International Journal of Parallel Programming (2025) 53:22
https://doi.org/10.1007/s10766-025-00799-y

Generating Sparse Matrices for Large-Scale Spectral Clustering on a Single GPU

Guanlin He[1,2,3] · **Stéphane Vialle**[2,3] · **Marc Baboulin**[2]

Received: 4 April 2022 / Accepted: 24 March 2025

Abstract
Spectral clustering has many fundamental advantages over *k*-means clustering, but comes at much higher time complexity and memory requirements mainly due to similarity matrix construction and eigenvectors computation. Thus, spectral clustering is prohibitively expensive for processing large datasets. In this paper we address the scalability challenge of spectral clustering on single-GPU architectures. An $n \times n$ similarity matrix generally contains many elements close to zero, and can become very sparse by applying a threshold on matrix elements. Then it can take advantage of sparse storage format like CSR if the matrix is generated directly in sparse format, which allows processing large datasets on just one GPU device. We obtain a sparse similarity matrix by constructing the ϵ-neighborhood similarity graph and generating the associated sparse matrix in the CSR format on the GPU. Then we leverage the spectral graph partitioning API of the GPU-accelerated nvGRAPH library for remaining computations especially the eigen-decomposition. Finally, we provide experiments on synthetic and real-world large datasets which demonstrate the performance and scalability of our GPU implementation for spectral clustering.

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

✉ Guanlin He
guanlin.he@mail.xhu.edu.cn

Stéphane Vialle
stephane.vialle@centralesupelec.fr

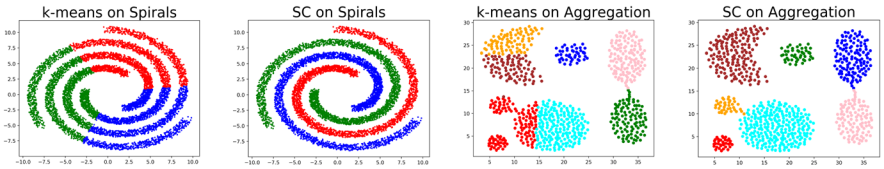
Marc Baboulin
marc.baboulin@upsaclay.fr

[1] School of Computer and Software Engineering, Xihua University, Chengdu 610039, China

[2] LISN, CNRS, Université Paris-Saclay, Orsay 91405, France

[3] CentraleSupélec, Gif-sur-Yvette 91192, France

Published online: 26 May 2025



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

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

1 Introduction

Data clustering, also known as cluster analysis, refers to an automatic process that discovers the natural groupings (i.e., clusters) of a set of unlabeled data instances [17]. 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 to minimize inter-cluster similarity.

Various kinds of approaches for clustering have been proposed in the literature. A well-known one is the *k*-means algorithm [21], which tries to minimize intra-cluster distance iteratively. Although *k*-means has the virtue of simplicity and speediness, it usually forms convex clusters even if they do not really exist, as shown in Fig. 1a. Besides, *k*-means suffers from the "curse of dimensionality" because the Euclidean distance metric used by *k*-means will lose sensitivity in high-dimensional space [6, 16]. Another disadvantage of *k*-means is 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 [4].

Spectral clustering [24] is a more recent clustering method with many fundamental advantages over *k*-means. 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 [25]. Essentially, spectral clustering 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).[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, d. 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

[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.

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 [20, 38, 41].

Spectral clustering is attractive with the above features, but its classical algorithms have a serious disadvantage: $\mathcal{O}(n^3)$ time complexity [39], 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), for a dataset with n instances in d dimensions. Moreover, storing the similarity matrix and the graph Laplacian matrix requires $\mathcal{O}(n^2)$ memory space. Therefore, the high complexities of both computational and memory space requirements lead to a great challenge when processing large datasets with spectral clustering.

One way to address 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 computations. Due to the computational cost for 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 resources. How to store the memory-demanding similarity matrix and the graph Laplacian matrix on the GPU remains an important concern.

In this paper, we focus on the parallelization of spectral clustering algorithm in order to address large datasets on a single GPU. Our main contributions are three different algorithms and associated optimized parallel implementations for constructing an ϵ-neighborhood similarity graph and storing associated sparse matrix in Compressed Sparse Row (CSR) format on a single GPU (without using any temporary full matrix in dense format). This can achieve significant performance improvements, reduce substantial memory space requirements on the GPU, and make it possible to take advantage of NVIDIA's GPU-accelerated nvGRAPH library for subsequent computations of spectral clustering.

This paper is a significantly extended version of our previous paper [15] published in the proceedings of NPC 2021. First, we enhance the performance of the two algorithms initially proposed in our previous paper by improving the CUDA kernels and their grid and block configurations. Second, we propose a new algorithm based on a chunkwise dense-to-CSR approach which outperforms the previous two algorithms in some benchmarks. We give much more details about our parallel implementations and code optimizations on the GPU. Besides, we also complement the investigation into related works. Finally, a more comprehensive experimental evaluation is provided, including performance comparison against an optimized CPU version code.

The remainder of this paper is organized as follows. Section 2 reviews the principles of spectral clustering and Sect. 3 summarizes related works. Then Sect. 4 describes our algorithms and optimized GPU implementations for the construction of the similarity graph/matrix in CSR format. We present in Sect. 5 the exploitation of nvGRAPH eigensolvers for spectral graph partitioning on the GPU. Finally the experimental evaluation is given in Sect. 6 and we conclude in Sect. 7.

2 Spectral Clustering Principles

Given a set of data instances $X = \{x_1, \ldots, x_n\}$ with x_i in $\mathbb{R}^d$ and the number of desired clusters k_c, the first step of spectral clustering is to construct the similarity graph and generate the corresponding similarity matrix $S = [s_{ij}]_{i,j=1,\ldots,n}$ (a.k.a. affinity matrix or adjacency matrix in the literature). 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 [20]. 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 [16]. Note that by definition $s_{ij} = 0$ if $i = j$, i.e., the diagonal elements of the similarity matrix are always 0. *(2) How to construct the similarity graph?* There are several common ways, such as *full connection*, *ε-neighborhood* and *k-nearest neighbor* [20]. 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.

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

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

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 (together called eigenpairs) are associated with many properties of graphs [20]. 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 [20] 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 [26]).

From the graph cut point of view, clustering on a dataset X corresponds to partitioning a graph G into k_c partitions by finding a minimum balanced cut. Therefore,

spectral clustering is similar to spectral graph partitioning, except that the former includes similarity graph/matrix construction step while the latter does not. *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 algorithm can be approximated from the first k_c eigenvectors (associated with the smallest k_c eigenvalues) of graph Laplacian matrix [20, 22]. Let U denote the $n \times k_c$ matrix containing the eigenvectors 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 Related Works

We have investigated not only the related works on GPU-accelerated spectral clustering, but also the existing works on similarity graph construction and on the calculation of the first few eigenpairs since they constitute the two most computationally expensive steps of spectral clustering.

3.1 GPU-Accelerated Spectral Clustering

The first paper on this topic [42] that we found was published in 2008, shortly after CUDA came out. It parallelizes spectral clustering algorithm on multi-core CPU

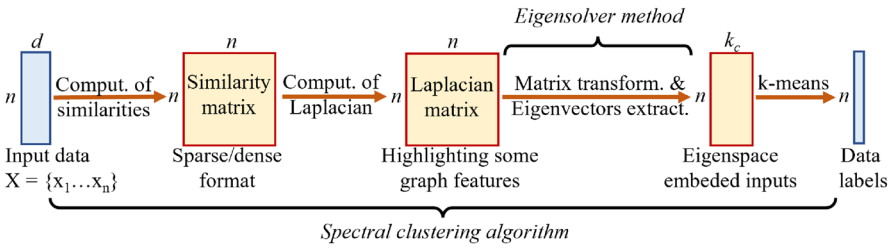


Fig. 2 Main computation steps in spectral clustering

and on GPU. However, dense matrices are constructed and the 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 [35], but unfortunately the authors introduced too briefly their parallelization details and did not give performance analysis of their parallel implementation.

Another work [18] 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 [8, 9, 22], and released the products in the nvGRAPH library [25] and RAPIDS cuGraph library [31]. 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.

3.2 Similarity Graph/Matrix Construction

To the best of our knowledge, most existing works on graph construction [2, 3, 7, 12] target *k-nearest neighbor* graph. We found few related works on the construction of *ε-neighborhood* graph in sparse format on the GPU. The sole work on it that we found [18] constructs sparse similarity matrix in 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 Sect. 4).

Note that there are many works and libraries on CSR-based sparse matrix–vector multiplication on GPU, e.g., Greathouse and Daga [11], Gao et al. [10], NVIDIA [27], but they usually assume CSR format matrices are already provided.

3.3 Eigensolver Methods and GPU Implementations

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 [33]: 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 based on the *QR* algorithm). This is a generic but computationally expensive method.
- *Lanczos* method [33]: 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) [22].

- *LOBPCG* method [19]: 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. [22].

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, nvGRAPH, cuGraph, MAGMA, AmgX, and ViennaCL.

The cuSOLVER library [26] is a GPU-accelerated library from NVIDIA providing LAPACK-like features (decompositions and linear system solutions) for both dense and sparse matrices. The sole sparse eigensolver within cuSOLVER (including cuSolverSP) is `csreigvsi`, which 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 [25] 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 show in Sect. 5 the use of these algorithms with more details. However, since the last release in November 2019 with CUDA 10.2, NVIDIA does not actively develop the nvGRAPH product any more. Despite this situation, the code and installation guide of nvGRAPH library is publicly available,[2] which provides a way for nvGRAPH users to continue using nvGRAPH after the CUDA Toolkit stops releasing it. In place of nvGRAPH, NVIDIA has been actively developing the 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 [31], 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[3] on our machine, and we succeeded in using the C++/CUDA API of cuGraph's graph partitioning

[2] https://github.com/rapidsai/nvgraph.

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

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[4] [36] is a public domain linear algebra library optimized for "multi-core + multi-GPU" hybrid architectures. The sole sparse eigensolver of MAGMA is a generic GPU implementation of the LOBPCG method (not particularly designed for the eigenvalue problem of spectral clustering). We tried it (with MAGMA 2.5.4) to calculate the eigenvectors of the graph Laplacian matrix, but unfortunately it failed (see Sect. 6.1.2).

The Algebraic Multigrid Solver (AmgX) library[5] [23] 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. Besides, the ViennaCL library[6] [32] 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. But we have yet to test the eigensolvers of AmgX and ViennaCL libraries for spectral clustering.

Certainly, it is also possible to implement a new eigensolver and a new spectral clustering library by leveraging existing works on CSR-based sparse matrix–vector multiplication on GPU [10, 11]. However, this would be a huge work.

Based on the above findings, we mainly rely on the sparse eigensolvers embedded in nvGRAPH's graph partitioning algorithms.

4 Construction of the Similarity Graph/Matrix in Sparse Format

Initially we implemented spectral clustering on the GPU by constructing the similarity matrix and Laplacian matrix in dense format, then exploiting the dense symmetric eigensolver `syevdx` of cuSOLVER library and finally applying the GPU implementation of the k-means algorithm [14]. However, as the number of data instances n grows over the order of 10^4, it becomes impossible to store the dense-format square matrices with limited GPU memory.

In this section we focus on the design of 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 computational cost and memory requirements.

[4] https://icl.utk.edu/magma/index.html.
[5] https://github.com/NVIDIA/AMGX.
[6] http://viennacl.sourceforge.net.

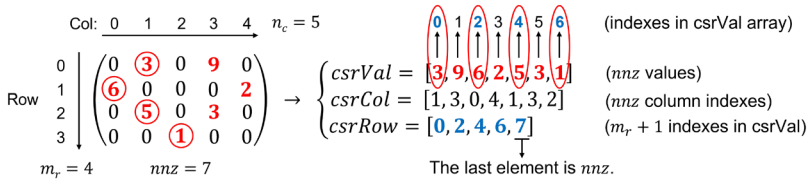


Fig. 3 An example of CSR format for storing an $m_r \times n_c$ matrix

4.1 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 many zeros. Even for the similarity matrix associated with *fully connected* graph, we observe that usually a significant portion of elements are very close to 0. By setting a small threshold for similarity and regarding those below-threshold similarities as 0, 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 0 and while the latter sets similarities associated with over-threshold distances to 0. For simplicity, we call both of the related graphs as *ε -neighborhood-like* graph in this paper. Storing the similarity matrix in a sparse format will require much less memory space than a dense storage and thus will increase significantly the size of datasets able to be processed on the GPU.

There are various formats for storing a sparse matrix. Several commonly used ones are: Coordinate format (COO), Compressed Sparse Row format (CSR), Compressed Sparse Column format (CSC), and Ellpack format [5, 8, 30]. We choose 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 [8] and usually requires less memory than COO and Ellpack formats. Moreover, the CSR format is efficient for matrix–vector computations[7]. 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.

The CSR format of a sparse matrix consists of three arrays. We call them `csrVal[]`, `csrCol[]`, `csrRow[]`. Figure 3 gives a CSR example with the $m_r \times n_c$ matrix. `csrVal[]` and `csrCol[]` store the values and column indexes of all nonzero matrix elements in row-major format, respectively. `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 requirements for CSR format are $2 \times nnz + m_r + 1$,

[7] From SciPy API reference: `scipy.sparse.csr_matrix`.

where *nnz* represents the total number of nonzeros in a matrix (see annotations on right side of Fig. 3). In graph analytics, the CSR representation of similarity graph corresponds to an adjacency list, where for each vertex v_i, the neighbors $v_j, \ldots, v_k$ and optionally the corresponding edge weights $w_j, \ldots, w_k$ are listed.

We also introduce the Ellpack format as it will be used as an intermediate storage format later in one of our proposed algorithms. It consists of two arrays. We call them `elpVal[]` and `elpCol[]`. Figure 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. For each row, a segment of size *maxNnzRow* is reserved 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 extra space will be wasted, as marked '*' in the figure. Therefore, the memory requirements for Ellpack format are $2 \times m_r \times maxNnzRow$.

4.2 Difficulties

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 Sect. 4.4).

We point out that in this paper 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. In the following we propose 3 different algorithms and their associated

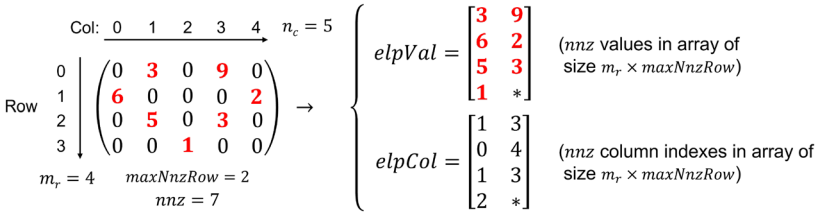
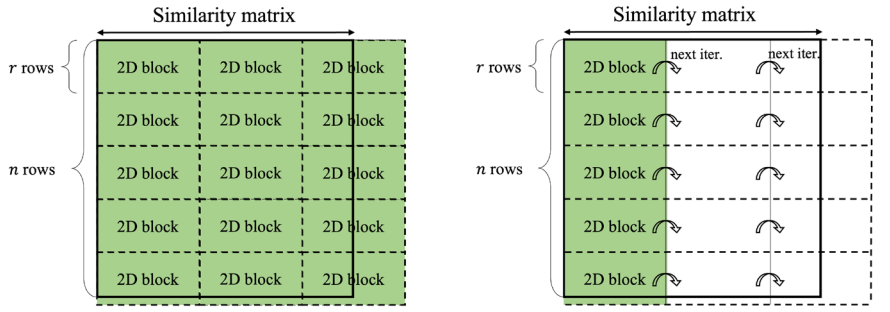


Fig. 4 An example of Ellpack format for storing an $m_r \times n_c$ matrix



(a) 2D grid with 2D blocks for the `1stPass` kernel of Algo CSR-1 & for the `chkPass` kernel of Algo CSR-3

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

Fig. 5 Grid and block configuration for our CUDA kernels. Each solid frame stands for a similarity matrix, each green box stands for a block of threads, a thick dash box stands for similarities processed by one block of threads

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. Due to space limitation, we only present the code listing of the most complex kernel (Listing 1).

4.3 Algo CSR-1: Straightforward CSR

Algorithm 1 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[]` after computing each similarity and finding nonzeros that satisfy a predefined threshold. Then 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 recomputing all similarities and recomparing them against the threshold (they could not be saved in the first pass).

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

Input:
(1) A set of data instances $X = \{x_1, ..., x_n\}$ with x_i in $\mathbb{R}^d$
(2) Similarity metric (e.g., Gaussian similarity, cosine similarity)
and connectivity parameters (e.g., σ, threshold) **Output:** Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`
Conduct the **first pass** (`1stPass` kernel) that:
- computes similarities and finds nonzeros that satisfy the threshold;
- counts the number of nonzeros per row and stores in `nnzPerRow[]`.
Perform an exclusive scan on `nnzPerRow[]` to obtain `csrRow[]`;
Get nnz from `csrRow[]` and allocate memory for `csrVal[]` and `csrCol[]`;
Conduct the **second pass** (`2ndPass` kernel) that:
1 - recomputes similarities and finds again nonzeros that satisfy the threshold;
2 - stores the values and column indexes of nonzeros into `csrVal[]` & `csrCol[]`, respectively.

For the `1stPass` kernel, we choose to create a 2D grid with 2D blocks of threads. As shown in Fig. 5 (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. Then the number of nonzeros is first accumulated within each block into shared memory using the `atomicAdd_block` operation. 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. 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 ($2^{31} - 1 = 2\,147\,483\,647$) so the calculated number of blocks in y dimension 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. Finally, the first pass of our Algo CSR-1, launching one or several grids, processes the entire $n \times n$ similarity matrix as long as this matrix and the n input data instances fit into the memory of a single GPU. Fortunately, as we store the similarity matrix in CSR format, we can process very large datasets on a single GPU.

For the `2ndPass` kernel, we choose to create a 1D grid with 2D blocks. Several points need to be noted: (1) As shown in Fig. 5b, each block of threads processes some rows of the similarity matrix in an iterative fashion (illustrated by arrows), 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. Then only the threads in the first column of each block copy the nonzeros from shared to global memory. (3) Particularly, when testing whether an element in shared memory is nonzero or not, 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. (4) 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 but regard it as in y dimension.

For the exclusive scan step, we leverage the easy-to-use `exclusive_scan` API of NVIDIA's Thrust library [29].

4.4 Algo CSR-2: Ellpack-to-CSR

Algorithm 2 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: (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 restarting 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*), `csrRow[]`, 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 restarting indexes recorded by the first pass.

Algorithm 2 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 (e.g., Gaussian similarity, cosine similarity)
and connectivity parameters (e.g., σ, threshold)
(3) Hypothetical maximum number of nonzeros in a row: $hypoMaxNnzRow$
Output: Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`
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;
Conduct a **full pass** (`fullPass` kernel) across the similarity matrix that:
- computes similarities and finds nonzeros that satisfy the 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 restarting indexes that may be used for a supplementary pass for storing remaining nonzeros.
Find $maxNnzRow$ from `nnzPerRow[]`;
Perform an exclusive scan on `nnzPerRow[]` to obtain `csrRow[]`;
Get nnz from `csrRow[]` and allocate memory for `csrVal[]` and `csrCol[]`;
Launch the `ellpackToCSR` kernel to fill `csrVal[]` and `csrCol[]` with accumulated nonzeros stored in `csrValMax[]` and `csrColMax[]`;
if $maxNnzRow > hypoMaxNnzRow$ **then**
Conduct a **supplementary pass** (`supPass` kernel) across a part of similarity matrix that:
- recomputes similarities from restarting indexes and finds remaining nonzeros that satisfy the threshold;
- completes `csrVal[]` and `csrCol[]` by storing the values and column indexes of remaining nonzeros.
end

For each kernel, we choose to create a 1D grid with 2D blocks, as shown in Fig 5b. Like the `2ndPass` kernel of Algo CSR-1, the points (1)(3)(4) also apply to the kernels of Algo CSR-2.

For the `fullPass` kernel (shown in Listing 1), we declare several shared memory arrays for storing similarities in dense format, storing nonzeros in Ellpack format, and some other uses (line 7). 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 (lines 11–27). 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 (lines 34–41). Meanwhile these threads also record the restarting column indexes (aligned to multiples of 32 memory words for performance concern) and corresponding restarting nonzero element indexes in each row in case the number of nonzeros per row exceeds *hypoMaxNnzRow* (lines 35 & 45). 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 (lines 23, 31 & 49). This helps to reduce warp divergence. Similarly, we set a flag once the *hypoMaxNnzRow* is reached so as to avoid unnecessary operations (lines 33 & 42). Additionally, the number of nonzeros per iteration is accumulated into the number of nonzeros per row.

Listing 1: `fullPass` kernel of Algo CSR-2

```
__global__ void fullPass (...)
{
  int row = blockDim.y*blockIdx.x + threadIdx.y;
  int col = threadIdx.x; int maxCol = ((n-1)/blockDim.x +1)*blockDim.x;
  int flagReachHypo = 0;  int idxNzRowRestart = 0;  int colRestart = n;
  int yofs = threadIdx.y*blockDim.x;  int ymofs = threadIdx.y*hypo;
  ... // Declare & initialize shared memory arrays (dynamic allocation)

  // Each block processes some rows in an iterative fashion
  while (col < maxCol && row < n) {
    if (col < n) {
      // Use Gaussian similarity with threshold for squared distance
      #ifdef GAUSS_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;      }
        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 metrics (\textcolor{red}{e.g.,} cosine
    ) with threshold
    } // End of if (col < n)
    __syncthreads();

    // Copy nonzeros to shared Ellpack arrays & records restarting idx
    if (shNnzIter[threadIdx.y] > 0 && threadIdx.x == 0) {
      int i = 0, idxNzRow = shNnzRow[threadIdx.y];
      if (flagReachHypo == 0) {
        for (; i < blockDim.x && idxNzRow < hypo && 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 == hypo)  flagReachHypo = 1;
      } // End of if (flagReachHypo == 0)
      for (; idxNzRow == hypo && i < blockDim.x && col + i < n; i++) {
        if (i%32==0) {idxNzRowRestart = hypo; 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

  // Update restarting indexes
  if (threadIdx.x == 0 && row < n) {
    if (shNnzRow[threadIdx.y] <= hypo)
      {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 (...) {
    GPU_csrValMax[..]=shNzValMax[..]; GPU_csrColMax[..]=shNzColMax[..];
    col += blockDim.x; }

  // Store nnz per row and restarting indexes into global memory
  if (threadIdx.x == 0 && row < n) {
    GPU_nnzRow[row] = shNnzRow[threadIdx.y];
    GPU_idxNzRowRestart[row] = idxNzRowRestart;
    GPU_colRestart[row] = colRestart; }
}
```

After finishing the outermost loop, a fraction of threads update the restarting indexes in case the number of nonzeros in a row is no more than *hypoMaxNnzRow* (Listing 1, lines 57–61). Now since the nonzeros are contiguously stored in shared Ellpack arrays, each block of threads can parallelly and iteratively copy the nonzeros into global Ellpack arrays with coalescence (lines 66–69). Finally, a fraction of threads store the number of nonzeros per row and the restarting indexes into global memory arrays (lines 72–75). By the way, we point out that for all the kernels in this paper, we ensure most of the global memory accesses are coalesced (e.g., line 16), and 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 [28].

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

The `supPass` kernel 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 restarting indexes recorded before 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 the easy-to-use `exclusive_scan` API of NVIDIA's Thrust library to implement the scan step.

4.5 Algo CSR-3: Chunkwise Dense-to-CSR

Algorithm 3 describes the construction of the CSR format similarity matrix based on a chunkwise dense-to-CSR approach. As mentioned in Sect. 4.2, 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.

For each chunk of the similarity matrix, we launch a typical kernel called `chkPass` to construct the matrix chunk in dense format (according to the grid and block configuration shown in Fig 5a) and we leverage the

Table 1 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*
Number 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 implement-ation	`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)

`cusparseDenseToSparse_xxx` functions of NVIDIA's GPU-accelerated cuSPARSE library [27] to convert it into CSR format. Note that the chunks should be constructed and converted one by one in order, so that we can accumulate the number of nonzeros and continuously update `csrRow[]` using the `transform` API of Thrust library. Finally, 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.

Algorithm 3 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 (e.g., Gaussian similarity, cosine similarity)
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:** Similarity matrix in CSR format: `csrVal[]`, `csrRow[]`, `csrCol[]`
Allocate memory for `csrVal[]` and `csrCol[]` according to $spMaxNzPct$;
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;
Determine automatically the number of chunks in the way that only one chunk can fit into the allocated GPU memory;
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.
Perform the *mergeCSR* step to merge the CSR results of all chunks and obtain the CSR format of the whole similarity matrix.

4.6 Comparison of the Three Algorithms

Table 1 compares in many aspects our three algorithms for constructing the similarity matrix in CSR format on a single 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 Sect. 6.3, 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., *maxNnzRow*, *nnz*, sparsity) before running any faster algorithms, then Algo CSR-1 is the very choice.

According to our experiments, we think hundreds of dimensions may be regarded as *large d*, and millions of instances may be regarded as *large n* in the context of our algorithm selection guide.

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 Sect. 4.2) 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.

5 Spectral Graph Partitioning using nvGRAPH

With the CSR format similarity matrix constructed in Sect. 4, the remaining steps of spectral clustering can be completed on a single 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 second algorithm, but it utilizes the LOBPCG eigensolver to handle the constructed Laplacian matrix.

Compared to Lanczos method, LOBPCG can handle eigenvalues with multiplicity [22] (which often happens in spectral clustering). Moreover, the NVIDIA implementation of LOBPCG is able to restart the computation when it encounters

numerical instabilities. Thus the 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 Sect. 2 and Fig. 2). Note that the nvGRAPH documentation [25] 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 constructs the Laplacian matrix and finds its smallest eigenpairs).

Listing 2: Usage of the `nvGRAPH` spectral graph partitioning API

```
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 the 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
```

Listing 2 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 first k_c eigenpairs (lines 25–26).

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.

6 Experiments and Discussion

6.1 Experimental Framework

In this section we experiment and evaluate our GPU implementation for spectral clustering.[8] It mainly concerns the GPU algorithms for constructing the similarity graph/matrix in CSR format and the use of the nvGRAPH library for graph partitioning.

6.1.1 CSR Format Similarity Graph/Matrix Construction

Apart from the GPU algorithms and implementations for CSR graph/matrix construction, we have also developed a well optimized parallel CPU implementation related to Algorithm 1 as a baseline for performance comparison[9] (we did not find any other established baseline for valid comparison). It is parallelized with OpenMP for multi-threaded execution. In order to be efficient, we have implemented a single large parallel region, so that threads are activated then deactivated only once during the algorithm execution. In this unique region, we have parallelized external loops of heavy and quasi regular loop nests with and without OpenMP's `reduction`. Moreover, internal loops have been designed to facilitate auto-vectorization with gcc for AVX units. To differentiate each implementation of CSR matrix construction, we call them "CPU CSR-1", "GPU CSR-1", "GPU CSR-2" and "GPU CSR-3" in this section.

So, we will compare our GPU and CPU optimized implementations for the CSR matrix construction step.

6.1.2 Spectral Graph Partitioning using LOBPCG Eigensolver

We take advantage of nvGRAPH's LOBPCG-embedded algorithm for the graph partitioning step on GPU, as explained in Sect. 5. We tried to compare the performance of nvGRAPH's LOBPCG eigensolver with MAGMA's LOBPCG eigensolver since they are both GPU implementations of LOBPCG. However, the former is refined to specifically address the Laplacian (generalized) eigenvalue problem [22] and is encapsulated in the spectral graph partitioning/clustering API of nvGRAPH, while the latter is a generic GPU implementation of LOBPCG without necessary adaptation for spectral clustering. Therefore, we failed to run MAGMA's LOBPCG

[8] Our code is available on https://github.com/guanlin-he/clustering-release.

[9] The baseline code is available on https://github.com/guanlin-he/clustering-release/blob/main/modules/spectral_clustering/constr_sim_matrix_on_cpu.cc.

Table 2 Datasets and parameter settings of our benchmarks

Dataset	(n, d, k_c)	Similarity metric	Threshold	Supposed max. % of nonzeros for Algo 3(%)	Tolerance for eigensolver
MNIST60K	(60 K, 784, 10)	Cosine	0.8 (sim.)	1	0.005
MNIST120K	(120 K, 784, 10)	Cosine	0.8 (sim.)	1	0.005
MNIST240K	(240 K, 784, 10)	Cosine	0.8 0.84*(sim.)	1	0.005
Syn1M	(1 M, 4, 4)	Gaussian($\sigma = 0.01$)	0.0008 (sq. dist.)	0.01	0.001
Syn5M	(5 M, 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 MNIST60K and MNIST120K, while a higher threshold (0.84) is required for MNIST240K. - [1] Although the thresholds for Syn1M and Syn5M appear ×4 greater than those used in our previous paper [15], they have equivalent effects because the former are for the squared distance while the latter are in fact for the squared distance divided by the # of dimensions (4)

eigensolver for spectral clustering (encountered an unexpected "floating point exception" error).

Then, we have compared with *scikit-learn*'s `SpectralClustering` API which is a CPU implementation that encapsulates a LOBPCG eigensolver. Our experimental results presented in [13] 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 ×8 to ×28 when processing 10^4 instances (processing larger datasets would be too time-consuming for *scikit-learn*).

So we did not succeed in establishing a valid comparison against CPU or GPU implementations for the graph partitioning step.

6.1.3 Hardware and Software Configuration

All experiments are performed on a server consisting of two Intel Xeon Silver 4114 processors as CPU with 96 GB RAM, and a NVIDIA GeForce RTX 3090 as GPU. Each CPU processor has the following features: released in 2017, Skylake architecture, 10 physical cores (20 logical cores), 2.20 GHz base frequency, AVX/AVX2/AVX512 support, 13.75 MB L3 cache. In constrast, the GPU has the following features: released in 2020, Ampere architecture, 1.70 GHz max clock rate, 5248 CUDA cores, 24 GB RAM. The CPU and GPU are connected by a PCIe 3.0 x16 bus with a theoretical bandwidth of 16 GB/s and a experimental end-to-end bandwidth close to 12.5 GB/s.

The operating system is Ubuntu 20.04.3. The CPU code is compiled by gcc 9.3.0 with `-Ofast -funroll-loops -march=native` optimization flag

Table 3 Characteristics of the constructed sparse similarity matrices

Dataset	Max nnz in a row	Avg. nnz per row	Total nnz (M)	Sparsity (%)
MNIST60K	2196	251	15.1	99.581
MNIST120K	3310	299	35.9	99.751
MNIST240K	5552	478	114.8	99.801
MNIST240K*	3520	199	47.8	99.917
Syn1M	54	23	23.4	99.998
Syn5M	64	29	149.9	99.999

and `-fopenmp` flag. The GPU code is compiled by nvcc of CUDA Toolkit 11.5[10] with `-gpu-architecture=sm_86`. Computations are in single precision.

6.2 Datasets and Parameter Settings

For all our experiments we focus on large datasets. Table 2 summarizes the datasets and algorithmic parameter settings used in our experiments. The datasets can be classified into two categories:

- *MNIST-based*. They include MNIST60K, MNIST120K and MNIST240K. The first one is the training set of the well-known MNIST database of handwritten digits,[11] while the last two ones are produced using the InfiMNIST code.[12] They all have 784 dimensions and 10 clusters.
- *Synthetic*. We create two million-scale datasets called Syn1M and Syn5M with our data generator.[13] They all have 4 dimensions and 4 convex clusters.

In the beginning we perform feature scaling for the synthetic datasets to transform every dimension into the range of [0, 1].

We adopt the cosine similarity metric for the MNIST-based data because it is more effective than the Gaussian similarity for high-dimensional data (as introduced in Ina et al. [16] and as we verified empirically). In contrast, we use Gaussian similarity for our low-dimensional synthetic datasets.

We impose a threshold on the similarity value or the squared distance to construct the *ε-neighborhood-like* graph and associated sparse similarity matrix (see explanation in Sect. 4.1). 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

[10] 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 Sect. 3.3).

[11] http://yann.lecun.com/exdb/mnist/.

[12] https://leon.bottou.org/projects/infimnist.

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

Table 4 Optimal block size configuration for our CUDA kernels

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
MNIST60K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (N/A) <2048>	(32, 16)
MNIST120K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
MNIST240K	(32, 16)	(64, 4)	(64, 2) <512> (128, 4) <1024> (128, 1) <2048>	(128, 4) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
MNIST240K*	(32, 16)	(64, 4)	(64, 2) <512> (256, 4) <1024> (128, 1) <2048>	(128, 2) <512> (128, 2) <1024> (128, 2) <2048>	(32, 16)
Syn1M	(32, 8)	(32, 4)	(32, 4) <16> (32, 4) <32> (32, 4) <54>	(128, 1) <16> (128, 1) <32> (N/A) <54>	(32, 16)
Syn5M	(32, 8)	(32, 4)	(32, 4) <16> (32, 4) <32> (32, 4) <64>	(128, 1) <16> (128, 1) <32> (N/A) <64>	(32, 16)

BSX and BSY represent the block size in x and y dimensions, respectively. (N/A) 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

clustering quality for MNIST60K and MNIST120K, but we need a higher threshold (0.84, marked with *) for MNIST240K to achieve similar clustering quality. For Syn1M and Syn5M, 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 Syn1M and 0.001% for Syn5M. 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 2). We simply set the maximal number of iterations to the nvGRAPH default values (defined by NVIDIA), i.e., 4000 for the LOBPCG eigensolver and 200 for the final k-means. However, we found that the approximation

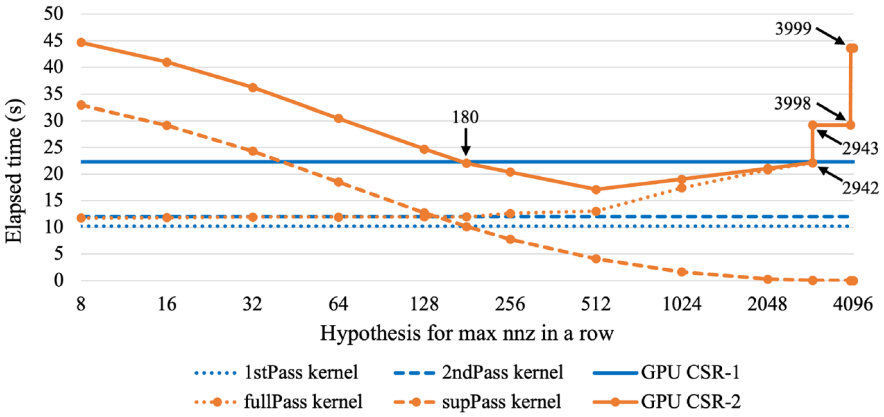


Fig. 6 Performance comparison of GPU CSR-1 vs. GPU CSR-2 on MNIST120K dataset

tolerance for the eigensolver needs to be tuned with care because it has a significant impact on the clustering quality and the execution time. After some experimental tests, we set the tolerance of eigensolver to 0.005 for the MNIST-based datasets, 0.001 for Syn1M and 0.0001 for Syn5M. Besides, the tolerance for the *k*-means algorithm is set to a classical value of 0.0001 for all benchmarks (the result is not very sensitive to this parameter).

6.3 Performance of the Similarity Matrix Construction

Table 3 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.

6.3.1 Tuning of the Grid and Block Configuration

As we all know, the grid and block configuration (i.e., dimension and size) for a CUDA kernel can have a significant impact on the kernel performance. One of the suggestions [28] 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 [15], 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 Sects. 4.3, 4.4 and 4.5).

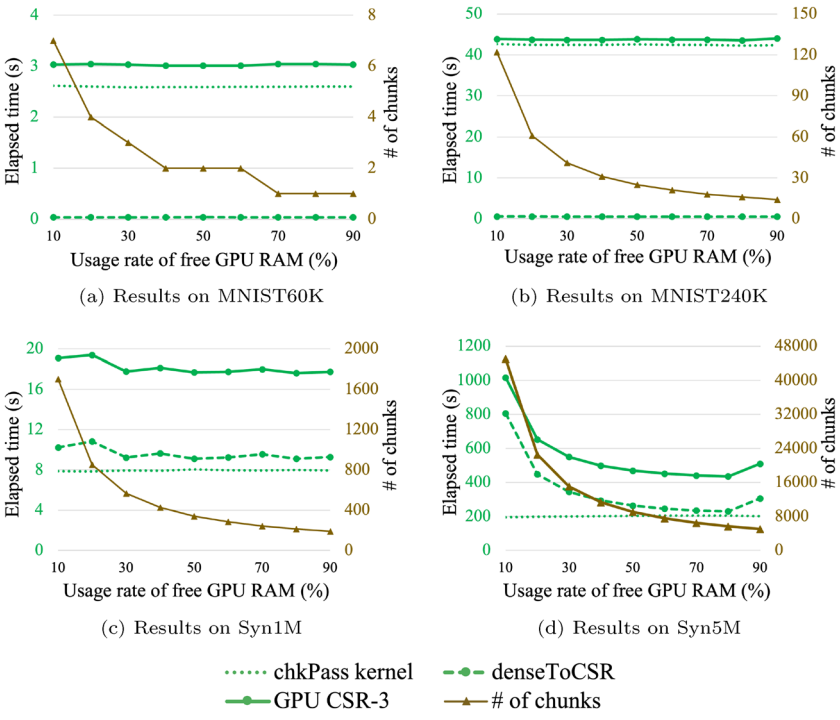


Fig. 7 Impact of free GPU RAM usage rate on the performance of GPU CSR-3 (In each figure, the left axis shows the elapsed time related to green lines, while the right axis shows the number of chunks related to the brown line)

Table 4 shows the optimal block size in x and y dimensions that we found experimentally by trying many possible configurations for each kernel and each benchmark. For each kernel, we tried approximately 10 configurations of grid and blocks, each requiring the modification and recompilation of the source code, and the re-execution of the new binary code, which is time-consuming. Due to space limitation, we do not illustrate here the impact of block size on kernel performance, but essentially, block sizes close to the optimal ones are often sub-optimal choices, i.e., a block size close to the optimal is often a relatively good or acceptable choice, while block sizes far from the optimal ones may lead to about ×2 to ×5 lower kernel performance. It can be observed from the table that on our RTX 3090 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. Particularly, the optimal block size for the `chkPass` kernel of GPU CSR-3 is constant (BSX=32, BSY=16) for all benchmarks. 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 Sect. 4.4 for explanation).

6.3.2 Tuning of the *hypo* Parameter for GPU CSR-2

As shown in Fig. 6, we take the MNIST120K benchmark as an example to study the performance of each kernel of GPU CSR-1 and GPU CSR-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. The `ellpackToCSR` kernel of GPU CSR-2 consumes so little time compared to the other kernels that we have omitted it in the figure. Note that each time in Fig. 6 was measured 3 to 5 times and appeared stable.

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 Fig. 6 up to a hypothesis of 512. However, this kernel allocates shared memory in an amount proportional to the hypothesis, which can lead to an increase in execution time. This phenomenon is suspected to be the cause of the two sudden slowdowns when the hypothesis grows from 2942 to 2943 and from 3998 to 3999 (see Fig. 6).

A Stream Multiprocessor (SM) of GPU may have several resident blocks of threads which can help the scheduler of blocks hide the latency time of global memory accesses. However, each SM has a limited amount of shared memory, L1 cache and registers, and supports a limited number of threads, therefore the number of blocks that can reside simultaneously in a Stream Multiprocessor depends on the amount of shared memory and registers required by each block and the number of threads per block. Moreover, on our RTX 3090 GPU device (Ampere architecture), L1 cache and shared memory are unified in a 128 KB of fast memory. So increasing shared memory would decrease L1 cache and could limit the number of resident blocks. In our code, the amount of shared memory per block depends on the *hypoMaxNnzRow* parameter, and the turning point at 2943 corresponds to crossing the 24 KB per block. However, due to the complexity of SM resource configurations, we were unable to calculate and clearly demonstrate a change in the number of resident blocks or a significant decrease in the amount of L1 cache.[14] Nevertheless, using large values for *hypoMaxNnzRow* requires more and more resources for each block, and introduces major disruptions in our kernel's performance.

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 Fig. 6).

[14] The NVIDIA Nsight Compute tool offers an occupancy calculator that may be helpful towards this, however we failed to employ this tool on our remote testbed due to seriously slow reactions of the graphical user interface.

Table 5 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>	
MNIST60K	1815	146	74	5.53	3.91 <512>	4.49 <1024>	5.42 <2048>	**3.04**
MNIST120K	7427	590	301	22.27	17.12 <512>	19.05 <1024>	21.09 <2048>	**11.09**
MNIST240K	28,293	2502	1266	91.47	77.78 <512>	83.07 <1024>	85.29 <2048>	**43.84**
MNIST240K*	29,520	2650	1244	91.05	62.71 <512>	59.03 <1024>	72.52 <2048>	**43.56**
Syn1M	2845	194	151	14.01	7.71 <16>	5.71 <32>	**5.66** <54>	17.62
Syn5M	too long	5772	3928	363.69	242.69 <16>	176.06 <32>	**145.89** <64>	435.76

The best result (the shortest execution time) in each row is highlighted in bold

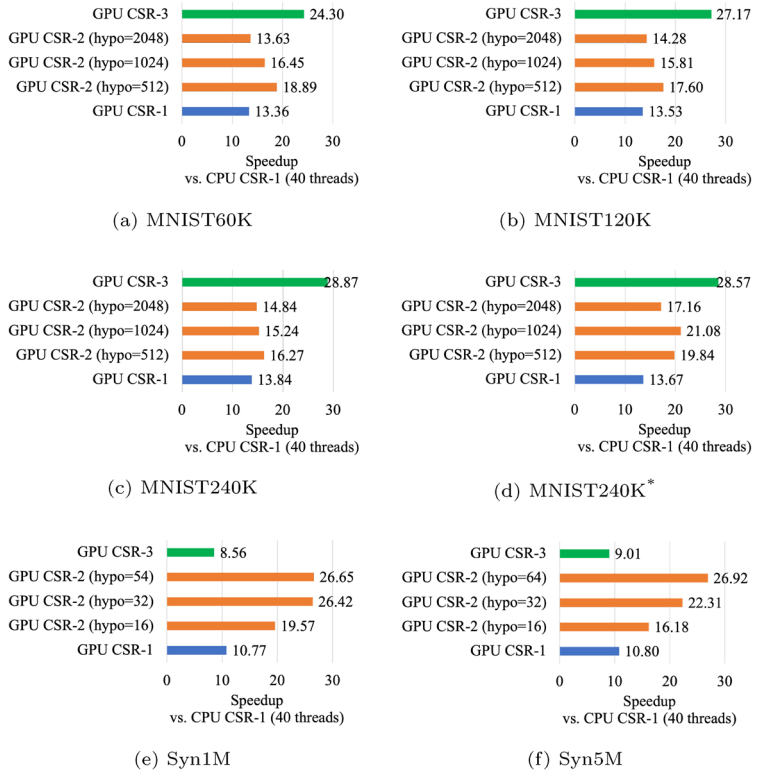


Fig. 8 Speedup of the similarity matrix construction in CSR format on GPU vs. CPU

GPU CSR-2 can therefore run faster but requires some tests to identify the interesting hypothesis range.

6.3.3 Tuning of the Dense Matrix Chunk Size for GPU CSR-3

As presented in Algorithm 3, our Algo CSR-3 runs the `chkPass` kernel and calls routines of cuSPARSE library, both need memory space for data structures to generate a chunk of similarity matrix. A small chunk requires a small part of available memory, leaving a lot of memory to the library, but processing a small chunk in parallel can be inefficient for our kernel. Processing a large chunk leaves a small part of memory for the library which could be slowed down. We experimentally investigated this problem.

The easiest way to tune the chunk size was to set the percent of available GPU memory allocated for the chunk, the rest remaining available for the library. So, Fig. 7 investigates the impact of the percent of free GPU RAM allocated for a chunk of similarity matrix (see explanation in the penultimate paragraph of Sect. 6.2) on the performance breakdown of GPU CSR-3 (green lines) indicated

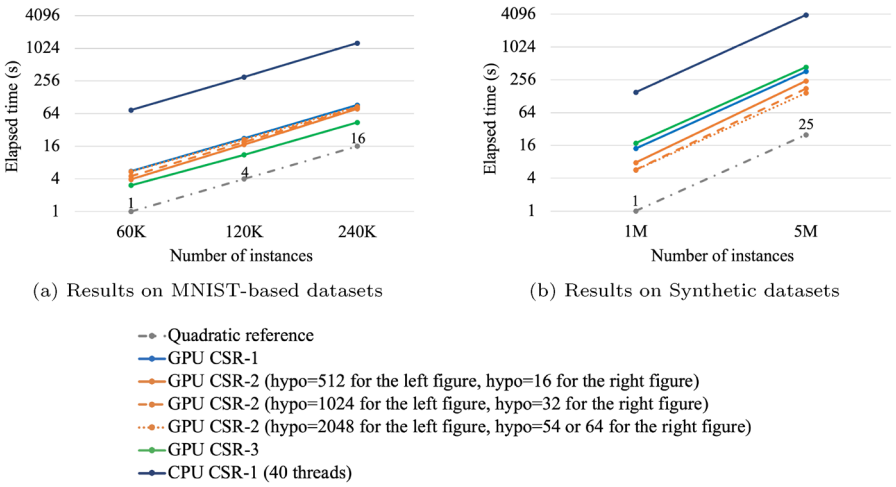


Fig. 9 Scalability of the similarity matrix construction in CSR format

Table 6 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	NMI	Min.	Max.	Average
MNIST60K	0.44	0.66	2.30	3.34	2.88
MNIST120K	0.50	0.67	3.48	4.59	3.95
MNIST240K*	0.56	0.73	4.41	5.90	5.01
Syn1M	1.00	1.00	3.63	5.18	4.08
Syn5M	1.00	1.00	17.67	19.15	18.25

on left y-axis. Again, each time was measured 3 to 5 times and appeared stable. Besides, the number of chunks is presented by a brown line indicated on right y-axis. Note that we consider only the range from 10 to 90% for the free memory usage rate because it is meaningless to process a too small chunk on GPU and it is necessary to leave some memory for cuSPARSE functions. For all benchmarks, the initialization of our algorithm takes only 0.1% of GPU CSR-3's global time and the final *mergeCSR* step costs less than 0.2%, so they are not presented in the figure. According to Fig. 7, it can be seen from the green lines that:

- 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 Syn1M and Syn5M, the `chkPass` kernel performance is also very stable as the memory usage rate changes. However, the execution time of the *denseToCSR*

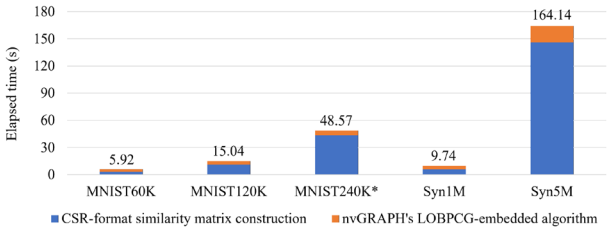


Fig. 10 Global performance of spectral clustering on the GPU

step becomes significant and less stable on Syn1M and makes the time of GPU CSR-3 less stable. This instability becomes more severe on Syn5M. Specifically, the performance deteriorates when the memory usage rate decreases under 30% for Syn1M, and the memory usage rate resulting in optimal performance is 80% for Syn5M. The performance deterioration in case of decreasing memory usage rate (which leads to a growing number of chunks as shown by brown lines) appears in the *denseToCSR* step implemented by calling cuSPARSE functions (including matrix descriptor creations and matrix conversions) which are performed once for each chunk.

In any case, a free memory usage rate of 80% for storing a matrix chunk appears the best choice on our GPU device.

6.3.4 GPU vs. CPU Performance Comparison

Table 5 compares the performance of CPU and GPU algorithms on different datasets. Each result on GPU is the average time of 5 consecutive runs with low fluctuations, while each result on CPU is the rounded time of a single run as it takes much longer (which also showed slight fluctuations whenever we checked it). 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 h) on the Syn5M dataset so we did not get the final time. Figure 8 visualizes the speedup of GPU algorithms versus the best performance of CPU CSR-1 (using 40 threads and auto-vectorization). Note that the speedups are based on the unrounded time of CPU CSR-1, thus slightly different from those based on the rounded ones presented in Table 5.

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

With the chosen values for *hypoMaxNnzRow*, GPU CSR-2 can outperform GPU CSR-1 and this superiority is especially significant on MNIST240K*, Syn1M and Syn5M benchmarks. This is because the gain from reducing similarity computations surpasses the cost of recording restarting indexes for GPU CSR-2 (see Sect. 4.4).

Compared to the other GPU algorithms, GPU CSR-3 is around ×2 faster on the MNIST-based datasets but is significantly slower on Syn1M and Syn5M. 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 similarity computations, GPU CSR-3 achieves a speedup. However, on Syn1M and Syn5M datasets which have only 4 dimensions and million-scale n, it reaches less performance than the other GPU algorithms.

To reveal a possible scalability phenomenon, we plot the $T(n)$ curve in logarithmic scales (on both axes T and n) based on the results in Table 5, where T denotes the elapsed time and n denotes the number of data instances. We obtain the scalability graph of Fig. 9 showing 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 $\mathcal{O}(n^2 d)$ 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.

6.4 Performance of nvGRAPH's LOBPCG-Embedded Algorithm

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 Sect. 5). Table 6 presents the elapsed time of the nvGRAPH algorithm and the final clustering quality measured by two commonly used metrics: Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI) [37]. Both 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 and NMI scores achieved by our spectral clustering implementation are around 0.5 and 0.7 respectively. The NMI score is close to that obtained in Yang et al. [40] by spectral clustering algorithm and is better than that obtained by the k-means algorithm. The clustering quality on Syn1M and Syn5M (formed by convex clusters) is perfect.

For all benchmarks we observed a certain degree of performance fluctuations. Although the theoretical complexity of eigenvector computation is $\mathcal{O}(n^3)$ in the worst case, our experiments exhibit a low complexity close to $\mathcal{O}(log(n))$ on the MNIST-based datasets and close to $\mathcal{O}(n)$ on Syn1M and Syn5M 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 s with CUDA 11.5, and it is not included in the performance measurements.

6.5 Global Performance of Spectral Clustering

Figure 10 presents the global performance of spectral clustering on the GPU, consisting in the performance of the best algorithm for CSR-format similarity matrix construction 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 MNIST120K, MNIST240K* and Syn5M, mainly due to its $\mathcal{O}(n^2 d)$ time complexity. The elapsed time consumed by the LOBPCG-embedded algorithm appears to take the second place.

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. Our GPU server includes a bus PCIe 3.0×16 with a theoretical bandwidth of 16 GB/s but our experimental measurements always show a bandwidth close to 12.5 GB/s. Our largest benchmark (MNIST240K) requires to transfer 0.718 GB of input data and output results, leading to a communication time close to 0.06 s plus the time of locking and unlocking related CPU memory (due to our implementation). The data allocations and deallocations are performed in the beginning and the end of our program or during computation steps for the purpose of calculations, so their time is not counted in the transfer time. Finally, in all our benchmarks, our complete transfer time always remained less than 0.20 s. Compared to the computation time which ranges from several seconds to hundreds of seconds, the transfer time is thus negligible and is 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.

7 Conclusion and Perspectives

In this paper we have addressed the scalability of spectral clustering on single-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, without storing the entire matrix in dense format, enables us to save a large amount of memory space compared to the dense format storage, which is crucial on a single 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).

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 eigensolvers (including Lanczos and LOBPCG) and the 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 eigenvalues with multiplicity and 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 and in running really faster than an optimized CPU code on a dual CPU server.

Since the nvGRAPH library (which we rely on for its LOBPCG eigensolver) is no longer being actively developed by NVIDIA after 2019, it may be necessary to find some alternative sparse eigensolvers (especially LOBPCG) that are optimized for the GPU. For instance, the AmgX library [23] contains multiple GPU-accelerated eigensolvers including the LOBPCG solver, and the ViennaCL library [32] also includes some GPU-accelerated eigensolvers. However, their effectiveness of being used for spectral clustering remains to be tested in the future.

Using a LOBPCG eigensolver based on the Compressed Sparse Blocks (CSB) matrix storage format could be also an interesting optimization, since this format stores a collection of sparse blocks, compatible with a cache-blocking computation strategy [1]. This strategy could be very efficient on multi-core CPU servers, and enhance our CPU implementation. However, a LOBPCG solver using CSB format and adapted to spectral clustering would remain to be implemented.

Globally, it would be worth comparing against other GPU libraries that neither require nor support CSR data format for spectral clustering, and show how the proposed work in this paper advances them, either in wall-clock time or required memory. However, we could not find such libraries for valid comparison.

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. Our CSR algorithms for similarity matrix construction could be adapted to multi-GPU architectures by going through some consecutive rows of the similarity matrix on each GPU. However, the LOBPCG-embedded algorithm of the nvGRAPH library on which we rely does not have a multi-GPU version yet, but multi-GPU versions of other sparse Top-K eigensolvers could be tested (e.g., [34]). The data transfers between multiple GPUs could be achieved using the NVIDIA Collective Communication Library (NCCL). Another approach to addressing larger datasets would be CPU-GPU algorithms incorporating the *representative* extraction technique on CPU to reduce the number of data instances that need to be processed on GPU [13, 39].

Acknowledgements 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. The authors sincerely thank the reviewers for their valuable comments which helped a lot to improve this paper.

References

1. Aktulga, H.M., Afibuzzaman, M., Williams, S., et al.: A high performance block Eigensolver for nuclear configuration interaction calculations. IEEE Trans. Parallel Distrib. Syst. **28**(6), 1550–1563 (2017)
2. Anastasiu, D.C., Karypis, G.: 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, pp. 791–800, (2015)
3. Anastasiu, D.C., Karypis, G.: Parallel cosine nearest neighbor graph construction. J. Parallel Distrib. Comput. **129**, 61–82 (2019)
4. 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)
5. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation (2008)
6. Domingos, P.: A few useful things to know about machine learning. Commun. ACM **55**(10), 78–87 (2012)
7. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proceedings of the 20th International Conference on World Wide Web, pp. 577–586 (2011)
8. Fender, A.: Parallel solutions for large-scale eigenvalue problems arising in graph analytics. Ph.D. Thesis, Université Paris-Saclay (2017)
9. Fender, A., Emad, N., et al.: Accelerated hybrid approach for spectral problems arising in graph analytics. Proc. Comput. Sci. **80**, 2338–2347 (2016)
10. Gao, J., Qi, P., He, G., et al.: Efficient CSR-based sparse matrix-vector multiplication on GPU. Math. Probl. Eng. (2016)
11. Greathouse, J.L., Daga, M.: Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, pp. 769–780, (2014)
12. Hajebi, K., Abbasi-Yadkori, Y., Shahbazi, H., et al.: Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)
13. He, G.: Parallel algorithms for clustering large datasets on CPU-GPU heterogeneous architectures. Theses, Université Paris-Saclay, https://theses.hal.science/tel-04114475 (2022)
14. He, G., Vialle, S., Baboulin, M.: Parallel and accurate k-means algorithm on CPU–GPU architectures for spectral clustering. Concurr. Comput. Pract. Exp. **34**(14), e6621 (2022)
15. He, G., Vialle, S., Sylvestre, N., et al.: Scalable algorithms using sparse storage for parallel spectral clustering on GPU. In: Cérin, C., Qian, D., Gaudiot, J.L., et al. (eds.) Network and Parallel Computing, pp. 40–52. Springer, Cham (2022)
16. Ina, T., Hashimoto, A., Iiyama, M., et al.: Outlier cluster formation in spectral clustering. arXiv preprint arXiv:1703.01028 (2017)
17. Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern Recognit. Lett. **31**(8), 651–666 (2010)
18. 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)
19. Knyazev, A.V.: Toward the optimal preconditioned Eigensolver: locally optimal block preconditioned conjugate gradient method. SIAM J. Sci. Comput. **23**(2), 517–541 (2001)
20. Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. 17(4) (2007)
21. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297, (1967)
22. Naumov, M., Moon, T.: Parallel spectral graph partitioning. Technical report, NVIDIA Technical Report, NVR-2016-001 (2016)
23. Naumov, M., Arsaev, M., Castonguay, P., et al.: AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. SIAM J. Sci. Comput. **37**(5), S602–S626 (2015)
24. Ng, A.Y., Jordan, M.I., Weiss, Y.: 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], pp. 849–856, (2001)
25. NVIDIA: NVGRAPH Library User's Guide (DU-08010-001_v10.2). https://docs.nvidia.com/pdf/nvGRAPH_Library.pdf (2019)
26. NVIDIA: cuSOLVER Library (DU-06709-001_v12.0). https://docs.nvidia.com/cuda/cusolver/index.html (2022a)
27. NVIDIA: cuSPARSE Library (DU-06709-001_v12.0). https://docs.nvidia.com/cuda/cusparse/index.html (2022b)
28. NVIDIA: CUDA C++ Best Practices Guide (Release 12.2). https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2023a)
29. NVIDIA: Thrust (Release 12.2). https://docs.nvidia.com/cuda/thrust/index.html (2023b)
30. NVIDIA: NVPL Storage Formats. https://docs.nvidia.com/nvpl/_static/sparse/storage_format/sparse_matrix.html (2024)
31. RAPIDS Development Team: RAPIDS: Libraries for End to End GPU Data Science. https://rapids.ai (2023)
32. Rupp, K., Tillet, P., Rudolf, F., et al.: ViennaC—linear algebra library for multi-and many-core architectures. SIAM J. Sci. Comput. **38**(5), S412–S439 (2016)
33. Saad, Y.: Numerical methods for large eigenvalue problems: Revised edition. SIAM (2011)
34. Sgherzi, F., Parravicini, A., Santambrogio, M.D.: A mixed precision, multi-GPU design for large-scale Top-K sparse eigenproblems. In: 2022 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1259–1263, (2022)
35. 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)
36. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5 &6), 232–240 (2010)
37. 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)
38. Xiang, T., Gong, S.: Spectral clustering with eigenvector selection. Pattern Recognit. **41**(3), 1012–1029 (2008)
39. 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)
40. Yang, X., Deng, C., Zheng, F., et al.: Deep spectral clustering using dual autoencoder network. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4066–4075, (2019)
41. Zelnik-Manor, L., Perona, P.: Self-tuning spectral clustering. In: Advances in Neural Information Processing Systems 17 (NIPS 2004), December 13–18, 2004, Vancouver, Canada, pp. 1601–1608, (2004)
42. Zheng, J., Chen, W., Chen, Y., et al.: Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In: 2008 13th Asia-Pacific Computer Systems Architecture Conference, IEEE, pp. 1–8 (2008)