A High Performance Implementation of Spectral Clustering on CPU-GPU Platforms

Yu Jin
Institute for Advanced Computer Studies
Department of Electrical and Computer Engineering
University of Maryland, College Park, USA
Email: yuj@umd.edu

Joseph F. JaJa
Institute for Advanced Computer Studies
Department of Electrical and Computer Engineering
University of Maryland, College Park, USA
Email: joseph@umiacs.umd.edu

***Abstract*—Spectral clustering is one of the most popular graph clustering algorithms, which achieves the best performance for many scientific and engineering applications. However, existing implementations in commonly used software platforms such as Matlab and Python do not scale well for many of the emerging Big Data applications. In this paper, we present a fast implementation of the spectral clustering algorithm on a CPU-GPU heterogeneous platform. Our implementation takes advantage of the computational power of the multi-core CPU and the massive multithreading and SIMD capabilities of GPUs. Given the input as data points in high dimensional space, we propose a parallel scheme to build a sparse similarity graph represented in a standard sparse representation format. Then we compute the smallest k eigenvectors of the Laplacian matrix by utilizing the reverse communication interfaces of ARPACK software and cuSPARSE library, where k is typically very large. Moreover, we implement a very fast parallelized k-means algorithm on GPUs. Our implementation is shown to be significantly faster compared to the best known Matlab and Python implementations for each step. In addition, our algorithm scales to problems with a very large number of clusters.**

***Keywords*-CPU-GPU platform; spectral clustering; sparse similarity graph;reverse communication interface; k-means clustering**

I. Introduction

Spectral clustering algorithm has recently gained popularity in handling many graph clustering tasks such as those reported in [1, 2, 3]. Compared to traditional clustering algorithms, such as k-means clustering and hierarchical clustering, spectral clustering has a very well formulated mathematical framework and is able to discover non-convex regions which may not be detected by other clustering algorithms. Moreover, spectral clustering can be conveniently implemented by linear algebra operations using popular scientific software environments such as Matlab and Python. Most of the available software implementations are built upon CPU-optimized Basic Linear Algebra Subprograms (BLAS), usually accelerated using multi-thread programming. However, such implementations scale poorly as the problem size or the number of clusters grow very large. Recent results show that GPU accelerated BLAS significantly outperforms multi-threaded BLAS libraries such as the Intel MKL package, LAPACK and Goto BLAS [4, 5]. Moreover, hybrid computing environments, which collaboratively combine the computational advantages of GPUs and CPUs, further boost the overall performance and are able to achieve very high performance on problems whose sizes grow up to the capacity of CPU memory [6, 7, 8, 9, 10, 11]. In this paper, we present a hybrid implementation of the spectral clustering algorithm which significantly outperforms the known implementations, most of which are purely based on multi-core CPUs.

There have been reported efforts on parallelizing the spectral clustering algorithm. Zheng et al. [12] presented both CUDA and OpenMP implementations of spectral clustering. However, the implementation was targeted for a much smaller data size than the work in this paper, and moreover, their implementation achieve a relatively limited speedup. Matam et al. [13] implemented a special case of spectral clustering, namely the spectral bisection algorithm, which was shown to achieve high speed-ups compared to Matlab and Intel MKL implementations. Chen et al. [14, 15] implemented the spectral clustering algorithm on a distributed environment using Message Passing Interface (MPI), which is targeted for problems whose sizes that could not fit in the memory of a single machine. Tsironis and Sozio [16] proposed an implementation of spectral clustering based on MapReduce. Both implementations were targeted for clusters, and involve frequent data communications which will clearly constrain the overall performance.

In this paper, we present a hybrid implementation of spectral clustering on a CPU-GPU heterogeneous platform which significantly outperforms all the best implementations we are aware of, which are based on existing parallel platforms. We highlight the main contributions of our paper as follows:

- Our algorithm is the first work to comprehensively explore the hybrid implementation of spectral clustering algorithm on CPU-GPU platforms.
- Our implementation makes use of sparse representation of the corresponding graphs and can handle extremely large input sizes and generate a very large number of clusters.
- The hybrid implementation is highly efficient and is shown to make a very good use of available resources.

- Our experimental results show superior performance relative to the common scientific software implementations on multicore CPUs.

The rest of the paper is organized as follows. Section II gives an overview of the spectral clustering algorithm, while describing the important steps in some detail. Section III describes the operating environment and the necessary software dependencies. Section IV provides a description of our parallel implementation, while Section V evaluates the performance of our algorithm with a comparison with Matlab and Python implementations on both synthetic and real-world datasets.

II. Overview of Spectral Clustering Algorithm

Spectral clustering was first introduced in 1973 to study the graph partition problem [17]. Later, the algorithm was extended in [18, 19], and generalized to a wide range of applications, such as computational biology [20, 21], medical image analysis [2, 3], social networks [22, 23] and information retrieval [24, 25]. A standard procedure of the spectral clustering algorithm to compute k clusters is described next [26],

- Step 1: Given a set of data points $x_1, x_2, ..., x_n \in \mathbb{R}^d$ and some similarity measure $s(x_i, x_j)$, construct a sparse similarity matrix W that captures the significant similarities between the pairs of points.
- Step 2: Compute the normalized graph Laplacian matrix as $L_n = D^{-1}L$ where L is the unnormalized graph Laplacian matrix defined as $L = D - W$ and D is the diagonal matrix with each element $D_{i,i} = \sum_{j=1}^{n} W_{i,j}$.
- Step 3: Compute the k eigenvectors of the normalized graph Laplacian matrix L_n corresponding to the smallest k nonzero eigenvalues.
- Step 4: Apply the k-means clustering algorithm on the rows of the matrix whose columns are the k eigenvectors to obtain the final clusters.

Given the similarity graph defined by the similarity matrix W, the basic idea behind spectral clustering is to partition the graph into k partitions such that some measure of the cut between the partitions is minimized. The traditional graph cut is defined as follows:

$$\text{Cut}(A_1, A_2, ..., A_k) = \frac{1}{2}\sum_{i=1}^{k} W(A_i, \bar{A}_i); \tag{1}$$

$$W(A, \bar{A}) := \sum_{i\in A, j\in \bar{A}} w_{ij} \tag{2}$$

To ensure that the each partition represents a meaningful cluster of reasonable size, two alternative cut measures are often used, namely **RatioCut** and normalized cut **Ncut**. Note that we use below $|A_i|$ as the number of nodes in A and $vol(A)$ as the sum of the degrees of all the nodes in A.

Table I. CPU and GPU specifics

CPU Model	Intel Xeon E5-2690
CPU Cores	8
DRAM Size	128GB
GPU Model	Tesla K20c
Device Memory Size	5GB GDDR5
SMs and SPs	13 and 192
Compute Capability	3.5
CUDA SDK	7.5
PCIe Bus	PCIe x16 Gen2

$$\text{RatioCut}(A_1, A_2, A_k) = \frac{1}{2}\sum_{i=1}^{k} \frac{W(A_i, \bar{A}_i)}{|A_i|}; \tag{3}$$

$$\text{Ncut}(A_1, A_2, A_k) = \frac{1}{2}\sum_{i=1}^{k} \frac{W(A_i, \bar{A}_i)}{vol(A_i)}; \tag{4}$$

In our implementation, we focus on the problem of minimizing the **Ncut** which has an equivalent algebraic formulation as defined next.

$$\min_{H} \operatorname{trace}(H'LH) \text{ subject to } H'DH = I \tag{5}$$

That is, we need to determine a matrix $H \in \mathbb{R}^{n\times k}$ whose columns are indicator vectors, which minimizes the objective function introduced above.

Since this problem is NP-hard, we relax the discrete constraints on H are removed, thereby allowing H to be any matrix in $\mathbb{R}^{n\times k}$. Note that there is no theoretical guarantee on the quality of the solution of the relaxed problem compared to the exact solution of the discrete version. It turns out that the relaxed problem is a well-known trace minimization problem, which can be exactly solved by taking H as the eigenvectors with the smallest k eigenvalues of the matrix $L_n = D^{-1}L$ or equivalently the k generalized eigenvectors corresponding to the smallest k eigenvalues of $Lx = \lambda Dx$. The k-means clustering is then applied on the rows of H to obtain the desired clustering.

The algorithm described above begins with a set of d-dimensional data points and builds the similarity graph explicitly from the pair-wise similarity metric. The similarity graph is usually stored in a sparse matrix representation, which often reduces the memory requirement and computational cost to linear instead of quadratic. For the general graph clustering whose input is specified as a graph, our spectral clustering algorithm starts directly in Step 2. Otherwise, we build our sparse graph representation from the given set of data points.

III. Environment Setup

A. The Heterogeneous System

The CPU-GPU heterogeneous system used in our implementation is specified in Table I.

The CPU and the GPU communicate through the PCIe bus whose theoretical peak bandwidth is 8 GB/s. The cost of data communication can be quite significant for large-scale problems. To achieve the best overall performance, our implementation leverages the GPU to compute the most computationally expensive part while minimizing the data transfer between the host and the device.

B. CUDA Platform

CUDA is a general-purpose multithreaded programming model that leverages the large number of GPU cores to solve complex data parallel problems. The CUDA programming model assumes a heterogeneous system with a host CPU and several GPUs as co-processors. Each GPU has an array of Streaming Multiprocessors (**SM**), each of which has a number of Streaming Processors (**SP**) that execute instructions concurrently. The parallel computation on GPU is invoked by calling customized kernel functions using thousands of threads. The kernel function is executed by blocks of threads independently. Each block of threads can be scheduled on any Streaming Multiprocessors (SP) as shown in Figure 1. The kernel function takes as parameters the number of blocks and the number of threads within a block.

In addition, NVIDIA provides efficient BLAS libraries for both sparse[1] and dense[2] matrix computations. Our implementation relies on the Thrust library, which resembles the C++ Standard Template Library (STL) that provides efficient operations such as sort, transform, which greatly improves productivity.

C. ARPACK Software

ARPACK is a software package designed to solve large-scale eigenvalue problems [27]. ARPACK is reliable and achieves high accuracy, and is widely used in modern scientific software environments. It contains highly optimized Fortran subroutines that are able to solve symmetric, non-symmetric and generalized eigenproblems. ARPACK is based on the Implicitly Restarted Arnoldi Method (IRAM) with non-trivial numerical optimization techniques [28, 29]. In our implementation, we adopt ARPACK++ [3] that provides C++ interfaces to the original ARPACK Fortran packages and utilizes efficient matrix solver libraries such as LAPACK, SuperLU. The eigenvalue problem is efficiently solved by collaboratively combining the interfaces of ARPACK++ and cuSPARSE library.

D. OpenBLAS

OpenBLAS[4] is an open-source CPU-based BLAS library utilized by ARPACK++. It supports multi-threaded acceleration through pthread programming or OpenMP by specifying corresponding environment variables. OpenBLAS is a highly optimized BLAS library developed based on GotoBLAS2, which has been shown to surpass other CPU-based BLAS libraries [4].

[1]http://docs.nvidia.com/cuda/cusparse/
[2]http://docs.nvidia.com/cuda/cublas/
[3]http://reuter.mit.edu/software/arpackpatch/
[4]http://www.openblas.net/

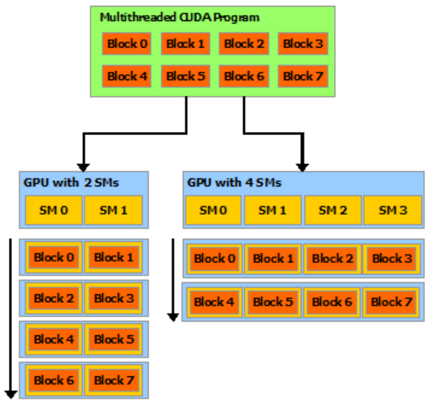

Figure 1: CUDA Program Model

IV. Implementation

A. Data Preprocessing

Given the d-dimensional data points, the preprocessing step constructs the similarity matrix from the data points. The clustering problem is reformulated as a graph clustering where the graph is represented by the similarity matrix.

As mentioned before, the similarity matrix is usually constructed to be sparse, which reduces the memory requirement and enables high computational efficiency. The sparsity patterns of the similarity matrices are highly dependent on the specific application. The following are several common ways to construct a sparse similarity matrix [26].

- λ-threshold graph: The similarity graph is constructed where data points are connected if their similarity measure is above the threshold λ.
- ε-distance graph: The similarity matrix is construct by only connecting data points that are within a spatial distance ε.
- k-nearest-neighbor graph: The similarity graph is constructed where two data points x_i and x_j are connected only if either x_i is among the k most similar data points of x_j, or x_j is among the k most similar data points of x_i. Note that the parameter k is unrelated to the number k of clusters used in the next section.

The notion of the similarity measure between data points also varies depending on the application. Typical measures are the following.

- Cosine Similarity Measure

$$\text{CosineDist}(x_i, x_j) = \frac{\langle x_i, x_j \rangle}{\|x_i\|_2 \|x_j\|_2} \tag{6}$$

Algorithm 1 Construction of Sparse Similarity Matrix

```
1. Transfer the input data $X$ and edge lists $E$ from CPU to GPU.
2. Initialize $n$-length vectors $X_{average}$ and $X_{norm}$ on GPU.
3. Initialize $nnz$-length vector $val$ on GPU.
4. Execute kernel function compute_average where each thread $i$ computes $X_{average}(i) = \frac{1}{d}\sum_{j=1}^{d} X_{ij}$
5. Execute kernel function update_data where each thread $i$ updates one row of data $X_{ij} = X_{ij} - X_{average}(i)$ and compute $X_{norm}(i) = \sqrt{\sum_{j=1}^{d} X_{ij}^2}$
6. Execute kernel function compute_similarity where each thread $i$ computes the similarity between the $i^{th}$ pair of data points in $E$.
7. The edge list and the vector $val$ form the sparse graph represented in the Coordinate Format (COO) format.
```

- Cross Correlation

$$\text{CrossCorr}(x_i, x_j) = \frac{\langle x_i - \bar{x_i}, x_j - \bar{x_j}\rangle}{\|x_i - \bar{x_i}\|_2 \|x_j - \bar{x_j}\|_2} \tag{7}$$

- Exponential decay function

$$\text{ExpDecay}(x_i, x_j) = e^{\frac{\|x_i - x_j\|_2}{2\sigma^2}} \tag{8}$$

Although the sparse patterns and similarity measures are different depending on the application, the general construction of the similarity matrix can be accelerated under the CUDA programming model regardless of the preprocessing used. Here we provide a parallel implementation for a specific sparsity pattern and similarity measure.

We consider the input data as a matrix $X \in \mathbb{R}^{n \times d}$ where n is the number of data points and d is the dimension of each data point. The goal is to construct a sparse matrix representation of the similarity graph using the ε-distance graph structure and cross correlation as the similarity measure. We assume the neighborhood information is given by a list $E \in \mathbb{R}^{nnz \times 2}$, which contains all pairs of indices of data points that are within ε-distance. The number nnz of such pairs is the number of edges in the graph. The procedure for constructing the sparse similarity matrix represented in Coordinate Format (COO) format is described in Algorithm 1.

The above procedure is highly data parallel and easy to implement under the CUDA programming model. In general, there are two sparse matrix representations that we use in our work.

- Coordinate Format (**COO**): this format is the simplest sparse matrix representation. Essentially, COO uses tuples (i, j, w_{ij}) to represent all the non-zero entries. This can be done through three separate nnz-length arrays that respectively store the row indices, column indices, and the corresponding non-zero matrix values.
- Compressed Sparse Row Format (**CSR**): this consists of three arrays, one containing the non-zero values, the second containing the column indices of the corresponding non-zero values, and the third contains the prefix sums of the number of nonzero entries of the rows.

Algorithm 2 Parallel Computation of $D^{-1}W$

```
1. Initialize a n-length vector $x$ with 1.0 for all elements.
2. Compute the vector $y = Wx$ where each element $y_i = d_{ii}$ by calling cusparseDcsrmv in cuSPARSE library
3. Execute the kernel function ScaleElements where each thread i processes one item in COO format < r, c, val > and scales the element value by the inverse of $y_i$.
4. Compress the row indices through the cuSPARSE interface cusparseXcoo2csr.
5. The compressed row indices, the column indices and the updated element value form the CSR representation of $D^{-1}W$
```

Other sparse formats such as Compressed Sparse Column Format (**CSC**), Block Compressed Sparse Row Format (**BSR**) are also supported in our implementation.

B. Parallel Eigensolvers

Given the similarity graph W represented in some sparse format and the desired number of clusters k, this step computes the k eigenvectors corresponding to the smallest k eigenvalues of normalized Laplacian $L_n = I - D^{-1}W$ where W is the sparse matrix and D is the diagonal matrix with each element $D_{i,i} = \sum_{j=1}^{n} W_{i,j}$. We assume that $D_{i,i}$ are all positive, otherwise the isolated nodes can be removed from the graph. The eigenvectors corresponding to the smallest k eigenvalues of the normalized Laplacian are exactly the eigenvectors corresponding to the largest k eigenvalues of $D^{-1}W$. Since computing the largest eigenvalues results in better numerical stability and convergent behavior, we focus our attention on computing the eigenvectors corresponding to the largest k eigenvalues of $D^{-1}W$.

The sparse matrix multiplication $D^{-1}W$ can easily be computed as follows:

$$\begin{bmatrix} d_{11}^{-1} & & & \\ & d_{22}^{-1} & & \\ & & \dots & \\ & & & d_{nn}^{-1} \end{bmatrix} \times \begin{bmatrix} W_{1j} \\ W_{2j} \\ \dots \\ W_{nj} \end{bmatrix} = \begin{bmatrix} d_{11}^{-1} W_{1j} \\ d_{22}^{-1} W_{2j} \\ \dots \\ d_{nn}^{-1} W_{nj} \end{bmatrix} \tag{9}$$

The corresponding computation is data parallel and has complexity O(nnz). We assume that the sparse similarity matrix initially resides in the device memory, represented in **COO** format. The parallel computation is described in Algorithm 2. Note that the $D^{-1}W$ will be transformed to the **CSR** format to perform the sparse matrix-vector multiplication at the next step.

An important feature of the ARPACK software is the **reverse communication interfaces**, which facilitate the process of solving large-scale eigenvalue problems. The reverse communication interfaces are CPU-based interfaces that encapsulate implicitly restarted Arnoldi/Lanczos method, which is an iterative method to obtain the required eigenvalues and corresponding eigenvectors. For each iteration, the interface provides a n-length vector used as input and the output of sparse matrix-vector multiplication is provided back to the interface. ARPACK interfaces combine the

Algorithm 3 Parallel Eigensolver

```
1. Initialize the object Prob with parameters.
2. While !Prob.converge()
    Prob.TakeStep().
    Transfer the data located at Prob.GetVector() from host to device.
    Call cusparseDcsrmv to perform matrix-vector multiplication on device.
    Transfer the result from device to host and put it at the location addressed by Prob.PutVector().
3. Compute the eigenvectors by Prob.FindEigenvectors().
```

optimized Fortran routines and CPU-based BLAS library OpenBLAS, which is one of the most efficient CPU-based BLAS library. ARPACK provides the flexibility in choosing any matrix representation format and the function to obtain the results of matrix-vector multiplication. In our implementation, the matrix-vector multiplication is performed on the GPU. For each iteration, the input vector is transferred from the CPU to the GPU and the output vector is transfered back to the interface. The detailed implementation is shown in Algorithm 3.

The object `Prob` is initialized as the eigenvalue problem for the symmetric real matrix with the k largest-magnitude eigenvalues. `TakeStep()` is an interface that performs the necessary matrix operations based on the multi-threaded OpenBLAS library. For each iteration, the multiplication of sparse matrix and dense vector is computed on the GPU where 1) the sparse matrix is $D^{-1}W$ reside on GPU; 2) the input vector, whose location is indicated by `Prob.GetVector()`, is transferred from CPU to GPU; 3) the result is transfered back from GPU to CPU to the position `Prob.PutVector()`. After the object `Prob` reaches convergence, the eigenvectors are computed by `Prob.FindEigenvectors()`.

The complexity of Algorithm 3. largely depends on the interfaces `TakeStep()` and `FindEigenvectors()`. Both routines depend on the number m of Arnoldi/Lanczos vectors, which is usually set as $m = \max(n, 2k)$. `TakeStep()` involves the eigenvalue decomposition and iteratively QR factorization of $m \times m$ matrix, as well as a few dense matrix-vector multiplication. Therefore the complexity for `TakeStep()` is at least $(O(m^3)+O(nm)\times O(m-k))$. Moreover, the general complexity for sparse matrix-vector multiplication is $O(nnz \cdot m)$. The number of iteration $\#$ depends on the initial vector and properties of the matrix. The complexity `FindEigenvectors()` is $O(nmk)$. Hence the overall complexity is,

$$(O(m^3) + O(nm^2) + O(nnz \cdot m)) \times \# + O(nmk) \quad (10)$$

As far as we know, the procedures described in Algorithm 3 are currently the most efficient and convenient way to solve general eigenvalue problems for large-scale matrices. We leverage the existing software ARPACK on CPU to perform the complex eigensolver procedures and the GPU to perform

Algorithm 4 Parallel K-means Algorithm

```
1. Transfer the data $V \in \mathbb{R}^{n\times d}$ from the CPU to the GPU.
2. Randomly select $k$ points as the centroids of the k clusters stored in $C \in \mathbb{R}^{k\times d}$
3. While (the centroids change) do
    Compute the pairwise distances $S \in \mathbb{R}^{n\times k}$ between data points and the centroids.
    Update the new label of each data point.
    Compute the new centroids of the clusters.
4. Transfer the labeling result from GPU to CPU.
```

Algorithm 5 Parallel k-means++ Initialization

```
1. Pick the initial data point uniformly at random from 1 to n.
2. Initialize the n-length vector Dist where each element is the shortest distance between the data point $v_i$ and the current centroids.
3. for $i$ = 2 to k
    Compute the n-length vector $P$ such that $P_j = \frac{\text{Dist}_j^2}{\sum_{l=1}^{n}\text{Dist}_l^2}$
    Choose the $i^{\text{th}}$ centroid as the data point $x$ with probability $P_x$
    Compute the vector newDist such that each $i^{\text{th}}$ element as the distance between the data point $v_i$ and the new centroid
    Update Dist $\text{Dist}_j = \text{minimum}(\text{Dist}_j, \text{newDist}_j)$
```

the expensive matrix computations. Results in Section V. will show that the data communication overhead is negligible compared to the overall computational cost and the overall implementation is very efficient compared to other software that relies on CPU-based sparse matrix-vector multiplication.

C. Parallel k-means clustering

The k-means clustering algorithm is an iterative algorithm to partition the input data points into k clusters whose objective function is to minimize the sum of squared distances between each point and its representative. In spectral clustering, the k-means algorithm is used to cluster the rows of the matrix consisting of the eigenvectors. Each such row can in fact be viewed as a reduced dimension representation of the original data point. There are several GPU-based implementations of the k-means clustering such as [30, 31]. However, none of these implementations seem to be efficient for large-scale problems, especially when k is very large. Our implementation is a revised version from an open-source project [5] which efficiently utilizes the Thrust and CUBLAS libraries and achieve significant speedups.

We assume that the low-dimensional representation $V \in \mathbb{R}^{n\times k}$ initially resides in the CPU memory where n is the number of data points and k is the desired number of clusters. The implementation is described in Algorithm 4.

Step 2 is the most common way to initialize the centroids. However, we use a more effective initialization strategy, referred to as the k-means++ initialization, which has been shown to converge faster and achieve better results than the traditional k-means algorithm [32]. This initialization is simple to implement in parallel using basic routines in CUDA Thrust library, as described in Algorithm 5,

[5]https://github.com/bryancatanzaro/kmeans

Step 3 in Algorithm 4 is the main loop that iteratively updates the labels of the data points and the corresponding centers of the clusters until convergence (or the maximum number of iterations is reached). Given the data points $V \in \mathbb{R}^{n\times d}$ and centroids $C \in \mathbb{R}^{k\times d}$, the pair-wise distance matrix $S \in \mathbb{R}^{n\times k}$ is computed as follows.

$$S_{ij} = \sum_{l=1}^{d}(V_{il} - C_{jl})^2 \tag{11}$$

After expanding the right hand side, the distance matrix S can be expressed as

$$S_{ij} = \sum_{l=1}^{d}(V_{il})^2 + \sum_{l=1}^{d}(C_{jl})^2 - 2\sum_{l=1}^{d}V_{il}C_{jl} \tag{12}$$

Hence, we compute two additional vectors $V_{norm} \in \mathbb{R}^{n\times 1}$ and $C_{norm} \in \mathbb{R}^{n\times 1}$,

$$V_{norm}(i) = \sum_{l=1}^{d}(V_{il})^2, \tag{13}$$

$$C_{norm}(j) = \sum_{l=1}^{d}(C_{jl})^2 \tag{14}$$

The matrix S can be initialized as the sum of the corresponding elements in V_{norm} and C_{norm}

$$S_{ij} = V_{norm}(i) + C_{norm}(j) \tag{15}$$

The pair-wise distance matrix S is then computed by level-3 BLAS function provided in the cuBLAS library.

$$S = S - 2VC^T \tag{16}$$

For each data point, the new label is updated by as the index of centroid which has the minimum distance to the data point. Meanwhile, a global variable is maintained to record the number of label changes during the update.

The new centroids are updated as the mean value of all the data points sharing the same label. To identify the points in each cluster, we sort the data points according to their new labels. Each GPU thread will then independently work on a consecutive portion of the sorted data points where most of these points share the same label.

The entire workflow of our implementation is summarized in Figure 2.

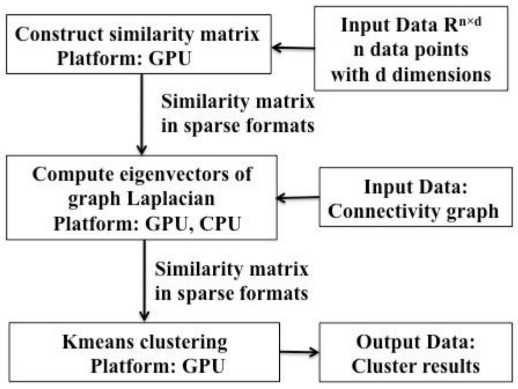

Figure 2: Parallel Implementation of Spectral Clustering

V. Evaluation

A. Datasets

We evaluate our parallel implementation on several real-world and synthetic datasets. The **Diffusion Tensor Imaging (DTI)** dataset is given as a set of data points, each of which is characterized by a 90-dimensional array. The other datasets are specified by an undirected graph data where the edges are given by an edge list. The problem sizes and the numbers of clusters generated are shown in Table II. A brief description of each dataset is given next.

- **DTI**: The Diffusion Tensor Imaging(**DTI**) dataset is the brain image data of a subject chosen from a publicly accessible medical dataset provided by Nathan Kline Institute (NKI). The dataset captures the diffusion of the water molecules in the brain tissues, which can be used to deduce information about the fiber connectivity in the human brain. After preprocessing steps [2], the input data consists of $142K$ data points, each of which represents a 2mm×2mm×2mm brain voxel. The entire data points constitute the brain volume. Each data point is characterized by a 90-dimensional array representing the connectivity strength of the voxel to 90 brain regions (representing a segmentation of the grey matter). The task is to cluster the voxels that share similar connectivity profiles. To facilitate the construction of the similarity matrix, an edge list is provided which contains all pair of voxels that are within 4 millimeter distance.
- **FB**: This dataset is a dataset collected by a Facebook application. It contains the graph where each node represents an anonymous user and edges exist between users that share similar political interests[33].
- **DBLP**: This dataset consists of a comprehensive co-authorship network in a computer science bibliography. The nodes represent the authors. Authors are connected if they coauthored at least one publication[33]. The dataset contains more than 5000 communities. Here we set the number of clusters to 500 for experimental purposes.
- **Syn200**: The synthetic dataset is randomly generated by the **stochastic block model** [34]. The stochastic block model assumes that the data points are partitioned into r disjoint subsets, $C_1, C_2, ..., C_r$. A symmetric $r \times r$ matrix P is provided to model the inter-community edge probability. The synthetic sparse graph is randomly generated such that two nodes are connected

Table II. Datasets

Dataset	Nodes	Edges	Clusters
DTI	142541	3992290	500
FB	4039	88234	10
DBLP	317080	1049866	500
Syn200	20000	773388	200

with probability $p = 0.3$ if they are within the same cluster and $q = 0.01$ if they are in different clusters.

B. Environment and Software

The computing environment is a heterogeneous CPU-GPU platform with CPU and GPU specifics shown in Table I. The software and packages used are as follows,

- **Matlab**: Matlab is a high-level language that provides interactive programing environment, which is widely used by scientists and engineers. The version of Matlab used for our implementation is 2015a. The sparse matrix representation and operations are the built-in functions. The k-means clustering is the function in Statistical and Machine Learning toolbox.
- **Python**: Python software packages, such as Numpy, Scipy and sklearn, are popular tools to perform scientific computations. The version of Python binary for our implementation is 2.7.11. The sparse representation and functions to solve the eigenvalue problems are from *Scipy* package. The k-means clustering function is from *sklearn.cluster* module. The module versions are Numpy-1.10.4, Scipy-0.16.1 and sklearn-0.17 respectively.

Linear algebra and numeric functions are by default multi-threaded in Matlab on multicore and multiprocessor machines [6]. In addition, the Python packages are built on highly optimized CPU-based BLAS routines, some of which have been accelerated using multi-threaded programming.

C. Performance Analysis

We measure the running time of our spectral clustering algorithm on the three components separately: 1) computation of the similarity matrix; 2) sparse matrix eigensolver; and 3) the k-means clustering algorithm. For the CUDA implementation, we measure the time costs that include both the computational time as well as the extra time for library initialization time and data communication. Specifically, we evaluate the performance of each of the following components:

- **Computation of the similarity matrix**:
  - initialize CUDA libraries.
  - transfer data and edge list from CPU to GPU.
  - construct the similarity matrix.
- **Sparse matrix eigensolver**:

[6]http://www.mathworks.com/discovery/matlab-multicore.html

Table III. Running Time of Spectral Clustering on DTI Dataset

Time/s	CUDA	Matlab	Python
Compute Similarity Matrix	**0.0331**	221.249	220.880
Sparse Eigensolver	**475.442**	603.165	3281.973
K-means Clustering	**5.407**	1785.17	2154.7818

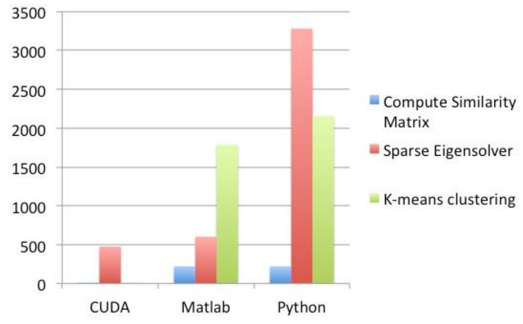

Figure 3: Time Costs of Spectral Clustering on DTI Dataset

  - data communication between CPU and GPU;
  - computation of the eigenvectors;
  - transfer of the eigenvectors from CPU to GPU.
- **K-means clustering**:
  - perform the k-means clustering;
  - tranfer the clustering result from GPU to CPU.

Figure 3. and Table III. show the time costs of each step corresponding to the **DTI** dataset.

It is clear that our CUDA implementation significantly outperforms the currently fastest known Matlab and Python implementations at each step. Since the computation of the similarity matrix is highly parallel, the CUDA implementation achieves linear speedups by taking advantage of the GPU with thousands of threads computing the cross correlation coefficients concurrently. For the Matlab and Python implementations, the results are based on the serial implementation which loops over the edge list and computes the correlation coefficient explicitly using the built-in function. We also tested an alternative implementation which takes advantage of *vectorization* techniques that recast the loop-based operation into matrix and vector operations. The optimized Matlab and Python implementationd take $5.753s$ and $6.271s$ trespectively to compute the similarity matrix.

Both Matlab and Python packages utilize the *reverse communication interfaces* of ARPACK to compute the eigenvectors of large-scale symmetric matrix, and hence all of the three implementations share similar procedures and interfaces. The basic difference is related to the function to compute the sparse matrix-vector multiplication. Our CUDA implementation utilizes the GPU and the cuSPARSE library to compute the multiplication while Matlab and Python utilize their built-in routines. Since the GPU performs significantly better than the CPU on BLAS operations [5],

Table IV. Running Time of Spectral Clustering on FB Dataset

Time/s	CUDA	Matlab	Python
Sparse Eigensolver	**0.0216**	0.1027	0.0851
K-means Clustering	**0.007251**	0.0205	0.0259

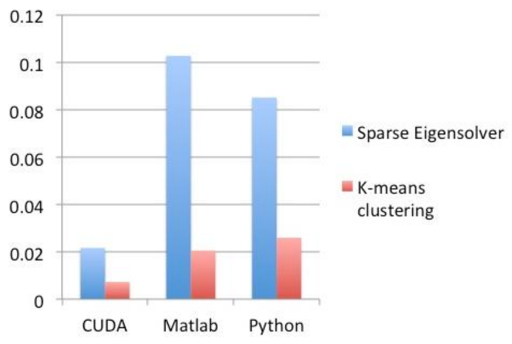

Figure 4: Time Costs of Spectral Clustering on FB Dataset

the CUDA implementation achieves better performance than Matlab and Python even with the communication overhead. However, since the time complexity of implicitly restarted Lanczos method is approximately $O(m^3 + nm^2)$, the time spent on the *reverse communication interfaces* scales relatively poorly, which may become the most computationally expensive part when k is large.

As for the *kmeans clustering* algorithm, our CUDA implementation achieves more than 300x speedup over the Matlab and Python implementations. The running time of this step depends on the centroid initialization. The CUDA and Python implementations utilize the k-means++ initialization, which leads to fewer number of iterations in general than Matlab. Moreover, in the CUDA implementation, the process of transforming the computation of the pair-wise distance matrix to the BLAS operations significantly accelerates the running time of the algorithm.

The performance results for the graph datasets (**FB**, **Syn200**, **dblp**) are shown in Table IV through Table VI and Figure 4 through Figure 6. Similar to the previous results, our CUDA implementation achieves the best performance among the three implementations at each step. However, the speedup ratio depends on the specific problem size.

The FB dataset contains a very small graph with 4039 nodes and involves very few clusters $k = 10$. Because the number of clusters is small, the most expensive computation of *sparse eigensolver* is the sparse matrix-vector multiplication. Therefore for this step,the CUDA implementation achieves around 5x speedup over the other implementations. For the *k-means clustering* step, the CUDA implementation shows only a minor speedup by a factor of around 4x.

The Syn200 dataset contains a medium-sized synthetic graph with 200 clusters. The CUDA implementation achieves a slight improvement in computing the eigenvectors since the performance is mainly constrained by the CPU-based routines. For the of k-means clustering step, the CUDA implementation achieves over 100x speedup.

Table V. Running Time of Spectral Clustering on Syn200 Dataset

Time/s	CUDA	Matlab	Python
Sparse Eigensolver	**4.1153**	6.9531	18.915
K-means Clustering	**0.02478**	38.3728	2.4719

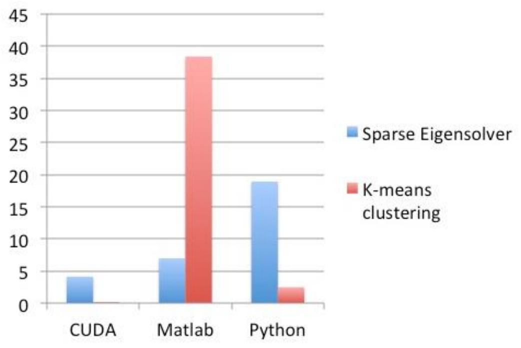

Figure 5: Time Costs of Spectral Clustering on Syn200 Dataset

Table VI. Running Time of Spectral Clustering on dblp Dataset

Time/s	CUDA	Matlab	Python
Sparse Eigensolver	**682.643**	1885.2303	9338.31
K-means Clustering	**1.79456**	1012.92	719.686

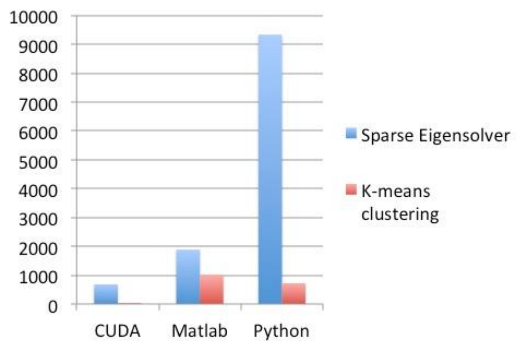

Figure 6: Time Costs of Spectral Clustering on dblp Dataset

The dblp dataset contains a large-scale graph with 500 clusters. Both Matlab and Python implementations perform poorly for such a problem size. Our CUDA implementation achieve around 3x speedup in *sparse eigensolver* in spite of the fact that the performance is still constrained by the CPU-based interfaces. In the k-means clustering step, the CUDA implementation achieves over 400x speedup.

Table VII shows a comparison between data communication time and computation time for the CUDA implementation on each of our four datasets. The data communication

Table VII. Comparison Between Data Communication Time and Computation Time

Time/s	Communication	Computation
DTI	2.248	475.213
FB	0.002131	0.02635
DBLP	2.731	680.31
Syn200	0.0741	3.8201

time includes 1) input data transfered from CPU to GPU; 2) data communication between CPU and GPU during the execution of the eigensolver stage; 3) output results that are transferred from GPU to CPU. Given that the bandwidth remains constant during the execution of the algorithm, the time complexity of data communication is $O(n^2 + m \times \# + nk)$ depending on the sparsity ratio of the similarity matrix and the number of Arnoldi iterations # n; the time complexity of computation is $O(nd^2 + O(nm^2) \times \# + O(n^2k))$. Therefore we expect the data communication time to be less than the computational time as in fact illustrated in the Table VII, especially for large-scale problems.

In conclusion, our CUDA implementation always achieves better performance than Matlab and Python implementations for each step. The speedup ratio largely depends on the specific problem size. Our traget applications involve problems with a large number of clusters. Our implementation achieves significant speedups for the steps of *computing the similarity matrix* and the *k-means clustering* due to the massive computational power of GPU. Moreover, we always achieve some speedups for the *sparse eigensolver* step by accelerating the computations involving matrix-vector multiplications.

VI. Conclusion

We presented a high performance implementation of the spectral clustering algorithm on CPU-GPU platforms. Our implementation leverages the GPU to accelerate highly parallel computations and Basic Linear Algebra Subprograms (BLAS) operations. We focused on the acceleration of the three major steps of the spectral clustering algorithm: 1) construction of the similarity matrix; 2) computation of eigenvectors for large-scale similarity matrices; 3) k-means clustering algorithm. We believe that we are the first to accelerate the large-scale eigenvector computation by combining the interfaces of traditional CPU-based software packages ARPACK and GPU-based CUDA library. Such a combination achieves good speedups compared to other CPU-based software. We deploy a smart seeding strategy and utilize BLAS operations to implement the fast k-means clustering algorithm. Our implementation is shown to achieve significant speedup compared to Matlab and Python software packages, especially for large-scale problems.

Acknowledgment

We gratefully acknowledge funding provided by The University of Maryland/Mpowering the State through the Center for Health-related Informatics and Bioimaging (CHIB) and the NVIDIA Research Excellence Center at the University of Maryland.

References

[1] Y. van Gennip, B. Hunter, R. Ahn, P. Elliott, K. Luh, M. Halvorson, S. Reid, M. Valasik, J. Wo, G. E. Tita, *et al.*, "Community detection using spectral clustering on sparse geosocial data," *SIAM Journal on Applied Mathematics*, vol. 73, no. 1, pp. 67–83, 2013.

[2] Y. Jin, J. F. JaJa, R. Chen, and E. H. Herskovits, "A data-driven approach to extract connectivity structures from diffusion tensor imaging data," in *2015 IEEE International Conference on Big Data*, IEEE, 2015.

[3] R. C. Craddock, G. A. James, P. E. Holtzheimer, X. P. Hu, and H. S. Mayberg, "A whole brain fmri atlas generated via spatially constrained spectral clustering," *Human brain mapping*, vol. 33, no. 8, pp. 1914–1928, 2012.

[4] D. Eddelbuettel, *Benchmarking single-and multi-core blas implementations and gpus for use with r*.

[5] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," *Internet: Www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.

[6] C. Lee, W. W. Ro, and J.-L. Gaudiot, "Boosting cuda applications with cpu–gpu hybrid computing," *International Journal of Parallel Programming*, vol. 42, no. 2, pp. 384–404, 2014.

[7] J. Agulleiro, F Vazquez, E. Garzon, and J. Fernandez, "Hybrid computing: Cpu+ gpu co-processing and its application to tomographic reconstruction," *Ultramicroscopy*, vol. 115, pp. 109–114, 2012.

[8] J. Wu and J. JaJa, "Optimized fft computations on heterogeneous platforms with application to the poisson equation," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, pp. 2745–2756, 2014.

[9] J. Wu and J. JaJa, "Achieving native gpu performance for out-of-card large matrix multiplication," *Parallel Processing Letters*, 2015.

[10] K. Li, W. Yang, and K. Li, "A hybrid parallel solving algorithm on gpu for quasi-tridiagonal system of linear equations," *IEEE Transactions on Parallel and Distributed Systems*,

[11] E. N. Akimova and D. V. Belousov, "Parallel algorithms for solving linear systems with block-tridiagonal matrices on multi-core cpu with gpu," *Journal of Computational Science*, vol. 3, no. 6, pp. 445–449, 2012.

[12] J. Zheng, W. Chen, Y. Chen, Y. Zhang, Y. Zhao, and W. Zheng, "Parallelization of spectral clustering algorithm on multi-core processors and gpgpu," in *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*, IEEE, 2008, pp. 1–8.

[13] K. K. Matam and K. Kothapalli, "Gpu accelerated lanczos algorithm with applications," in *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, IEEE, 2011, pp. 71–76.

[14] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 3, pp. 568–586, 2011.

[15] Y. Song, W.-Y. Chen, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering," in *Machine Learning and Knowledge Discovery in Databases*, Springer, 2008, pp. 374–389.

[16] S. Tsironis, M. Sozio, M. Vazirgiannis, and L.-E. Poltechnique, "Accurate spectral clustering for community detection in mapreduce," Citeseer.

[17] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," *IBM Journal of Research and Development*, vol. 17, no. 5, pp. 420–425, 1973.

[18] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 8, pp. 888–905, 2000.

[19] A. Y. Ng, M. I. Jordan, Y. Weiss, *et al.*, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 2, pp. 849–856, 2002.

[20] W. Pentney and M. Meila, "Spectral clustering of biological sequence data," in *AAAI*, vol. 5, 2005, pp. 845–850.

[21] D. J. Higham, G. Kalna, and M. Kibble, "Spectral clustering and its use in bioinformatics," *Journal of computational and applied mathematics*, vol. 204, no. 1, pp. 25–37, 2007.

[22] S. White and P. Smyth, "A spectral clustering approach to finding communities in graph.," vol. 5, SIAM, 2005, pp. 76–84.

[23] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan, "Clustering social networks," in *Algorithms and Models for the Web-Graph*, Springer, 2007, pp. 56–67.

[24] A.-G. Chifu, F. Hristea, J. Mothe, and M. Popescu, "Word sense discrimination in information retrieval: A spectral clustering-based approach," *Information Processing & Management*, vol. 51, no. 2, pp. 16–31, 2015.

[25] B. McFee and D. P. Ellis, "Analyzing song structure with spectral clustering."

[26] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[27] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK users' guide: Solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998, vol. 6.

[28] R. B. Lehoucq and D. C. Sorensen, "Deflation techniques for an implicitly restarted arnoldi iteration," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 789–821, 1996.

[29] D. C. Sorensen, "Implicit application of polynomial filters in ak-step arnoldi method," *Siam journal on matrix analysis and applications*, vol. 13, no. 1, pp. 357–385, 1992.

[30] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via cuda," in *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*, IEEE, 2009, pp. 7–15.

[31] J. Wu and B. Hong, "An efficient k-means algorithm on cuda," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 1740–1749.

[32] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[33] J. Leskovec and A. Krevl, *Snap datasets: Stanford large network dataset collection*, http://snap.stanford.edu/data.

[34] B. Karrer and M. E. Newman, "Stochastic blockmodels and community structure in networks," *Physical Review E*, vol. 83, no. 1, p. 016 107, 2011.