Scalable Algorithms Using Sparse Storage for Parallel Spectral Clustering on GPU

Guanlin He, Stephane Vialle, Nicolas Sylvestre, Marc Baboulin

Scalable Algorithms Using Sparse Storage for Parallel Spectral Clustering on GPU

Guanlin He[1,2(✉)], Stephane Vialle[1,2], Nicolas Sylvestre[1], and Marc Baboulin[1,3]

[1] Université Paris-Saclay, CNRS, LISN, Orsay 91405, France
guanlin.he@lisn.fr, {nicolas.sylvestre,marc.baboulin}@upsaclay.fr
[2] CentraleSupélec, Gif-sur-Yvette 91192, France
stephane.vialle@centralesupelec.fr
[3] Université Paris-Saclay, CNRS, CEA, Maison de la Simulation, Gif-sur-Yvette 91191, France

Abstract. Spectral clustering has many fundamental advantages over k-means, but has high computational complexity ($\mathcal{O}(n^3)$) and memory requirement ($\mathcal{O}(n^2)$), making it prohibitively expensive for large datasets. In this paper we present our solution on GPU to address the scalability challenge of spectral clustering. First, we propose optimized algorithms for constructing similarity matrix directly in CSR sparse format on the GPU. Next, we leverage the spectral graph partitioning API of the GPU-accelerated nvGRAPH library for remaining computations especially for eigenvector extraction. Finally, experiments on synthetic and real-world large datasets demonstrate the high performance and scalability of our GPU implementation for spectral clustering.

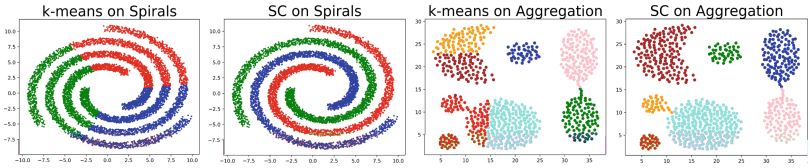
Keywords: Spectral clustering · GPU computing · Similarity matrix construction · Sparse matrix format · Parallel code scalability

1 Introduction and Positioning

Data Clustering. Also known as cluster analysis, data clustering refers to an automatic process that discovers the natural groupings (i.e. clusters) of a set of unlabeled data instances [7]. It belongs to unsupervised machine learning and is one of the most important and challenging tasks in data analysis and pattern recognition. Generally, the clustering process seeks to maximize intra-cluster similarity and minimize inter-cluster similarity.

Various kinds of approaches to clustering have been proposed in the literature. The most well-known one might be the k-means algorithm, which tries to minimize intra-cluster distance iteratively. Although k-means has the virtue of simplicity and speediness, it only forms convex clusters, as shown in Fig. 1a & 1c. Besides, k-means usually suffers from the "curse of dimensionality" because the Euclidean distance metric typically used in the k-means algorithm will lose sensitivity in high-dimensional space [2,6]. Another disadvantage of k-means is

Published by Springer Nature Switzerland AG 2022
C. Cérin et al. (Eds.): NPC 2021, LNCS 13152, pp. 40–52, 2022.
https://doi.org/10.1007/978-3-030-93571-9_4



(a) bad (b) good (c) bad (d) good

Fig. 1. k-means vs. spectral clustering (SC) on 2D shape datasets

the sensitivity to randomized centroid initialization with respect to the result of clustering. Consequently, k-means often gets stuck in local minima solutions, and sometimes even generates arbitrarily bad clusterings, as shown in Fig. 1c. A better centroid initialization approach is the k-means++ seeding method, which chooses centroids by adaptive probabilistic sampling and generally improves both the accuracy and the speed of k-means [1].

Spectral Clustering. A more recent clustering method with many fundamental advantages over k-means is spectral clustering [10]. Based on graph theory, it has a close connection with spectral graph partitioning which tries to minimize the volume of connections between clusters relatively to their size, also known as minimizing balanced cut [12]. Essentially, spectral clustering embeds data into the sub-eigenspace of graph Laplacian and then performs k-means on the embedded representation.[1] However, contrary to k-means, spectral clustering can discover non-convex clusters and is more likely to find the global minimum owing to the *embedding* step, as shown in Fig. 1b & 1d. Moreover, 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. Additionally, when k_c is unknown, the eigenvalues and eigenvectors calculated in spectral clustering algorithm can be exploited to estimate the natural k_c [10,16,18].

Spectral clustering is attractive with the above features, but the algorithm has an important disadvantage: $\mathcal{O}(n^3)$ time complexity [17], mainly due to the calculation of eigenvectors ($\mathcal{O}(n^3)$ when using direct methods) and the construction of similarity matrix ($\mathcal{O}(n^2 d)$), for a dataset with n instances in d dimensions. Thus, spectral clustering will have a prohibitive computational cost as n grows. On the other hand, storing the similarity matrix and the graph Laplacian matrix both need $\mathcal{O}(n^2)$ memory space. Therefore, the high complexities of both computational and memory space requirements lead to a great challenge when addressing large datasets with spectral clustering.

One way to meet the scalability challenge of spectral clustering is to employ modern parallel architectures, such as multi-core CPU and many-core GPU. 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 highly parallel

[1] Therefore, spectral clustering may also be regarded as the combination of a heavy *preprocessing* step (including main computations) and a classical k-means step.

computations. Since spectral clustering needs to construct similarity matrix, and to calculate eigenvectors through linear algebra computations, both with a high degree of parallelism, it appears more interesting to exploit the massively parallel nature of the GPU. However, the GPU has limited global memory resources. How to store the memory-demanding similarity matrix and graph Laplacian matrix on the GPU remains an important concern.

Related Works. There are existing studies related to GPU-accelerated spectral clustering, but we observed the following limitations. First, the benchmark datasets are usually of limited size [8,19]. Second, many studies [3,4,11] are oriented to spectral graph partitioning instead of spectral clustering, thus they typically assume the edge list or the adjacency list of a graph is available, ignoring the construction of similarity matrix that would take $\mathcal{O}(n^2d)$ arithmetical operations in the general case of spectral clustering. Third, two research works have reported some limitations in the speedup of similarity matrix construction [19] and eigenvector computation [8]. Therefore, although adapted to the GPU architecture, a particular attention should be paid to the parallelization of these two calculation steps. An example of video segmentation through spectral clustering in pixel level has been successfully implemented on a cluster of GPUs [14] but unfortunately the authors introduced too briefly their parallelization details and did not give performance analysis of their parallel implementation.

Positioning. In this paper, we focus on the parallelization of spectral clustering algorithm on the GPU in order to address large datasets. The main contributions are our optimized parallel algorithms on the GPU for constructing similarity matrix directly in Compressed Sparse Row (CSR) format. This can achieve significant performance improvements, reduce substantial memory space requirement on the GPU, and make it possible to take advantage of the GPU-accelerated nvGRAPH library for subsequent computations of spectral clustering. Moreover, we analyze the effectiveness and performance of eigensolver-embedded algorithms in the nvGRAPH library.

The remainder of this paper is organized as follows. Section 2 reviews the principles of spectral clustering. Section 3 illustrates our solutions and optimized parallel algorithms for similarity matrix construction directly in CSR format on the GPU. Then we present in Sect. 4 our analysis and exploitation of the nvGRAPH eigensolver-embedded algorithms. Finally we conclude in Sect. 5.

2 Spectral Clustering Principles

Given a set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$ and the number of desired clusters k_c, the first step of spectral clustering is to construct similarity graph and generate corresponding similarity matrix (see Algorithm 1). 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, cosine similarity, etc. The choice of metric should depend on the domain the data comes from and no general advice can be given [10]. The most commonly

used metric seems to be the Gaussian similarity function (see Eq. 2.1), 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. 2.2) appears to be more effective for data in high-dimensional space [6]. **(2) How to construct the similarity graph.** There are several common ways, such as *full connection*, *ε-neighborhood* and *k-nearest neighbors* [10]. The first way generates a dense matrix. The last two ways yield typically a sparse similarity matrix by setting the similarity s_{ij} to zero 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 neighbors* seems more computationally expensive as it requires sorting operations.

Algorithm 1: Spectral clustering algorithm

Inputs: A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$, the number of desired clusters k_c

Outputs: Cluster labels of n data instances

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

$$\text{Gaussian similarity metric: } s_{ij} = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right) \tag{2.1}$$

$$\text{Cosine similarity metric: } s_{ij} = \frac{x_i \bullet x_j}{\| x_i \| \| x_j \|} \tag{2.2}$$

The generated similarity matrix S is symmetric and of $n \times n$ size. Then we derive the diagonal degree matrix D with $deg_i = \sum_{j=1}^{n} s_{ij}$. Next, we calculate the (unnormalized) graph Laplacian $L = D - S$ which does not depends on the diagonal elements of the similarity matrix and whose eigenvalues and eigenvectors are associated with many properties of graphs [10]. Moreover, L can be further normalized as the symmetric matrix $L_{sym} := D^{-1/2} L D^{-1/2}$ or the non-symmetric matrix $L_{rw} := D^{-1} L$. In order to achieve good clustering in broader cases, it is argued and advocated [10] 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 `syevdx` symmetric eigensolver in the cuSOLVER library).

From the graph cut point of view, clustering on a dataset X is equivalent 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, the solution can be approximated from the first k_c eigenvectors (associated with the smallest k_c eigenvalues) of graph Laplacian matrix [10,11]. Let U denote the $n \times k_c$ matrix containing the eigenvectors

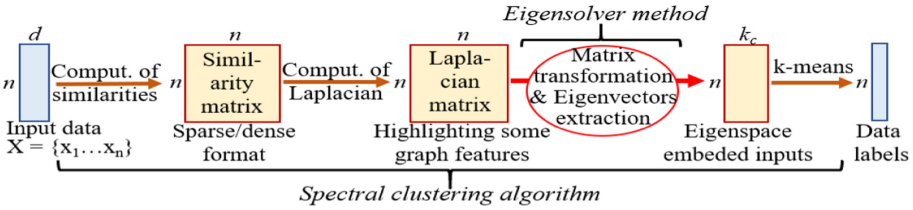


Fig. 2. Main computation steps in spectral clustering

as columns. Then 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.

Finally, the k-means algorithm is applied on the embedded representation by regarding 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, before performing the final k-means, it is customary to scale each row of matrix U to unit length to improve the clustering result.

To summarize, spectral clustering involves several data transformation steps, illustrated in Fig. 2. A similarity matrix is computed based on the nature of the dataset and the clustering objective to model a connectivity graph, and 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.

3 Similarity Matrix Construction in CSR on GPU

In this section, we focus on the design of GPU parallel algorithms for constructing similarity matrix directly in CSR sparse format, in order to address both the $\mathcal{O}(n^2 d)$ computational challenge and the $\mathcal{O}(n^2)$ memory requirement of similarity matrix construction.

3.1 Need of Passing from Dense Format to Sparse Format

Initially we started from constructing the similarity matrix S simply in dense format on the GPU. Then we chose to compute the normalized graph Laplacian matrix L_{sym} (symmetric) instead of L_{rw} (non-symmetric) so that we could calculate its smallest k_c eigenvectors with `syevdx`, a dense symmetric eigensolver in the GPU-accelerated cuSOLVER library from NVIDIA. Finally, we applied our GPU parallel algorithm of k-means [5] on the points defined by the rows of k_c-eigenvector matrix to obtain the clustering result.

However, we quickly ran out of limited GPU memory when trying to process datasets with larger n since the similarity matrix and the graph Laplacian matrix were both constructed in dense format with $\mathcal{O}(n^2)$ memory requirement.

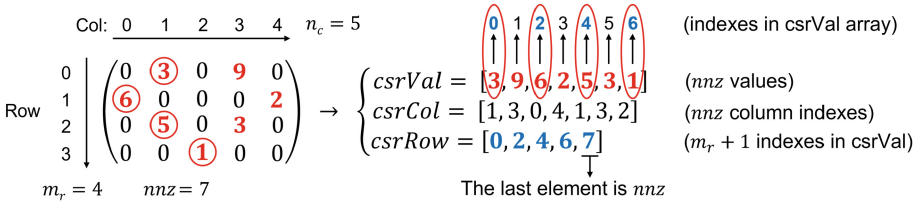


Fig. 3. An example for CSR format with an $m_r \times n_c$ matrix

On the other hand, the similarity matrix associated with *ε-neighborhood* graph or *k-nearest neighbors* graph generally has a sparse pattern, i.e. containing numerous zeros. Even for the similarity matrix associated with fully connected graph, we observed typically a significant portion of elements are close to zero. By setting a small threshold (e.g. 0.01) and regarding those under-threshold similarities as zeros, we are likely to obtain a sparse similarity matrix. Storing this array in a sparse format rather than a dense format significantly saves GPU memory, which greatly increases the scale of datasets that can be processed.

3.2 Choice of CSR Sparse Format

There are several commonly used sparse formats for storing a sparse matrix, such as Coordinate Format (COO), Compressed Sparse Row Format (CSR), Compressed Sparse Column Format (CSC), Ellpack, etc. In our case of sparse similarity matrix, we choose to use the CSR format for two reasons. First, the CSR format can usually achieve a good trade-off between memory space requirement and operation flexibility, and is efficient both for regular sparsity pattern and for power-law distribution [3]. With these advantages, the CSR format has been widely used and supported in most libraries. Second, we intend to employ the nvGRAPH spectral graph partitioning API but it supports only the CSR format for graph representation.

A sparse matrix represented in CSR format consists of three arrays. We call them `csrVal`, `csrCol`, `csrRow`. Figure 3 gives a simple example of CSR representation. The `csrVal` stores all nonzero values of the matrix in row-major format. The `csrCol` contains the column index of every nonzero element. Considering the first nonzero element in each row of the matrix (i.e. the circled red numbers in Fig. 3), the `csrRow` holds their indices that count in the `csrVal` array (i.e. the blue numbers circled by red ellipses) and contains in the end the total number of nonzeros in the matrix.

3.3 Difficulties

We have not found any existing work on the construction of similarity matrix in CSR format on the GPU, neither in the literature nor in numerous GPU-accelerated libraries. Studies [3,4,11,12] in the field of graph partitioning often

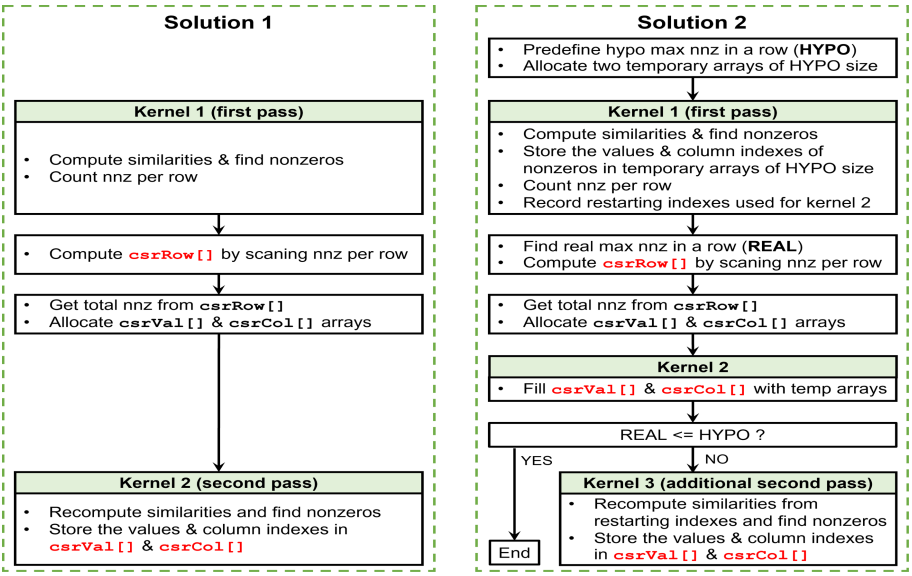


Fig. 4. Our two solutions for CSR similarity matrix construction on GPU

target graphs in COO topology represented by an edge list or in CSR topology represented by an adjacency list. They typically assume the availability of these sparse format lists, and do not consider or do not need the construction process. Another work [8] constructs sparse similarity matrix in COO format but on the assumption that the neighborhood information is given by an edge list.

In the standard case of spectral clustering, we have no such edge list or adjacency list available, but only n data instances in $\mathbb{R}^d$ dimensions. To obtain the similarity matrix in CSR format and save memory space, it makes no sense to first construct a sparse similarity matrix in dense format and then transform it from dense to CSR format. Thus, the construction of similarity matrix must be directly in CSR format. However, this is difficult to be done in parallel especially on the GPU, because (1) the total number of nonzeros is unknown, so we cannot allocate memory for the `csrVal` and `csrCol` arrays; (2) the number of nonzeros per row is unknown, so we cannot know in which segment of `csrVal` and `csrCol` we should store the value and column index of each nonzero, respectively; (3) although GPU threads can compute similarities and find nonzeros in parallel, they cannot parallelly store nonzeros (values and column indexes) at the right place of `csrVal` and `csrCol`, since each thread does not know the number of nonzeros ahead of it.

3.4 Our Solutions

We propose two solutions for the similarity matrix construction in CSR on GPU.

Table 1. Datasets and parameter settings of our benchmarks

Dataset	(n, d, k_c)	Similarity metric	Threshold	Block size of CUDA kernels
MNIST-60K	(60K, 784, 10)	Cosine	0.8 (similarity)	64
MNIST-120K	(120K, 784, 10)	Cosine	0.8 (similarity)	64
MNIST-240K	(240K, 784, 10)	Cosine	0.8 or 0.84* (similarity)	64
Synthetic-1M	(1M, 4, 4)	Gaussian ($\sigma = 0.01$)	0.0002 (distance)	128
Synthetic-5M	(5M, 4, 4)	Gaussian ($\sigma = 0.01$)	0.0001 (distance)	128

Table 2. Performance comparison on GPU of our 2 solutions (S1 vs. S2)

Dataset	Max nnz in a row	Average nnz per row	Total nnz	Sparsity (% of 0)	S1 (s)	S2 (s)		Best speedup S2 *vs* S1
						<1st HYPO>	<2nd HYPO>	
MNIST-60K	2196	251	15.1M	99.581%	7.16	5.29 <1024>	5.95 <2048>	**1.35**
MNIST-120K	3310	299	35.9M	99.751%	29.98	24.5 <1024>	24.39 <2048>	1.23
MNIST-240K	5552	478	114.8M	99.801%	126.85	103.75 <1024>	104.77 <2048>	1.22
MNIST-240K*	3520	199	47.8M	99.917%	125.17	91.75 <1024>	102.10 <2048>	**1.36***
Synthetic-1M	54	23	23.4M	99.998%	13.57	10.20 <16>	8.35 <54>	**1.63**
Synthetic-5M	64	29	149.9M	99.999%	362.79	318.45 <32>	312.48 <64>	1.16

Solution 1. As shown in the left part of Fig. 4, our Solution 1 mainly consists of two kernels with two complete passes across all the elements in similarity matrix. The first pass in Kernel 1 computes the similarities of all pair of instances, find the over-threshold similarities as nonzeros, and count the number of nonzeros (nnz) per row. Then the `csrRow` can be obtained by performing an exclusive scan on the array of nnz per row. Moreover, the total nnz can be known from the last element of `csrRow` and we can therefore allocate `csrVal` and `csrCol` arrays. With all these materials, we then launch the second pass in Kernel 2 to recompute all the similarities, find nonzeros as the first pass, and finally fill the `csrVal` and `csrCol` arrays.

Solution 2. As shown in the right part of Fig. 4, our Solution 2 is primarily composed of three kernels performing a single pass or possibly two passes (when running Kernel 3). It requires to predefine a hypothesis (HYPO) for the maximum number of nonzeros in a row, and allocate two temporary arrays of HYPO size for `csrVal` and `csrCol`. Then the first pass in Kernel 1 needs to undertake several tasks: not only compute all similarities, find nonzeros, count nnz per row, but also store the information of nonzeros in the temporary arrays, and meanwhile record the restarting places for the additional pass in Kernel 3 in case that our hypothesis is wrong. Then we can find the real maximal nnz in a row (REAL) from the nnz per row array. Next, we compute the `csrRow`, the total nnz, and allocate `csrVal` and `csrCol` arrays as our Solution 1. Thanks to the results stored in the temporary arrays in Kernel 1, we can just fill `csrVal` and `csrCol` arrays with Kernel 2. If our hypothesis is correct (REAL <= HYPO), then we will obtain the complete result in Kernel 2. Otherwise, we need to conduct an additional second pass in Kernel 3 to find the nonzeros out of hypothesis and store them at the right place in `csrVal` and `csrCol`. Particularly, the additional

pass will not traverse all elements but will only start from the restarting indices recorded in Kernel 1.

Parallel Implementation on GPU. Both solutions are mainly implemented with our optimized CUDA kernels. For each of the CUDA kernels in Fig. 4, we create a 1D grid containing n 1D blocks. Each block of threads process one row of the matrix in a loop fashion. For Solution 2, we pay particular attention to the memory address alignment of restarting indexes on multiples of 32 memory words, which proves to have an significant impact on performance.

3.5 Experiments with Our Two Solutions

We tested our two solutions on a GeForce RTX 3090 with the datasets and parameter settings shown in Table 1. For the reason of comparison, we set the same threshold (0.8) for all MNIST-based datasets (https://leon.bottou.org/projects/infimnist). However, only the first two datasets yield satisfying clusterings with this threshold, while the MNIST-240K needs a higher threshold (0.84, marked with *) to generate a good clustering. Before clustering, we perform feature scaling for the synthetic datasets to transform all values into the same scale within the range [0,1]. Note that we do not have a version or benchmark on CPU yet.

Table 2 shows the results of our benchmarks. We observed that the similarity matrices are extremely sparse although they contain millions of nonzeros. Both two solutions are scalable up to 5 million instances. With appropriate hypotheses, our Solution 2 outperforms Solution 1 with a speedup from $\times$1.16 up to $\times$1.63.

However, this is conditional. Figure 5 shows the impact of hypothesis on the performance of our Solution 2 with the MNIST-120K set. When the hypothesis is small (HYPO $<$ 256) or very large (HYPO $>$ 2399), our Solution 2 is less efficient than Solution 1. Specifically, a smaller hypothesis will leave more computations and more memory accesses to Kernel 3 of Solution 2, while a greater hypothesis will need Kernel 1 of Solution 2 to compute more and record more in global memory. In particular, we observed two sudden increases of time when the hypothesis grows from 2398 to 2399 and from 3038 to 3039. We think they are caused by the decrease of the number of resident blocks in Stream Multiprocessors, because we use much shared memory when the hypothesis is large. Although our Solution 2 can be influenced by the value of hypothesis, we found a fairly wide range [256, 2399] where Solution 2 achieves better performance than Solution 1. Hence finding an appropriate hypothesis should not be difficult if we avoid extreme values. The Kernel 2 of Solution 2 consumes little time compared to other kernels, so is omitted here.

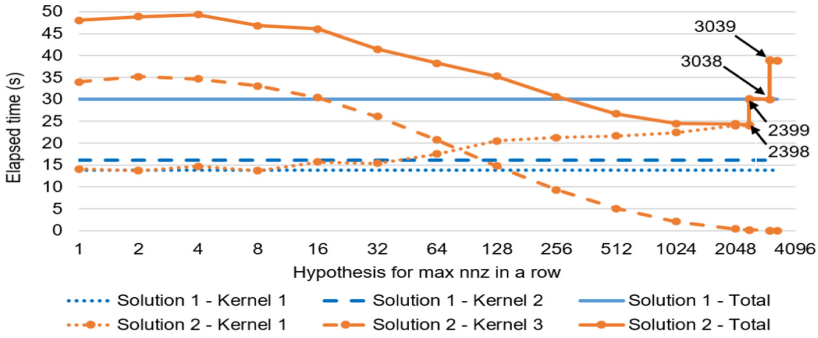


Fig. 5. Detailed comparison of our 2 solutions on the MNIST-120K dataset

4 Eigenvector Extraction for Spectral Clustering on GPU

4.1 Eigensolver Methods

As presented in Sect. 2, a key step of spectral clustering is to calculate the first k_c eigenvectors of Laplacian matrix. This can be done with eigensolver methods. They include new matrix transformations to facilitate the eigenvectors extraction and are not specific to spectral clustering. Three well-known methods are the following:

- **Arnoldi's** method [13]: it takes any input matrix (like L, L_{sym} or L_{rw}, see Sect. 2) and transforms it into an Hessenberg matrix, then calls an eigensolver (usually the solver QR). This is a generic but computationally expensive method.
- **Lanczos** method [13]: similar to Arnoldi's method but requires a real and symmetric (or Hermitian) input matrix (like L or L_{sym}) which it transforms into a tridiagonal matrix, before calling an eigensolver (like QR). This is an efficient method but it suffers from numerical instabilities.
- **LOBPCG** method [9]: 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.

Implementations of these algorithms exist in different libraries. They require input matrices in dense or sparse format and are sometimes improved to be more robust to numerical instabilities.

4.2 Eigensolver-Embedded Algorithms in the nvGRAPH Library

With the similarity matrix in CSR sparse format defined in Sect. 3, the remaining steps of spectral clustering can be completed by calling the spectral graph partitioning API of nvGRAPH library [12], designed by NVIDIA for GPU-accelerated

graph analytics. The API takes as input graph the similarity matrix only in CSR format, and then performs spectral graph partitioning with the following three selectable algorithms:

- **Maximization of the *modularity* with Lanczos method**. The *modularity* measures how well a partitioning applies to the target graph compared to a random graph. It can be approximately maximized by looking at the largest eigenpairs of *modularity* matrix [3]. This modularity matrix instead of the Laplacian matrix is constructed before eigenpair computation with the Lanczos solver.
- **Minimization of the balanced cut with Lanczos method**. This algorithm aims at minimizing the volume of inter-cluster connections relative to the size of clusters (i.e. balanced cut). It constructs the Laplacian matrix and then calls the Lanczos solver.
- **Minimization of the balanced cut with LOBPCG method**. Similar principle to the second algorithm, but utilizes the LOBPCG solver to handle the constructed Laplacian matrix.

Compared to Lanczos method, LOBPCG can handle eigenvalues with multiplicity (which often happens in spectral clustering). Moreover, the NVIDIA implementation is able to restart the computation when it encounters numerical instabilities. Thus this LOBPCG-embedded algorithm has appeared as the most reliable solution on our benchmarks. Note that all the three algorithms contain the k-means step at the end.

4.3 Experiments with nvGRAPH Library

Following the similarity matrix construction in CSR format (Sect. 3), we tested the nvGRAPH spectral graph partitioning API to complete spectral clustering on the GPU. The LOBPCG-embedded algorithm is selected for our benchmarks. Several parameters need to be specified. 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 because it has a significant impact on the clustering quality and the execution time. Depending on the benchmarks, a too small tolerance may lead to eigensolver divergence and too much execution time, while a too large tolerance can result in bad clusterings. Besides, the tolerance for k-means is set to 0.0001.

Table 3 presents the elapsed time of our spectral clustering (including the nvGRAPH API) on the GeForce RTX 3090, as well as the clustering quality measured by three metrics: Rand Index (RI), Ajusted Rand Index (ARI), Normalized Mutual Information (NMI) [15]. The last two metrics are stricter than the first one. All metrics return a score less or equal to 1, and a score closer to 1 indicates a better clustering. Our benchmarks on the MNIST-based datasets (handwritten digits) yielded relatively good clustering quality, while we found perfect clusterings on the synthetic datasets (convex clusters).

Table 3. Elapsed time and clustering quality of spectral clustering on GPU

Dataset	Elapsed time (s)				Quality metric		
	Data transfers	Similarity matrix constr. in CSR	nvGRAPH LOBPCG <eigen. tolerance>	Total	RI	ARI	NMI
MNIST-60K	0.015	**5.29**	2.35 <0.005>	7.66	0.93	0.63	0.74
MNIST-120K	0.031	**24.39**	4.03 <0.005>	28.45	0.89	0.50	0.65
MNIST-240K	0.062	**91.75**	5.42 <0.005>	97.23	0.88	0.47	0.69
Synthetic-1M	0.002	**8.35**	3.61 <0.001>	11.96	1.00	1.00	1.00
Synthetic-5M	0.008	**312.48**	29.75 <0.0001>	342.24	1.00	1.00	1.00

With respect to the elapsed time, the similarity matrix construction turns out to be the most time-consuming step of spectral clustering, mainly due to its $\mathcal{O}(n^2d)$ time complexity. Although the theoretical complexity of eigenvector computation is $\mathcal{O}(n^3)$, the nvGRAPH LOBPCG eigensolver did not take much time compared to the similarity matrix construction. This is mainly due to the fact that the eigensolver adopts an iterative and approximate method instead of direct methods. The data transfers between CPU to GPU appears negligible. Finally, the large datasets demonstrate the scalability on a GPU of our implementation of spectral clustering with algorithms optimized for CSR sparse matrix generation (although it still dominates the runtime).

5 Conclusion and Future Work

We have presented our scalable parallel algorithms for spectral clustering on a single GPU. We have proposed two solutions for the construction of similarity matrix directly in CSR sparse format on GPU, which can save a large amount of GPU memory space compared to dense format storage. Moreover, our matrix generation in CSR format is compatible with nvGRAPH's eigensolver-embedded algorithms that require CSR matrices. With our sparse matrix construction and nvGRAPH's eigensolvers, we have obtained a parallelized end-to-end spectral clustering implementation on one GPU. Finally, our experiments show that our GPU implementation succeeds to scale up to millions of data instances.

To address even larger datasets, it would be interesting to parallelize spectral clustering on multi-GPU machines which provide more computing power and memory space. Another solution would be CPU-GPU algorithms incorporating the *representative* extraction technique on the CPU to reduce the number of data instances to process on the GPU. Moreover, a comparison with a purely CPU multithreaded and vectorized version would be interesting in the future.

Acknowledgement. This work was supported in part by the China Scholarship Council (No. 201807000143). The experiments were conducted on the research computing platform supported in part by Région Grand-Est, Metz-Métropole and Moselle Departement.

References

1. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, USA (2007)
2. Domingos, P.: A few useful things to know about machine learning. Commun. ACM **55**(10), 78–87 (2012)
3. Fender, A.: Parallel solutions for large-scale eigenvalue problems arising in graph analytics. Ph.D. thesis, Université Paris-Saclay (2017)
4. Fender, A., Emad, N., et al.: Accelerated hybrid approach for spectral problems arising in graph analytics. Procedia Comput. Sci. **80**, 2338–2347 (2016)
5. He, G., Vialle, S., Baboulin, M.: Parallel and accurate k-means algorithm on CPU-GPU architectures for spectral clustering. Concurr. Comput. Pract. Exp., e6621 (2021). https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6621
6. Ina, T., Hashimoto, A., Iiyama, M., Kasahara, H., Mori, M., Minoh, M.: Outlier cluster formation in spectral clustering. arXiv preprint arXiv:1703.01028 (2017)
7. Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern Recogn. Lett. **31**(8), 651–666 (2010)
8. Jin, Y., JáJá, J.F.: A high performance implementation of spectral clustering on CPU-GPU platforms. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Chicago, IL, USA, pp. 825–834 (2016)
9. Knyazev, A.V.: Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. SIAM J. Sci. Comput. **23**(2), 517–541 (2001)
10. von Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**(4), 395–416 (2007)
11. Naumov, M., Moon, T.: Parallel spectral graph partitioning. Technical report, NVIDIA Technical Report, NVR-2016-001 (2016)
12. NVIDIA: NVGRAPH library user's guide (2019)
13. Saad, Y.: Numerical Methods for Large Eigenvalue Problems. SIAM, Philadelphia (2011)
14. Sundaram, N., Keutzer, K.: 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)
15. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: variants, properties, normalization and correction for chance. J. Mach. Learn. Res. **11**, 2837–2854 (2010)
16. Xiang, T., Gong, S.: Spectral clustering with eigenvector selection. Pattern Recognit. **41**(3), 1012–1029 (2008)
17. Yan, D., Huang, L., Jordan, M.I.: Fast approximate spectral clustering. In: Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining, Paris, France, 2009 (2009)
18. Zelnik-Manor, L., Perona, P.: Self-tuning spectral clustering. In: Advances in Neural Information Processing Systems 17 (NIPS 2004), Vancouver, Canada, 13–18 December 2004, pp. 1601–1608 (2004)
19. Zheng, J., Chen, W., Chen, Y., Zhang, Y., Zhao, Y., Zheng, W.: Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In: 2008 13th Asia-Pacific Computer Systems Architecture Conference, pp. 1–8. IEEE (2008)