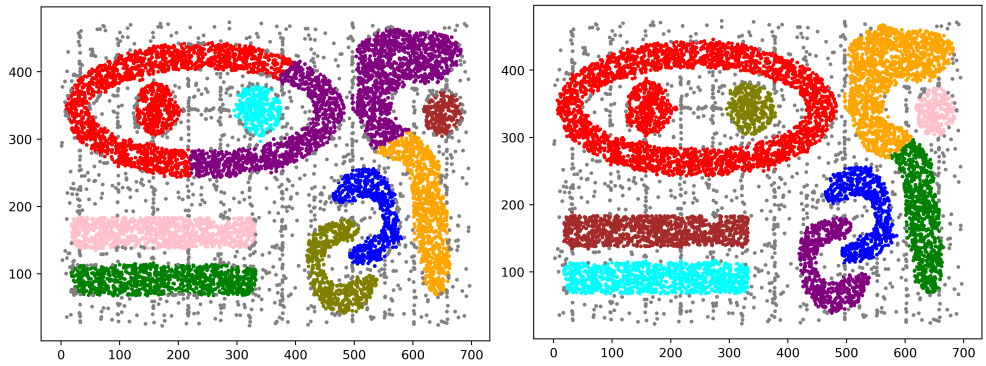


(e) Noise robust SC ($k_c = 10$, noise filtering based on nnz per row, $tholdNF = 0.39$, noise cluster in gray)



(f) Noise robust SC ($k_c = 10$, noise filtering based on vertex degree, $tholdNF = 0.31$, noise cluster in gray)

Figure 4.2: Spectral clustering (abbr. SC) on the Cluto_t7 dataset (part 1)



(a) Noise robust SC ($k_c = 10$, noise filtering based on nnz per row, $tholdNF = 0.38$, noise cluster in gray)

(b) Noise robust SC ($k_c = 10$, noise filtering based on vertex degree, $tholdNF = 0.30$, noise cluster in gray)

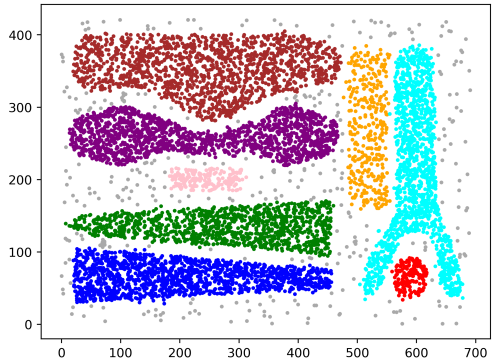
Figure 4.3: Spectral clustering (abbr. SC) on the Cluto_t7 dataset (part 2)

Compound

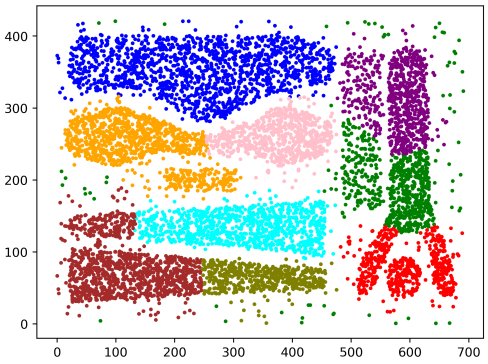
For the Compound dataset (Figure 4.6), our noise filtering algorithm can perfectly identify the noise points in the right area, but unfortunately the outer points with lower densities of the two clusters in the upper left area are also identified as noise by our algorithm.

Figures 4.7 and 4.8 present the impact of $tholdNF$ on the clustering quality measured by ARI and NMI scores[3] for the above four noisy datasets. There are some missing values in Figure 4.7 (d) and Figure 4.8 (a) and (b), because in these cases the nvGRAPH API fails in execution (probably due to eigensolver failure). Nevertheless, it can be clearly seen that our noise filtering algorithm can significantly improve the ARI and NMI scores of spectral clustering in a wide range of $tholdNF$ for Cluto_t7, Cluto_t8, and Compound datasets. In contrast, for the Cure_t2 dataset, the ARI scores of spectral clustering without noise filtering are higher than those with noise filtering and the NMI scores without noise filtering are close to those with noise filtering, as shown in Figure 4.8 (a) and (b). This is because the ARI and NMI scores are calculated based on the ground truth clustering, which favors the partitioning of the point distributions into three clusters in the upper area of Figure 4.5 (a). However, by comparing Figure 4.5 (b), (e) and (f), it is obvious that spectral clustering with noise filtering achieves better results than without noise filtering.

[3]The AMI scores are not displayed because they are usually equal to or very close to the NMI scores in our experiments.



(a) Ground truth clustering ($k_c = 9$, noise cluster in gray)



(b) SC ($k_c = 9$, without filtering noise)

chome.metz.supelec.fr - PuTTY

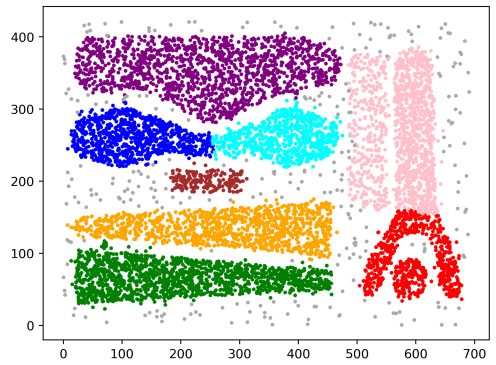
```
Noise filtering begins ...
Min nnz in a row:  0
Max nnz in a row:  94
Histogram of the scaled nnz per row:
0.00 ~ 0.05 |*** 153
0.05 ~ 0.10 |* 63
0.10 ~ 0.15 |* 60
0.15 ~ 0.20 |** 86
0.20 ~ 0.25 |***** 208
0.25 ~ 0.30 |******** 341
0.30 ~ 0.35 |****** 284
0.35 ~ 0.40 |************ 511
0.40 ~ 0.45 |******************** 806
0.45 ~ 0.50 |***************** 712
0.50 ~ 0.55 |********************* 820
0.55 ~ 0.60 |***************** 708
0.60 ~ 0.65 |***************** 680
0.65 ~ 0.70 |************** 581
0.70 ~ 0.75 |***************** 712
0.75 ~ 0.80 |************** 590
0.80 ~ 0.85 |******* 310
0.85 ~ 0.90 |****** 260
0.90 ~ 0.95 |** 95
0.95 ~ 1.00 | 18
```

(c) Histogram of scaled nnz per row

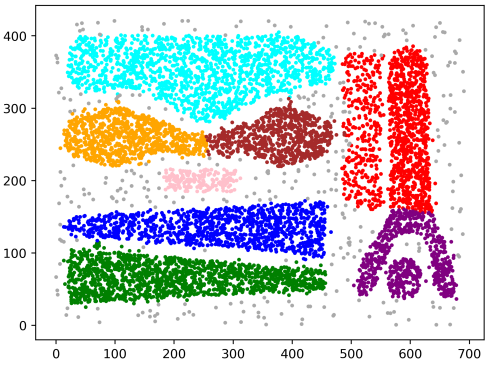
chome.metz.supelec.fr - PuTTY

```
Noise filtering begins ...
Min degree:  0.000000
Max degree:  47.518974
Histogram of the scaled vertex degree:
0.00 ~ 0.05 |**** 160
0.05 ~ 0.10 |** 84
0.10 ~ 0.15 |* 60
0.15 ~ 0.20 |** 98
0.20 ~ 0.25 |**** 190
0.25 ~ 0.30 |******* 289
0.30 ~ 0.35 |******* 309
0.35 ~ 0.40 |************ 501
0.40 ~ 0.45 |***************** 698
0.45 ~ 0.50 |********************* 788
0.50 ~ 0.55 |**************** 658
0.55 ~ 0.60 |***************** 701
0.60 ~ 0.65 |***************** 677
0.65 ~ 0.70 |******************* 773
0.70 ~ 0.75 |***************** 693
0.75 ~ 0.80 |************** 566
0.80 ~ 0.85 |********** 395
0.85 ~ 0.90 |***** 207
0.90 ~ 0.95 |*** 121
0.95 ~ 1.00 | 31
```

(d) Histogram of scaled degrees of vertices

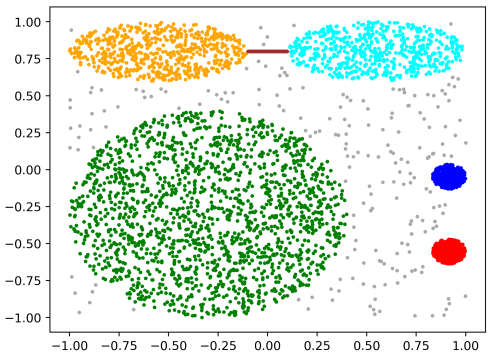


(e) Noise robust SC ($k_c = 9$, noise filtering based on nnz per row, $tholdNF = 0.15$, noise cluster in gray)

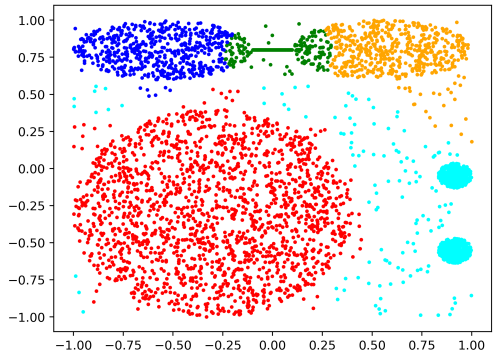


(f) Noise robust SC ($k_c = 9$, noise filtering based on vertex degree, $tholdNF = 0.15$, noise cluster in gray)

Figure 4.4: Spectral clustering (abbr. SC) on the Cluto_t8 dataset



(a) Ground truth clustering ($k_c = 7$, noise cluster in gray)



(b) SC ($k_c = 5$, without filtering noise)

chome.metz.supelec.fr - PuTTY

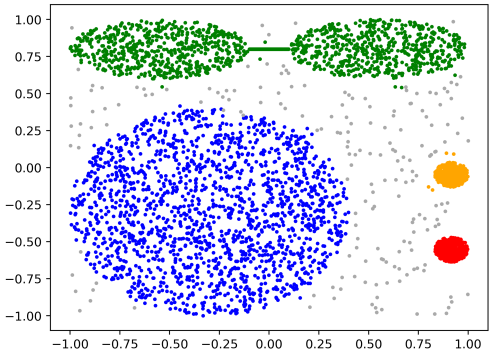
```
Noise filtering begins ...
Min nnz in a row:  0
Max nnz in a row:  397
Histogram of the scaled nnz per row:
0.00 ~ 0.02 |*** 189
0.02 ~ 0.04 |* 103
0.04 ~ 0.06 |*********** 607
0.06 ~ 0.08 |******************** 1037
0.08 ~ 0.10 |******** 442
0.10 ~ 0.12 |******* 383
0.12 ~ 0.14 |***** 294
0.14 ~ 0.16 |* 66
0.16 ~ 0.18 | 6
0.18 ~ 0.20 | 7
0.20 ~ 0.22 | 8
0.22 ~ 0.24 | 3
0.24 ~ 0.26 | 4
0.26 ~ 0.28 | 5
0.28 ~ 0.30 | 5
0.30 ~ 0.32 | 4
0.32 ~ 0.34 | 23
0.34 ~ 0.36 | 18
0.36 ~ 0.38 | 33
0.38 ~ 0.40 | 30
```

(c) Histogram of scaled nnz per row

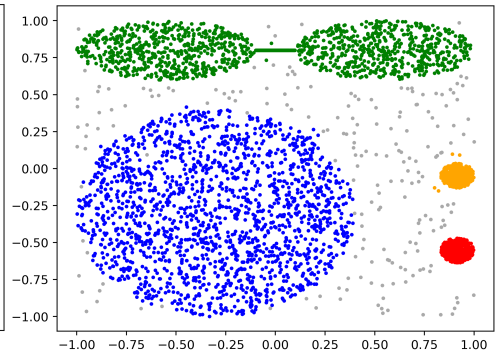
chome.metz.supelec.fr - PuTTY

```
Noise filtering begins ...
Min degree:  0.000000
Max degree:  164.974930
Histogram of the scaled vertex degree:
0.00 ~ 0.02 |**** 199
0.02 ~ 0.04 |*** 172
0.04 ~ 0.06 |***************** 743
0.06 ~ 0.08 |******************** 875
0.08 ~ 0.10 |********* 433
0.10 ~ 0.12 |********* 431
0.12 ~ 0.14 |***** 229
0.14 ~ 0.16 |* 46
0.16 ~ 0.18 | 12
0.18 ~ 0.20 | 2
0.20 ~ 0.22 | 2
0.22 ~ 0.24 | 5
0.24 ~ 0.26 | 3
0.26 ~ 0.28 | 1
0.28 ~ 0.30 | 2
0.30 ~ 0.32 | 3
0.32 ~ 0.34 | 2
0.34 ~ 0.36 | 6
0.36 ~ 0.38 | 5
0.38 ~ 0.40 | 14
```

(d) Histogram of scaled degrees of vertices

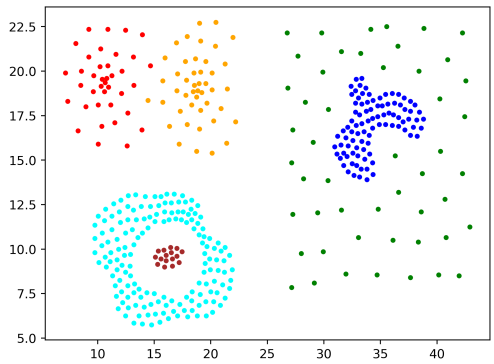


(e) Noise robust SC ($k_c = 5$, noise filtering based on nnz per row, $tholdNF = 0.02$, noise cluster in gray)

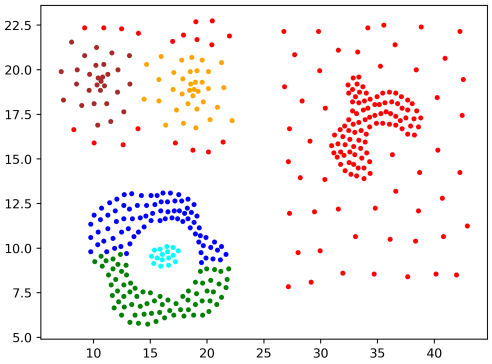


(f) Noise robust SC ($k_c = 5$, noise filtering based on vertex degree, $tholdNF = 0.02$, noise cluster in gray)

Figure 4.5: Spectral clustering (abbr. SC) on the Cure_t2 dataset



(a) Ground truth clustering ($k_c = 6$)



(b) SC ($k_c = 6$, without filtering noise)

chome.metz.supelec.fr - PuTTY

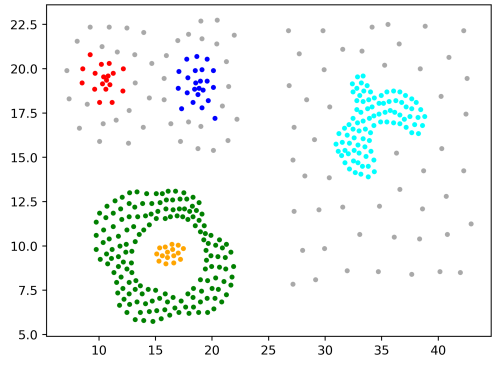
```
Noise filtering begins ...
Min nnz in a row:  0
Max nnz in a row:  22
Histogram of the scaled nnz per row:
0.00 ~ 0.05 |****************** 68
0.05 ~ 0.10 |**** 16
0.10 ~ 0.15 |** 8
0.15 ~ 0.20 |* 4
0.20 ~ 0.25 |* 6
0.25 ~ 0.30 |** 8
0.30 ~ 0.35 |**** 15
0.35 ~ 0.40 |***** 21
0.40 ~ 0.45 |****** 25
0.45 ~ 0.50 |******** 29
0.50 ~ 0.55 |******************** 72
0.55 ~ 0.60 |******* 27
0.60 ~ 0.65 |******* 26
0.65 ~ 0.70 |***** 19
0.70 ~ 0.75 |*** 14
0.75 ~ 0.80 |** 9
0.80 ~ 0.85 |*** 14
0.85 ~ 0.90 |* 5
0.90 ~ 0.95 |** 9
0.95 ~ 1.00 | 3
```

(c) Histogram of scaled nnz per row

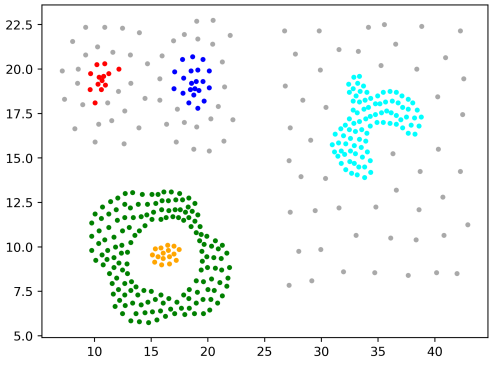
chome.metz.supelec.fr - PuTTY

```
Noise filtering begins ...
Min degree:  0.000000
Max degree:  6.000996
Histogram of the scaled vertex degree:
0.00 ~ 0.05 |******************** 86
0.05 ~ 0.10 |*** 13
0.10 ~ 0.15 |* 6
0.15 ~ 0.20 | 4
0.20 ~ 0.25 |** 9
0.25 ~ 0.30 |***** 23
0.30 ~ 0.35 |***** 24
0.35 ~ 0.40 |****** 26
0.40 ~ 0.45 |******* 32
0.45 ~ 0.50 |********* 40
0.50 ~ 0.55 |******* 31
0.55 ~ 0.60 |***** 25
0.60 ~ 0.65 |*** 16
0.65 ~ 0.70 |*** 17
0.70 ~ 0.75 |***** 24
0.75 ~ 0.80 |** 12
0.80 ~ 0.85 |* 5
0.85 ~ 0.90 | 2
0.90 ~ 0.95 | 3
0.95 ~ 1.00 | 0
```

(d) Histogram of scaled degrees of vertices

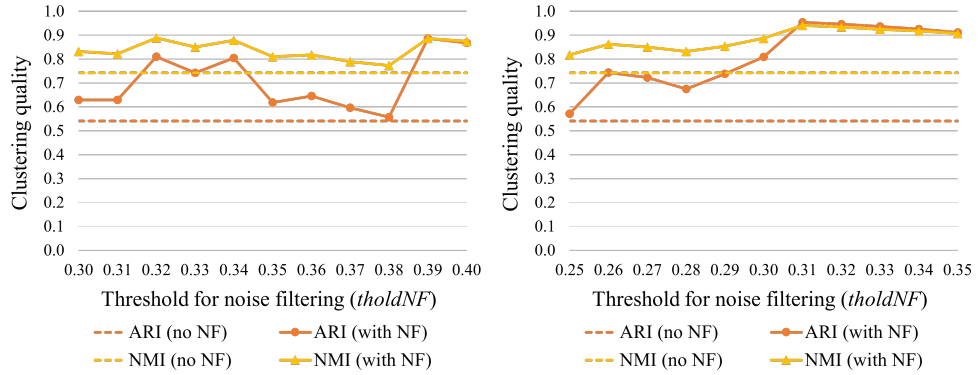


(e) Noise robust SC ($k_c = 6$, noise filtering based on nnz per row, $tholdNF = 0.15$, noise cluster in gray)



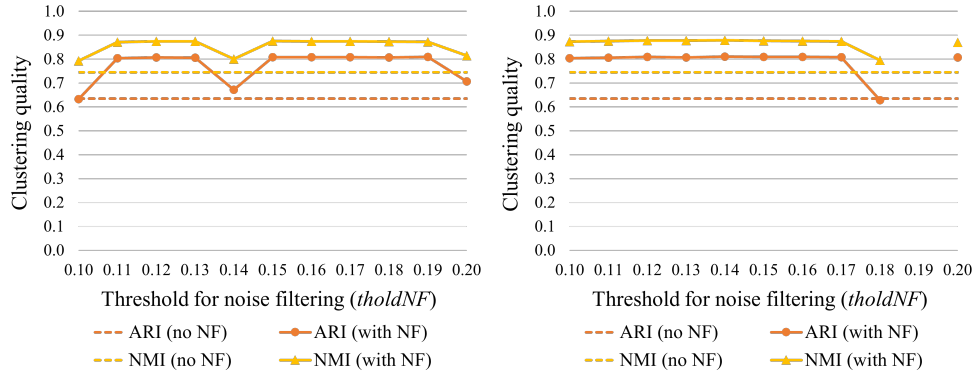
(f) Noise robust SC ($k_c = 6$, noise filtering based on vertex degree, $tholdNF = 0.10$, noise cluster in gray)

Figure 4.6: Spectral clustering (abbr. SC) on the Compound dataset



(a) On Cluto_t7 (with noise filtering based on nnz per row)

(b) On Cluto_t7 (with noise filtering based on vertex degree)



(c) On Cluto_t8 (with noise filtering based on nnz per row)

(d) On Cluto_t8 (with noise filtering based on vertex degree)

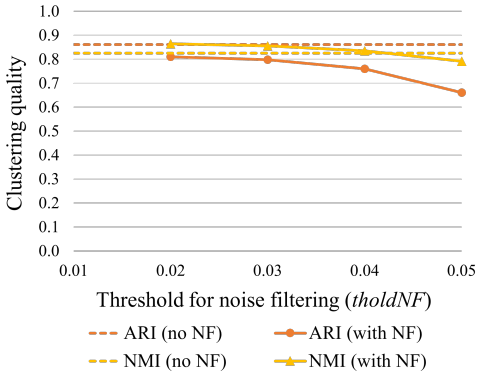
Figure 4.7: Impact of the threshold for noise filtering ($tholdNF$) on spectral clustering quality on Cluto_t7 and Cluto_t8 datasets

Table 4.2: Time (ms) of spectral clustering on GPU with noise filtering based on nnz per row

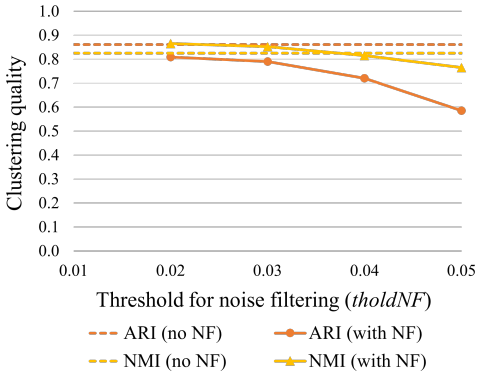
Dataset	Data transfers	CSR sim. constr.	Noise filtering	nvGRAPH API	Total
Compound	0.58	0.46	**0.69**	1115.41	1117.13
Cure_t2	0.74	1.06	**1.28**	1533.20	1536.27
Cluto_t8	0.70	3.70	**1.28**	1138.15	1143.82
Cluto_t7	0.69	5.56	**1.73**	1121.68	1129.66

Table 4.3: Time (ms) of spectral clustering on GPU with noise filtering based on vertex degree

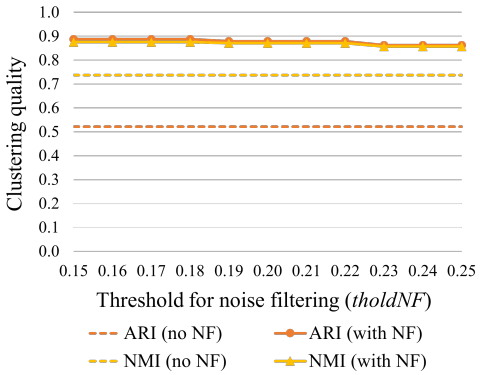
Dataset	Data transfers	CSR sim. constr.	Noise filtering	nvGRAPH API	Total
Compound	0.59	0.45	**0.80**	1205.59	1207.44
Cure_t2	0.70	1.03	**1.41**	1456.39	1459.53
Cluto_t8	0.74	3.80	**1.43**	1201.67	1207.62
Cluto_t7	0.74	5.55	**1.77**	1404.72	1412.78



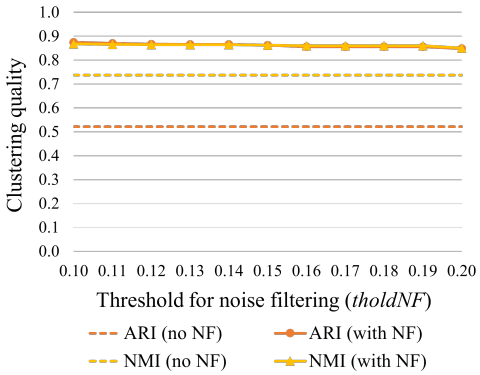
(a) On Cure_t2 (with noise filtering based on nnz per row)



(b) On Cure_t2 (with noise filtering based on vertex degree)



(c) On Compound (with noise filtering based on nnz per row)



(d) On Compound (with noise filtering based on vertex degree)

Figure 4.8: Impact of the threshold for noise filtering ($tholdNF$) on spectral clustering quality on Cure_t2 and Compound datasets

4.5.3 . Time overhead of noise filtering

Tables 4.2 and 4.3 give the running time breakdown of spectral clustering on GPU with the respective noise filtering method. For simplicity, `BSX`=128 and `BSY`=2 are set for all the CUDA kernels related to CSR format similarity matrix

construction (using Algorithm 4 in Section 3.3.3) and noise filtering (see Section 4.4). The $tholdNF$ is set with the appropriate values given previously for each dataset.

Obviously, the nvGRAPH API accounts for the vast majority of the total runtime. The noise filtering takes more time than the CSR similarity matrix construction on the Compound (with $n = 399$) and Cure_t2 (with $n = 4200$) datasets. However, it is the opposite situation on the Cluto_t8 (with $n = 8000$) and Cluto_t7 (with $n = 10000$) datasets, due to the high time complexity ($\mathcal{O}(n^2 d)$) of similarity matrix construction. Moreover, we observed that noise filtering based on vertex degree takes a little more time than that based on nnz per row, and so does the following nvGRAPH API. Essentially, the time overhead of our GPU implementation for noise filtering is insignificant for spectral clustering.

4.6 . Summary

In this chapter we have proposed an efficient noise robust spectral clustering algorithm based on two noise filtering approaches and provided its parallel implementation on GPU. Experiments on various noisy datasets show that our noise filtering implementation can significantly improve the quality of spectral clustering while introducing low time overhead. However, the tuning of the noise filtering threshold can be tricky in some cases and an auto-tuning method remains to be developed.

5 - Large-scale Representative-based Spectral Clustering on CPU-GPU Platforms

5.1 . Introduction

In the previous Chapter 3, due to optimized parallel computations with sparse data storage format, we have succeeded in scaling spectral clustering up to millions of data instances on a single GPU. However, it would be difficult to move forward to an even larger scale mainly because of two issues: (1) the similarity matrix construction would be too time-consuming considering the $\mathcal{O}(n^2d)$ time complexity; (2) the maximum number of edges that the nvGRAPH eigensolver-embedded API can handle is about 2 billion.

To address the above issues, we incorporate the use of representatives in this chapter, where representatives are assumed to be some existing or calculated points that can reflect the distributional characteristics of a dataset. The basic idea is to first extract some representative points from the original dataset (preprocessing step), then perform spectral clustering on the representatives (spectral processing step), and finally obtain the clustering result of the original dataset by assigning each instance to its nearest representative (postprocessing step). The goal of using representatives is to reduce the amount of data on which the computationally expensive spectral clustering is performed, while retaining the clustering quality with little degradation compared to spectral clustering on the original dataset.

In fact, the idea of using representatives is not new in the field of cluster analysis. For example, in 1998, Guha, Rastogi, and Shim [67] proposed an algorithm called CURE (short for Clustering Using Representatives), which produces some representative points for each cluster and then conducts hierarchical clustering on the representatives. In 2009, Yan, Huang, and Jordan [203] proposed a general framework for fast approximate spectral clustering based on the use of representatives, and they suggested two methods for extracting representatives: the k-means algorithm and the random projection tree. More information can be found in Section 1.4, where other approximation methods are also introduced.

In this chapter, we adopt the general framework proposed by Yan et al. and combine it with parallel computing to achieve large-scale spectral clustering on CPU-GPU platforms. Three methods for extracting representatives are considered: random sampling, k-means algorithm (see Section 1.2.1), and k-means++ algorithm (see Section 1.2.2). In Section 5.2 we empirically study the performance of each extraction method. Then, in Section 5.3, we consider three different usage scenarios and propose associated parallel processing chains for representative-based spectral clustering on CPU architectures, GPU architectures, or CPU-GPU heterogeneous architectures. As expected, the work presented in Chapter 2 (parallel

k-means on CPU and GPU) and the work presented in Chapter 3 (parallel spectral clustering on GPU) can serve as modules in the proposed processing chains. Finally, experiments on large-scale datasets demonstrate the high scalability and performance of the proposed spectral clustering chains.

5.2 . Extraction of representatives

5.2.1 . Using random sampling vs. k-means vs. k-means++

We consider three methods for extracting representatives: random sampling, k-means, and k-means++. Their performance is experimentally investigated with our testbed *john3* using four synthetic large-scale 2D datasets: Spirals-75M, Smile2-100M, Aggregation-78.8M, Complex9-303M. These datasets consist of clusters with highly dense point distributions. More information about the datasets and the testbed can be found in Appendices A and B, respectively.

Benchmarking settings

The following settings are used in our experiments:

- 40 OpenMP threads are created for the parallelization of each method on CPU, because using 40 threads allows hyperthreading which often results in better performance.
- The tolerance is set to 0.01 for the k-means and k-means++ algorithms (see explanation in Section 1.2.1).
- Two-level summation with 1000 packages are used in the *Update* step of k-means and k-means++ to handle the effect of rounding errors (see explanation in Section 2.2).
- Our parallel CPU implementation for the seeding step of k-means++ is displayed in Appendix G Listing G.6. Essentially it utilizes OpenMP directives and some Thrust functions (`reduce`, `inclusive_scan`, `exclusive_scan`) but does not change the high-level sequential nature of seeding, i.e. select initial centroids one by one (see Section 1.2.2).
- Computations are mainly in single precision, except that double precision is used for the Thrust functions exploited in the seeding step of k-means++ to handle the effect of rounding errors.

Distributions of the extracted k_r representatives

Figures 5.1, 5.2, 5.3, 5.4 show the distributions of k_r representatives extracted by each method on the four test datasets, respectively. It can be observed that:

- When k_r is relatively small (e.g. $k_r = 100$ or 500), the k_r representatives extracted by k-means or by k-means++ have much better distributions than

those extracted by random sampling. In fact, since the number of clusters in each benchmark dataset is relatively small ($k_c = 3$ to 9) and all clusters are in 2D space, extracting a relatively small number of representatives by k-means or k-means++ is usually sufficient to capture the distributional features of all 2D clusters.

- When k_r is relatively small, it is difficult or even impossible to correctly cluster the k_r representatives extracted by random sampling since their distributions are irregular. To obtain acceptable distributions of representatives with random sampling, k_r needs to be large enough (e.g. $k_r = 2500$). However, the calculation cost of spectral clustering on representatives would increase with k_r, at least quadratically.

- Compared to the representatives extracted by k-means, those extracted by k-means++ have slightly more uniform distributions, and therefore are more likely and easier to be correctly clustered, especially when k_r is small.

Elapsed time of extraction

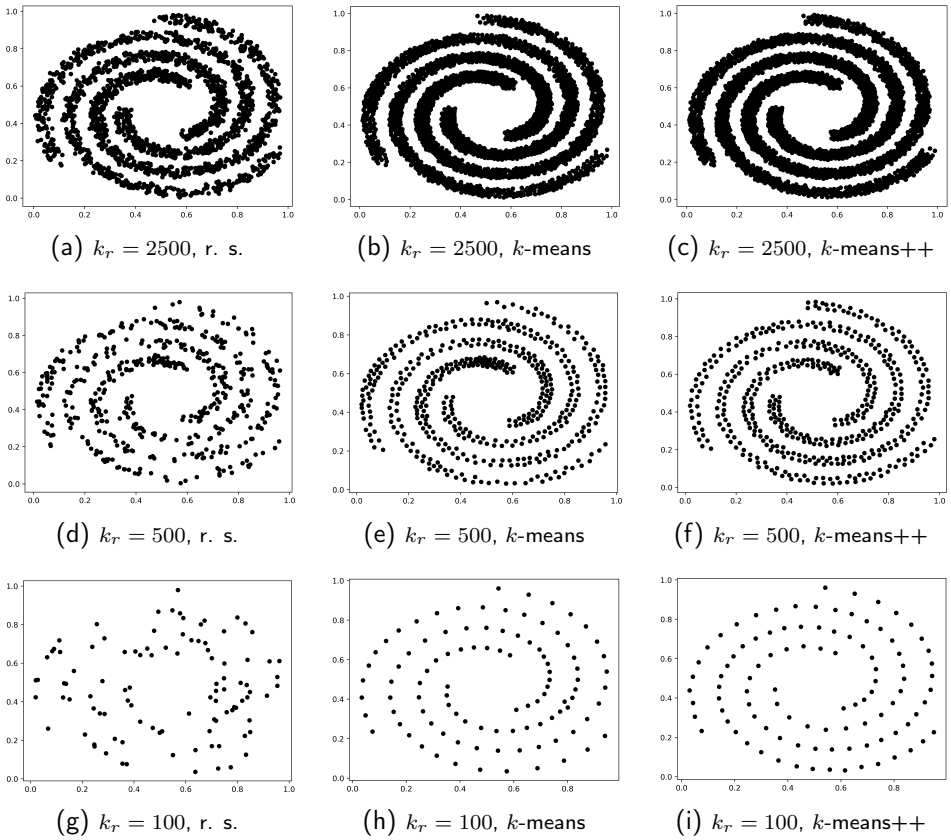
Tables 5.1, 5.2, 5.3 and 5.4 present the elapsed time of k_r representatives extraction using each method on the four datasets, respectively. It can be seen that:

- The representatives extraction by random sampling is much faster than k-means and k-means++. Note that it consists of not only the selection of k_r representatives by random sampling, but also the attachment of each data instance to its nearest representative. The former takes little time, while the latter consumes the same time as one iteration of the *ComputeAssign* step.

- k-means++ needs fewer iterations than k-means, but the former still takes significantly more time due to the expensive seeding step. The *Update* step per iteration takes little time, while the elapsed time of the *ComputeAssign* step per iteration is much higher and grows approximately linearly with k_r, due to the $\mathcal{O}(n \times k_r \times d)$ time complexity.

Strategy for choosing an extraction method

In summary, extracting more representatives generally better captures the distributional features of each cluster, but increases both the calculation cost of extracting representatives and the calculation cost of spectral clustering on representatives. After weighing the distribution quality of representatives against the cost of extraction, we found that k-means seems to be the best choice for extracting a relatively small number of representatives, while random sampling seems to be a preferable choice for extracting a large number of representatives. Particularly, k-means++ may be necessary to achieve better clustering results than k-means in case of extracting a very small number of representatives.



(a) $k_r = 2500$, r. s. (b) $k_r = 2500$, k-means (c) $k_r = 2500$, k-means++

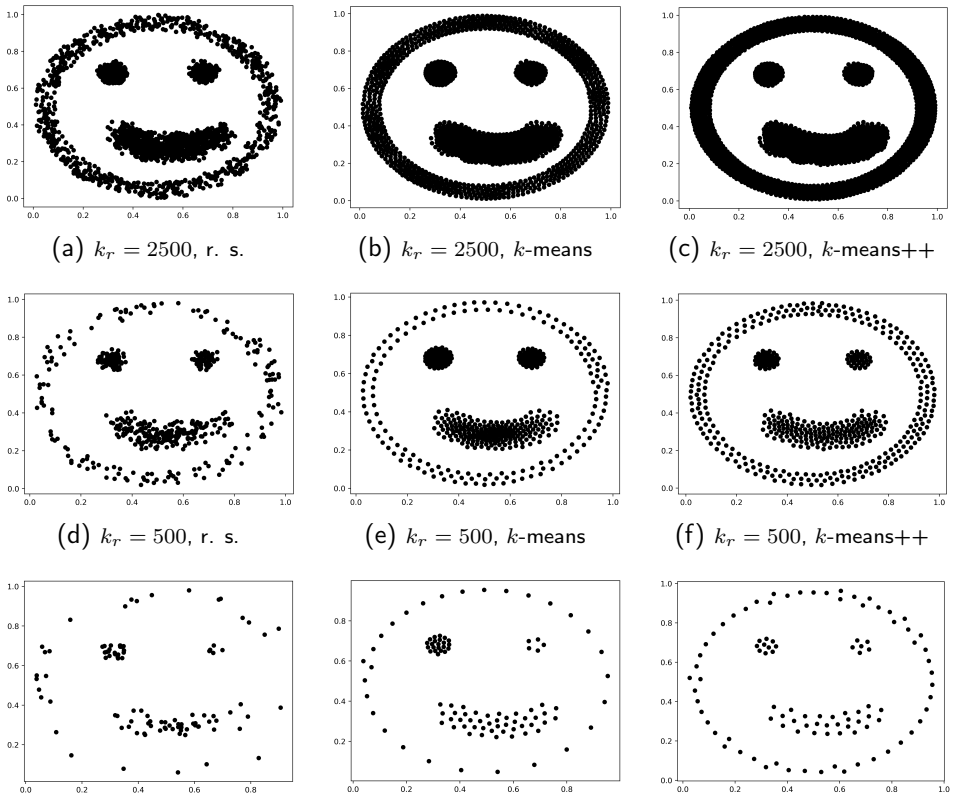
(d) $k_r = 500$, r. s. (e) $k_r = 500$, k-means (f) $k_r = 500$, k-means++

(g) $k_r = 100$, r. s. (h) $k_r = 100$, k-means (i) $k_r = 100$, k-means++

Figure 5.1: k_r representatives extracted from the Spirals-75M dataset by 3 different methods (1st column: random sampling; 2nd column: k-means with tolerance = 0.01; 3rd column: k-means++ with tolerance = 0.01)

Table 5.1: Elapsed time of k_r representatives extraction on Spirals-75M $(n, d, k_c) = (75M, 2, 3)$

k_r	Method	Time (s)				
		Initialize centroids	*ComputeAssign* per iter.	*Update* per iter.	Nb of iters.	Total
2500	r.s.	0.0002	12.49	N/A	1	12.49
	km	0.0002	12.24	0.04	24	294.72
	km+	1083.48	12.32	0.02	20	1330.28
500	r.s.	0.0002	2.66	N/A	1	2.66
	km	0.0002	2.50	0.03	20	50.63
	km+	225.32	2.52	0.02	15	263.41
100	r.s.	0.0003	0.63	N/A	1	0.63
	km	0.0003	0.53	0.02	21	11.54
	km+	49.40	0.51	0.02	16	57.91



(a) $k_r = 2500$, r. s. (b) $k_r = 2500$, k-means (c) $k_r = 2500$, k-means++

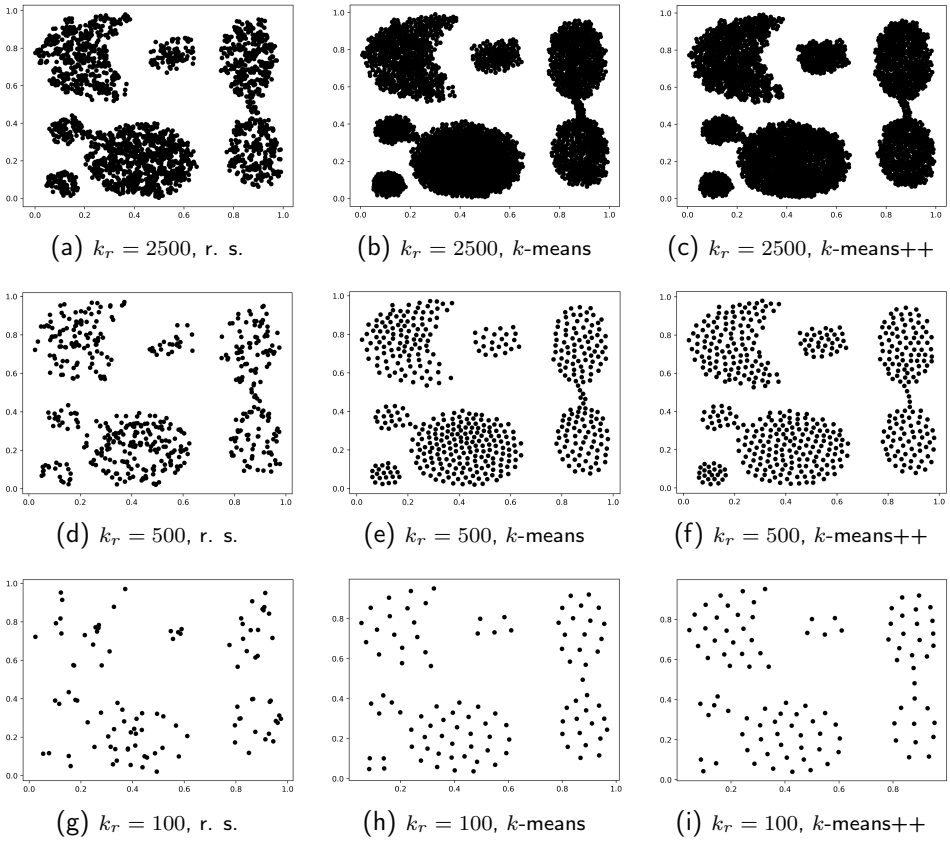
(d) $k_r = 500$, r. s. (e) $k_r = 500$, k-means (f) $k_r = 500$, k-means++

(g) $k_r = 100$, r. s. (h) $k_r = 100$, k-means (i) $k_r = 100$, k-means++

Figure 5.2: k_r representatives extracted from the Smile2-100M dataset by 3 different methods (1st column: random sampling; 2nd column: k-means with tolerance = 0.01; 3rd column: k-means++ with tolerance = 0.01)

Table 5.2: Elapsed time of k_r representatives extraction on Smile2-100M $(n, d, k_c) = (100M, 2, 4)$

k_r	Method	Time (s)				
		Initialize centroids	*ComputeAssign* per iter.	*Update* per iter.	Nb of iters.	Total
2500	r.s.	0.0003	16.60	N/A	1	16.60
	km	0.0004	16.38	0.05	35	574.91
	km+	1417.81	16.42	0.04	20	1747.11
500	r.s.	0.0003	3.57	N/A	1	3.57
	km	0.0004	3.36	0.04	31	105.20
	km+	293.86	3.36	0.02	22	368.27
100	r.s.	0.0003	0.83	N/A	1	0.84
	km	0.0002	0.71	0.03	27	19.89
	km+	63.60	0.69	0.02	16	74.98



(a) $k_r = 2500$, r. s. (b) $k_r = 2500$, k-means (c) $k_r = 2500$, k-means++

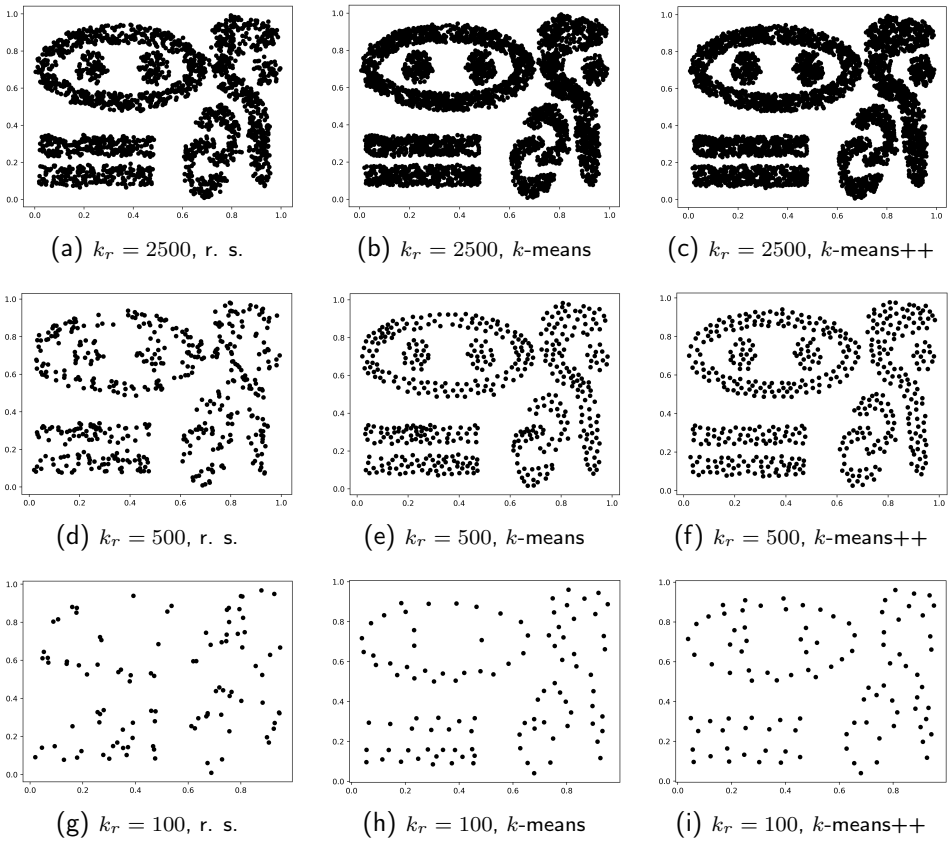
(d) $k_r = 500$, r. s. (e) $k_r = 500$, k-means (f) $k_r = 500$, k-means++

(g) $k_r = 100$, r. s. (h) $k_r = 100$, k-means (i) $k_r = 100$, k-means++

Figure 5.3: k_r representatives extracted from Aggregation-78.8M dataset by 3 different methods (1st column: random sampling; 2nd column: k-means with tolerance = 0.01; 3rd column: k-means++ with tolerance = 0.01)

Table 5.3: Elapsed time of k_r representatives extraction on Aggregation-78.8M $(n, d, k_c) = (78.8M, 2, 7)$

k_r	Method	Time (s)				
		Initialize centroids	***ComputeAssign* per iter.**	***Update* per iter.**	**Nb of iters.**	**Total**
2500	r.s.	0.0003	13.09	N/A	1	13.09
	km	0.0003	12.91	0.04	30	388.32
	km+	1132.37	12.90	0.03	21	1403.78
500	r.s.	0.0003	2.77	N/A	1	2.77
	km	0.0002	2.69	0.03	23	62.48
	km+	231.21	2.63	0.01	18	278.83
100	r.s.	0.0002	0.67	N/A	1	0.67
	km	0.0003	0.56	0.02	19	11.16
	km+	48.49	0.52	0.01	17	57.63



(a) $k_r = 2500$, r. s. (b) $k_r = 2500$, k-means (c) $k_r = 2500$, k-means++

(d) $k_r = 500$, r. s. (e) $k_r = 500$, k-means (f) $k_r = 500$, k-means++

(g) $k_r = 100$, r. s. (h) $k_r = 100$, k-means (i) $k_r = 100$, k-means++

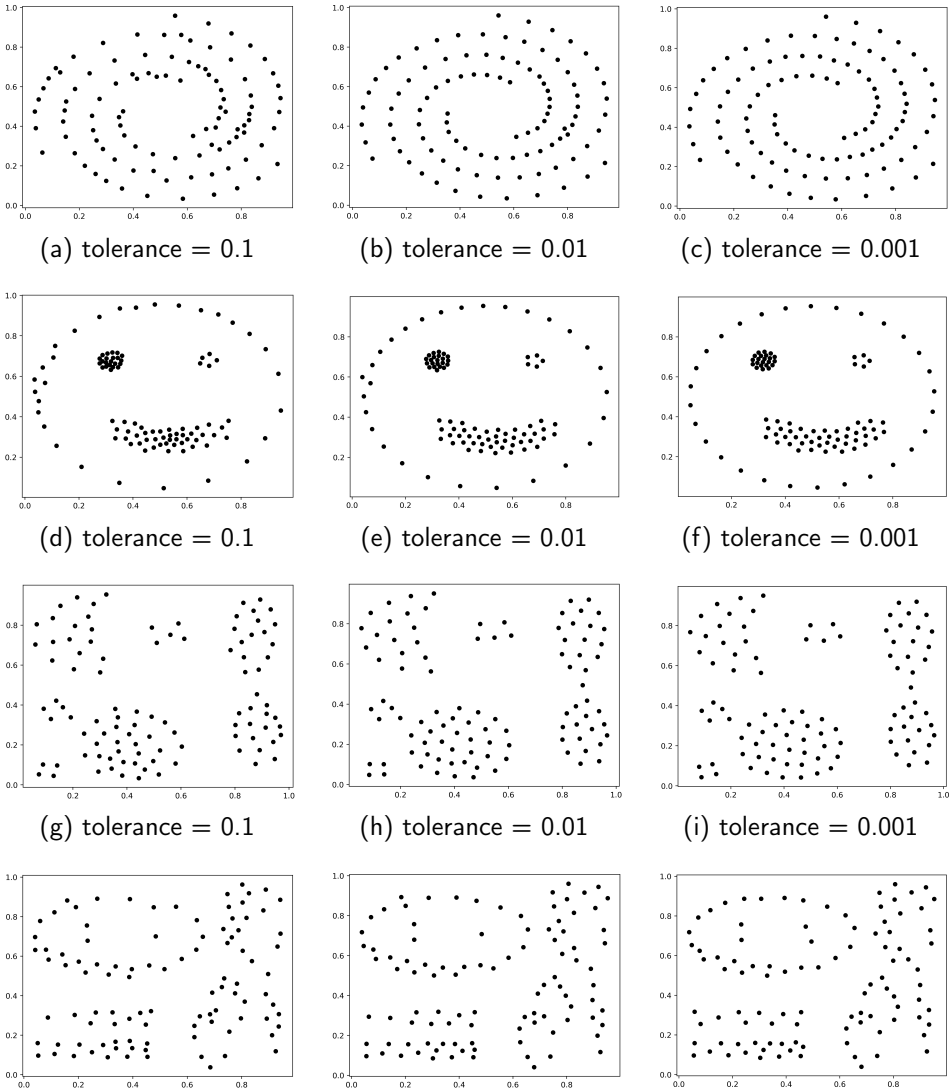
Figure 5.4: k_r representatives extracted from Complex9-303M dataset by 3 different methods (1st column: random sampling; 2nd column: k-means with tolerance = 0.01; 3rd column: k-means++ with tolerance = 0.01)

Table 5.4: Elapsed time of k_r representatives extraction on Complex9-303M $(n, d, k_c) = (303.1M, 2, 9)$

k_r	Method	Time (s)				
		Initialize centroids	*ComputeAssign* per iter.	*Update* per iter.	Nb of iters.	Total
2500	r.s.	0.0004	50.21	N/A	1	50.21
	km	0.0003	49.44	0.18	25	1240.64
	km+	4220.51	49.51	0.13	20	5213.32
500	r.s.	0.0003	10.74	N/A	1	10.74
	km	0.0003	10.13	0.13	20	205.10
	km+	844.16	10.08	0.06	18	1026.66
100	r.s.	0.0003	2.67	N/A	1	2.67
	km	0.0002	2.12	0.11	14	31.13
	km+	186.23	2.06	0.07	11	209.65

5.2.2 . Impact of the tolerance of k-means

For the results in previous figures and tables, the tolerances of k-means and k-means++ are directly set to 0.01. Nevertheless, the tolerance actually plays an important role in the performance of extracting representatives.



(a) tolerance = 0.1 (b) tolerance = 0.01 (c) tolerance = 0.001

(d) tolerance = 0.1 (e) tolerance = 0.01 (f) tolerance = 0.001

(g) tolerance = 0.1 (h) tolerance = 0.01 (i) tolerance = 0.001

(j) tolerance = 0.1 (k) tolerance = 0.01 (l) tolerance = 0.001

Figure 5.5: 100 representatives extracted from each benchmark dataset by using k-means with different tolerances (1st row: Spirals-75M; 2nd row: Smile2-100M; 3rd row: Aggregation-78.8M; 4th row: Complex9-303M)

Figure 5.5 displays the distributions of 100 representatives extracted by k-means using different tolerances, and Table 5.5 presents the associated elapsed time. It can be found that:

- With decreasing tolerance, the representatives extracted by k-means have improved distributions, i.e. better capture the distributional characteristics of each cluster. However, the number of iterations required to achieve convergence increases significantly, thus leading to a considerable augmentation of elapsed time.
- Generally, setting the tolerance of k-means to 0.01 seems to be a good compromise on various datasets, i.e. achieving relatively good distributions of representatives in an acceptable amount of time.

Table 5.5: Elapsed time of extracting 100 representatives using k-means with different tolerances

Dataset	Tolerance	Time (s)				
		Initialize centroids	*ComputeAssign* per iter.	*Update* per iter.	Nb of iters.	Total
Spirals-75M	0.1	0.0002	0.54	0.02	3	1.67
	0.01	0.0003	0.53	0.02	21	11.54
	0.001	0.0002	0.51	0.02	108	57.01
Smile2-100M	0.1	0.0003	0.75	0.08	3	2.48
	0.01	0.0002	0.71	0.03	27	19.89
	0.001	0.0003	0.69	0.02	194	138.42
Aggregation-78.8M	0.1	0.0003	0.59	0.02	3	1.81
	0.01	0.0003	0.56	0.02	19	11.16
	0.001	0.0002	0.53	0.02	69	38.06
Complex9-303M	0.1	0.0003	2.29	0.11	3	7.21
	0.01	0.0002	2.12	0.11	14	31.13
	0.001	0.0002	2.06	0.07	80	170.29

5.3 . Representative-based spectral clustering on CPU-GPU platforms

5.3.1 . Different scenarios and adapted parallel processing chains

To achieve large-scale high-performance representative-based spectral clustering on modern CPU-GPU platforms, we need to design adapted parallel processing chains by considering the following aspects:

- **Strengths and limitations of modern CPU vs. GPU architectures.** Basically, the GPU is specialized for large fine-grained parallel computations but usually has much less RAM than the CPU. See Section 1.5.1 for more description.

- **Advantages and disadvantages of different methods for k_r representatives extraction.** Random sampling is a fast but naive method. Getting a good distribution of representatives using random sampling usually requires k_r to be large enough. In contrast, k-means is a high-quality method that preserves cluster properties well even when k_r is relatively small, but the calculation cost is nontrivial especially for large k_r. See Section 5.2.1 for illustration.

- **Dataset characteristics.** The size of a dataset ($n \times d$ elements) may exceed the size of GPU RAM, in which case the data cannot be entirely loaded onto the GPU. Besides, a dataset may have only a few clusters in low-dimensional space, thus extracting a small number of representatives may be sufficient to represent the properties of all clusters; or a dataset may have a large number of clusters or dimensions, thus a large number of representatives (at least $k_r \gg k_c$) can be required.

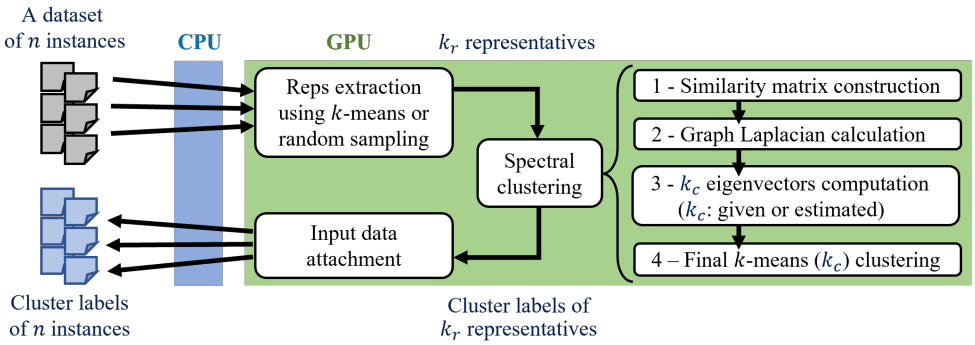
Depending on whether the GPU RAM is sufficient with respect to the data size and whether the number of representatives to be extracted is small or large, we propose an associated parallel processing chain for each scenario, as shown in Figure 5.6.

Scenario I: sufficient GPU RAM

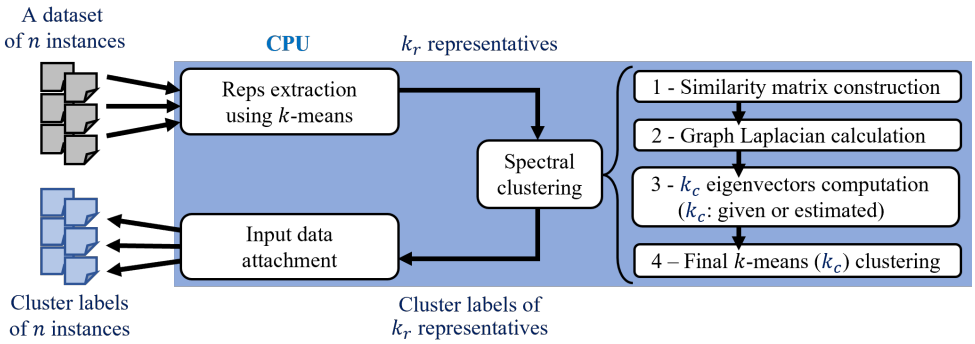
Let us first consider the scenario where there is sufficient GPU RAM to store the entire dataset but direct spectral clustering would still take too much time. Figure 5.6 (a) presents our parallel processing chain in this case. After reading data instances from a disk file to CPU RAM, we suggest transferring all data from CPU to GPU, and then performing representative-based spectral clustering entirely on GPU. This is because computations on GPU are usually faster than computations on CPU according to the experimental results presented in Chapter 2 and Chapter 3. If the number of representatives to be extracted (k_r) is relatively small, then we suggest using k-means as the extraction method, otherwise random sampling for large k_r should be a better choice in terms of execution time.

Scenario II: insufficient GPU RAM + small k_r

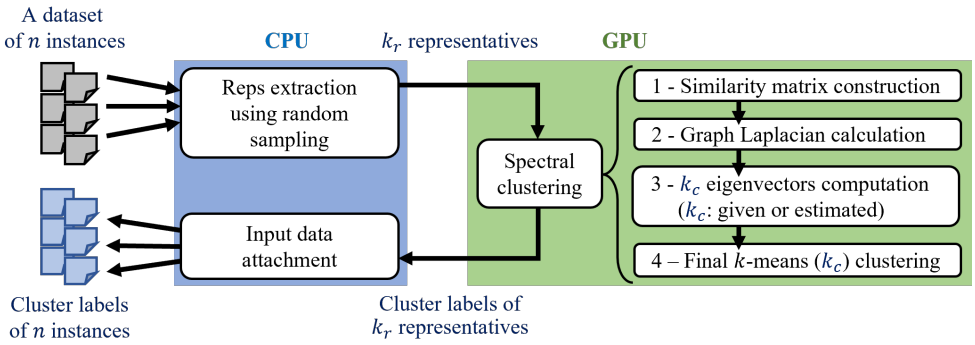
Secondly, we consider the scenario of lacking GPU RAM to store the entire dataset. In this case, the representatives extraction has to be done on the CPU. Suppose that only a small number of representatives (k_r) needs to be extracted, then as shown in Figure 5.6 (b), k-means should be adopted for representatives extraction because of its ability to obtain a good representation of data. Moreover, spectral clustering on just a few k_r representatives can remain on CPU (e.g. using scikit-learn [150]) and so does the input data attachment. Therefore the GPU becomes unnecessary.



(a) Chain I for Scenario I: sufficient GPU RAM (+ small/large k_r)



(b) Chain II for Scenario II: insufficient GPU RAM + small k_r



(c) Chain III for Scenario III: insufficient GPU RAM + large k_r

Figure 5.6: Three scenarios and associated parallel processing chains for large-scale spectral clustering using representatives

Scenario III: insufficient GPU RAM + large k_r

Finally, if GPU RAM is insufficient with respect to the data size and meanwhile a large number of representatives are required, we propose a CPU-GPU heterogeneous processing chain, as exhibited in Figure 5.6 (c). The representatives ex-

traction has to be done on the CPU and should use random sampling instead of k-means, because the former is much faster and can get a good distribution of representatives as long as k_r is large enough. However, spectral clustering on a large number of representatives is computationally intensive and thus performing it on the GPU is preferable.

The above three scenarios and associated processing chains can be integrated into a global workflow as shown in Figure 5.7.

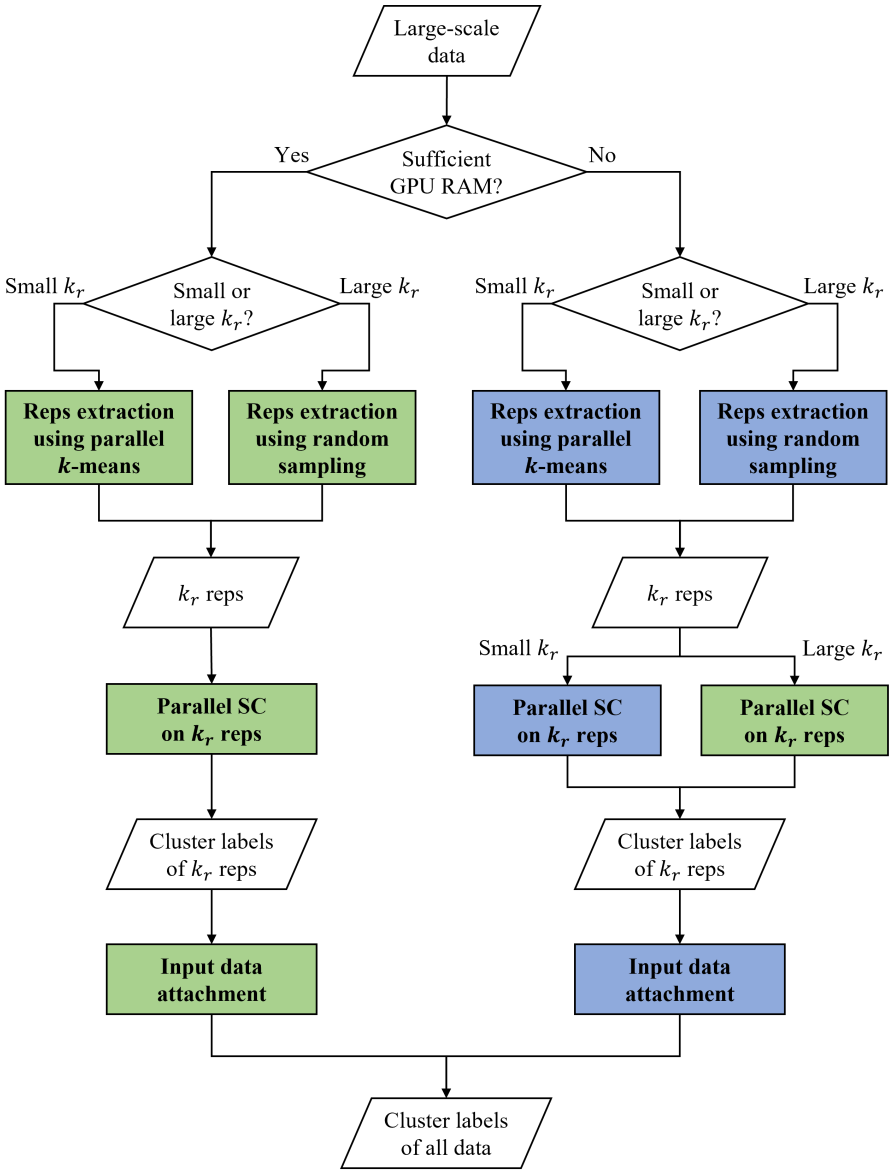


Figure 5.7: Global workflow for large-scale representative-based spectral clustering on CPU-GPU platforms
(blue box: on CPU; green box: on GPU)

5.3.2 . Global experiments

We experimented with the proposed processing chains (see Figure 5.6) for representative-based spectral clustering on the four large-scale datasets mentioned in Section 5.2 (Spirals-75M, Smile2-100M, Aggregation-78.8M, Complex9-303M). The testbed is our *john3* server consisting of two Xeon Silver 4114 processors as CPU (20 physical cores in total) and a GeForce RTX 3090 as GPU (24GB RAM). More details about the datasets and the testbed are available in Appendix A and B. Note that the size of each benchmark dataset does not exceed the GPU RAM yet, but they can be used to evaluate the performance of each proposed chain.

Benchmarking approach & experimental settings

Our benchmarking approach and its experimental settings are described as follows:

- For the extraction of representatives using k-means or k-means++, the tolerance is always set to 0.01. Besides, when $\frac{n}{k_r} > 10^4$, the two-level summation method using 1000 packages will be activated for the *Update* step of k-means and k-means++, in order to handle the effect of rounding errors (see explanation in Section 2.2). Otherwise, when $\frac{n}{k_r} \leq 10^4$, the *Update* step does not activate the two-level summation method.

- For the similarity matrix construction on k_r representatives, the Gaussian similarity is used and the values of connectivity parameters (σ, upper bound threshold for squared distance, lower bound threshold for similarity) are set as shown in Table 5.6. The similarity matrix construction in the proposed chain I and chain III is performed using our Algo CSR-1 (see Section 3.3.3).

- After the similarity matrix construction, the remaining steps of spectral clustering are conducted using the LOBPCG-embedded algorithm of the nvGRAPH library (see Section 3.4) for the proposed chain I and chain III. The tolerance for the LOBPCG eigensolver is always set to 1E-5. Other parameter settings of the nvGRAPH's algorithm are the same as in Section 3.6.2.

- For the proposed chain II, spectral clustering on k_r representatives is implemented using *scikit-learn* [150] version 1.1.1. Specifically, the similarity matrix construction is implemented using the `pairwise_kernels` function[1] (with parameter settings: `metric='rbf'` i.e. Gaussian similarity, `gamma`=$1/(2\sigma^2)$), then spectral clustering based on the precomputed similarity matrix is implemented using the `SpectralClustering` function[2] (with parameter settings: `eigen_solver='lobpcg'`, `eigen_tol=1E-5`, `random_state=1`, `n_init=1`).

[1] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_kernels.html

[2] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html

Particularly, although both functions include the parameter `n_jobs` to define the number of jobs for parallel computing, we found in practice that it is difficult to control parallelization through this parameter. In fact, defining multiple jobs accelerated the `pairwise_kernels` function, but decelerated significantly the `SpectralClustering` function, which led to a decrease of overall performance. Also, defining multiple jobs only for the `pairwise_kernels` function did not change the situation. Nevertheless, we found practically that, without defining the parameter `n_jobs`, executing each of the two functions involved automatically a combination of mono-core computing and multi-core computing. Therefore, we decided not to interfere with the computations through `n_jobs`.

Finally, the input data attachment of chain II is implemented using the *Numpy* library [74].

- Our CPU implementations for the representatives extraction step and the input data attachment step always use 40 OpenMP threads, which equals the number of logical cores of our CPU. The block size configurations for our CUDA kernels are mainly set as `BSX=128` and `BSY=2`.
- Computations are mainly in single precision, except that double precision is used for the Thrust functions exploited in the seeding step of k-means++ to handle the effect of rounding errors (see Appendix G).

Table 5.6: Settings of connectivity parameters

Chain	k_r	Extract. method	Sprials-75M		Smile2-100M		Aggregation-78.8M		Complex9-303M	
			σ	Threshold	σ	Threshold	σ	Threshold	σ	Threshold
I	10^2	km+	0.1	0.01 (sq. dist.)	0.1	0.007 (sq. dist.)	0.1	0.009 (sq. dist.)	0.1	0.01 (sq. dist.)
	10^2	km	0.1	0.006 (sq. dist.)	0.1	0.008 (sq. dist.)	0.1	0.009 (sq. dist.)	0.1	0.01 (sq. dist.)
	10^3	r.s.	0.1	0.001 (sq. dist.)	0.1	0.003 (sq. dist.)	0.1	0.005 (sq. dist.)	0.1	0.005 (sq. dist.)
	10^4	r.s.	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.002 (sq. dist.)
	10^5	r.s.	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)
	10^6	r.s.	0.1	0.0001 (sq. dist.)	N/A	N/A	N/A	N/A	N/A	N/A
II	10^2	km+	0.1	0.01 (sim.)	0.1	0.01 (sim.)	0.1	0 (sim.)	0.1	0 (sim.)
	10^2	km	0.1	0.01 (sim.)	0.1	0.01 (sim.)	0.1	0 (sim.)	0.1	0 (sim.)
	10^3	r.s.	0.01	0.0005 (sim.)	0.01	0.0005 (sim.)	0.01	0.0001 (sim.)	0.01	0.0001 (sim.)
	10^4	r.s.	0.01	0.001 (sim.)	0.01	0.001 (sim.)	0.01	0.001 (sim.)	0.01	0.001 (sim.)
III	10^2	km+	0.1	0.008 (sq. dist.)	0.1	0.005 (sq. dist.)	0.1	0.007 (sq. dist.)	0.1	0.01 (sq. dist.)
	10^2	km	0.1	0.007 (sq. dist.)	0.1	0.008 (sq. dist.)	0.1	0.008 (sq. dist.)	0.1	0.01 (sq. dist.)
	10^3	r.s.	0.1	0.002 (sq. dist.)	0.1	0.002 (sq. dist.)	0.1	0.003 (sq. dist.)	0.1	0.005 (sq. dist.)
	10^4	r.s.	0.1	0.001 (sq. dist.)	0.1	0.002 (sq. dist.)	0.1	0.002 (sq. dist.)	0.1	0.002 (sq. dist.)
	10^5	r.s.	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)	0.1	0.001 (sq. dist.)

[1] N/A (short for Not Available): we tried some parameter settings but always got the execution failure of the nvGRAPH's algorithm.
[2] km+: k-means++; km: k-means; r.s.: random sampling.

Performance comparison of the proposed processing chains

Tables 5.7, 5.8, 5.9 and 5.10 present the performance of our parallel processing chains for representative-based spectral clustering on the four large-scale datasets, respectively. Globally, the achieved clustering quality (measured by ARI and NMI scores introduced in Section 1.1.2)[3] and the elapsed time basically validate our strategy for the choice of representatives extraction methods according to the value of k_r (see Section 5.2.1).

In terms of running time comparison, it can be observed that:

- When the GPU RAM is sufficient to store the entire dataset (which is the case for the four benchmark datasets), then chain I (computations entirely on the GPU) should be adopted because it runs significantly faster than chain II (computations entirely on CPU) and chain III (computations on CPU+GPU).
- When the number of representatives to be extracted (k_r) is small ($< 10^4$), chain II achieves similar global performance to chain III, which validates the reasonability of staying on CPU for spectral clustering of k_r representatives.
- When k_r becomes large ($\geq 10^4$), spectral clustering on k_r representatives takes much more time on CPU than on GPU, hence chain III becomes preferable to chain II.

Besides, the spectral graph partitioning consumes more time than the similarity matrix construction, which is contrary to the results achieved on other datasets in Section 3.6.6. It seems that the LOBPCG eigensolver takes more iterations to converge on datasets with nonconvex clusters. The elapsed time of CPU-GPU data transfers in chain I and chain III is insignificant, and the input data attachment consumes little time regardless of the processing chain and the number of representatives.

Globally, our experiments validate the reasonability, high performance and good scalability of our parallel processing chains on CPU-GPU platforms for large-scale representative-based spectral clustering.

[3]Some scores of clustering quality achieved with a better extraction method or with more representatives do not appear to be higher, because we did not find better settings for connectivity parameters due to lack of time. Besides, the AMI scores are not displayed in the tables because they are usually equal to or very close to the NMI scores in our experiments.

Table 5.7: Performance of representative-based spectral clustering on Spirals-75M $(n, d, k_c) = (75M, 2, 3)$

Chain	k_r	Reps extract. method	Quality		Time (s)					
			ARI	NMI	Reps extract.	CPU-GPU transfers	SC on k_r reps		Attach.	Total
							Constr.	Partition.		
I	10^2	km+	0.156	0.300	1.26	0.27	0.0004	1.06	0.0009	2.59
	10^2	km	0.301	0.399	0.61	0.28	0.0005	1.07	0.0009	1.96
	10^3	km	1.000	1.000	5.86	0.28	0.0007	1.34	0.0009	7.48
	10^4	r.s.	1.000	1.000	1.92	0.27	0.007	1.64	0.0009	3.84
	10^5	r.s.	1.000	1.000	19.29	0.27	0.40	3.39	0.001	23.35
	10^6	r.s.	1.000	1.000	199.55	0.28	19.90	293.80	0.002	513.53
II	10^2	km+	0.010	0.009	55.66	N/A	0.003	0.08	0.38	56.12
	10^2	km	0.011	0.010	11.33	N/A	0.004	0.08	0.38	11.79
	10^3	km	1.000	1.000	115.98	N/A	0.06	1.55	0.32	117.91
	10^4	r.s.	1.000	1.000	48.99	N/A	3.31	42.31	0.38	94.99
III	10^2	km+	0.938	0.927	55.66	0.03	0.0005	1.05	0.01	56.75
	10^2	km	0.599	0.631	11.33	0.03	0.0005	1.03	0.01	12.40
	10^3	km	1.000	1.000	115.98	0.03	0.001	1.35	0.01	117.37
	10^4	r.s.	1.000	1.000	48.99	0.03	0.01	1.50	0.02	50.55
	10^5	r.s.	1.000	1.000	487.42	0.03	0.43	3.00	0.02	490.90

[1] Constr.: similarity matrix/graph construction; Partition.: spectral graph partitioning; Attach.: input data attachment.
[2] N/A: short for Not Applicable, because chain II is entirely on CPU.

Table 5.8: Performance of representative-based spectral clustering on Smile2-100M $(n, d, k_c) = (100M, 2, 4)$

Chain	k_r	Reps extract. method	Quality		Time (s)					
			ARI	NMI	Reps extract.	CPU-GPU transfers	SC on k_r reps		Attach.	Total
							Constr.	Partition.		
I	10^2	km+	1.000	1.000	1.75	0.35	0.0004	1.19	0.001	3.29
	10^2	km	0.727	0.811	1.01	0.35	0.0004	1.11	0.001	2.47
	10^3	km	1.000	1.000	10.18	0.34	0.0006	1.66	0.001	12.18
	10^4	r.s.	1.000	1.000	2.56	0.34	0.007	2.73	0.001	5.64
	10^5	r.s.	1.000	1.000	25.78	0.35	0.49	18.73	0.002	45.35
II	10^2	km+	0.535	0.597	72.15	N/A	0.004	0.07	0.46	72.68
	10^2	km	0.011	0.010	19.13	N/A	0.004	0.07	0.44	19.64
	10^3	km	1.000	1.000	200.36	N/A	0.07	0.98	0.50	201.91
	10^4	r.s.	1.000	1.000	65.33	N/A	3.43	16.24	0.51	85.51
III	10^2	km+	1.000	1.000	72.15	0.03	0.0005	1.11	0.01	73.30
	10^2	km	0.732	0.814	19.13	0.03	0.0006	1.21	0.01	20.38
	10^3	km	1.000	1.000	200.36	0.03	0.001	1.78	0.02	202.19
	10^4	r.s.	1.000	1.000	65.33	0.03	0.008	1.87	0.02	67.26
	10^5	r.s.	1.000	1.000	650.01	0.03	0.51	15.83	0.03	666.41

Table 5.9: Performance of representative-based spectral clustering on Aggregation-78.8M $(n, d, k_c) = (78.8M, 2, 7)$

Chain	k_r	Reps extract. method	Quality		Time (s)					
			ARI	NMI	Reps extract.	CPU-GPU transfers	SC on k_r reps		Attach.	Total
							Constr.	Partition.		
I	10^2	km+	0.989	0.983	1.32	0.28	0.0005	1.08	0.0009	2.68
	10^2	km	0.990	0.986	0.66	0.28	0.0005	1.10	0.0009	2.04
	10^3	km	0.990	0.986	7.51	0.27	0.001	1.09	0.0009	8.87
	10^4	r.s.	0.991	0.987	2.03	0.27	0.006	1.79	0.0009	4.09
	10^5	r.s.	0.989	0.984	20.35	0.27	0.40	2.94	0.001	23.96
II	10^2	km+	0.984	0.978	59.74	N/A	0.004	0.08	0.38	60.20
	10^2	km	0.011	0.010	10.52	N/A	0.004	0.09	0.38	10.99
	10^3	km	1.000	1.000	131.43	N/A	0.06	0.92	0.37	132.78
	10^4	r.s.	1.000	1.000	51.64	N/A	3.07	15.06	0.43	70.20
III	10^2	km+	0.994	0.991	59.74	0.03	0.0006	1.08	0.02	60.87
	10^2	km	0.984	0.978	10.52	0.03	0.0006	1.07	0.02	11.64
	10^3	km	0.992	0.988	131.43	0.03	0.001	1.12	0.02	132.60
	10^4	r.s.	0.994	0.991	51.64	0.03	0.007	1.22	0.02	52.92
	10^5	r.s.	0.993	0.991	512.05	0.03	0.41	2.56	0.02	515.08

Table 5.10: Performance of representative-based spectral clustering on Complex9-303M $(n, d, k_c) = (303.1M, 2, 9)$

Chain	k_r	Reps extract. method	Quality		Time (s)					
			ARI	NMI	Reps extract.	CPU-GPU transfers	SC on k_r reps		Attach.	Total
							Constr.	Partition.		
I	10^2	km+	0.470	0.726	4.82	1.01	0.0004	1.27	0.003	7.10
	10^2	km	0.557	0.780	1.83	1.03	0.0006	1.16	0.003	4.02
	10^3	km	0.454	0.715	24.40	1.05	0.001	1.44	0.003	26.89
	10^4	r.s.	0.563	0.811	7.74	1.04	0.007	1.90	0.003	10.69
	10^5	r.s.	0.699	0.874	79.30	1.04	0.38	4.22	0.004	84.94
II	10^2	km+	0.321	0.597	213.59	N/A	0.003	0.10	1.62	215.31
	10^2	km	0.341	0.617	30.72	N/A	0.004	0.08	1.61	32.41
	10^3	km	1.000	1.000	461.82	N/A	0.07	1.45	1.55	464.89
	10^4	r.s.	1.000	1.000	197.94	N/A	3.22	21.38	1.64	224.18
III	10^2	km+	0.433	0.734	213.59	0.03	0.0005	1.29	0.09	215.00
	10^2	km	0.383	0.670	30.72	0.03	0.0006	1.27	0.11	32.13
	10^3	km	0.521	0.764	461.82	0.03	0.001	1.45	0.10	463.40
	10^4	r.s.	0.660	0.850	197.94	0.03	0.010	1.52	0.07	199.57
	10^5	r.s.	0.840	0.930	1969.35	0.04	0.39	6.76	0.09	1976.63

5.4 . Summary

In this chapter, we have proposed three parallel processing chains on CPU-GPU platforms for representative-based spectral clustering on large-scale datasets. Our processing chains cover several scenarios depending on whether the GPU RAM is sufficient to store the entire dataset and on the number of representatives to be extracted. For each scenario, one of the chains makes the best use of the different hardware and different methods for representatives extraction. Experiments demonstrates the performance and scalability of the proposed chains.

Conclusion and Perspectives

In this dissertation, we have studied the efficient parallelization of some clustering algorithms on CPU and GPU platforms and succeeded in making them run significantly faster and being able to handle large-scale datasets. Our main contributions are summarized as follows.

A bibliographic study on clustering, especially on k-means clustering and spectral clustering. First of all, we provided an overview of clustering, covering its concept, applications, categorization of algorithms, typical algorithms in each category, and evaluation metrics. Then, we reviewed k-means clustering and spectral clustering from several aspects such as foundation, classical algorithms, strengths, weaknesses, and approaches to improvement. Particularly, we found that approximation and parallelization are two main approaches to address the scalability challenge of spectral clustering, and we conducted a dedicated survey of each approach.

Parallel, scalable and accurate k-means clustering: a CPU version and a GPU version. When applying k-means to large datasets using single precision arithmetic, we observed the numerical accuracy issue which results from the accumulation of rounding errors in the summation step of updating centroids. Thus, we proposed a two-level summation method based on the use of packages, which can reduce the accumulation of rounding errors and achieve satisfactory numerical accuracy without using double precision arithmetic. Then, we designed two optimized parallel implementations of the numerically accurate k-means algorithm on CPU and on GPU, respectively. Our CPU version employs OpenMP multithreading and auto-vectorization, while our GPU version exploits dynamic parallelism, multiple streams and shared memory. Finally, experimental results on large datasets demonstrate the numerical accuracy and high performance of our k-means implementations.

Parallel scalable spectral clustering on GPU. Classical algorithms of spectral clustering suffer from $\mathcal{O}(n^3)$ calculation cost and $\mathcal{O}(n^2)$ memory space requirements, where n is the number of data instances. Essentially, we exploit massively parallel GPU computing to deal with the high calculation cost, while performing matrix sparsification and using sparse storage format to reduce most of the memory space requirements. Specifically, we designed three different algorithms (named Algo CSR-1, Algo CSR-2, Algo CSR-3) and their optimized parallel implementations to construct the similarity matrix in CSR format on GPU. Our implementations mainly consist of home-made CUDA kernels with various optimizations and the use of some functions of NVIDIA's GPU-accelerated libraries (Thrust, cuSPARSE). Then, we take advantage of the spectral graph partitioning algorithms of

NVIDIA's nvGRAPH library to conduct the remaining steps of spectral clustering on GPU (including Laplacian matrix computation, eigenvectors computation, and final k-means clustering). In particular, the CSR format is used for all partitioning algorithms, but the algorithm with the LOBPCG eigensolver is preferred in our use.

Not surprisingly, experimental results demonstrate that our GPU implementations for the CSR matrix construction can run $\times 8.5$ to $\times 28.8$ faster than an optimized parallel CPU implementation of Algo CSR-1, while NVIDIA's LOBPCG-embedded algorithm (on GPU) can run $\times 8$ to $\times 28$ times faster than *scikit-learn*'s LOBPCG-embedded algorithm (on CPU).

Finally, our global GPU implementation for spectral clustering is scalable to millions of data instances in just a few seconds to a few minutes!

Parallel noise filtering on GPU for spectral clustering. The effectiveness of traditional spectral clustering algorithms can be easily corrupted by noise instances in the dataset. We independently designed two noise filtering methods for spectral clustering: one based on the number of nonzeros per row in the similarity matrix, and the other based on the degrees of vertices in the associated similarity graph. They both leverage the previously constructed similarity matrix in CSR format. We integrated them into a noise robust spectral clustering algorithm and developed an efficient parallel implementation on GPU. Experiments on various datasets show that our noise filtering implementation can greatly improve the robustness of spectral clustering to noise with small time overhead.

Representative-based spectral clustering on CPU-GPU architectures. Although our spectral clustering implementation on GPU can handle million-scale datasets in just a few seconds to a few minutes, it would start to be too time-consuming to address datasets larger than the million scale ($n \geq 10^7$). To break this bottleneck, we adopted the representative-based approximation framework for spectral clustering and integrated it with parallel computing on CPU-GPU platforms. We compared three methods for representatives extraction: random sampling, k-means, and k-means++. Then, we considered three possible scenarios and proposed an adapted parallel processing chain for each scenario. As expected, our preceding works (i.e. parallel k-means on CPU and GPU, parallel spectral clustering on GPU) readily serve as modules in the proposed chains. Finally, global experiments exhibit the high performance and scalability of our spectral clustering chains on CPU-GPU platforms. For example, spectral clustering on hundreds of millions (instead of millions) of data instances also takes only a few seconds to a few minutes!

Finally, we suggest several research directions that seem most relevant to extending our work:

- Our spectral clustering implementation on GPU relies heavily on the eigensolver-embedded algorithms of NVIDIA's nvGRAPH library. Unfortunately, NVIDIA is no longer actively developing the nvGRAPH product since its last release in November 2019. Thus, it would be better to find some alternative sparse eigensolvers (especially LOBPCG) that are optimized for the GPU. For instance, the AmgX library [134] contains multiple GPU-accelerated eigensolvers including the LOBPCG solver, but their effectiveness of being used for spectral clustering remains to be tested.

- It would be interesting to study the parallelization of clustering algorithms on other parallel architectures such as multi-GPU machines. For spectral clustering, we think our CSR algorithms for similarity matrix construction can be easily adapted to multi-GPU architectures by going through some consecutive rows of the similarity matrix on each GPU, however we cannot find a multi-GPU implementation of the LOBPCG eigensolver. Nevertheless, we can at least transfer the results of all GPUs to a single GPU and then call nvGRAPH's LOBPCG-embedded API on that single GPU. The data transfers between multiple GPUs can be achieved using the NVIDIA Collective Communication Library (NCCL). For k-means clustering, we can parallelize the *ComputeAssign* step on multiple GPUs by performing the distance computations between some data instances and k_c centroids on each GPU. Then the local summation per cluster on some packages of instances can be performed on each GPU. Finally, the summation results of all GPUs can be transferred to a single GPU, where the global summation results are computed and the centroids are updated (*Update* step).

- We have adopted the representative-based approximation for spectral clustering. It is also attractive to try other approximation methods, such as Nyström-based [59] and landmark-based [34] methods, and study their efficient parallelization on modern parallel architectures.

- For a given dataset, tuning multiple parameters to achieve good clustering quality or optimize code performance is usually a complex task in practice, requiring a lot of experimental attempts. It would be very useful to reduce the number of parameters that need to be carefully tuned, or to develop some efficient methods for automatic tuning, or to find some practical rules of thumb from experiments.

- k-means and spectral clustering are just two of the countless clustering algorithms. Therefore, a broad area of research is to explore parallel acceleration of other effective but computationally intensive clustering algorithms, such as density-based algorithms [53] and affinity propagation [61].

On the other hand, we did not concern ourselves with the energy consumption of our computations. Nevertheless, this issue has become very important during recent years. A possibility would be to adapt our clustering algorithms to FPGA processors, in order to achieve better energy efficiency rather than only high computation speed [211]. Some research efforts within the ParSys team of LISN are also starting to explore this way for other types of parallel computations.

Appendices

A - Benchmark datasets

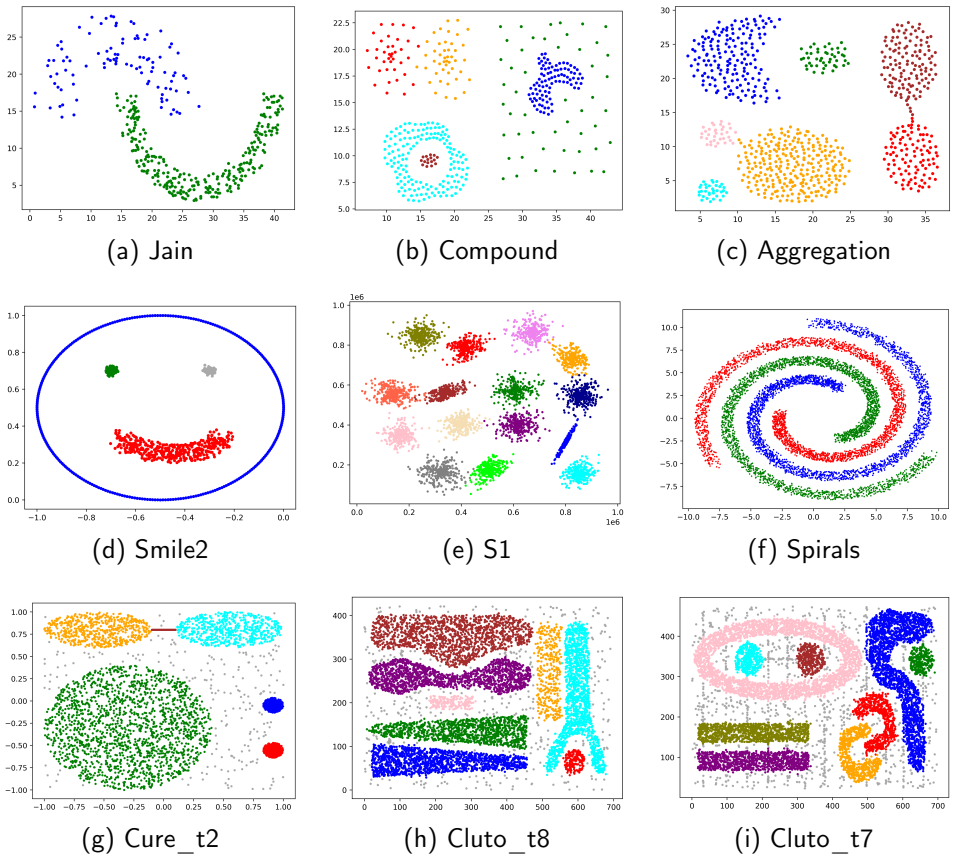
In this dissertation, we experimented on a variety of datasets ranging from synthetic to real-world datasets, from small-scale to large-scale datasets, from 2D to high-dimensional datasets, from convex shape datasets to arbitrary shape datasets, and from noise-free to noise-laden datasets. Except some home-made synthetic datasets (HMSD), most benchmark datasets are from the following open sources: Clustering Basic Benchmark (CBB)[1], GitHub Clustering Benchmarks (GCB)[2], UC Irvine Machine Learning Repository (UCI)[3], The MNIST Database of Handwritten Digits (MNIST)[4], The Infinite MNIST Datasets (InfiMNIST)[5], Lancaster University (LU)[6].

Table A.1 gives an overview of the 2D small-scale benchmark datasets, and Figure A.1 exhibits their ground truth clusterings.

Table A.1: Features of small-scale benchmark datasets

Dataset	**# instances** (n)	**# dims** (d)	**# clusters** (k_c)	**.txt file size**	**Source**
Jain	373	2	2	5 KB	CBB
Compound	399	2	6	5 KB	CBB
Aggregation	788	2	7	9 KB	CBB
Smile2	1 000	2	4	20 KB	GCB
S1	5 000	2	15	74 KB	CBB
S4	5 000	2	15	74 KB	CBB
Spirals	7 500	2	3	189 KB	LU
Cure_t2	4 200	2	7	82 KB	GCB
Complex9	3 031	2	9	49 KB	GCB
Cluto_t7	10 000	2	10	217 KB	GCB
Cluto_t8	8 000	2	9	180 KB	GCB

[1] http://cs.joensuu.fi/sipu/datasets/
[2] https://github.com/deric/clustering-benchmark
[3] https://archive.ics.uci.edu/ml/index.php
[4] http://yann.lecun.com/exdb/mnist/
[5] https://leon.bottou.org/projects/infimnist
[6] https://www.lancaster.ac.uk/pg/hyder/Downloads/downloads.html



(a) Jain (b) Compound (c) Aggregation

(d) Smile2 (e) S1 (f) Spirals

(g) Cure_t2 (h) Cluto_t8 (i) Cluto_t7

Figure A.1: 2D small-scale datasets

Table A.2 specifies the medium-scale and large-scale benchmark datasets. They can be divided into four types depending on how the data is produced.

- **Synthetic 4D datasets: Syn4D-1M, Syn4D-5M, Syn4D-50M.** They are produced by our data generator[7]. The generated n instances are uniformly distributed in 4 convex clusters ($\frac{n}{4}$ instances in each cluster). Each cluster has a radius of 9 and the centroids are supposed to be (40, 40, 60, 60), (40, 60, 60, 40), (60, 40, 40, 60) and (60, 60, 40, 40), respectively.

- **Synthetic 2D datasets: Spirals-75M, Smile2-100M, Aggregation-78.8M, and Complex9-303M sets.** They are produced by our data amplifier[8] which amplifies small-scale 2D datasets by incorporating random fluctuations, as shown in Figure A.2. Let a denote the amplification factor and f denote the fluctuation factor. For each point of a small-scale

[7] https://gitlab-research.centralesupelec.fr/Stephane.Vialle/cpu-gpu-kmeans

[8] https://github.com/guanlin-he/clustering-release

2D dataset, we create $a-1$ new points that randomly lie in its f-related neighborhood, thus the amplified dataset has a times the size of its original dataset. We control the value of f such that the clusters of the amplified dataset are similarly separated compared to the original dataset.

- **MNIST-based datasets: MNIST-60K, MNIST-120K, MNIST-240K.** The first one is the training set of the well-known MNIST database of handwritten digits[9], while the other MNIST-based sets are produced using the InfiMNIST code[10]. They all have 784 dimensions and 10 clusters.

- **Real-world datasets: Household power consumption (HPO), US census 1990 (USC).** They come from the UCI Machine Learning Repository [50] and their ground truth clusterings are unavailable.

  The Household power consumption (HPO) dataset contains 2 075 259 measurements of electric power consumption in a household over a period of nearly 4 years. Each measurement has 9 attributes. We remove the measurements containing missing values and also remove the first 2 attributes that record the date and time of measurements. The remaining set that we use for evaluation contains 2,049,280 measurements with 7 numerical attributes, i.e. $n = 2\,049\,280$, $d = 7$.

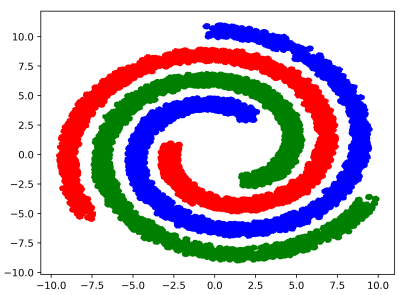
  The US census 1990 (USC) dataset contains 2 458 285 instances with 68 categorical attributes (i.e. $n = 2\,458\,285$, $d = 68$). It is actually a simplified and discretized version of the USCensus1990raw dataset which contains one percent sample drawn from the full 1990 US census data.

[9] http://yann.lecun.com/exdb/mnist/
[10] https://leon.bottou.org/projects/infimnist

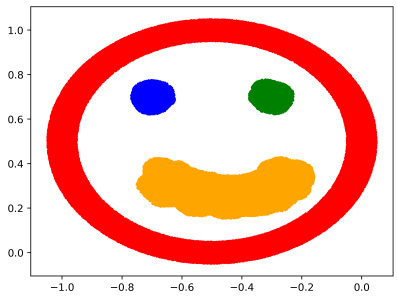
Table A.2: Features of medium-scale and large-scale benchmark datasets

Dataset	# instances (n)	# dims (d)	# clusters (k_c)	.txt file size	Source
Spirals-75M	75 000 000	2	3	1.4 GB	HMSD
Smile2-100M	100 000 000	2	4	1.9 GB	HMSD
Aggregation-78.8M	78 800 000	2	7	1.5 GB	HMSD
Complex9-303M	303 100 000	2	9	6.4 GB	HMSD
Syn4D-1M	1 000 000	4	4	30 MB	HMSD
Syn4D-5M	5 000 000	4	4	181 MB	HMSD
Syn4D-50M	50 000 000	4	4	1.77 GB	HMSD
HPO*	2 049 280	7	Unknown	56 MB	UCI
USC	2 458 285	68	Unknown	326 MB	UCI
MNIST-60K	60 000	784	10	104 MB	MNIST
MNIST-120K	120 000	784	10	210 MB	InfiMNIST
MNIST-240K	240 000	784	10	421 MB	InfiMNIST

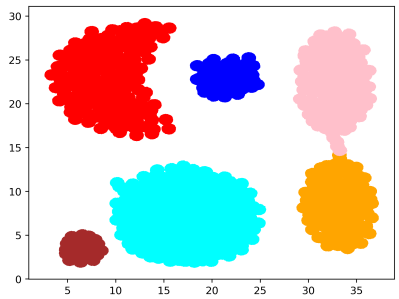
* This dataset is made available under the "Creative Commons Attribution 4.0 International (CC BY 4.0)" license.



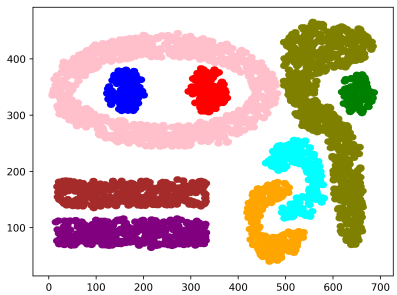
(a) Spirals-75M based on Spirals ($a = 10^4$, $f = 1\%$)



(b) Smile2-100M based on Smile2 ($a = 10^5$, $f = 5\%$)



(c) Aggregation-78.8M based on Aggregation ($a = 10^5$, $f = 2\%$)



(d) Complex9-303M based on Complex9 ($a = 10^5$, $f = 1\%$)

Figure A.2: Large-scale datasets generated by amplifying small-scale datasets with random fluctuations (a denotes amplification factor, f denotes fluctuation factor)

B - Testbed features

All experiments in this dissertation have been carried out on a server named *john3* located at the Metz Campus of CentraleSupelec. The hardware and software features of *john3* are presented in Tables B.2 and B.1, respectively. Essentially, *john3* consists of two Intel Xeon Silver 4114 processors as CPU and one NVIDIA GeForce RTX as GPU. Particularly, we upgraded the GPU hardware, the CUDA version, and the OS during the preparation of this dissertation. For this reason, the experimental results in Chapter 2 were obtained with RTX 2080 Ti, CUDA 10.2, and Ubuntu 18, while those in Chapters 3, 4 and 5 were obtained with RTX 3090, CUDA 11.5, and Ubuntu 20. The CPU and GPU are always connected by a PCIe 3.0 x16 bus.

Table B.1: Software features of our *john3* server

Operating system	Ubuntu 18 (for Chapter 2) → Ubuntu 20.04.3 (for Chapters 3, 4, 5)
CPU code compiler	gcc 9.3.0
CPU parallelization tool	OpenMP
Python version	3.8.10
scikit-learn version	0.22.2.post1 → 1.1.1 (only for Section 5.3.2)
CUDA version	10.2 (for Chapter 2) 11.5 (for Chapters 3, 4, 5)

Table B.2: Hardware features of our *john3* server

CPU	2 Intel Xeon Silver 4114 Processors	
Launch date	July 2017	
Architecture	Skylake	
# of physical cores per processor	10	
Processor base frequency	2.20 GHz	
Hyper-threading	Yes	
Instruction set extensions	SSE4.2, AVX, AVX2, AVX-512	
# of AVX-512 FMA units per processor	1	
L3 cache size per processor	13.75 MB	
RAM size	96 GB	
Max. memory speed	2.40 GHz	
SGEMM perf. per processor	619.2 GFLOPS[a]	
PCIe bus	PCI Express 3.0 x16	
Launch date	November 2010	
Bandwidth x16	16 GB/s (theoretical), 12.5 GB/s (experimental)	
GPU*	1 NVIDIA GeForce RTX 2080 Ti (for Chapter 2)	1 NVIDIA GeForce RTX 3090 (for Chapters 3, 4, 5)
Launch date	September 2018	September 2020
Architecture	Turing	Ampere
Compute capability	7.5	8.6
GPU max clock rate	1.65 GHz	1.70 GHz
Stream Multiprocessors (SM)	68	82
CUDA cores per SM	64	128
CUDA cores	4352	10496
Peak single precision (FP32) perf.	14.2 TFLOPS[b]	35.6 TFLOPS[c]
Memory clock rate	7 GHz	9.75 GHz
Memory bus width	352-bit	384-bit
Total # of registers per block	65536	65536
Total shared memory per block	48 KB	48 KB
Total shared memory per SM	64 KB	100 KB
L2 cache size	5.5 MB	6 MB
GPU global memory	11 GB	24 GB
Warp size	32	32
Max. # of threads per block	1024	1024
Max. # of threads per SM	1024	1536
Max. dim. size of a block (x, y, z)	(1024, 1024, 64)	(1024, 1024, 64)
Max. dim. size of a grid (x, y, z)	$(2^{31}-1, 65535, 65535)$	$(2^{31}-1, 65535, 65535)$

* We upgraded the GPU from RTX 2080 Ti to RTX 3090 during the dissertation preparation. Only one GPU was present on *john3* at a time.

[a] Source: https://gadgetversus.com/processor/dual-intel-xeon-silver-4114-specs/

[b] Source: https://wccftech.com/review/nvidia-geforce-rtx-2080-ti-and-rtx-2080-review/2/

[c] Source: https://www.sie.es/wp-content/uploads/2021/02/GPU_Evaluation_report_IFIC.pdf

C - GPU implementation for Algo CSR-1

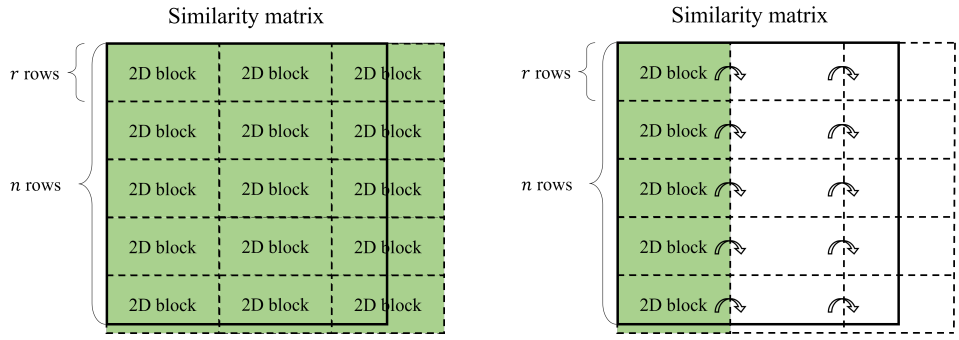
Listings C.1, C.2 and C.3 show the host code and two optimized CUDA kernels of our GPU implementation for Algo CSR-1 (Algorithm 4 in Section 3.3.3).

```
1  #include <thrust/...> // Include Thrust library functions
2  ... // Declaration, memory allocation & initialization
3
4  // Launch the first-pass kernel
5  Db.x = BSX;   Db.y = BSY;;
6  Dg.x = n/Db.x + (n%Db.x > 0 ? 1 : 0);
7  Dg.y = n/Db.y + (n%Db.y > 0 ? 1 : 0);
8  shMemSize = sizeof(int)*Db.y;
9  1stPass<<<Dg,Db,shMemSize>>>(...,             // input
10                              GPU_nnzPerRow);  // output
11
12 // Find the minimum and maximum number of nonzeros in a row
13 thrust::device_ptr<int> d_nnzPerRow(GPU_nnzPerRow);
14 thrust::pair<...> extrema =
15       thrust::minmax_element(..., d_nnzPerRow, d_nnzPerRow + n);
16 minNnzRow = *extrema.first;
17 maxNnzRow = *extrema.second;
18
19 // Compute csrRow by an exclusive scan on nnzPerRow
20 thrust::device_ptr<int> d_csrRow(GPU_csrRow);
21 thrust::exclusive_scan(..., d_nnzPerRow, d_nnzPerRow + n+1, d_csrRow);
22
23 // Get the nnz of sim. matrix and allocate memory for csrVal & csrCol
24 nnz = d_csrRow[n];
25 cudaMalloc((void**) &GPU_csrCol, sizeof(int)*nnz);
26 cudaMalloc((void**) &GPU_csrVal, sizeof(float)*nnz);
27
28 // Launch the second-pass kernel
29 Db.x = BSX;                              Db.y = BSY;
30 Dg.x = n/Db.y + (n%Db.y > 0 ? 1 : 0);    Dg.y = 1;
31 shMemSize = sizeof(float)*Db.y*Db.x + sizeof(int)*(Db.y*Db.x + Db.y);
32 2ndPass<<<Dg,Db,shMemSize>>>(GPU_csrRow, ...,  // input
33                              GPU_csrVal,       // output
34                              GPU_csrCol);      // output
```

Listing C.1: Host code of GPU implementation for Algo CSR-1

For the `1stPass` kernel, we choose to create a 2D grid with 2D blocks of threads (Listings C.1, lines 5-7). As shown in Figure C.1 (a), the grid covers all the elements of similarity matrix. Thus each thread takes care of one matrix element, and count it as a nonzero if the predefined threshold is satisfied (Listings C.2, lines 17-25). Then the number of nonzeros is first accumulated within each block into shared memory using the `atomicAdd_block` operation (Listings C.2, line 28). Finally we accumulate the results of blocks of the same row to get the number of nonzeros per row into global memory using classic `atomicAdd` operation (Listings C.2, lines 34-36). Although the design of this kernel is typical, it should be noted that the maximum y-dimension of a grid (65535) is far smaller than the maximum x-dimension of a grid so the calculated number of blocks in y dimension

(Listings C.1, line 7) may exceed the limit if n is large enough. In this case, we consider the horizontal partitioning of the similarity matrix into chunks as large as possible and launch one grid for each chunk.



(a) 2D grid with 2D blocks for the `1stPass` kernel of Algo CSR-1

(b) 1D grid with 2D blocks for the `2ndPass` kernel of Algo CSR-1 & for all the kernels of Algo CSR-2

Figure C.1: Grid and block configuration of the CUDA kernels for CSR format similarity matrix construction

For the `2ndPass` kernel, we choose to create a 1D grid with 2D blocks (Listings C.1, lines 29-30). Several points need to be noted:

1. As shown in Figure C.1 (b), each block of threads processes some rows of the similarity matrix in an iterative fashion, i.e. moving forward segment by segment, so that each block knows its own sections for storing nonzeros in `csrVal[]` and `csrCol[]` (according to `csrRow[]`) and meanwhile different blocks can work independently in parallel.

2. In each iteration, each block of threads parallelly computes a segment of similarity matrix, finds threshold-satisfied nonzeros and stores them into shared memory arrays (Listings C.3, lines 23-38). Then only the threads in the first column of each block copy the nonzeros from shared to global memory (Listings C.3, lines 42-51).

3. Since usually only a fraction of elements are nonzeros, we count the number of nonzeros per iteration into a shared memory array (Listings C.3, line 32) and check whether there are nonzeros in each iteration before proceeding (Listings C.3, line 42), which should avoid many unnecessary operations.

4. Particularly, when testing whether an element in shared memory is nonzero or not (Listings C.3, line 44), we choose to check its column index (vs. -1) instead of its similarity value (vs. 0) because there is a risk that the floating-point underflow may occur for the similarity value if it is too small.

5. Again considering the maximum y-dimension of a grid (65535) may be insufficient in case of large n while the maximum x-dimension of a grid ($2^{31} - 1 = 2\,147\,483\,647$) is usually sufficiently large, we choose to create the 1D grid in x dimension (Listings C.1, line 30) but regard it as in y dimension (Listings C.3, line 5).

For the other steps, we leverage some easy-to-use APIs of NVIDIA's Thrust library [145]. Based on the number of nonzeros per row obtained in the first pass, the `minmax_element` API is used to find the minimum and maximum number of nonzeros in a row (Listings C.1, lines 13-17) and the `exclusive_scan` API is used to derive `csrRow[]` (Listings C.1, lines 20-21).

```
1  // Starting address for dynamic allocation of shared memory
2  extern __shared__ float shBuff[];
3
4  __global__ void 1stPass (...)
5  {
6    // 2D blocks, 2D grid
7    int col = blockIdx.x * blockDim.x + threadIdx.x;
8    int row = blockIdx.y * blockDim.y + threadIdx.y;
9
10   // Declaration & initialization
11   int nnzThread = 0;
12   int *shNnz = (int*)shBuff;
13   if (threadIdx.x == 0)  shNnz[threadIdx.y] = 0;
14   __syncthreads();
15
16   if (col < n && row < n) {
17     // Uniform or Gaussian sim. with threshold for squared distance
18     #ifdef defined(UNI_SIM_WITH_SQDIST_THOLD) ||
19            defined(GAUSS_SIM_WITH_SQDIST_THOLD)
20       ... // Calculate squared distances (sqDist)
21       if (sqDist < tholdSqDist && row != col)  nnzThread++;
22     #endif
23
24     // Other similarity metrics with threshold
25     #ifdef ... #endif
26
27     // Accumulate the results within a block
28     if (nnzThread > 0) atomicAdd_block(&shNnz[threadIdx.y], nnzThread);
29   }
30
31   __syncthreads();
32
33   // Store the final result into global memory
34   if (threadIdx.x == 0 && row < n)
35     if (shNnz[threadIdx.y] > 0)
36       atomicAdd(&GPU_nnzRow[row], shNnz[threadIdx.y]);
37 }
```

Listing C.2: `1stPass` kernel for Algo CSR-1

```
__global__ void 2ndPass (...)
{
  // 1D block in x-axis, 1D grid in x-axis but regarded as in y-axis
  int col = threadIdx.x;
  int row = blockDim.y * blockIdx.x + threadIdx.y;

  // Declaration & initialization
  int maxCol = ((n - 1)/blockDim.x + 1) * blockDim.x;
  int yofs = threadIdx.y * blockDim.x;
  int nnzOffset;
  // Pointers to dynamic shared memory arrays (must be 1D)
  float *shValIter = shBuff;  // size: blockDim.y*blockDim.x
  int   *shColIter =          // size: blockDim.y*blockDim.x
                     (int*)&shValIter[blockDim.y*blockDim.x];
  int   *shNnzIter =          // size: blockDim.y
                     &shColIter[blockDim.y*blockDim.x];
  if (threadIdx.x == 0)  shNnzIter[threadIdx.y] = 0;
  if (row < n) nnzOffset = GPU_csrRow[row];
  __syncthreads();

  // Each block finds & records the nonzeros of 1 row in a loop fashion
  while (col < maxCol && row < n) {
    if (col < n) {
      // Gaussian similarity with threshold for squared distance
      #ifdef GAUSS_SIM_WITH_SQDIST_THOLD
        ... // Calculate squared distances (sqDist)
        shColIter[yofs + threadIdx.x] = -1;
        if (sqDist < tholdSqDist && row != col) {
          shValIter[yofs + threadIdx.x] =
                    __expf((-1.0f)*sqDist/(2.0f*sigma*sigma));
          shColIter[yofs + threadIdx.x] = col;
          atomicAdd(&shNnzIter[threadIdx.y], 1);
        }
      #endif

      // Other similarity metrics with threshold
      #ifdef ... #endif
    }
    __syncthreads();

    // Store nonzeros into global CSR arrays
    if (shNnzIter[threadIdx.y] > 0 && threadIdx.x == 0) {
      for (int i = 0; i < blockDim.x && col + i < n; i++) {
        if (shColIter[yofs + i] != -1) {
          GPU_csrVal[nnzOffset] = shValIter[yofs + i];
          GPU_csrCol[nnzOffset] = col + i; // i.e. shColIter[yofs + i]
          offset++;
        }
      }
      shNnzIter[threadIdx.y] = 0;
    }
    __syncthreads();

    col += blockDim.x;
  } // End of while
}
```

Listing C.3: `2ndPass` kernel for Algo CSR-1

D - GPU implementation for Algo CSR-2

Listings D.1, D.2, D.3, D.4, D.5, D.6 show the host code and three optimized CUDA kernels of our GPU implementation for Algo CSR-2 (Algorithm 5 in Section 3.3.4). For each kernel, we choose to create a 1D grid with 2D blocks of threads (Listings D.1, lines 5-6, 35-36, 48-49) so that each block is in charge of a few rows of the similarity matrix. The points 1, 4 and 5 related to the `2ndPass` kernel of Algo CSR-1 (see Section 3.3.3) also apply to the kernels of Algo CSR-2.

For the `fullPass` kernel, we declare several shared memory arrays for storing similarities in dense format, storing nonzeros in Ellpack format, and some other uses (Listings D.2, lines 18-28). Note that 2D blocks will demand too much shared memory if $hypoMaxNnzRow$ is large, so to support larger hypothesis we need to reduce block y dimension (e.g. use 1D blocks). In each iteration, each block of threads parallelly computes a segment of similarity matrix, finds threshold-satisfied nonzeros and stores all similarities of the segment into shared memory arrays in dense format (Listings D.3, lines 7-21). Then the nonzeros stored in the dense-format shared arrays are found and accumulated into Ellpack-format shared arrays by only the threads in the first column of each block (Listings D.3, lines 29-36). Meanwhile these threads also record the restart column indexes (aligned to multiples of 32 memory words for performance concern) and corresponding restart nonzero element indexes in each row in case the number of nonzeros per row exceeds $hypoMaxNnzRow$ (Listings D.3, lines 30 & 40). Since usually only a fraction of elements are nonzeros, we also record the number of nonzeros found per iteration so that we can avoid the accumulating and recording operations in case no nonzero is found in an iteration (Listings D.3, lines 16, 25 & 44). This helps to reduce warp divergence. Similarly, we set a flag once the $hypoMaxNnzRow$ is reached so as to avoid unnecessary operations (Listings D.3, lines 28 & 37). Additionally, the number of nonzeros per iteration is accumulated into the number of nonzeros per row. After finishing the outermost loop, a fraction of threads update the restart indexes in case the number of nonzeros in a row is no more than $hypoMaxNnzRow$ (Listings D.4, lines 8-11). Now since the nonzeros are contiguously stored in shared Ellpack arrays, each block of threads parallelly and iteratively copy the nonzeros into global Ellpack arrays with coalescence (Listings D.4, lines 18-24). Finally, a fraction of threads store the number of nonzeros per row and the restart indexes into global memory arrays (Listings D.4, lines 27-31).

For the `ellpackToCSR` kernel, each block of threads first loads its global starting offsets and per-row ending offsets for storing nonzeros (Listings D.5, lines 8-12). Then the nonzeros that have been successfully recorded in global Ellpack arrays are iteratively copied into global CSR arrays with coalescence (Listings D.5, lines 15-20). Finally a fraction of threads record the global restart index (for storing nonzeros) by adding the global starting offsets and per-row ending offsets

(Listings D.5, lines 23-25).

```
#include <thrust/...> // Include Thrust library functions
... // Declaration & memory allocation

// Launch the full-pass kernel
Db.x = BSX;                                Db.y = BSY;
Dg.x = n/Db.y + (n%Db.y > 0 ? 1 : 0);   Dg.y = 1;
shMemSize = sizeof(float)*(Db.y*Db.x + Db.y*hypoMaxNnzRow) +
            sizeof(int)*(Db.y*Db.x + Db.y*hypoMaxNnzRow + Db.y*3);
fullPass<<<Dg,Db,shMemSize>>>(hypoMaxNnzRow, ...,  // input
                              GPU_nnzPerRow,       // output
                              GPU_csrValMax,       // output
                              GPU_csrColMax,       // output
                              GPU_idxNzRowRestart, // output
                              GPU_colRestart);     // output

// Find the minimum and maximum number of nonzeros in a row
thrust::device_ptr<int> d_nnzPerRow(GPU_nnzPerRow);
thrust::pair<...> extrema =
      thrust::minmax_element(..., d_nnzPerRow, d_nnzPerRow + n);
minNnzRow = *extrema.first;
maxNnzRow = *extrema.second;

// Compute csrRow by an exclusive scan on nnzPerRow
thrust::device_ptr<int> d_csrRow(GPU_csrRow);
thrust::exclusive_scan(..., d_nnzPerRow, d_nnzPerRow + n+1, d_csrRow);

// Get the nnz of sim. matrix and allocate memory for csrVal & csrCol
nnz = d_csrRow[n];
cudaMalloc((void**) &GPU_csrCol, sizeof(int)*nnz);
cudaMalloc((void**) &GPU_csrVal, sizeof(float)*nnz);

// Launch a kernel to fill csrVal and csrCol
// with valid nonzeros stored in csrValMax and csrColMax
int *GPU_idxNzTotalRestart = GPU_idxNzRowRestart;
Db.x = BSX;                                Db.y = BSY;
Dg.x = n/Db.y + (n%Db.y > 0 ? 1 : 0);   Dg.y = 1;
ellpackToCSR<<<Dg,Db>>>(GPU_csrRow,             // input
                        GPU_idxNzRowRestart,    // input
                        hypoMaxNnzRow,          // input
                        GPU_csrValMax,          // input
                        GPU_csrColMax,          // input
                        GPU_csrVal,             // output
                        GPU_csrCol,             // output
                        GPU_idxNzTotalRestart); // output

if (maxNnzRow > hypoMaxNnzRow) {
  // Launch the supplementary-pass kernel
  Db.x = BSX;                                Db.y =BSY;
  Dg.x = n/Db.y + (n%Db.y > 0 ? 1 : 0);   Dg.y = 1;
  shMemSize = sizeof(float)*Db.y*Db.x + sizeof(int)*(Db.y*Db.x+Db.y);
  supPass<<<Dg,Db,shMemSize>>>(GPU_csrRow,            // input
                               GPU_idxNzTotalRestart, // input
                               GPU_colRestart,        // input
                               hypoMaxNnzRow, ...,    // input
                               GPU_csrVal,            // output
                               GPU_csrCol);           // output
}

... // Memory deallocation for auxiliary arrays
```

Listing D.1: Host code of GPU implementation for Algo CSR-2

The supPass kernel (Listings D.6) is similar to the 2ndPass kernel of Algo CSR-1. However, the difference is that each block of threads in the supPass kernel starts the work from the restart indexes recorded before (Listings D.6, lines 6-12) while in the 2ndPass kernel of Algo CSR-1 each block of threads starts the work from the beginning of each row.

Similar to Algo CSR-1, we leverage some easy-to-use APIs of NVIDIA's Thrust library to implement the other steps of Algo CSR-2, i.e. the minmax_element API is used to find the minimum and maximum number of nonzeros in a row (Listings D.1, lines 17-21) and the exclusive_scan API is used to derive the csrRow array (Listings D.1, lines 24-25).

```
1  // Starting address for dynamic allocation of shared memory
2  extern __shared__ float shBuff[];
3
4  __global__ void fullPass (...)
5  {
6    // 2D block, 1D grid in x-axis but regarded as in y-axis
7    int col = threadIdx.x;
8    int row = blockDim.y * blockIdx.x + threadIdx.y;
9    // Declaration & initialization
10   int maxCol = ((n - 1)/blockDim.x + 1) * blockDim.x;
11   int flagReachHypo = 0;
12   int idxNzRowRestart = 0;
13   int colRestart = n;
14   int yofs = threadIdx.y*blockDim.x;
15   int ymofs = threadIdx.y*hypoMaxNnzRow;
16
17   // Pointers to dynamic shared memory arrays (must be 1D)
18   float *shValIter  = shBuff;  // size: blockDim.y*blockDim.x
19   float *shNzValMax = &shValIter[blockDim.y*blockDim.x];
20                     // size: blockDim.y*hypoMaxNnzRow
21   int   *shColIter  = (int*)&shNzValMax[blockDim.y*hypoMaxNnzRow];
22                     // size: blockDim.y*blockDim.x
23   int   *shNzColMax = &shColIter[blockDim.y*blockDim.x];
24                     // size: blockDim.y*hypoMaxNnzRow
25   int   *shNnzIter  = &shNzColMax[blockDim.y*hypoMaxNnzRow];
26                     // size: blockDim.y
27   int   *shNnzRow   = &shNnzIter[blockDim.y];       // size: blockDim.y
28   int   *shIdxNzRowRestart = &shNnzRow[blockDim.y]; // size: blockDim.y
29
30   if (threadIdx.x == 0) {
31     shNnzIter[threadIdx.y] = 0;
32     shNnzRow[threadIdx.y] = 0;
33   }
34   __syncthreads();
35
36   ... // Part 2 of the kernel
37   ... // Part 3 of the kernel
38 }
```

Listing D.2: fullPass kernel (part 1) for Algo CSR-2

```
__global__ void fullPass (...)
{
  ... // Part 1 of the kernel

  // Each block processes some rows in a loop fashion
  while (col < maxCol && row < n) {
    if (col < n) {
      // Gaussian similarity with threshold for squared distance
      #ifdef GAUSS_SIM_WITH_SQDIST_THOLD
        ... // Calculate squared distances (sqDist)
        shColIter[yofs + threadIdx.x] = -1;
        if (sqDist < tholdSqDist && row != col) {
          shValIter[yofs + threadIdx.x] =
                    __expf((-1.0f)*sqDist/(2.0f*sigma*sigma));
          shColIter[yofs + threadIdx.x] = col;
          atomicAdd(&shNnzIter[threadIdx.y], 1);
        }
      #endif

      #ifdef ... #endif // Other similarity metrics with threshold
    } // End of if (col < n)
    __syncthreads();

    // Copy nonzeros to shared Ellpack arrays & records restart indexes
    if (shNnzIter[threadIdx.y] > 0 && threadIdx.x == 0) {
      int idxNzRow = shNnzRow[threadIdx.y];
      int i = 0;
      if (flagReachHypo == 0) {
        for (; i<blockDim.x && idxNzRow<hypoMaxNnzRow && col+i<n; i++){
          if (i%32==0) {idxNzRowRestart=idxNzRow; colRestart=col+i;}
          if (shColIter[yofs + i] != -1) {
            shNzValMax[ymofs + idxNzRow] = shValIter[yofs + i];
            shNzColMax[ymofs + idxNzRow] = col + i; //shColIter[yofs+i]
            idxNzRow++;
          }
        } // End of for loop
        if (idxNzRow == hypoMaxNnzRow)  flagReachHypo = 1;
      } // End of if (flagReachHypo == 0)
      for (; idxNzRow==hypoMaxNnzRow && i<blockDim.x && col+i<n; i++){
        if (i%32==0) {idxNzRowRestart=hypoMaxNnzRow; colRestart=col+i;}
        if (shColIter[yofs + i] != -1)  idxNzRow++;
      } // End of for loop
      shNnzRow[threadIdx.y]  += shNnzIter[threadIdx.y];
      shNnzIter[threadIdx.y] = 0;
    } // End of if (*shNnzIter > 0 && threadIdx.x == 0)

    __syncthreads();
    col += blockDim.x;
  } // end of while

  ... // Part 3 of the kernel
}
```

Listing D.3: `fullPass` kernel (part 2) for Algo CSR-2

```
__global__ void fullPass (...)
{
  ... // Part 1 of the kernel
  ... // Part 2 of the kernel

  // Update restart indexes
  if (threadIdx.x == 0 && row < n) {
    if (shNnzRow[threadIdx.y] <= hypoMaxNnzRow) {
      idxNzRowRestart = shNnzRow[threadIdx.y];
      colRestart = n;
    }
    shIdxNzRowRestart[threadIdx.y] = idxNzRowRestart;
  }
  __syncthreads();

  // Each block of threads parallelly store nonzeros from shared
  // Ellpack arrays into global Ellpack arrays in the coalesced way
  col = threadIdx.x;
  while (col < shIdxNzRowRestart[threadIdx.y] && row < n) {
    size_t csrMaxIdx = (size_t)row*(size_t)hypoMaxNnzRow + (size_t)col;
    GPU_csrValMax[csrMaxIdx] = shNzValMax[ymofs + col];
    GPU_csrColMax[csrMaxIdx] = shNzColMax[ymofs + col];
    col += blockDim.x;
  }

  // Store nnz per row and restart indexes into global memory
  if (threadIdx.x == 0 && row < n) {
    GPU_nnzRow[row] = shNnzRow[threadIdx.y];
    GPU_idxNzRowRestart[row] = idxNzRowRestart;
    GPU_colRestart[row] = colRestart;
  }
}
```

Listing D.4: `fullPass` kernel (part 3) for Algo CSR-2

```
__global__ void ellpackToCSR (...)
{
  // 2D block, 1D grid in x-axis but regarded as in y-axis
  int col = threadIdx.x;
  int row = blockDim.y * blockIdx.x + threadIdx.y;

  // Read nnz offset and restart index per row from global memory
  int nnzOffset, idxNzRowRestart;
  if (row < n) {
    nnzOffset = GPU_csrRow[row];
    idxNzRowRestart = GPU_idxNzRowRestart[row];
  }

  // Copy nonzeros from Ellpack arrays to CSR arrays in coalesced way
  int idxOffset = row*hypoMaxNnzRow;
  while (row < n && col < idxNzRowRestart) {
    GPU_csrVal[nnzOffset + col] = GPU_csrValMax[idxOffset + col];
    GPU_csrCol[nnzOffset + col] = GPU_csrColMax[idxOffset + col];
    col += blockDim.x;
  }

  // Store the global restart index into global memory
  if (threadIdx.x == 0 && row < n) {
    GPU_idxNzTotalRestart[row] = nnzOffset + idxNzRowRestart;
  }
}
```

Listing D.5: `ellpackToCSR` kernel for Algo CSR-2

```
__global__ void supPass (...)
{
  // 2D block, 1D grid in x-axis but regarded as in y-axis
  int row = blockDim.y * blockIdx.x + threadIdx.y;
  int yofs = threadIdx.y*blockDim.x;
  int col, colRestart, maxCol, nnzOffset;
  if (row < n) {
    colRestart = GPU_colRestart[row];
    col        = colRestart + threadIdx.x;
    maxCol     = colRestart+((n-colRestart-1)/blockDim.x+1)*blockDim.x;
    nnzOffset  = GPU_idxNzTotalRestart[row];
  }

  ... // Similar to the 2ndPass kernel of Algo CSR-1
}
```

Listing D.6: `supPass` kernel for Algo CSR-2

E - GPU implementation for Algo CSR-3

Listing E.1 shows the host code of our GPU implementation for Algo CSR-3 (Algorithm 6 in Section 3.3.5). The number of chunks is determined based on the size of free GPU memory to use (lines 6-12). For each chunk of the similarity matrix, we launch a typical kernel called `chkPass` to construct the matrix chunk in dense format (lines 28-30) and we leverage the `cusparseDenseToSparse_xxx` functions of NVIDIA's GPU-accelerated cuSPARSE library [144] to convert it into CSR format (lines 33-41). Note that the chunks should be constructed and converted one by one in order, so that we can accumulate the number of nonzeros (lines 38-39) and continuously update `csrRow[]` using the `transform` API of Thrust library (lines 45-50). After the loop, we exploit the `cusparseXcsrsort` and `cusparseSgthr` functions of the cuSPARSE library to merge the CSR results obtained from each chunk so that we obtain the CSR format of the whole similarity matrix (lines 56-59). Finally we derive other necessary results from `csrRow[]` (lines 63).

```
#include <cusparse.h>  // Include cuSPARSE library
#include <thrust/...>  // Include Thrust library functions
... // Declaration & memory allocation

// Auto-tuning of nbChunks
cudaMemGetInfo(&freeGPUMem, &totalGPUMem);
useGPUMem = (size_t)((double)freeGPUMem*(double)memUsePercent/100.0);
maxNbRows = useGPUMem / (sizeof(float)*((size_t)n));
if (maxNbRows > ((size_t)n))
  nbChunks = 1;
else
  nbChunks = n/((int)maxNbRows) + (n%((int)maxNbRows) > 0 ? 1 : 0);

// Initialization
q = n/nbChunks; // quotient
r = n%nbChunks; // remainder
nnz = 0;   nnzOffset = 0;
Db.x = BSX;   Db.y = BSY;
Dg.x = n/Db.x + (n%Db.x > 0 ? 1 : 0);

// Chunkwise similarity matrix construction on the GPU
for (int b = 0; b < nbChunks; b++) {
  chunkOffset = (c < r ? ((q + 1) * c) : (q * c + r));
  chunkSize = (c < r ? (q + 1) : q);
  Dg.y = chunkSize/Db.y + (chunkSize%Db.y > 0 ? 1 : 0);

  // Compute the similarity matrix in a chunkwise fashion
  chkPass<<<Dg,Db>>>(chunkOffset, chunkSize, // input
                     ...,                    // input
                     GPU_simChunk);          // output

  // Transform the chunk of similarity matrix from dense to CSR format
  cusparseCreateDnMat(...); // initialize dense mat. descriptor
  cusparseCreateCsr(...);   // initialize sparse mat. descriptor in CSR
  cusparseDenseToSparse_bufferSize(...); // return workspace size
  cusparseDenseToSparse_analysis(...);   // update sparse mat. descr.
  cusparseSpMatGetSize(...); // get the number of nonzeros
  if (c > 0)  nnzOffset += ((int)nnzChunk[c - 1]); // update nnzOffset
  nnz = nnzOffset + ((int)nnzChunk[c]); // update nnz
  cusparseCsrSetPointers(...);          // reset CSR pointers
  cusparseDenseToSparse_convert(...);   // dense-to-sparse conversion

  // Add nnzOffset to all values from "d_csrRow + chunkOffset"
  // to "d_csrRow + chunkOffset + chunkSize"
  if (c < nbChunks - 1)
    thrust::transform(..., d_csrRow + chunkOffset,
                      d_csrRow + chunkOffset + chunkSize, ...);
  else
    thrust::transform(..., d_csrRow + chunkOffset,
                      d_csrRow + chunkOffset + chunkSize + 1, ...);

  ... // Destroy matrix descriptors and deallocate memory
}

// Merge CSR format
cusparseXcsrsort_bufferSizeExt(...);    // allocate buffer
cusparseCreateIdentityPermutation(...); // set permutation vector
cusparseXcsrsort(...); // sort the column indices of CSR format
cusparseSgthr(...);    // gather sorted csrVal

... // Destroy matrix descriptors and deallocate memory

... // Derive nnzPerRow, minNnzRow, maxNnzRow from csrRow
```

Listing E.1: Host code of GPU implementation for Algo CSR-3

F - GPU implementation for noise filtering algorithm

We mainly present our GPU implementation for the noise filtering part of Algorithm 7, which consists in our CUDA kernels and some APIs of the Thrust library. The host code is shown in Listing F.1, and the CUDA kernels are shown in Listings F.2, F.3 and F.4.

The beginning part for determining the threshold for noise filtering ($tholdNF$) is uncomplicated and therefore omitted (Listing F.1, lines 4-15). Then we identify noise instances by launching the `findNoise` kernel using a 1D grid with 1D blocks of threads (Listing F.1, lines 18-22). Each thread accesses one element of the scaled `nnzPerRow` or the scaled degrees, and checks whether the element value is under or equal to $tholdNF$ (Listing F.2, lines 13-23). If the checked condition is satisfied, then the thread marks the corresponding instance as noise with `GPU_isNoise[*]=1`, records the index of the noise instance, and set the cluster label to "-1". In addition, the kernel counts the total number of identified noise instances (Listing F.2, lines 25-32). So the number of non-noise instances becomes clear (Listing F.1, lines 23-24).

To remove noise-related elements from the CSR format similarity matrix, we choose to first mark them as "-1" in `csrCol[]` (Listing F.1, lines 27-37) and then separate them from noise-unrelated elements (Listing F.1, lines 40-49). Specifically, we use the `exclusive_scan` API to derive the number of noise instances in front of each instance (`d_nbNoiseFront[]`) from the array that marks whether an instance is noise (`d_isNoise[]`). We use the `stable_partition` API to separate, in `d_idxNoise[]`, the indexes of noise instances from those of non-noise instances while preserving their relative order. With these prepared, we launch the `markNoiseInCSRCol` kernel using a 1D grid with 1D blocks of threads. Each block processes one row of the similarity matrix in an iterative and progressive way. Considering the maximum y-dimension of a grid (65535) may be insufficient for large number of instances while the maximum x-dimension of a grid ($2^{31} - 1 = 2\,147\,483\,647$) is sufficiently large, we choose to create the 1D grid in x dimension (Listings F.1, line 31) but regard it as in y dimension (Listings F.3, line 4). For the rows related to noise instances (Listings F.3, lines 12-20), the blocks of threads simply set the associated segment of `csrCol[]` to "-1", then the first thread of each block accumulates the number of noise-related nonzeros in that row into global memory and set the corresponding element in `nnzPerRow[]` to 0. For the rows related to non-noise instances (Listings F.3, lines 22-43), the blocks of threads update the elements of `csrCol[]` for noise-unrelated nonzeros, set the elements of `csrCol[]` to "-1" for noise-related nonzeros, then the first threads of each block accumulates the number of noise-related nonzeros in that row into

global memory and update the corresponding element in `nnzPerRow[]`.

```
#include <thrust/...>  // Include Thrust library functions
... // Declarations, memory allocations, initialization
... // CSR format similarity matrix construction
switch (filterApproach) {
  case 0 :
    ... // Get nnz per row, minimal and maximal # of nonzeros in a row
    break;
  case 1 :
    ... // Compute degrees of vertices, get minimal and maximal degree
    break;
}
... // Min-max scaling on nnzPerRow or degrees, compute the histogram
... // Estimate the optimal threshold for filtering noise (ONGOING work)
... // Print the histogram and the estimated optimal threshold
scanf("%f", &tholdNF);  // Let the user finalize the value of tholdNF

// Identify noise based on tholdNoise
Db.x = BSX;  Db.y = 1;  Dg.x = n/Db.x + (n%Db.x > 0 ? 1:0);  Dg.y = 1;
shMemSize = sizeof(int)*Db.x;
findNoise<<<Dg,Db,shMemSize>>>(GPU_scaledMetric, tholdNF, // input
                               GPU_nbNoise, GPU_isNoise,  // output
                               GPU_idxNoise, GPU_labels); // output
cudaMemcpy(&nbNoise, GPU_nbNoise, ..., cudaMemcpyDeviceToHost);
nNF = n - nbNoise;  // # of non-noise instances

// Mark noise as -1 in the CSR format of similarity matrix
thrust::device_ptr<int> d_...(GPU_...);
thrust::exclusive_scan(..., d_isNoise, d_isNoise + n, d_nbNoiseFront);
thrust::stable_partition(..., d_idxNoise, d_idxNoise + n,
                         is_not_minus_one());
Db.x = BSX;  Db.y = 1;  Dg.x = n;  Dg.y = 1;
markNoiseInCSRCol<<<Dg,Db>>>(GPU_csrRow, GPU_labels, nbNoise,// input
                             GPU_nbNoiseFront, GPU_idxNoise, // input
                             GPU_nnzNoise, GPU_nnzPerRow,    // output
                             GPU_csrCol);                    // output
cudaMemcpy(&nnzNoise, GPU_nnzNoise, ..., cudaMemcpyDeviceToHost);
nnzNF = nnz - nnzNoise;  // # of noise-free nonzeros

// Get in place the noise-free similarity matrix in CSR format
thrust::stable_partition(..., d_nnzPerRow, d_nnzPerRow + n + 1,
                         is_nonzero());
thrust::stable_partition(..., d_csrVal, d_csrVal + nnz, d_csrCol,
                         is_not_minus_one());
thrust::stable_partition(..., d_csrCol, d_csrCol + nnz,
                         is_not_minus_one());
thrust::exclusive_scan(..., d_nnzPerRow, d_nnzPerRow + nNF + 1,
                       d_csrRowNF);

int kcNF = kc - 1;  // # of noise-free clusters
... // Spectral graph partitioning using nvGRAPH, find kcNF clusters

// Get the labels of non-noise instances indexed in the original set
thrust::copy_if(..., d_nbNoiseFront, d_nbNoiseFront + n,
                d_isNoise, d_nbNoiseFrontNF, is_zero());
Db.x = BSX;                                  Db.y = 1;
Dg.x = nNF/Db.x + (nNF%Db.x > 0 ? 1 : 0);    Dg.y = 1;
mapLabels<<<Dg, Db>>>(nNF, GPU_labelsNF, GPU_nbNoiseFrontNF, // input
                      GPU_labels);                           // output
```

Listing F.1: Host code of GPU implementation for noise filtering algorithm

After completing the `markNoiseInCSRCol` kernel, we call the `stable_partition` API to get the noise-free version of `nnzPerRow[]`, `csrVal` and `csrCol` (Listings F.1, lines 40-45). Then we derive the noise-free `csrRow[]` from the noise-free `nnzPerRow[]` via the `exclusive_scan` API (Listings F.1, lines 46-47). Now we have the noise-free similarity matrix in CSR format, on which we perform spectral graph partitioning using the nvGRAPH library (see Section 3.4) to find $k_c - 1$ clusters (excluding a single cluster for noise instances).

Finally, we copy the elements of `d_nbNoiseFront` that are associated with non-noise instances into `d_nbNoiseFrontNF` (Listings F.1, lines 53-54), and launch the `mapLabels` kernel to map the labels of non-noise instances onto the original indexing structure (Listings F.1, lines 55-58). We use a 1D grid with 1D blocks of threads for the kernel. Each thread reads an element of `GPU_nbNoiseFrontNF` and copies the cluster label from the noise-free indexing array `GPU_labelsNF` into the original indexing array `GPU_labels` (Listing F.4, lines 6-9).

```
1  // Starting address for dynamic allocation of shared memory
2  extern __shared__ int shBuff[];
3  __global__ void findNoise(...)
4  {
5    // 1D block in x-axis, 1D grid in x-axis
6    int tid = blockIdx.x * blockDim.x + threadIdx.x;
7
8    // Pointers to dynamic shared memory arrays
9    int *shFlagNoise = shBuff;  // size: blockDim.x
10   shFlagNoise[threadIdx.x] = 0;
11
12   // Identify noise instances
13   if (tid < n) {
14     float scaledMetric = GPU_scaledMetric[tid];
15     GPU_isNoise[tid] = 0;
16     GPU_idxNoise[tid] = -1;
17     if (scaledMetric <= tholdNF) {
18       shFlagNoise[threadIdx.x] = 1;
19       GPU_isNoise[tid]         = 1;
20       GPU_idxNoise[tid]        = tid;
21       GPU_labels[tid]          = -1;
22     }
23   }
24
25   // Count the number of noise into GPU_nbNoise: two-part reduction
26   // 1 - Classic reduction of the shared array shFlagNoise[*]
27   ... // into shFlagNoise[0], kill useless warps step by step,
28   ... // only the 1st warp survives at the end.
29   // 2 - Final reduction into the global variable GPU_nbNoise
30   if (threadIdx.x == 0)
31     if (shFlagNoise[0] > 0)
32       atomicAdd(GPU_nbNoise, shFlagNoise[0]);
33 }
```

Listing F.2: `findNoise` kernel for noise filtering algorithm

```
__global__ void markNoiseInCSRCol (...)
{
  // 1D block in x-axis, 1D grid in x-axis but regarded as in y-axis
  int row = blockIdx.x;  int col = threadIdx.x;
  int nnzOffset = GPU_csrRow[row];
  int nnzRow    = GPU_csrRow[row + 1] - nnzOffset;
  int label     = GPU_labels[row];

  // Pointers to dynamic shared memory arrays
  int *shNnzNoise = shBuff;

  if (label == -1) {   // For noise instances
    while (col < nnzRow) {
      GPU_csrCol[nnzOffset + col] = -1;
      col += blockDim.x;
    }
    if (threadIdx.x == 0) {
      atomicAdd(GPU_nnzNoise, nnzRow);
      GPU_nnzPerRow[row] = 0;
    }

  } else {            // For non-noise instances
    if (threadIdx.x == 0)  *shNnzNoise = 0;
    __syncthreads();
    while (col < nnzRow) {
      int oldColIdx    = GPU_csrCol[nnzOffset + col];
      int nbNoiseFront = GPU_nbNoiseFront[oldColIdx];
      int newColIdx    = oldColIdx - nbNoiseFront;
      for (int i = 0; i < nbNoise; i++) {
        if (oldColIdx == GPU_idxNoise[i]) {
          newColIdx = -1;
          atomicAdd_block(shNnzNoise, 1);
        }
      }
      GPU_csrCol[nnzOffset + col] = newColIdx;
      col += blockDim.x;
    }
    __syncthreads();
    if (threadIdx.x == 0) {
      atomicAdd(GPU_nnzNoise, *shNnzNoise);
      GPU_nnzPerRow[row] -= *shNnzNoise;
    }
  }
}
```

Listing F.3: `markNoiseInCSRCol` kernel for noise filtering algorithm

```
__global__ void mapLabels (...)
{
  // 1D block in x-axis, 1D grid in x-axis
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (tid < nNF) {
    int nbNoiseFrontNF = GPU_nbNoiseFrontNF[tid];
    GPU_labels[tid + nbNoiseFrontNF] = GPU_labelsNF[tid];
  }
}
```

Listing F.4: `mapLabels` kernel for noise filtering algorithm

G - Parallel implementation for the seeding step of k-means++

Listings G.1 and G.2 display our GPU implementation for the seeding step of the k-means++ algorithm (see Algorithm 2), while Listing G.3 shows our CPU implementation. All of them use only single precision arithmetic.

Particularly, the thrust functions used in our implementations suffer from the effect of rounding errors when processing large-scale datasets using single precision for floating-point numbers. To handle this issue, we provide a mixed precision version for each CPU and GPU implementation (as presented in Listings G.4, G.5 and G.6), where double precision is used in some lines of code.

Other steps of k-means++ are the same as k-means (see Chapter 2).

```
#include <thrust/...> // Include Thrust library functions
void gpu_seeding (float *data, unsigned int seedbase, ...,  // input
                  float *cent)                              // output
{ unsigned int seed = seedBase;  int centIdx, *GPU_centIdx;
  float randValue, *GPU_d2, *GPU_d2Sum;
  float *GPU_prob, *GPU_inScanSum, *GPU_exScanSum;
  ... // Memory allocation
  Db.x = BSXN; Db.y = 1; Dg.x = n/Db.x + (n%Db.x > 0 ? 1:0); Dg.y = 1;
  shMemSize = sizeof(float)*Db.x + sizeof(float)*d;

  // Select initial centroids one by one
  for (int k = 0; k < kc; k++) {
    // Get the idx of an initial centroid
    if (k == 0) { centIdx = rand_r(&seed)/(float)RAND_MAX * n; }
    else {
      randValue = rand_r(&seed)/(float)RAND_MAX;
      findCentIdx<<<Dg,Db>>>(randValue, ...,                // input
                             GPU_inScanSum, GPU_exScanSum,  // input
                             GPU_centIdx);                  // output
      cudaMemcpy(&centIdx, GPU_centIdx, sizeof(int), ...DeviceToHost);
    }
    // Calculate GPU_d2 and GPU_d2Sum
    cudaMemset(GPU_d2Sum, 0, sizeof(float));
    calculateD2Sum<<<Dg,Db,shMemSize>>>(k, centIdx, GPU_cent, // input
                                        GPU_dataT, ...,       // input
                                        GPU_d2, GPU_d2Sum);   // output
    // Calculate sampling probabilities
    calculateProbability<<<Dg,Db>>>(GPU_d2, GPU_d2Sum, ..., // input
                                    GPU_prob);               // output
    // Calculate the results of inclusive scan & exclusive scan
    thrust::device_ptr<float> d_prob(GPU_prob), d_inSS(GPU_inScanSum),
                              d_exSS(GPU_exScanSum);
    thrust::inclusive_scan(thrust::device,d_prob,d_prob+n,d_inSS);
    thrust::exclusive_scan(thrust::device,d_prob,d_prob+n,d_exSS,0.0f);
  }
  ... // Memory deallocation
}
```

Listing G.1: Host code of GPU implementation for the seeding step of k-means++ (single precision version)

```
__global__ void findCentIdx (...)
{
  // 1D block in x-axis, 1D grid in x-axis
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Find the idx of a new initial centroid
  if (tid < n) {
    float exSS = GPU_exScanSum[tid], inSS = GPU_inScanSum[tid];
    if (randValue >= exSS && randValue < inSS)  *GPU_centerIdx = tid;
  }
}

// Starting address for dynamic allocation of shared memory
extern __shared__ float shBuff[];

__global__ void calculateD2Sum (...)
{
  // 1D block in x-axis, 1D grid in x-axis
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int dimIdx = threadIdx.x;

  // Declare shared memory arrays and initialize them
  float *shD2 = shBuff;                  // blockDim.x floats
  float *shCentDim = &shD2[blockDim.x];  // d          floats
  if (k > 0) shD2[threadIdx.x] = (tid < n ? GPU_d2[tid] : 0.0f);
  else       shD2[threadIdx.x] = (tid < n ? FLT_MAX : 0.0f);
  while (dimIdx < d) {
    shCentDim[dimIdx] = GPU_dataT[dimIdx*n + idx];
    GPU_cent[k*d + dimIdx] = shCentDim[dimIdx];
    dimIdx += blockDim.x;
  }
  __syncthreads();

  // Calculate d2
  if (tid < n) {
    float diff, sqDist = 0.0f;
    for (int j = 0; j < d; j++) {
      diff = GPU_dataT[j*n + tid] - shCentDim[j]; sqDist += diff*diff;
    }
    if (sqDist < shD2[threadIdx.x]) {
      shD2[threadIdx.x] = sqDist; GPU_d2[tid] = sqDist;
    }
  }

  // Calculate d2Sum
  ... // Recursive reduction of shD2[*] into shD2[0]
  if (threadIdx.x == 0)  atomicAdd(GPU_d2Sum, shD2[0]);
}

__global__ void calculateProbability (...)
{
  // 1D block in x-axis, 1D grid in x-axis
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Declare a shared memory variable and intialize it
  __shared__ float shD2;
  if (threadIdx.x == 0)  shD2 = *GPU_d2Sum;
  __syncthreads();

  // Calculate probability
  if (tid < n)  GPU_prob[tid] = GPU_d2[tid] / shD2;
}
```

Listing G.2: Devide code of GPU implementation for the seeding step of k-means++ (single precision version)

```
#include <thrust/...> // Include Thrust library functions
void cpu_seeding (float *data, unsigned int seedbase, ...,  // input
                  float *cent)                              // output
{
  // Declaration
  unsigned int seed = seedbase;  int centIdx;
  float randValue, *d2, d2Sum, *prob, *inScanSum, *exScanSum;
  omp_lock_t lock;  omp_init_lock(&lock);  // Initialize the lock
  ... // Memory allocation for d2, prob, inScanSum, exScanSum

  #pragma omp parallel
  {
    for (int k = 0; k < kc; k++) {// Select initial centroids one by one
      // Get the idx of an initial centroid
      #pragma omp single
      {
        if (k == 0)  centIdx = rand_r(&seed)/(float)RAND_MAX * n;
        randValue = rand_r(&seed)/(float)RAND_MAX;  d2Sum = 0.0f;
      }
      if (k > 0) {
        #pragma omp for
        for (int i = 0; i < n; i++) {
          float exSS = exScanSum[i], inSS = inScanSum[i];
          if (randValue >= exSS && randValue < inSS) {
            omp_set_lock(&lock);   // necessary due to -Ofast flag
            centIdx = i;
            omp_unset_lock(&lock); // necessary due to -Ofast flag
          }
        }
      }
      // Store the initial centroid
      #pragma omp for
      for (int j = 0; j < d; j++)  cent[k*d + j] = data[centIdx*d + j];
      // Calculate d2[]
      #pragma omp for
      for (int i = 0; i < n; i++) {
        float diff, sqDist = 0.0f;
        float minDistSq = (k > 0 ? d2[i] : FLT_MAX);
        for (int j = 0; j < d; j++) {
          diff = data[i*d + j] - data[centIdx*d + j];
          sqDist += diff*diff;
        }
        if (sqDist < minDistSq)  d2[i] = sqDist;
      }
      // Calculate d2Sum
      #pragma omp single
      { d2Sum = thrust::reduce(thrust::host, d2, d2 + n, 0.0f); }
      // Calculate sampling probabilities
      #pragma omp for
      for (int i = 0; i < n; i++)  prob[i] = d2[i] / d2Sum;
      // Calculate the results of inclusive scan & exclusive scan
      #pragma omp single
      {
        thrust::inclusive_scan(thrust::host,prob,prob+n,inScanSum);
        thrust::exclusive_scan(thrust::host,prob,prob+n,exScanSum,0.0f);
      }
    } // end for
  } // end pragma omp parallel

  omp_destroy_lock(&lock);  // Destroy lock
  ... // Memory deallocation
}
```

Listing G.3: CPU implementation for the seeding step of k-means++ (single precision version)

```
#include <thrust/...> // Include Thrust library functions
void gpu_seeding (float *data, unsigned int seedbase, ...,  // input
                  float *cent)                              // output
{
  ... // Same as the single precision version except the following code
  // To avoid the effect of rounding errors when processing
  // large-scale datasets, the following code uses
  // double instead of float, and 0.0 instead of 0.0f
  double randValue, *GPU_d2, *GPU_d2Sum;
  double *GPU_prob, *GPU_inScanSum, *GPU_exScanSum;
  if (k == 0)  centIdx = rand_r(&seed)/(double)RAND_MAX * n;
  randValue = rand_r(&seed)/(double)RAND_MAX;
  cudaMemset(GPU_d2Sum, 0, sizeof(double));
  thrust::device_ptr<double> d_prob(GPU_prob);
  thrust::device_ptr<double> d_inSS(GPU_inScanSum);
  thrust::device_ptr<double> d_exSS(GPU_exScanSum);
  thrust::exclusive_scan(thrust::device,d_prob,d_prob+n,d_exSS,0.0);
}
```

Listing G.4: Host code of GPU implementation for the seeding step of k-means++ (mixed precision version)

```
  ... // Same as the single precision version except the following code
  // To avoid the effect of rounding errors when processing
  // large-scale datasets, the following code uses
  // double instead of float
  double exSS = GPU_exScanSum[tid]; // in findCentIdx kernel
  double inSS = GPU_inScanSum[tid]; // in findCentIdx kernel
  __shared__ double shD2;  // in calculateProbability kernel
```

Listing G.5: Devide code of GPU implementation for the seeding step of k-means++ (mixed precision version)

```
#include <thrust/...> // Include Thrust library functions
void cpu_seeding (float *data, unsigned int seedbase, ...,  // input
                  float *cent)                              // output
{
  ... // Same as the single precision version except the following code
  // To avoid the effect of rounding errors when processing
  // large-scale datasets, the following code uses
  // double instead of float, and 0.0 instead of 0.0f
  double randValue, *d2, d2Sum, *prob, *inScanSum, *exScanSum;
  if (k == 0)  centIdx = rand_r(&seed)/(double)RAND_MAX * n;
  randValue = rand_r(&seed)/(double)RAND_MAX;  d2Sum = 0.0;
  double exSS = exScanSum[i], inSS = inScanSum[i];
  d2Sum = thrust::reduce(thrust::host, d2, d2 + n, 0.0);
  thrust::exclusive_scan(thrust::host,prob,prob+n,exScanSum,0.0);
}
```

Listing G.6: CPU implementation for the seeding step of k-means++ (mixed precision version)

Bibliography

[1] Z. Abbasi-Moud, H. Vahdat-Nejad, and J. Sadri. Tourism recommendation system based on semantic clustering and sentiment analysis. *Expert Systems with Applications*, 167:114324, 2021.

[2] J. Agarwal, R. Nagpal, and R. Sehgal. Crime analysis using k-means clustering. *International Journal of Computer Applications*, 83(4), 2013.

[3] A. Aggarwal, A. Deshpande, and R. Kannan. Adaptive sampling for k-means clustering. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 15–28. Springer, 2009.

[4] C. C. Aggarwal and C. K. Reddy. Data clustering. *Algorithms and applications. Chapman&Hall/CRC Data mining and Knowledge Discovery series, Londra*, 2014.

[5] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 94–105, 1998.

[6] E. Aljalbout, V. Golkov, Y. Siddiqui, M. Strobel, and D. Cremers. Clustering with deep learning: Taxonomy and new methods. *arXiv preprint arXiv:1801.07648*, 2018.

[7] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[8] L. F. Ana and A. K. Jain. Robust data clustering. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–II. IEEE, 2003.

[9] D. C. Anastasiu and G. Karypis. L2ap: Fast cosine similarity search with prefix l-2 norm bounds. In *2014 IEEE 30th International Conference on Data Engineering*, pages 784–795. IEEE, 2014.

[10] D. C. Anastasiu and G. Karypis. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 791–800, 2015.

[11] D. C. Anastasiu and G. Karypis. Parallel cosine nearest neighbor graph construction. *Journal of Parallel and Distributed Computing*, 129:61–82, 2019.

[12] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[13] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[14] M. Baboulin, A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.*, 180(12):2526–2533, 2009.

[15] S. Balakrishnan, M. Xu, A. Krishnamurthy, and A. Singh. Noise thresholds for spectral clustering. *Advances in Neural Information Processing Systems*, 24, 2011.

[16] G. H. Ball. Data analysis in the social sciences: What about the details? In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 533–559, 1965.

[17] T. Barton, T. Bruna, and P. Kordik. Chameleon 2: an improved graph-based clustering algorithm. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(1):1–27, 2019.

[18] M.-A. Belabbas and P. J. Wolfe. Spectral methods in machine learning and new strategies for very large datasets. *Proceedings of the National Academy of Sciences*, 106(2):369–374, 2009.

[19] A. Ben-Hur, D. Horn, H. T. Siegelmann, and V. Vapnik. Support vector clustering. *Journal of machine learning research*, 2(Dec):125–137, 2001.

[20] P. Bhattacharjee and P. Mitra. A survey of density based clustering algorithms. *Frontiers of Computer Science*, 15(1):1–27, 2021.

[21] A. H. Bhatti, A. Rahman, and A. A. Butt. Video segmentation using spectral clustering on superpixels. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 869–873. IEEE, 2016.

[22] J. Bhimani, M. Leeser, and N. Mi. Accelerating k-means clustering with parallel implementations and GPU computing. In *2015 IEEE High Performance Extreme Computing Conference, HPEC 2015, Waltham, MA, USA*, 2015.

[23] C. Böhm, M. Perdacher, and C. Plant. Multi-core k-means. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 273–281, 2017.

[24] A. Bojchevski, Y. Matkovic, and S. Günnemann. Robust spectral clustering for noisy data: Modeling sparse corruptions improves latent embeddings. In

Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 737–746, 2017.

[25] M. Bolla. Relations between spectral and classification properties of multigraphs. Technical report, No. DIMACS-91-27, Center for Discrete Mathematics and Theoretical Computer Science, 1991.

[26] C. Bouveyron and C. Brunet-Saumard. Model-based clustering of high-dimensional data: A review. *Computational Statistics & Data Analysis*, 71:52–78, 2014.

[27] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[28] T. Caliński and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods*, 3(1):1–27, 1974.

[29] R. J. Campello, P. Kröger, J. Sander, and A. Zimek. Density-based clustering. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, (2), 2019.

[30] R. J. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013.

[31] M. E. Celebi. *Partitional clustering algorithms*. Springer, 2014.

[32] M. Charrad, N. Ghazzali, V. Boiteau, and A. Niknafs. Nbclust: an R package for determining the relevant number of clusters in a data set. *Journal of statistical software*, 61:1–36, 2014.

[33] W. Chen, Y. Song, H. Bai, C. Lin, and E. Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(3), 2011.

[34] X. Chen and D. Cai. Large scale spectral clustering with landmark-based representation. In *Twenty-fifth AAAI conference on artificial intelligence*, 2011.

[35] X. Chen, F. Nie, J. Z. Huang, and M. Yang. Scalable normalized cut with improved spectral rotation. In *IJCAI*, pages 1518–1524, 2017.

[36] W. Cheng, W. Wang, and S. Batista. Grid-based clustering. In *Data clustering*, pages 128–148. Chapman and Hall/CRC, 2018.

[37] A. Choromanska, T. Jebara, H. Kim, M. Mohan, and C. Monteleoni. Fast spectral clustering via the Nyström method. In *International Conference on Algorithmic Learning Theory*, pages 367–381. Springer, 2013.

[38] J. Costeira and T. Kanade. A multi-body factorization method for motion analysis. In *Proceedings of IEEE International Conference on Computer Vision*, pages 1071–1076. IEEE, 1995.

[39] T. M. Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[40] S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, and G. Toraldo. A GPU-accelerated parallel K-means algorithm. *Computers & Electrical Engineering*, 75:262–274, 2019.

[41] Z. Dafir, Y. Lamari, and S. C. Slaoui. A survey on parallel clustering algorithms for big data. *Artificial Intelligence Review*, 54(4):2411–2443, 2021.

[42] D. L. Davies and D. W. Bouldin. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2):224–227, 1979.

[43] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[44] R. Diestel. *Graph Theory*. Springer Berlin, Heidelberg, 2010. Fourth edition.

[45] E. Dimitriadou and K. Hornik. cclust: Convex clustering methods and clustering indexes (version 0.6-23), November 2021. URL: https://cran.r-project.org/web/packages/cclust/index.html.

[46] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[47] W. Donath and A. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[48] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[49] K. Dowd and C. Severance. High performance computing. 2010.

[50] D. Dua and C. Graff. UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml.

[51] L. Duan, C. Aggarwal, S. Ma, and S. Sathe. Improving spectral clustering with deep embedding and cluster estimation. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 170–179. IEEE, 2019.

[52] P. D'Urso, L. De Giovanni, M. Disegna, and R. Massari. Bagged clustering and its application to tourism market segmentation. *Expert Systems with Applications*, 40(12):4944–4956, 2013.

[53] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[54] A. E. Ezugwu, A. M. Ikotun, O. O. Oyelade, L. Abualigah, J. O. Agushaka, C. I. Eke, and A. A. Akinyelu. A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects. *Engineering Applications of Artificial Intelligence*, 110:104743, 2022.

[55] A. Fender. *Parallel solutions for large-scale eigenvalue problems arising in graph analytics*. PhD thesis, Université Paris-Saclay, Dec. 2017.

[56] A. Fender, N. Emad, et al. Accelerated hybrid approach for spectral problems arising in graph analytics. *Procedia Computer Science*, 80:2338–2347, 2016.

[57] M. B. Ferraro and P. Giordani. Soft clustering. *Wiley Interdisciplinary Reviews: Computational Statistics*, 12(1):e1480, 2020.

[58] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.

[59] C. C. Fowlkes, S. J. Belongie, F. R. K. Chung, and J. Malik. Spectral grouping using the Nyström method. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(2), 2004.

[60] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American statistical association*, 78(383):553–569, 1983.

[61] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[62] G. Gan, C. Ma, and J. Wu. *Data clustering: theory, algorithms, and applications*. SIAM, 2020.

[63] K. Ghasedi Dizaji, A. Herandi, C. Deng, W. Cai, and H. Huang. Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In *Proceedings of the IEEE international conference on computer vision*, pages 5736–5745, 2017.

[64] A. Gonzalez-Pardo, J. J. Jung, and D. Camacho. Aco-based clustering for ego network analysis. *Future Generation Computer Systems*, 66:160–170, 2017.

[65] M. Gowanlock. Hybrid knn-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features. *Journal of Parallel and Distributed Computing*, 149:119–137, 2021.

[66] S. Guattery and G. L. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, 1998.

[67] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. *ACM Sigmod record*, 27(2):73–84, 1998.

[68] S. Guha, R. Rastogi, and K. Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information systems*, 25(5):345–366, 2000.

[69] L. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

[70] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[71] G. Hamerly and C. Elkan. Learning the k in k-means. *Advances in neural information processing systems*, 16, 2003.

[72] E. Han, P. Carbonetto, R. E. Curtis, Y. Wang, J. M. Granka, J. Byrnes, K. Noto, A. R. Kermany, N. M. Myres, M. J. Barber, et al. Clustering of 770,000 genomes reveals post-colonial population structure of north america. *Nature communications*, 8(1):1–12, 2017.

[73] J. Han, J. Pei, and M. Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[74] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi:10.1038/s41586-020-2649-2.

[75] G. He, S. Vialle, and M. Baboulin. Parallel and accurate k-means algorithm on CPU-GPU architectures for spectral clustering. *Concurrency and Computation: Practice and Experience*, page e6621, 2021. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6621.

[76] G. He, S. Vialle, and M. Baboulin. Parallelization of the k-means algorithm in a spectral clustering chain on CPU-GPU platforms. In *Euro-Par 2020: Parallel Processing Workshops*, volume 12480, LNCS, pages 135–147, Warsaw, Poland, 2021. Springer.

[77] G. He, S. Vialle, and M. Baboulin. Scalable spectral clustering on GPU. *International Journal of Parallel Programming*, page (Submitted), 2022.

[78] G. He, S. Vialle, S. Nicolas, and M. Baboulin. Scalable algorithms using sparse storage for parallel spectral clustering on GPU. In *18th Annual IFIP International Conference on Network and Parallel Computing (IFIP NPC)*, volume 13152, LNCS, pages 40–52, Paris, France, 2021. Springer.

[79] L. He, N. Ray, Y. Guan, and H. Zhang. Fast large-scale spectral clustering via explicit feature mapping. *IEEE transactions on cybernetics*, 49(3):1058–1071, 2018.

[80] K. A. Heller and Z. Ghahramani. Bayesian hierarchical clustering. In *Proceedings of the 22nd international conference on Machine learning*, pages 297–304, 2005.

[81] C. Hennig, M. Meila, F. Murtagh, and R. Rocci. *Handbook of cluster analysis*. CRC Press, 2015.

[82] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2002. Second edition.

[83] A. Hinneburg and H.-H. Gabriel. DENCLUE 2.0: Fast clustering based on kernel density estimation. In *International symposium on intelligent data analysis*, pages 70–80. Springer, 2007.

[84] A. Hinneburg, D. A. Keim, et al. An efficient approach to clustering in large multimedia databases with noise. In *KDD*, volume 98, pages 58–65, 1998.

[85] Z. Huang. Clustering large data sets with mixed numeric and categorical values. In *Proceedings of the 1st pacific-asia conference on knowledge discovery and data mining,(PAKDD)*, pages 21–34. Citeseer, 1997.

[86] Z. Huang. A fast clustering algorithm to cluster very large categorical data sets in data mining. *Dmkd*, 3(8):34–39, 1997.

[87] L. Hubert and P. Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[88] Z. Huo, G. Mei, G. Casolla, and F. Giampaolo. Designing an efficient parallel spectral clustering algorithm on multi-core processors in Julia. *Journal of Parallel and Distributed Computing*, 138:211–221, 2020.

[89] T. Ina, A. Hashimoto, M. Iiyama, H. Kasahara, M. Mori, and M. Minoh. Outlier cluster formation in spectral clustering. *arXiv preprint arXiv:1703.01028*, 2017.

[90] Intel. *Intel Intrinsics Guide v3.6.2*, Apr. 2022. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[91] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[92] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[93] R. Janani and S. Vijayarani. Text document clustering using spectral clustering algorithm with particle swarm optimization. *Expert Systems with Applications*, 134:192–200, 2019.

[94] F. Jézéquel, S. Graillat, D. Mukunoki, T. Imamura, and R. Iakymchuk. Can we avoid rounding-error estimation in HPC codes and still get trustful results? working paper or preprint, 2020. URL: https://hal.archives-ouvertes.fr/hal-02486753.

[95] Z. Jiang, Y. Zheng, H. Tan, B. Tang, and H. Zhou. Variational deep embedding: An unsupervised and generative approach to clustering. *arXiv preprint arXiv:1611.05148*, 2016.

[96] R. Jin, C. Kou, R. Liu, and Y. Li. Efficient parallel spectral clustering algorithm design for large data sets under cloud computing environment. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):1–10, 2013.

[97] Y. Jin and J. F. JáJá. A high performance implementation of spectral clustering on CPU-GPU platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Chicago, IL, USA*, pages 825–834, 2016.

[98] T. Kansal, S. Bahuguna, V. Singh, and T. Choudhury. Customer segmentation using k-means clustering. In *2018 international conference on computational techniques, electronics and mechanical systems (CTEMS)*, pages 135–139. IEEE, 2018.

[99] G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[100] L. Kaufman and P. J. Rousseeuw. Clustering large data sets. *Pattern Recognition in Practice*, pages 425–437, 1986.

[101] L. Kaufman and P. J. Rousseeuw. Clustering by means of medoids. In *Proc. Statistical Data Analysis Based on the L1 Norm Conference, Neuchatel, 1987*, pages 405–416, 1987.

[102] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 1990.

[103] W. Kim, A. Kanezaki, and M. Tanaka. Unsupervised learning of image segmentation based on differentiable feature clustering. *IEEE Transactions on Image Processing*, 29:8055–8068, 2020.

[104] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

[105] H.-P. Kriegel, P. Kröger, J. Sander, and A. Zimek. Density-based clustering. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(3):231–240, 2011.

[106] M. Kruliš and M. Kratochvíl. Detailed analysis and optimization of CUDA k-means algorithm. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[107] S. Kumar, M. Mohri, and A. Talwalkar. Sampling methods for the Nyström method. *The Journal of Machine Learning Research*, 13(1):981–1006, 2012.

[108] U. Kutbay et al. Partitional clustering. *Recent Applications in Data Clustering*, 2018.

[109] G. Laccetti, M. Lapegna, V. Mele, D. Romano, and L. Szustak. Performance enhancement of a dynamic K-means algorithm through a parallel adaptive strategy on multicore CPUs. *Journal of Parallel and Distributed Computing*, 2020.

[110] H. Lee, A. Battle, R. Raina, and A. Ng. Efficient sparse coding algorithms. *Advances in neural information processing systems*, 19, 2006.

[111] M. Li, X.-C. Lian, J. T. Kwok, and B.-L. Lu. Time and space efficient spectral clustering via column sampling. In *CVPR 2011*, pages 2297–2304. IEEE, 2011.

[112] S. Li and N. Amenta. Brute-force k-nearest neighbors search on the GPU. In *International Conference on Similarity Search and Applications*, pages 259–270. Springer, 2015.

[113] T. Li, Y. Zhang, D. Li, X. Liu, and Y. Peng. Fast compressive spectral clustering. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 949–954. IEEE, 2017.

[114] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data—experiments,

analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[115] Y. Li, J. Huang, and W. Liu. Scalable sequential spectral clustering. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[116] Z. Li, J. Jin, and L. Wang. High-performance k-means implementation based on a simplified map-reduce architecture. *arXiv preprint:1610.05601*, 2016.

[117] Z. Li, J. Liu, S. Chen, and X. Tang. Noise robust spectral clustering. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE, 2007.

[118] F. Lin and W. W. Cohen. Power iteration clustering. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 655–662, 2010.

[119] J. Liu, C. Wang, M. Danilevsky, and J. Han. Large-scale spectral clustering on graphs. In *Twenty-Third International Joint Conference on Artificial Intelligence*. Citeseer, 2013.

[120] W. Liu, J. He, and S.-F. Chang. Large graph construction for scalable semi-supervised learning. In *ICML*, 2010.

[121] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[122] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1(14), pages 281–297, 1967.

[123] P. D. McNicholas. Model-based clustering. *Journal of Classification*, 33(3):331–373, 2016.

[124] Message Passing Interface Forum. MPI: A message-passing interface standard version 4.0, June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[125] E. Min, X. Guo, Q. Liu, G. Zhang, J. Cui, and J. Long. A survey of clustering with deep learning: From the perspective of network architecture. *IEEE Access*, 6:39501–39514, 2018.

[126] B. Mirkin. Choosing the number of clusters. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3):252–260, 2011.

[127] S. Miyahara, Y. Komazaki, and S. Miyamoto. An algorithm combining spectral clustering and DBSCAN for core points. In *Knowledge and Systems Engineering*, pages 21–28. Springer, 2014.

[128] S. Miyamoto, H. Ichihashi, K. Honda, and H. Ichihashi. *Algorithms for fuzzy clustering*. Springer, 2008.

[129] S. Mouysset, J. Noailles, and D. Ruiz. Using a global parameter for gaussian affinity matrices in spectral clustering. In *International Conference on High Performance Computing for Computational Science*, pages 378–390. Springer, 2008.

[130] S. Mouysset, J. Noailles, D. Ruiz, and C. Tauber. Spectral clustering: interpretation and gaussian parameter. In *Data Analysis, Machine Learning and Knowledge Discovery*, pages 153–162. Springer, 2014.

[131] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[132] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview, ii. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(6):e1219, 2017.

[133] M. C. Nascimento and A. C. De Carvalho. Spectral methods for graph clustering–a survey. *European Journal of Operational Research*, 211(2):221–231, 2011.

[134] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, et al. Amgx: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015.

[135] M. Naumov and T. Moon. Parallel spectral graph partitioning. Technical report, NVIDIA Technical Report, NVR-2016-001, 2016.

[136] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pages 849–856, 2001.

[137] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of VLDB*, pages 144–155. Citeseer, 1994.

[138] F. Nielsen. Hierarchical clustering. In *Introduction to HPC with MPI for Data Science*, pages 195–211. Springer, 2016.

[139] NVIDIA. *NVGRAPH Library User's Guide (DU-08010-001_v10.2)*, November 2019. URL: https://docs.nvidia.com/pdf/nvGRAPH_Library.pdf.

[140] NVIDIA. *CUDA C++ Best Practices Guide (DG-05603-001_v11.6)*, 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[141] NVIDIA. *CUDA C++ Programming Guide (PG-02829-001_v11.7)*, 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[142] NVIDIA. *cuRAND Library (PG-05328-050_vRelease)*, 2022. URL: https://docs.nvidia.com/cuda/curand/index.html.

[143] NVIDIA. *cuSOLVER Library (DU-06709-001_v11.6)*, 2022. URL: https://docs.nvidia.com/cuda/cusolver/index.html.

[144] NVIDIA. *cuSPARSE Library (DU-06709-001_v11.6)*, 2022. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

[145] NVIDIA. *Thrust Quick Start Guide (DU-06716-001_v11.6)*, 2022. URL: https://docs.nvidia.com/cuda/thrust/index.html.

[146] E. J. Nyström. Über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Commentationes Physico-Mathematicae*, 4(15):1–52, 1928.

[147] OpenMP Architecture Review Board. *OpenMP API Specification Version 5.2*, Nov. 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[148] T. J. Park, K. J. Han, M. Kumar, and S. Narayanan. Auto-tuning spectral clustering for speaker diarization using normalized maximum eigengap. *IEEE Signal Processing Letters*, 27:381–385, 2019.

[149] C. Patil and I. Baidari. Estimating the optimal number of clusters k in a dataset using data depth. *Data Science and Engineering*, 4(2):132–140, 2019.

[150] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[151] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Icml*, volume 1, pages 727–734, 2000.

[152] X. Peng, L. Zhang, and Z. Yi. Scalable sparse subspace clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 430–437, 2013.

[153] P. Perona and W. Freeman. A factorization approach to grouping. In *European Conference on Computer Vision*, pages 655–670. Springer, 1998.

[154] D. T. Pham, S. S. Dimov, and C. D. Nguyen. Selection of k in k-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 219(1):103–119, 2005.

[155] B. Philippe and S. Yousef. Calcul des valeurs propres. 2008.

[156] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 11(3):430–452, 1990.

[157] J. Qin, W. Wang, C. Xiao, Y. Zhang, and Y. Wang. High-dimensional similarity query processing for data science. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 4062–4063, 2021.

[158] P. Ram and K. Sinha. Revisiting kd-tree for nearest neighbor search. In *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining*, pages 1378–1388, 2019.

[159] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[160] C. K. Reddy and B. Vinzamuri. A survey of partitional and hierarchical clustering algorithms. *Data clustering: Algorithms and applications*, 87, 2013.

[161] E. Rendón, I. Abundez, A. Arizmendi, and E. M. Quiroz. Internal versus external cluster validation indexes. *International Journal of computers and communications*, 5(1):27–34, 2011.

[162] A. Rodriguez and A. Laio. Clustering by fast search and find of density peaks. *science*, 344(6191):1492–1496, 2014.

[163] A. Rosenberg and J. Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.

[164] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[165] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, and S. Selberherr. ViennaCL—linear algebra library for multi- and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, 2016.

[166] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. SIAM, 2011.

[167] T. Sakai and A. Imiya. Fast spectral clustering with random projection and sampling. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 372–384. Springer, 2009.

[168] A. Saxena, M. Prasad, A. Gupta, N. Bharill, O. P. Patel, A. Tiwari, M. J. Er, W. Ding, and C.-T. Lin. A review of clustering techniques and developments. *Neurocomputing*, 267:664–681, 2017.

[169] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[170] E. Schubert, S. Hess, and K. Morik. The relationship of DBSCAN to matrix factorization and spectral clustering. In *LWDA*, 2018.

[171] G. L. Scott and H. C. Longuet-Higgins. Feature grouping by 'relocalisation' of eigenvectors of the proximity matrix. In *BMVC*, pages 1–6. Citeseer, 1990.

[172] G. Sheikholeslami, S. Chatterjee, and A. Zhang. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In *VLDB*, volume 98, pages 428–439, 1998.

[173] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.

[174] H. Shinnou and M. Sasaki. Spectral clustering for a large data set by reducing the similarity matrix size. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, 2008.

[175] P. H. Sneath and R. R. Sokal. Numerical taxonomy. 1973.

[176] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. 2000.

[177] H. Steinhaus et al. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci*, 1(804):801, 1956.

[178] A. Strehl and J. Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research*, 3(Dec):583–617, 2002.

[179] N. Sundaram and K. Keutzer. Long term video segmentation through pixel level spectral clustering on GPUs. In *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain*, 2011.

[180] N. Sylvestre. Résumé de recherches sur le spectral clustering, 2021.

[181] R. D. Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. URL: https://rapids.ai.

[182] F. Tian, B. Gao, Q. Cui, E. Chen, and T.-Y. Liu. Learning deep representations for graph clustering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[183] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2001.

[184] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010.

[185] N. Tremblay and A. Loukas. Approximating spectral clustering via sampling: a review. *Sampling Techniques for Supervised or Unsupervised Tasks*, pages 129–183, 2020.

[186] N. Tremblay, G. Puy, R. Gribonval, and P. Vandergheynst. Compressive spectral clustering. In *International conference on machine learning*, pages 1002–1011. PMLR, 2016.

[187] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854, 2010.

[188] U. von Luxburg. A tutorial on spectral clustering. *Stat. Comput.*, 17(4), 2007.

[189] S. Wagner and D. Wagner. Comparing clusterings-an overview. 2007.

[190] M. Walesiak and A. Dudek. clusterSim: Searching for optimal clustering procedure for a data set (version 0.49-2), January 2021. URL: https://cran.r-project.org/web/packages/clusterSim/index.html.

[191] L. Waltman, N. J. Van Eck, and E. C. Noyons. A unified approach to mapping and clustering of bibliometric networks. *Journal of informetrics*, 4(4):629–635, 2010.

[192] L. Wang and M. Dong. Multi-level low-rank approximation-based spectral clustering for image segmentation. *Pattern Recognition Letters*, 33(16):2206–2215, 2012.

[193] W. Wang, J. Yang, R. Muntz, et al. STING: A statistical information grid approach to spatial data mining. In *Vldb*, volume 97, pages 186–195. Citeseer, 1997.

[194] Y. Weiss. Segmentation using eigenvectors: a unifying view. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 975–982. IEEE, 1999.

[195] G. Wen. Robust self-tuning spectral clustering. *Neurocomputing*, 391:243–248, 2020.

[196] L. Wu, P.-Y. Chen, I. E.-H. Yen, F. Xu, Y. Xia, and C. Aggarwal. Scalable spectral clustering using random binning features. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2506–2515, 2018.

[197] R. Wyrzykowski and F. M. Ciorba. Algorithmic and software development advances for next-generation heterogeneous platforms, 2022.

[198] T. Xiang and S. Gong. Spectral clustering with eigenvector selection. *Pattern Recognit.*, 41(3):1012–1029, 2008.

[199] J. Xie, R. Girshick, and A. Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

[200] T. Xiong, S. Wang, A. Mayers, and E. Monga. Dhcc: Divisive hierarchical clustering of categorical data. *Data Mining and Knowledge Discovery*, 24(1):103–135, 2012.

[201] D. Xu and Y. Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

[202] X. Xu, M. Ester, H.-P. Kriegel, and J. Sander. A distribution-based clustering algorithm for mining in large spatial databases. In *Proceedings 14th International Conference on Data Engineering*, pages 324–331. IEEE, 1998.

[203] D. Yan, L. Huang, and M. I. Jordan. Fast approximate spectral clustering. In *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining, Paris, France, 2009*, 2009.

[204] D. Yan, Y. Wang, J. Wang, G. Wu, and H. Wang. Fast communication-efficient spectral clustering over distributed data. *IEEE Transactions on Big Data*, 7(1):158–168, 2019.

[205] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In *international conference on machine learning*, pages 3861–3870. PMLR, 2017.

[206] M.-S. Yang, C.-Y. Lai, and C.-Y. Lin. A robust EM clustering algorithm for gaussian mixture models. *Pattern Recognition*, 45(11):3950–3961, 2012.

[207] X. Yang, C. Deng, F. Zheng, J. Yan, and W. Liu. Deep spectral clustering using dual autoencoder network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4066–4075, 2019.

[208] T. Yu, W. Zhao, P. Liu, V. Janjic, X. Yan, S. Wang, H. Fu, G. Yang, and J. Thomson. Large-scale automatic k-means clustering for heterogeneous many-core supercomputer. *IEEE Transactions on Parallel and Distributed Systems*, 31(5):997–1008, 2019.

[209] F. Yuan, Z.-H. Meng, H.-X. Zhang, and C.-R. Dong. A new algorithm to get the initial centroids. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826)*, volume 2, pages 1191–1193. IEEE, 2004.

[210] L. Zelnik-Manor and P. Perona. Self-tuning spectral clustering. In *Advances in Neural Information Processing Systems 17 (NIPS 2004), December 13-18, 2004, Vancouver, Canada*, pages 1601–1608, 2004.

[211] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331, 2016.

[212] K. Zhang and J. T. Kwok. Clustered Nyström method for large scale manifold learning and dimension reduction. *IEEE Transactions on Neural Networks*, 21(10):1576–1587, 2010.

[213] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.

[214] W. Zhao, S. Tan, and P. Li. Song: Approximate nearest neighbor search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1033–1044. IEEE, 2020.

[215] J. Zheng, W. Chen, Y. Chen, Y. Zhang, Y. Zhao, and W. Zheng. Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In *2008 13th Asia-Pacific Computer Systems Architecture Conference*, pages 1–8. IEEE, 2008.

[216] Q. Zou, G. Lin, X. Jiang, X. Liu, and X. Zeng. Sequence clustering in bioinformatics: an empirical study. *Briefings in bioinformatics*, 21(1):1–10, 2020.

[217] M. Zubair, A. Iqbal, A. Shil, E. Haque, M. Moshiul Hoque, and I. H. Sarker. An efficient k-means clustering algorithm for analysing covid-19. In *International Conference on Hybrid Intelligent Systems*, pages 422–432. Springer, 2020.

Publications

International conferences

Guanlin He, Stéphane Vialle, and Marc Baboulin. Parallelization of the k-means algorithm in a spectral clustering chain on CPU-GPU platforms. In *Euro-Par 2020 : Parallel Processing Workshops*, volume 12480, LNCS, pages 135-147, Warsaw, Poland, 2021. Springer. ([76])

Guanlin He, Stéphane Vialle, Sylvestre Nicolas, and Marc Baboulin. Scalable algorithms using sparse storage for parallel spectral clustering on GPU. In *18th Annual IFIP International Conference on Network and Parallel Computing (IFIP NPC)*, volume 13152, LNCS, Paris, France, 2021. Springer. ([78])

International journals

Guanlin He, Stéphane Vialle, and Marc Baboulin. Parallel and accurate k-means algorithm on CPU-GPU architectures for spectral clustering. *Concurrency and Computation: Practice and Experience*, page e6621, 2021. ([75])

In submission

Guanlin He, Stéphane Vialle, and Marc Baboulin. Scalable spectral clustering on GPU. *International Journal of Parallel Programming*, 2022. ([77])