Parallel Visual Assessment of Cluster Tendency on GPU

Tao Meng and Bo Yuan(✉)

Intelligent Computing Lab, Division of Informatics,
Graduate School at Shenzhen, Tsinghua University,
Shenzhen 518055, People's Republic of China
zdhmengtao@163.com, yuanb@sz.tsinghua.edu.cn

Abstract. Determining the number of clusters in a data set is a critical issue in cluster analysis. The Visual Assessment of (cluster) Tendency (VAT) algorithm is an effective tool for investigating cluster tendency, which produces an intuitive image of matrix as the representation of complex data sets. However, VAT can be computationally expensive for large data sets due to its $O(N^2)$ time complexity. In this paper, we propose an efficient parallel scheme to accelerate the original VAT using NVIDIA GPU and CUDA architecture. We show that, on a range of data sets, the GPU-based VAT features good scalability and can achieve significant speedups compared to the original algorithm.

Keywords: Cluster analysis · Cluster tendency · VAT · GPU

1 Introduction

Cluster analysis is an important task in pattern recognition and data mining. In general, it consists of three steps: (1) assessing the cluster tendency (e.g., how many groups to seek); (2) partitioning the data into groups; (3) validating the clusters discovered [1]. For data that can be directly projected onto a 2D or 3D Euclidean space (e.g., with a scatter plot), direct observation can provide good insight on the appropriate number of clusters. However, for high-dimensional data, or when only the pairwise relationship between objects is available, advanced techniques are necessary.

Visual Assessment of (cluster) Tendency (VAT) [2] is one of the popular methods widely used to assess the cluster tendency. Given the dissimilarity matrix D of a set of n objects, VAT represents D as an $n \times n$ image $I(D^*)$ where the objects are reordered to reveal the hidden cluster structure as dark blocks along the diagonal of the image.

VAT works well on relatively small data sets (e.g., 500 or fewer objects). However, for data sets of moderate sizes (e.g., 20,000 data points), the computing time of VAT, with time complexity $O(N^2)$, may become intolerable. In view of the high computing time of VAT, several extensions such as reVAT [3], bigVAT [4] and sVAT [5] have been proposed. reVAT performs quasi-ordering of the objects based on a threshold parameter and replaces the intensity image with a series of one-dimensional profile graphs. However, the profile graphs are not as interpretable as the images produced by VAT. To address this problem, bigVAT uses the profile graphs to select a sample of

J. Kim et al. (Eds.): PAKDD 2017, Part II, LNAI 10235, pp. 429–440, 2017.
DOI: 10.1007/978-3-319-57529-2_34

objects and displays the quasi-ordered dissimilarity data of the sampled objects as a VAT-like intensity image. However, the resulting image may not be as descriptive as the VAT-ordered image. sVAT selects a sample of (approximately) size n from the full set of objects $O = \{o_1, o_2, \ldots, o_N\}$, and performs VAT on the sample. The sample is chosen so that it contains similar cluster structure as the original data set. However, if the original data set contains many clusters, the value of n needs to increase accordingly, creating a computational issue again.

GPU (Graphics Processing Unit) is an inexpensive, energy efficient and highly efficient SIMT (Single Instruction, Multiple Thread) parallel computing device, which can be found in many mainstream desktop computers and workstations. In this paper, we propose to improve the computational efficiency of VAT using CUDA-enabled GPUs by exploiting their massively parallel computing capability and the potential of parallelism of VAT.

This paper is organized as follows. Section 2 gives a brief review of VAT and its variations as well as GPU computing. Section 3 presents the details of the proposed parallel VAT algorithm based on GPU. The main experimental studies are reported in Sect. 4, focusing on the comparison between CPU-based VAT and GPU-based VAT. This paper is concluded in Sect. 5 with some discussions on future work.

2 Related Work

2.1 A Brief Review of VAT

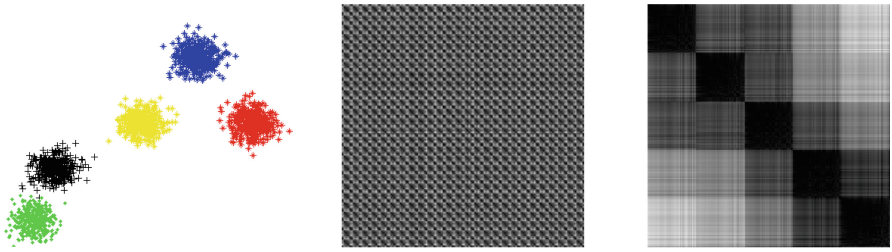
Let $O = \{o_1, o_2, \ldots, o_n\}$ denote n objects in the data set and D denote a matrix of pairwise dissimilarities between objects each element of which $d_{ij} = d(o_i, o_j)$ is the dissimilarity between objects o_i and o_j, with $0 \leq d_{ij} \leq 1; d_{ij} = d_{ji}; d_{ii} = 0$, for $1 \leq i, j \leq n$. Let K be the permutation of $\{1, 2, \ldots, n\}$ such that $K(i)$ is the index of the i^{th} element in the list. The reordered list is represented as: $\{o_{K(1)}, o_{K(2)}, \ldots, o_{K(n)}\}$. Let P be the permutation matrix with $p_{ij} = 1$ if $j = K(i)$ and 0 otherwise. The matrix D^* for the reordered list is a similarity transform of D by P: $D^* = P^T DP$.

The key idea is to find P so that D^* is as close to a block diagonal form as possible. VAT reorders the row and columns of D using a modified version of Prim's minimal spanning tree (MST) algorithm [6], and displays D^* as a gray-scale image. The main difference is that VAT does not form a MST. Instead, it identifies the order in which vertices are added and the initial vertex is selected based on the maximum edge weight in the underlying complete graph [7]. The general procedure of the VAT algorithm is shown in Table 1.

An example of VAT is shown in Fig. 1. Figure 1(a) is the scatter plot of 2,000 data points in 2D. The 5 visually apparent clusters are reflected by the 5 distinct dark blocks along the main diagonal in Fig. 1(c), which is the VAT image of the data. Compared to the image of D in the original order as shown in Fig. 1(b), it is evident that reordering is necessary to reveal the underlying cluster structure of the data.

Table 1. Algorithm I: The VAT algorithm

Input: A $N \times N$ dissimilarity matrix D
1: Set $I = \emptyset, J = \{1, 2, \ldots, N\}$ and $K = (0, 0, \ldots, 0)$.
2: Select $(i, j) \in \arg_{p \in I, q \in J} \max\{d_{pq}\}$.
3: Set $K(1) = i, I \leftarrow \{ i \}$ *and* $J \leftarrow J - \{ i \}$.
4: **for** $t = 2: N$ **do**
5: Select $(i, j) \in \arg_{p \in I, q \in J} \min\{d_{pq}\}$.
6: Set $K(t) = j$, update $I \leftarrow I \cup \{ j \}$ *and* $J \leftarrow J - \{ j \}$.
7: **end for**
8: Form the reordered matrix $D^* = [\, d^*_{ij} \,] = [\, d_{K(i)K(j)}]$, for $1 \le i, j \le n$.
Output: A gray-scale image $I(D^*)$ with max$\{\, d^*_{ij} \,\}$: white and min $\{\, d^*_{ij} \,\}$: black.



(a) (b) (c)

Fig. 1. An example of VAT: (a) the scatter plot of the 2D dataset; (b) the original dissimilarity image $I(D)$ and (c) the reordered VAT image $I(D^*)$

2.2 GPU High Performance Computing

In recent years, GPUs have evolved into highly parallel, multi-threaded, many-core processors and are widely used for general purpose computing [13]. Compared with CPU based distributed systems such as Hadoop, GPU based parallel computing systems are more lightweight, portable and energy-efficient. GPUs are well suited to problems that can be represented as data-parallel tasks where the same instruction is executed on massive data elements in parallel. It is also highly desirable that the arithmetic intensity is high, which is the ratio between the number of arithmetic operations and the number of memory operations.

CUDA (Compute Unified Device Architecture) is a general-purpose parallel computing platform and programming model that leverages the parallel computing engine in NVIDIA GPUs to solve challenging computational problems in a more efficient way than CPUs. It was introduced by NVIDIA in November 2006, which significantly reduces the difficulty faced by programmers for developing flexible parallel programs based on NVIDIA GPUs.

Threads and kernels are the core concepts in CUDA. Threads are lightweight processes executed on independent processors in GPU, and they are easy to be created

and synchronized. Kernels are functions executed on the GPU in parallel by massive threads organized into blocks and grids [14].

There are different types of memory in GPUs, which can significantly affect the performance of GPU programs. Each thread has its private local memory called register, which is the fastest type of memory. Each thread block features shared memory accessible by all threads within the same block, which can be as fast as registers if accessed properly. All threads have access to the same global memory, which is the largest and slowest storage and the only memory visible to CPU. Constant memory and texture memory are two read-only memory spaces accessible by all threads [15]. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

In data science, examples of successful GPU applications include matrix multiplication [16], databases [17–19], data stream mining [20], FIMI mining [21], subsequence search [22] and GPU-based primitives for database applications [19, 23].

3 GPU-Accelerated VAT

In this section, we present the design and implementation details of the proposed parallel VAT based on GPU. The VAT algorithm shown in Table 1 consists of three steps: (1) finding the maximum dissimilarity value and the objects involved; (2) generating the new order; (3) reordering the matrix. Our implementation follows the general workflow of the original algorithm. To make the algorithm more suitable for parallel implementation, we also make some changes (Table 2).

Table 2. Algorithm II: VAT based on GPU

Input: An $N \times N$ dissimilarity matrix D.
1: $I = (0, 0, \ldots, 0), J = (0, 0, \ldots, 0), K = \{1, 2, \ldots, N\}$ and $L = (0, 0, \ldots, 0)$.
2: Select $(i, j) \in \arg_{p \in \boldsymbol{K}, q \in \boldsymbol{K}} \max\{d_{pq}\}$. // **parallel**
3: Set $L(k) = \infty$, for $1 \le k \le N$. // **parallel**
4: Set $J(i) = 1$ *and* $I(1) = \{ i \}$.
5: **for** $t = 2$: N **do**
6: $\arg_{J(k)=0} L(k) = \min\{D_{[I(t-1)][k]}, L(k)\}$, for $1 \le k \le N$. // **parallel**
7: Select $i \in \arg_{J(k)=0} \min\{L(k)\}$. // **parallel**
8: Set $J(i) = 1$ *and* $I(t) = i$.
9: **end for**
10: Normalize D to D' as $d'_{pq} \in [0, 255]$, for $0 \le p, q \le N$. // **parallel**
11: Form the reordered matrix $D^* = [d^*_{ij}] = [\, d'_{I(i)I(j)} \,]$, for $1 \le i, j \le N$. // **parallel**
Output: A gray-scale image $I(D^*)$ with max$\{ d^*_{ij} \}$: white and min $\{ d^*_{ij} \}$: black.

3.1 Finding the Maximum Value

The reduction algorithm is a good choice for finding the maximum value of a matrix in GPU. Reduction refers to a class of parallel operations that pass over $O(N)$ input data

and generate $O(1)$ result, computed by a binary associative operator $\oplus$. Examples of such operations include minimum, maximum, sum, sum of squares, AND, OR, and the dot product of two vectors [24]. Unless the operator $\oplus$ is extremely expensive to evaluate, reduction tends to be bandwidth-bound. Figure 2 shows an example of parallel reduction that computes the maximum of an 8-element array. There are four threads in use, which are marked in different colors.

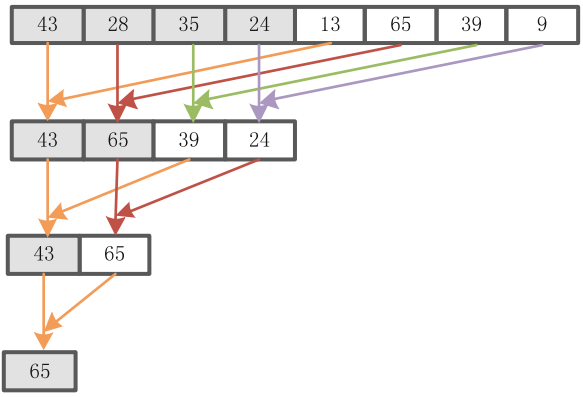


Fig. 2. An example of parallel reduction: finding the maximum value of a vector

Although Thrust, a popular library in CUDA, can find the maximum value efficiently, we employ a special reduction method as the index of the object with the maximum value is required. Furthermore, the input matrix itself is symmetric, which means that only half of the matrix needs to be processed. In this paper, we apply the Two-Pass Reduction algorithm [24] to find the maximum value and the maximum value's index in the matrix. The Two-Pass Reduction operates in two stages, as shown in Fig. 3. A kernel performs *NumBlocks* reductions in parallel, where *Numblocks* is the

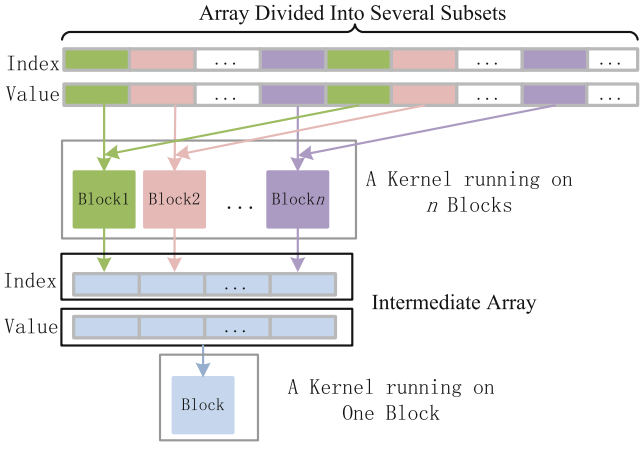


Fig. 3. An example of the Two-Pass Reduction algorithm

number of blocks used to invoke the kernel. Then, the results are stored into an intermediate array. The final result is generated by invoking the same kernel to perform a second pass on the intermediate array using a single block. Note that, this method imposes no requirement on the compute capability of GPUs, making it applicable to a wide range of GPU facilitates.

3.2 Generating New Order

Generating the new order of objects takes most of the time in VAT and its degree of parallelism has a significant impact on the overall speedup. It can be divided into two steps: computing the elements in L and finding the minimum value and the corresponding index in L.

Although it features a process of finding the minimum value and the corresponding index, we use the Reduction with Atomics algorithm with only a single kernel, instead of the Two-Pass Reduction algorithm. Note that, invoking a kernel, even a kernel that does nothing, involves a certain amount of overhead. In particular, in this step, the time spent in invoking a kernel is large compared to the time spent in the execution of the kernel. Furthermore, each kernel needs to be launched $N - 1$ times, where N is the width of the input matrix. Consequently, reducing the number of kernels in the algorithm is likely to be beneficial in terms of efficiency.

Similar to the Two-Pass Reduction algorithm, the Reduction with Atomics algorithm stores the result in an intermediate array. The difference is that the Reduction with Atomics algorithm uses a flag value for recording the number of exited blocks. As each block exits, it performs the `atomicAdd` function, a type of atomic operation in CUDA, to check whether it is the block that needs to perform the final reduction. Although the atomic operation does cost some extra time, the Reduction with Atomics algorithm is more efficient than Two-Pass Reduction when the size of data to be processed is small. Figure 4 shows the running times of the Reduction with Atomics algorithm and the Two-Pass Reduction algorithm.

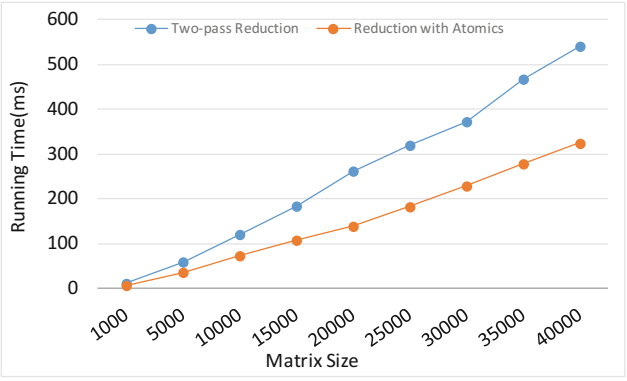


Fig. 4. Running times of the Reduction with Atomics algorithm and the Two-Pass Reduction algorithm on datasets of different sizes

3.3 Creating the Reordered Matrix

D needs to be transformed into $D^{'}$ where $d^{'}_{pq} \in [0, 255]$, to reflect the image density range [0, 255] in openCV [26]. Directly applying $N \times N$ threads may seem to be a straightforward way to create the reordered matrix. However, the memory in GPU is limited and in order to process more data, we need to transform the *double* type D to *unsigned char* type $D^{'}$ before applying $N \times N$ threads to execute $d^{*}_{ij} = d^{'}_{I(i)I(j)}$.

Since in GPU blocks are executed in an unordered manner [17], a memory space may be filled with new data before the original data has been read when multiple blocks are in use. To solve this issue, we perform the transformation of the first n elements using the same block where

$$n = \frac{N^2 \times size\ of\ (unsignedchar)}{size\ of\ (double)} \tag{1}$$

so that the rest elements can be safely processed in parallel with multiple blocks, as shown in Fig. 5.

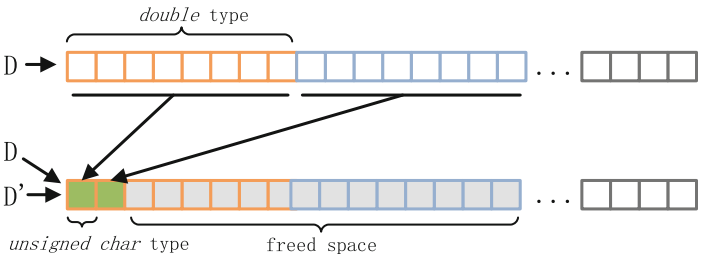


Fig. 5. Transformation of *double* type to *unsigned char* type

4 Experimental Results

We conducted the experiments on a workstation with two Intel Xeon E5-2640 v2 (2.00 GHz, 8 Cores) CPUs, 128 GB RAM and NVIDIA GeForce GTX TITAN X GPU. Powered by NVIDIA Maxwell architecture, the GeForce GTX TITAN X GPU features 3,072 CUDA cores and 12 GB GDDR5 memory. The programming environment was gcc-4.7 with CUDA 7.5 running on Ubuntu 15.04 (64 bit).

4.1 Test Datasets

We used a random dataset generator from *scikit-learn* [25]. Four different types of datasets (circles, moons, blobs and random) were generated (Fig. 6) and 10 instances (2D) were created for each type of dataset with 1,000 to 45,000 objects. We also used a dataset from UCI Machine Learning Repository [27] from which we sampled subsets

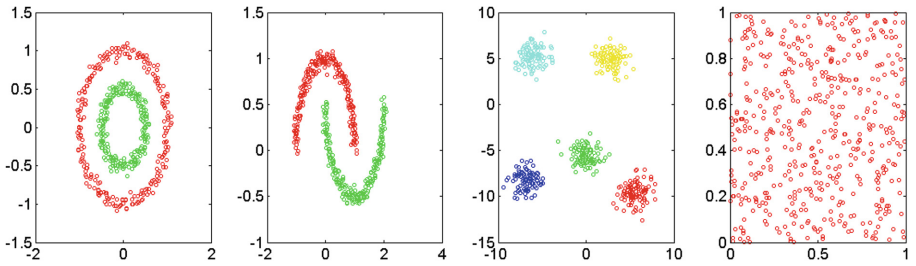


Fig. 6. Four different types of datasets used in the experiments. From left to right: circles, moons, blobs and random

with different sizes. Since the input of VAT is a dissimilarity matrix, once this matrix is given, the efficiency of VAT is fully determined, regardless of the dimension of the original dataset.

4.2 Results and Analysis

For each dataset, we compared the efficiency of the CPU-based VAT and our parallel VAT base on GPU. For the same data size, our algorithm achieved almost the same speedup rate on different datasets. So, we averaged the results and present the running time and speedup rate in Fig. 7. It is clear that, the running time of the original VAT increased rapidly due to its $O(N^2)$ time complexity. Meanwhile, the speedup rate increased steadily as the matrix size increased and reached around 37 for datasets with 40,000 objects.

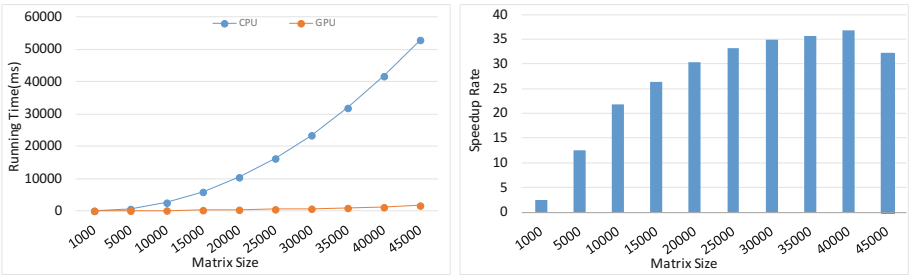


Fig. 7. Average running time and speedup rate on synthetic datasets of different sizes

Figures 8, 9 and 10 show the individual running time and speedup rate for each of the three major operations in VAT: finding the maximum value, generating the new order and creating the reordered matrix. Figure 11 shows the average time of data transmission between CPU and GPU. Figure 12 shows the average running time and speedup rate on real datasets of different sizes, showing an overall trend similar to Fig. 7. Note that, on datasets with 45,000 objects, we transformed the *double* type *D* to

an *unsigned char* type $D^{'}$ to reduce the space requirement before applying $N \times N$ threads to execute $d^{*}_{ij} = d^{'}_{I(i)I(j)}$. Due to the extra time cost, the overall speedup rates (e.g., Fig. 7) and the speedup rate for creating the reordered matrix (Fig. 10) both dropped slightly.

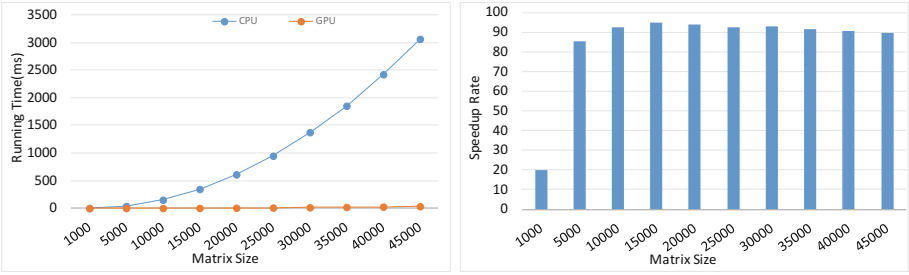


Fig. 8. Average running time and speedup rate of finding the maximum value on datasets of different sizes

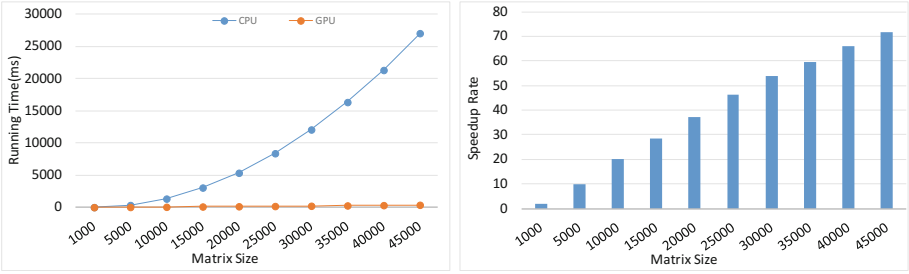


Fig. 9. Average running time and speedup rate of generating the new order on datasets of different sizes

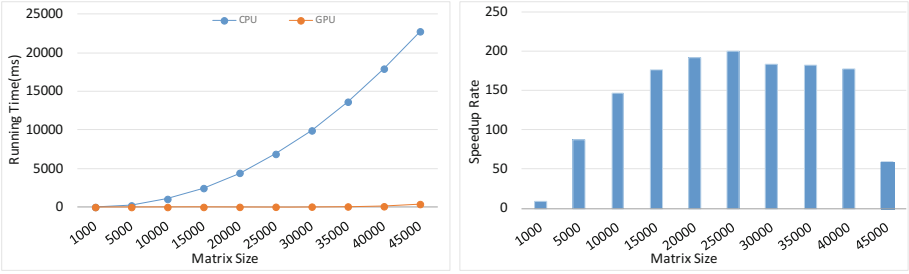


Fig. 10. Average running time and speedup rate of creating the reordered matrix on datasets of different sizes

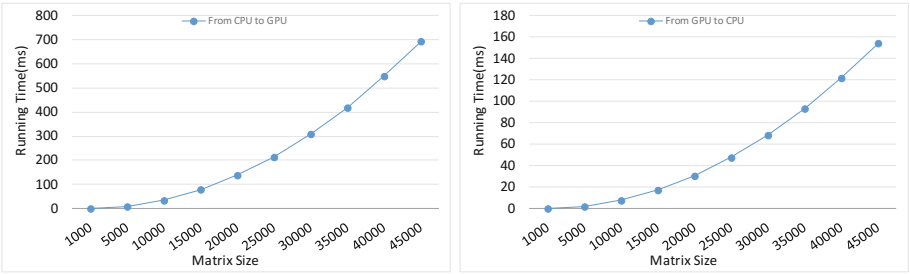


Fig. 11. Average running time of data transmission from CPU to GPU and from GPU to CPU

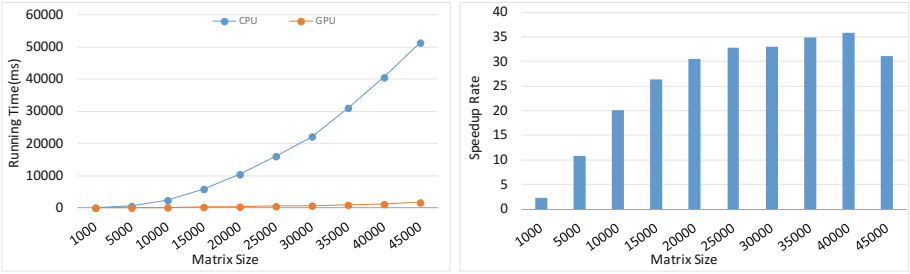


Fig. 12. Average running time and speedup rate on real datasets of different sizes

5 Conclusion

Visualizing the cluster tendency of datasets is important in both academic research and industrial applications. However, the applicability of VAT, one of the most popular visualization techniques in this domain, has been severely limited by its high time complexity. In this paper, we investigated the potential of parallelism of various components in VAT and proposed a GPU-based parallel VAT. Experiments on a variety of test datasets showed that the parallel VAT can achieve significant speedup rates and demonstrated good scalability in handling large datasets.

In recent years, a number of variations of VAT have been proposed to enhance its capability. For example, iVAT [8] and efiVAT [9] improve the ability of VAT to highlight cluster structure in $I(D^*)$ when D contains highly complex clusters. Havens et al. [10] performed data clustering in ordered dissimilarity images, and coVAT [11] extends VAT to rectangular dissimilarity data. CCE [12], DBE [15] and aVAT [8] use different schemes to automatically estimate the number of clusters in VAT images. Most of these VAT-like methods are built on the basic idea of the original VAT and our proposed parallel VAT algorithm can be potentially extended to these algorithms.

Acknowledgment. This work was partially supported by the NVIDIA GPU Education Center awarded to Tsinghua University.

References

1. Wang, L., Geng, X., Bezdek, J., Leckie, C., Kotagiri, R.: SpecVAT: enhanced visual cluster analysis. In: International Conference on Data Mining, pp. 638–647 (2008)
2. Bezdek, J.C., Hathaway, R.J.: VAT: a tool for visual assessment of (cluster) tendency. In: International Joint Conference on Neural Networks, vol. 3, pp. 2225–2230 (2002)
3. Huband, J.M., Bezdek, J.C., Hathaway, R.J.: Revised visual assessment of (cluster) tendency (reVAT). In: International Conference of the North American Fuzzy Information Processing Society, pp. 101–104 (2004)
4. Huband, J., Bezdek, J.C., Hathaway, R.: bigVAT: visual assessment of cluster tendency for large data sets. Pattern Recogn. **38**(11), 1875–1886 (2005)
5. Hathaway, R., Bezdek, J.C., Huband, J.: Scalable visual assessment of cluster tendency. Pattern Recogn. **39**(7), 1315–1324 (2006)
6. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36** (6), 1389–1401 (1957)
7. Pakhira, M.K.: Finding number of clusters before finding clusters. Procedia Technol. **4**, 27–37 (2012)
8. Wang, L., Nguyen, U.T.V., Bezdek, J.C., Leckie, C.A., Ramamohanarao, K.: iVAT and aVAT: enhanced visual analysis for cluster tendency assessment. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010. LNCS (LNAI), vol. 6118, pp. 16–27. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13657-3_5
9. Havens, T.C., Bezdek, J.C.: An efficient formulation of the improved visual assessment of cluster tendency (iVAT) algorithm. IEEE Trans. Knowl. Data Eng. **24**(5), 813–822 (2012)
10. Havens, T.C., Bezdek, J.C., Keller, J.M., Popescu, M.: Clustering in ordered dissimilarity data. Int. J. Intell. Syst. **24**(5), 504–528 (2009)
11. Bezdek, J.C., Hathaway, R., Huband, J.: Visual assessment of clustering tendency for rectangular dissimilarity matrices. IEEE Trans. Fuzzy Syst. **15**(5), 890–903 (2007)
12. Sledge, I., Huband, J., Bezdek, J.C.: (Automatic) Cluster count extraction from unlabeled datasets. In: Joint International Conference on Natural Computation and International Conference on Fuzzy Systems and Knowledge Discovery, vol. 1, pp. 3–13 (2008)
13. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/index.html
14. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Newnes, Oxford (2012)
15. Farber, R.: CUDA Application Design and Development. Elsevier, Amsterdam (2012)
16. Larsen, E.S., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, no. 43 (2001)
17. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 215–226 (2004)
18. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 325–336 (2006)
19. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 511–524 (2008)
20. Govindaraju, N.K., Raghuvanshi, N., Manocha, D.: Fast and approximate stream mining of quantiles and frequencies using graphics processors. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 611–622 (2005)

21. Fang, W., Lu, M., Xiao, X., He, B., Luo, Q.: Frequent itemset mining on graphics processors. In: Proceedings of the Fifth International Workshop on Data Management on New Hardware, pp. 34–42 (2009)
22. Sart, D., Mueen, A., Najjar, W., Keogh, E., Niennattrakul, V.: Accelerating dynamic time warping subsequence search with GPUs and FPGAs. In: 2010 IEEE International Conference on Data Mining, pp. 1001–1006 (2010)
23. He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, no. 46 (2007)
24. Nicholas, W.: The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley Professional, Boston (2013)
25. Pedregosa, et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
26. OpenCV User Guide. http://docs.opencv.org/2.4.13/doc/user_guide/user_guide.html
27. Bache, K., Lichman, M.: UCI Machine Learning Repository (2013)