Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Efficient parallel implementation of a density peaks clustering algorithm on graphics processing unit*

Ke-shi GE†, Hua-you SU†, Dong-sheng LI†‡, Xi-cheng LU
(*National Lab for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410003, China*)
†E-mail: gekeshi@nudt.edu.cn; huayousu@163.com; dsli@nudt.edu.cn

Received Dec. 17, 2016; Revision accepted Apr. 17, 2017; Crosschecked July 14, 2017

Abstract: The density peak (DP) algorithm has been widely used in scientific research due to its novel and effective peak density-based clustering approach. However, the DP algorithm uses each pair of data points several times when determining cluster centers, yielding high computational complexity. In this paper, we focus on accelerating the time-consuming density peaks algorithm with a graphics processing unit (GPU). We analyze the principle of the algorithm to locate its computational bottlenecks, and evaluate its potential for parallelism. In light of our analysis, we propose an efficient parallel DP algorithm targeting on a GPU architecture and implement this parallel method with compute unified device architecture (CUDA), called the 'CUDA-DP platform'. Specifically, we use shared memory to improve data locality, which reduces the amount of global memory access. To exploit the coalescing accessing mechanism of GPU, we convert the data structure of the CUDA-DP program from array of structures to structure of arrays. In addition, we introduce a binary search-and-sampling method to avoid sorting a large array. The results of the experiment show that CUDA-DP can achieve a 45-fold acceleration when compared to the central processing unit based density peaks implementation.

Key words: Density peak; Graphics processing unit; Parallel computing; Clustering
http://dx.doi.org/10.1631/FITEE.1601786 **CLC number:** TP301.6

1 Introduction

Clustering is an unsupervised classification technique that aims to separate unlabeled datasets into finite categories or clusters (Xu and Wunsch, 2005). Because clustering can find hidden patterns in datasets, it has been widely used in scientific research, such as machine learning (He *et al.*, 2016), computer vision, and bioinformatics. Several clustering algorithms have been proposed to fit various situations, but each algorithm has its drawbacks. K-means (MacQueen, 1967) and K-medoids (Park and Jun, 2009) methods take the centers of the data points as the corresponding cluster centers; however, such centers do not apply to nonspherical clusters, and the clustering results are influenced by the number of clusters. Hierarchical clustering algorithms (e.g., balanced iterative reducing and clustering using hierarchies (BIRCH) (Zhang *et al.*, 1997)) organize data into a hierarchical structure according to the proximity matrix. However, the time complexity of such algorithms is high, and the number of clusters needs to be specified in advance. A model-based clustering algorithm, self-organizing map (SOM) (Kohonen, 1990), sets a model for each cluster and finds the best fit for the model; however, the model is not necessarily correct, and the clustering result is sensitive to the parameters.

‡ Corresponding author
* Project supported by the National Basic Research Program (973) of China (No. 2014CB340303), the National Natural Science Foundation of China (Nos. 61502509 and 61222205), the Program for New Century Excellent Talents in University, and the Fok Ying-Tong Education Foundation (No. 141066)
ORCID: Dong-sheng LI, http://orcid.org/0000-0001-9743-2034

The density peak (DP) clustering algorithm was developed by Rodriguez and Laio (2014). It is based on density and distance theory. DP considers data points surrounded by neighboring points with lower local density as cluster centers. Here, 'lower' means that the local density of the point is lower than that of at least one point within a certain range. It is observed that cluster centers are far away from each other, meaning that the cluster centers have relatively large distance from data points with a higher local density.

Compared with other previous clustering algorithms, DP has many advantages. Unlike K-means and K-medoids (Park and Jun, 2009) which are effective only on spherical clusters, DP is able to detect arbitrary clusters. In addition, the accuracy of DP is unnecessarily depending on the capability of the trial probability to represent the data. In contrast with DP, the distribution-based algorithms (e.g., Gaussian mixture model (GMM) (Rasmussen, 2000)) require data points with a mix of predefined probability distribution functions to obtain good results. Furthermore, DP can automatically detect cluster centers with only the distance between data points, while K-means and density-based spatial clustering of applications with noise (DBSCAN) (Ester *et al.*, 1996) require prior knowledge about the dataset. For example, K-means requires the number of clusters, and ε and minPts are needed for DBSCAN. Last but not least, DP assigns all data points to clusters at one time, which means that it can avoid the convergence, whereas other algorithms may have local minimum results with improper initial states. As a convenient, novel, and effective clustering algorithm, DP has been widely applied in a variety of fields (e.g., time series analysis (Begum *et al.*, 2015), biochemical simulation (Zamuner *et al.*, 2015), and computer vision (Dean *et al.*, 2015)) and shown good results.

Although DP has many attractive characteristics, its computational complexity is very high, especially with the increase of data size and data dimension. This characteristic hampers the wide usage of the DP algorithm. To decide the cluster centers and correctly classify the points to the corresponding cluster, DP needs to compute two elements, local density ρ and distance δ, of each point with higher density. Both of these elements depend on the distances between all pairs of data points. Assuming that the size of the dataset is N, the computation complexity of the distance between each pair of points is $O(N^2)$. Moreover, similar to that of ε in DBSCAN, the magnitude of local density is sensitive to the threshold (denoted by d_c) which determines that the average number of neighbors is around 1% to 2% of the total number of data points. Generally, the procedure of computing d_c is implemented by sorting the values of all the distances, and then finding their locations in the sorted distance list. As we know, sorting a large array is very expensive. For large-scale and high-dimensional datasets, the time consumption is too high.

To relieve the time consumption problem, in the past few years, some researchers have devoted themselves to accelerating DP using parallelization and distribution methods. Zhang *et al.* (2016) proposed an approximate algorithm named LSH-DDP, which exploits locality-sensitive hashing (LSH) and parallelizes DP with a MapReduce model. Li *et al.* (2016) accelerated the computation of distance and threshold in DP with a graphics processing unit (GPU). The implementation is based on JCuda and an acceleration only about 7 folds is achieved; however, the power of GPU computation has not been effectively exploited.

In this paper, we propose a GPU-accelerated DP algorithm with compute unified device architecture DP (CUDA-DP). In this acceleration algorithm, we redesign the data structure of the data point array. We do not use the traditional array of structure (AOS), but the structure of array (SOA) form. Extracting data point features from the same dimension and putting them into a large array in which the features of the same dimension are stored contiguously, can exploit the global memory coalescing mechanism to avoid data access divergence. Considering the imprecision property of d_c, we introduce a sampling method to compute the approximate d_c and allow a trade-off between sampling and the binary search method. CUDA-DP also optimizes the process of computing δ. We use shared memory and perform reduction twice to obtain δ for each data point, avoiding circular dependencies. Compared with the central processing unit (CPU) based DP implementation, the CUDA-DP algorithm achieves an acceleration larger than 45 folds.

2 Density peak preliminaries

2.1 Introduction

DP is a new clustering algorithm proposed by Rodriguez and Laio (2014). The algorithm determines cluster centers based on the idea that cluster centers are surrounded by neighbors with a lower local density ρ, and at a relatively large distance δ from any other higher density points. The DP clustering process can be divided into five steps: (1) calculating the distance matrix, (2) estimating d_{c}, (3) computing the local density, (4) computing the minimum distance δ from a data point with higher density, and (5) finding the cluster centers and assigning the remaining data points to the corresponding cluster. The detailed DP algorithm is illustrated in Algorithm 1.

Algorithm 1 Density peak algorithm

Let $\{p_i\}_{i=1}^N$ be the data points;
Compute distance $\{d_{i,j}\}_{i,j=1}^N$;
Find a reasonable threshold d_{c};
Compute local density $\{\rho_i\}_{i=1}^N$;
Obtain distance $\{\delta_i\}_{i=1}^N$ and the index of the nearest point with higher density;
Decide cluster centers with relatively large ρ and δ;
Assign remaining points to the corresponding cluster centers.

To determine the cluster centers, two properties, local density ρ and distance δ, of each point should be calculated. The local density ρ_i of point i can be calculated by

$$\rho_i = \sum_{i \neq j}^{N} \exp\left(-\left(\frac{d_{i,j}}{d_{\mathrm{c}}}\right)^2\right), \tag{1}$$

where $d_{i,j}$ is the distance between data points i and j, and d_{c} is the threshold called the 'cutoff distance'. It should be noted that the definition of distance can be Euclidean distance, Manhattan distance, etc. There are also some other accurate measures to estimate the local density (Fukunaga and Hostetler, 1975; Cheng, 1995).

When the amount of data is small, the relative magnitude of the local density may be affected by the choice of d_{c}, whereas in large-scale data, the DP algorithm is strongly robust in the selection of d_{c}. Rodriguez and Laio (2014) suggested that one should choose a reasonable d_{c} so that the average number of neighbors with a higher local density is around 1% to 2% of the total number of points. To estimate the d_{c} that satisfies this requirement, an effective way is the binary search method.

The binary search method first finds a reasonable maximum value $d_{\mathrm{c_{max}}}$ and a minimum value $d_{\mathrm{c_{min}}}$ as a boundary. If the number of elements that are smaller than $(d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$ in the distance matrix is greater than $2\%N^2$, then $(d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$ is set to the new maximum value boundary $d_{\mathrm{c_{max}}}$; otherwise, it will be set as the new minimum value boundary $d_{\mathrm{c_{min}}}$. The detailed execution flow of the binary search method is shown in Algorithm 2.

Algorithm 2 Binary search method

Input: $d_{\mathrm{c_{max}}}, d_{\mathrm{c_{min}}}, p$ // $(d_{\mathrm{c_{max}}}, d_{\mathrm{c_{min}}})$ is the initial search // bound, and p controls the precision of d_{c}
Output: d_{c}

Let M be the distance matrix
while $d_{\mathrm{c_{max}}} - d_{\mathrm{c_{min}}} > p$ **do**
  $n \leftarrow \mathrm{CountIf}(M, (d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2)$ // count the // elements smaller than $(d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$
  if $n > 2\%N^2$ **then**
    $d_{\mathrm{c_{max}}} \leftarrow (d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$
  else
    $d_{\mathrm{c_{max}}} \leftarrow (d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$
  end if
end while
return $(d_{\mathrm{c_{min}}} + d_{\mathrm{c_{max}}})/2$

The distance δ_i of point i is defined as

$$\delta_i = \min_{j|\rho_j > \rho_i} (d_{i,j}), \tag{2}$$

where j is the index of the point that is the nearest neighbor in the dataset in which the data point has a higher density than point i. That is, among all the points with a higher density than point i, point j has the shortest distance (δ_i) from point i. For the point whose local density is the largest in the dataset, it is obvious that the point is a cluster center; thus, δ for this point is defined as $\delta_i = \max_j (d_{i,j})$. The points with a large δ and a relatively high ρ are considered as cluster centers; points with a relatively high ρ and a low δ are isolated.

The algorithm assigns the remaining point after finding the cluster centers in one step. Let q_i denote the index of sorted $\{\rho_i\}_1^N$ in descending order. Then we assign each point to the same cluster as its nearest neighbor from data points q_1 to q_N. After this

process, each of the data points will be assigned to the cluster to which it belongs, and the clustering process is completed.

2.2 Execution time analysis of the density peak

Given a dataset of size N, the time complexities of computing the distance matrix, estimating d_c, and calculating the local density and distance δ are all $O(N^2)$, whereas the computational complexity of assigning labels is only $O(N)$. To study the time overhead of the four major computational processes in the sequential algorithm, we test each part of the sequential program with different dataset sizes. The test results indicate that the execution time of the sequential program rises non-linearly with the number of data points (Fig. 1a). Among the four parts of the sequential program, the calculation of local density and the time overhead of estimating d_c grow fastest. The time required for computing threshold d_c is 600 ms and 2500 ms when the sizes of datasets are 10 000 and 20 000, respectively. The total execution time of the program can be calculated from the four components. According to Fig. 1a, we can obtain the time distribution of different parts easily, and component d_c consumes more than 30% of the total execution time. We further test the effect of the dimensions of the data points on the time overhead of computing the distance matrix. The results are shown in Fig. 1b. We can see that as the data dimensions are multiplied, the time of computing the distance matrix grows linearly.

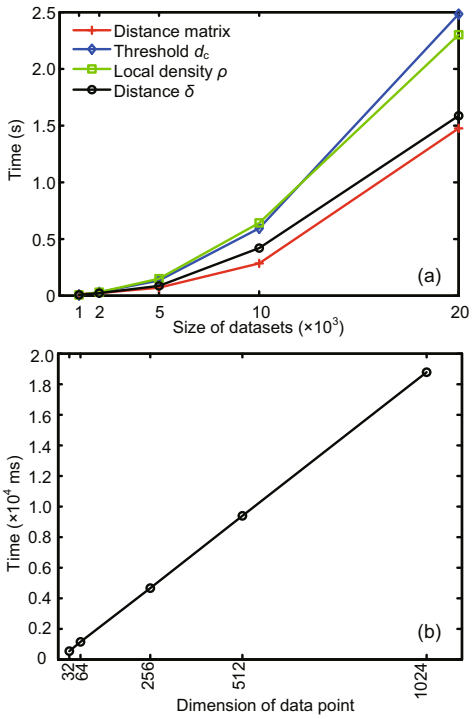

Fig. 1 Time overheads of the sequential program for 2D datasets with different sizes (a) and under different data point dimensions (b) (Each of the datasets contains 1024 points)

2.3 Parallel potential and challenges

Compared with that of a traditional CPU, GPU computation capability is significantly higher. As mentioned earlier, the distance matrix and local density computations require a mass of floating-point operations, which gives GPU a huge opportunity to accelerate these computation-intensive components. Because the DP algorithm shows a good data-level parallelization, a large number of threads start processing different points simultaneously with a GPU. Moreover, the GPU memory system is more efficient than that of CPU. The bandwidth of GPU's global memory is several times that of CPU's main memory. This gives GPU an advantage when accessing large-scale data. In the DP algorithm, finding cluster centers is based on the distance matrix, meaning that the large distance matrix with size N^2 will be read and written at least once. Higher memory bandwidth can significantly reduce the data access overhead.

As explained in the previous subsection, the time taken to calculate parameter d_c is relatively long. Although the binary search method can quickly find d_c in a reasonable range, it needs several iterations to reduce the d_c selection range until the accuracy requirement is met. In each iteration, counting the number of elements whose value is less than d_c over the whole distance matrix consumes much time. In fact, the counting process can be sped up significantly with the GPU's shared memory.

The same operation is executed for each data point when computing ρ and δ. The local density of each data point i requires transform and the reduce operations of the distance from point i to all of the other points. The process of calculating δ is to loop through all the data points to find the distances from the higher density points. Because calculating ρ and δ for a data point does not affect the calculation of other data points, we can calculate these two metrics for the N data points in parallel.

Each element $d_{i,j}$ of the distance matrix is calculated by reading the M-dimensional data of data points i and j. The computations of the elements of the distance matrix are independent of each other; thus, parallel implementation is appropriate for the computation of the distance matrix; the acceleration effect of parallel implementation is more remarkable as data amount N and data dimension M increase.

Parallelization with GPU requires solving the following issues: Calculating the distance matrix requires two memory accesses for each element; however, the computational overhead of a pair of data points is not large. As a result, the computational resource utilization rate of GPU is not high, and the memory access cost becomes the bottleneck in calculating the distance matrix. Minimizing the number of memory accesses is the key to improving the efficiency of parallel implementation. In other GPU-based implementations, a lot of duplicate memory accesses waste computing resources. Moreover, d_c is a robust parameter; thus, it is not necessary to compute the exact d_c in large-scale datasets. The process of estimating d_c requires a trade-off between accuracy and efficiency. Furthermore, computing the local density of each data point and the distance δ needs to be repeated N times, and there are circular dependencies in these two processes. Efficient parallel implementations should avoid these loops.

2.4 Related works

Previous studies on the parallelization of clustering algorithms include parallel clustering algorithms based on distributed systems and GPU-based parallel accelerations. For the characteristics of distributed systems, researchers have applied some new data structures to improve parallelism in the clustering algorithm (Arlia and Coppola, 2001; Xu *et al.*, 2002). Some researchers have studied boosting the performance of parallel clustering algorithms by improving the load balance of distributed systems (Garg *et al.*, 2006). In widely used distributed architectures, such as MapReduce and Spark, some researchers have developed efficient parallel clustering algorithms (Zhao *et al.*, 2009; Sarazin *et al.*, 2014; Meng *et al.*, 2016). Because GPUs are widely applied to parallel acceleration, a number of studies have focused on the use of GPU-accelerated clustering algorithms. A GPU-based K-means implementation (Shalom *et al.*, 2008) achieves 7- to 22-fold gain by avoiding the need for data and cluster information transfer between GPU and CPU. G-DBSCAN (Andrade *et al.*, 2013) can be 100 times faster than the sequential version by using graphs to explore various parallelization opportunities.

Similar to our study, fast search and find of density peaks (FSFDP) focuses on paralleling the calculation of the distance matrix and the d_c using GPU (Li *et al.*, 2016). Compared with our work, the work of Li *et al.* (2016) has no optimization of data structure, and the acceleration of distance matrix calculation is only 4.39-fold in a 15-dimensional dataset containing 10 126 data points. Unlike our research, which uses an approximate method to estimate d_c, the work of Li *et al.* (2016) tries to find a more accurate d_c, which is not necessary for the final clustering result. Li *et al.* (2016) achieved a 15.75-fold acceleration on the same dataset mentioned above.

3 Parallelization on GPU and optimization

3.1 Parallelization on GPU

Based on the analysis in Section 2, we initially propose the GPU parallel strategy, and discuss the opportunity for further optimization.

Distance matrix $\boldsymbol{M}$: A simple and intuitive model that exploits thread parallelism involves launching one thread to calculate the corresponding distance. Fig. 2 illustrates the idea of a memory accessing strategy for calculating distances $d_{i,j}, d_{i+1,j}, ...$ within a thread block. Each thread accesses two data points and calculates the Euclidean distance between them.

In fact, the calculation of the distance matrix in row i reads the same data point i; thus, the calculation of the same row distance matrix does not need to access the global memory $2N$ times. Because the data in the shared memory are visible to all threads in the same thread block, we allocate the shared memory to store the data points that are shared within the thread block. To increase the compute-to-global memory access ratio in a single thread, we make one thread handle x elements of the distance matrix. Fig. 3 illustrates the memory accessing strategy with the shared memory. As shown in Fig. 3, x data points are read to shared memory in advance. Each thread in the thread block reads one

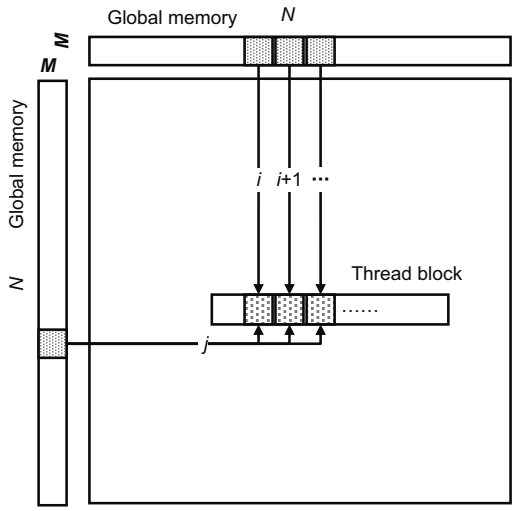

Fig. 2 Sketch for computing distance matrix

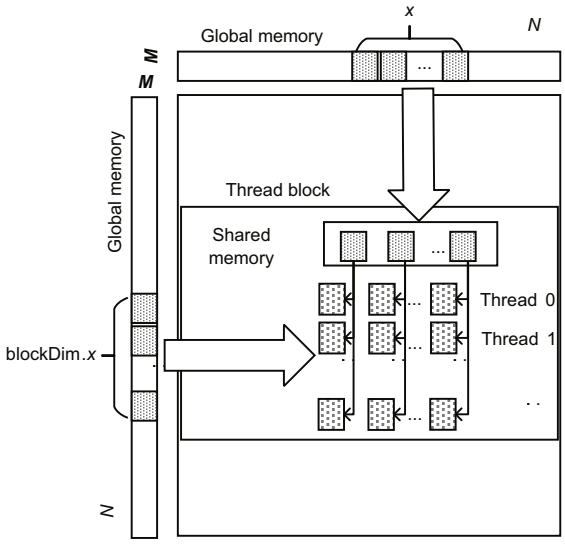

Fig. 3 Sketch of the computing distance matrix using shared memory (blockDim.x: dimension of the thread block in GPU)

data point from the shared memory and the corresponding data point from global memory to calculate the distance simultaneously. When the procedure is completed, these threads use the next data point in shared memory to calculate the rest of the elements of the distance matrix. Because the shared memory bandwidth is more than 10 times larger than the global memory, this strategy will significantly reduce the program's memory access time. Moreover, the thread uses the data points stored in the registers to calculate multiple distances, making it effective in exploiting the floating-point computing power of the GPU.

Estimated cutoff distance d_c: In CUDA-DP, for each iteration of the binary search method, we launch a kernel function to do the counting job. In the kernel function, each thread block counts the elements in one row of the distance matrix when distance $d_{i,j}$ is smaller than $(d_{c_{\min}} + d_{c_{\max}})/2$. We also use shared memory to store the partial sum in the counting process. Finally all the thread blocks obtain the total number of the points that meet the requirement through the atomic-add operation.

Local density ρ: We create one thread block for each data point, while each thread block allocates an array in the shared memory. The size of the shared memory array is the same as that of the thread block. Each thread saves a temporary value of local density ρ_i calculated by Eq. (1) in its corresponding element in the array. After that, the local density of each data point is calculated by a reduction operation on the shared memory array.

Distance δ: For each data point, the computation flow of δ is illustrated in Algorithm 3.

Algorithm 3 Computing δ for data point i

Input: Distance matrix $\boldsymbol{M}$ and density array $\boldsymbol{\rho}$

flag← **false**
$\delta \leftarrow \infty$
index $\leftarrow 0$ // 'index' stores the index of the nearest // point with a higher density
for $j \leftarrow 0$ to N **do**
  if $\rho[i] < \rho[j]$ **then**
    if !flag **then**
      $\delta \leftarrow M[i][j]$
      index $\leftarrow j$
      flag $\leftarrow$ **true**
      continue
    end if
    if $\delta < M[i][j]$ **then**
      $\delta \leftarrow M[i][j]$
      index← j
    end if
  end if
end for

For each data point, Algorithm 3 finds its nearest point with a higher density and stores its corresponding index. There are circular dependencies, and the maximum distance for the maximum density point needs to be searched again. As the hardware in GPU lacks loop and branch instruction optimization, if we create a single thread to compute distance δ, the parallelism would be low. However, it would also take a long time for thread execution.

In the optimized δ computation flow, we obtain δ and the corresponding index by two reduction operations. Similar to the reduction operation in computing the local density, we allocate a shared memory array to implement reduction, and the size of the array is the same as the size of the thread block.

The reduction process first obtains the maximum distance from each data point to other points, and then initializes the array in the shared memory used for parallel lookup of δ with this maximum value. Thus, in the process of a parallel search for δ, the δ value of the maximum-density point will be the initial value of the array, while the δ's of other data points and their corresponding indices will be reduced to the first element in the shared memory array.

3.2 Data structure optimization

Because the data points are stored in a memory system with AOS form, threads within the same thread block span multiple memory spaces when accessing the same dimension of adjacent data points. The data access is illustrated in Fig. 4a. In the hardware implementation of GPU, the thread block executes the program by grouping the threads into units of warps. Grouping of threads into warps is relevant not only to computation, but also to memory access. The device coalesces memory loads and stores that are issued by threads of a warp into as few transactions as possible. The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to that of cache lines which are necessary to service all of the threads of the warp (NVIDIA, 2016). For example, for 32D floating-point data, a single coalesced memory access of a warp can read only one data point. To allow all threads within a warp to calculate the distance in parallel, the program also needs memory access operations several times. This means that the array of structures will result in accessing a larger cache and more instances of cache line access, which increases the number of concurrent accesses. This is the reason why the basic GPU parallelization method can still be optimized.

As illustrated in Fig. 4a, the traditional way to organize data points is to use AOS. It means that the data of one dimension is contiguous with its neighbor dimension, but is far from the same dimension of the next data point with the same dimension in memory. Actually, multiple threads execute the calculation of data points of the same dimension simultaneously. To obtain data of the same dimension in one memory access as much as possible, we need to store them contiguously.

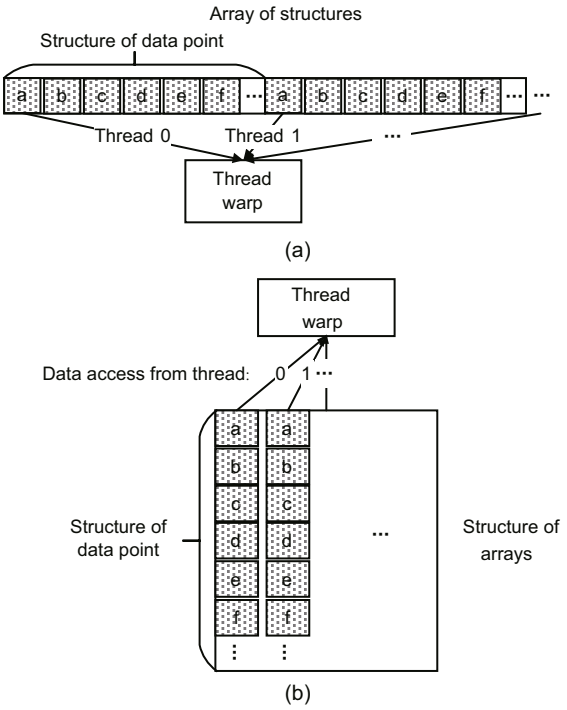

Fig. 4 Memory access to array of structures (a) and structure of arrays (b)

We reorganize the data structure from an AOS to SOA to address this problem. SOA is a method of storing an array of multidimensional data points. As shown in Fig. 4b, one row represents an array structure containing the same dimension of all data points, and the size of this array is equal to the number of data points. Although the values for different dimensions from one data point are separated in a different array, it has no negative influence on the acceleration effect. In our CUDA-DP implementation, we store the data point as an SOA in the memory when we get data, and then transfer it into the global memory of GPU. SOA is located in global memory. As the data point in shared memory is copied from global memory, the data structure in shared memory is also an SOA. Only in the step of computing the distance matrix, does the data point need to be accessed in global memory and SOA to be used. It is unnecessary to add adaptive coding for the algorithm.

In this approach, the values for the data points of the same dimension are stored continuously. For example, 32D floating-point values (4 bytes) are buffered in a cache line of 128 bytes, and a single coalesced transaction can service that memory access. This obviously decreases the number of memory accesses and makes full use of the parallelism and concurrency of the thread warp.

3.3 Estimating d_c with sampling method

Although we try to exploit GPU to speed up the calculation process, multiple iterations of the binary search method can not be parallelized further. As the number of iterations increases, the additional overhead of invoking the kernel function will be greater. Fortunately, the choice of d_c has little effect on the order of local density in large-scale data; thus, the clustering results are robust to d_c selection. Based on this observation, we use the sampling method as the other way to estimate d_c to avoid this problem of the binary search method.

The sampling method estimates d_c by sorting a subset of the distance matrix in which the elements are randomly selected. Suppose we select x elements in each row of the distance matrix. We start N threads; one thread handles one row of matrix elements, and each thread generates x random integers in the range of 0 to N and stores the corresponding element in a global memory array. After that, we invoke the Thrust library to sort the global memory array and estimate d_c.

The choice between these two measures depends on the dataset. If the scale of the dataset is small, a high accuracy of d_c is the major consideration, and the binary search method will be a better choice. On the other hand, for a large-scale dataset, the sampling method can meet the requirements of accuracy and processing time of the algorithm.

4 Experimental evaluation

4.1 Experimental setup

In this section, we experiment with acceleration of the CUDA-DP algorithm. Then the optimization results of the GPU program before and after adjusting the data structure are compared. Finally, we test whether the estimated d_c affects the clustering effectiveness. The experiment platform is the NVIDIA Kepler K40m GPU, whose configuration used for evaluation is shown in Table 1. The acceleration comparison object is the sequential algorithms running in the Intel Xeon E5-2620 with six cores and 256 GB of memory. For 50 000 2D floating data points, 10 GB of memory space is required to store the distance matrix. As a result, memory space must be sufficient for DP.

Table 1 GPU (NVIDIA Kepler K40m) configuration

Parameter	Value
Peak performance (double-precision)	1.43 Tflops
Peak performance (single-precision)	4.29 Tflops
Warp size	32
GlobalMem	12 GB
CUDA core	2880
Core frequency	745 MHz
Memory bandwidth	288 GB/s

We use CUDA to implement CUDA-DP. CUDA provides a number of efficient function interfaces and helps reduce development difficulty. In the CUDA-DP algorithm, sorting, reduce, transform, and other operations can be implemented with the Thrust library.

The datasets used for the experiment are listed in Table 2. The first dataset (Zhang *et al.*, 1997) contains 100 000 2D data points. In the first part of the experiment, we use a series of subsets of this dataset to test the acceleration of CUDA-DP. In the second part, the five intermediate datasets (Franti *et al.*, 2006) collectively referred to as Dim-sets, are used to test the acceleration of the distance matrix for different data dimensions and to compare the acceleration effect of the optimization in the data structure. The last two datasets (Veenman *et al.*, 2002) are used to compare the clustering effects of the sampling method and the binary search method.

Table 2 Datasets for the experiment

Dataset	Dimension	Number of instances
BIRCH	2	100 000
Dim32	32	1024
Dim64	64	1024
Dim256	256	1024
Dim512	512	1024
Dim1024	1024	1024
D31	2	3100
R15	2	600

4.2 Acceleration of density peak algorithm with CUDA

The experiment in this subsection consists of two parts. First, we test the acceleration effect of the CUDA-DP algorithm under different scales, some of which are subsets of BIRCH, which is a 2D dataset (Table 2). Limited to the memory of a single GPU, the maximum size of the experimental dataset reaches 46 000 data points. A sequential program for the experiments allocates an array to store data and minimize unnecessary function calls to reduce the time overhead. Because parameter d_c obtained from the binary search method is more accurate (for more details see Section 4.4), we use the binary search method to calculate d_c in the CUDA-DP acceleration test. Each thread block in the test starts with 128 threads. We record CUDA-DP acceleration and the four computing steps on different dataset sizes. The acceleration of each step with datasets of different sizes is shown in Fig. 5a. When the amount of data is 46 000, computing the distance matrix achieves a 45-fold acceleration. Meanwhile, the speed of d_c calculation achieves 22 times the speed ratio, and computing speed δ achieves a 64-fold acceleration. The local density computation is dramatically increased by 150 folds. In the sequential program, calculating the local density invokes the exponential function N^2 times. However, the exponential function is just invoked once per thread in GPU, and the stream processor executes the exponential functions faster. Therefore, in the calculation of local density, the CUDA-DP acceleration effect is significant.

Fig. 5b represents the acceleration of the whole application when compared with the sequential program, including the four steps illustrated in Fig. 5a and other overhead (e.g., the I/O and some serial codes). As shown in Fig. 5b, the acceleration of CUDA-DP initially increases linearly with the number of datasets. After reaching 10 000 points, the acceleration ratio still increases, but not significantly. When the data size reaches 46 000, our implementation can accelerate the execution by a factor of up to 45, compared to the sequential version of DP.

Then, we test the acceleration effect of different thread block sizes. This experiment uses a subset of BIRCH containing 20 000 data points. We still use the binary search method to calculate d_c. The experiment starts with 32 threads per thread block

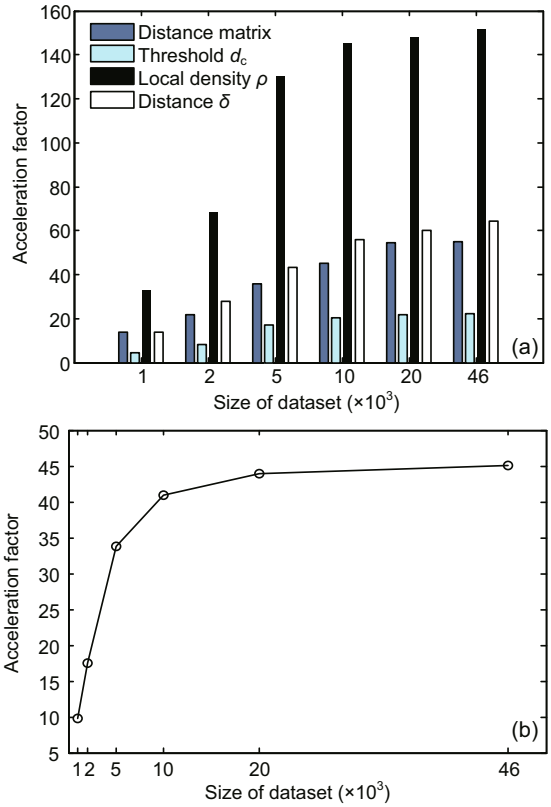

Fig. 5 Acceleration of the four computing procedures (a) and of the density peak algorithm with CUDA including the computing process and data transferring operations (b) for 2D datasets of different sizes with a maximum data size of 46 000

and doubles the number of threads until it reaches 1024, which is the maximum number of threads for a 1D thread block. A warp in the K40m GPU consists of 32 threads; thus, the thread block size in our experiments is a multiple of 32. The results of this experiment are shown in Fig. 6.

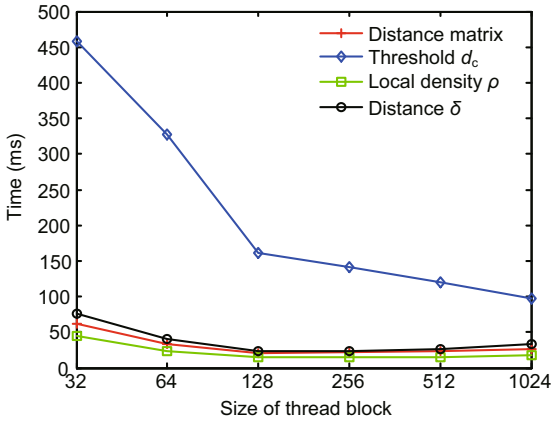

Fig. 6 Time taken to execute the four computational procedures with thread blocks of different dimensions

As we know, threads within the same thread block share data through shared memory, and every stream-multiprocessor (SM) has its own shared memory in GPU; thus, threads within a thread block must be executed within the same SM. Warp is the smallest unit of CUDA program execution. While a warp is accessing memory, another warp can be executed in SM at the same time. Therefore, a thread block consisting of multiple warps can make full use of SM's computing and storage resources. It can be observed that the computation time decreases as the thread block size increases from 32 to 128. However, as the size of the thread block continues to increase, the number of thread blocks that can be scheduled on an SM is reduced, affecting the scheduling efficiency of SM. Thus, when the size of the thread block continues to increase to 1024, the time overheads of computing the distance matrix, ρ, and δ do not continue to decrease, but increase slightly.

It is worth noting that the time overhead of computing d_c decreases as the thread block size increases. SM has no branch prediction part and no error recovery mechanism. Therefore, when a branch is encountered, SM must wait for branch address calculation to complete the subsequent instruction and continue to work. This means that when each thread unavoidably performs some branch operations, serial operation is executed sequentially and the branch overhead cannot be hidden by branch prediction. In CUDA-DP, the calculation of d_c is very simple for each thread, while the other three computation portions have more loops and branch instructions. Thus, when the thread block size exceeds 128, the performance gain from increase in the number of threads in these three portions is less than the loss.

4.3 Effect of data structure optimization

In this section, we test the impact of optimization of the data structure on accelerating the calculation of the distance matrix. Calculating the distance matrix is also a time-consuming part of the sequential program. To make full use of warp concurrent memory access characteristics and reduce the number of memory accesses, this study optimizes AOS into SOA. To test the effectiveness of this optimization method, we use the Dim-sets in which data points have various data dimensions for testing. The number of data points in the dataset is 1024, and the data dimension increases from 32 to 1024. The results of this experiment are shown in Fig. 7.

The results show that the program with SOA achieves an acceleration factor of 5.4, compared to that with AOS in 32-dimensional data. As the data point dimension increases, the acceleration also increases. The acceleration reaches 9.2-fold when the dimension of data points is 1024. In AOS, the access memory stride between adjacent threads increases when the dimension increases. In this case, the probability of the missing cache will increase, resulting in more memory access overhead. As a result, when the data dimension increases, the optimization effect will be better.

4.4 Trade-off between binary search method and sampling method

In this section, we compare the effectiveness and efficiency of the sampling and binary search methods. We observe the clustering results from these two methods on different-scale data to verify their effectiveness. By testing the time overhead of the two methods, we compare their efficiencies. We randomly selected 1% distance samples. The test results on different datasets are shown in Table 3.

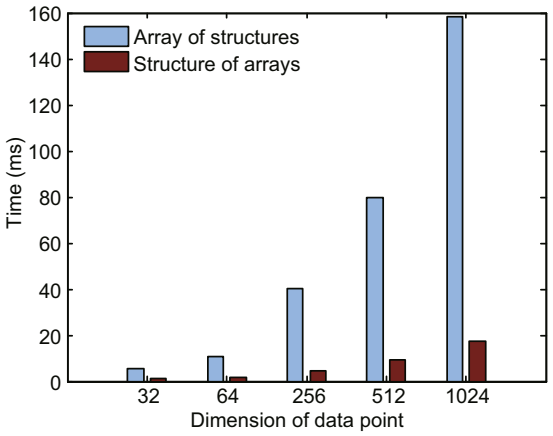

Fig. 7 Running time under datasets with different dimensions using array of structures and structure of arrays (All datasets contain 1024 points)

Table 3 Comparison between the binary search method and sampling method

Dataset	Time (ms)		Value	
	Binary search	Sampling	Binary search	Sampling
R15	1.3	1.0	0.347	0.393
D31	3.8	1.1	1.151	1.073
BIRCH (5000)	11.8	1.4	7.653	7.364

Using sampling to estimate d_c will contribute to inevitable bias. The deviation is more obvious in small-scale datasets. As shown in Table 3, on R15, the value deviation between the sampling method and the binary search method is 0.046, which is 13% of the value from the binary search method. The value deviation from the experiment on BIRCH (5000) is significantly lower than that on R15, with only 3.8%. Concerning the time overhead, when the size is 600, the sampling method takes 1.0 ms to evaluate d_c, while binary search takes 1.3 ms, causing little reduction in time. With a larger dataset, the time overhead of the sampling method increases slightly, while it increases much faster for the binary search method. The ampling method takes only 1.4 ms, and can save nearly 90% of the time overhead compared with the binary search method when the number of data points is 5000. As indicated in Section 2.2, the time complexity of calculating d_c increases non-linearly with the number of data points, and the time deviation between these two methods will be more significant on large-scale datasets. According to the results of the experiment, when the amount of data is further increased, it can be predicted that the sampling method will speed up the calculation of d_c more significantly and accurately.

We obtain the decision graphs and clustering results to assess the impact of sampling on the clustering results. The coordinates in these decision graphs are based on ρ and δ. The cluster centers are determined by choosing the points with large ρ and δ. Figs. 8 and 9 show the decision graphs obtained by these two methods on datasets R15 and D31, respectively.

Figs. 8a and 9a are obtained by the binary search method, while Figs. 8b and 9b are drawn using (ρ, δ) obtained by the sampling method. We can see that the two decision graphs are basically the same. The number of density peaks selected for the cluster center is the same. Although there are some differences in the coordinates (ρ, δ) in the two graphs, they do not affect the selection of the cluster centers and the assignment of the remaining data points to the cluster. The clustering results corresponding to the decision graph are shown in Figs. 10 and 11. These clustering results are highly consistent, proving that the estimated d_c's are valid for the CUDA-DP algorithm.

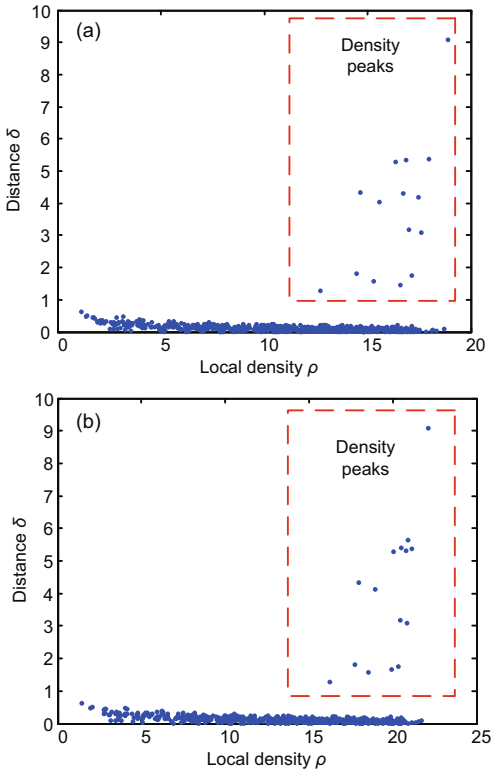

Fig. 8 Decision graphs obtained by the binary search method (a) and the sampling method (b) on R15

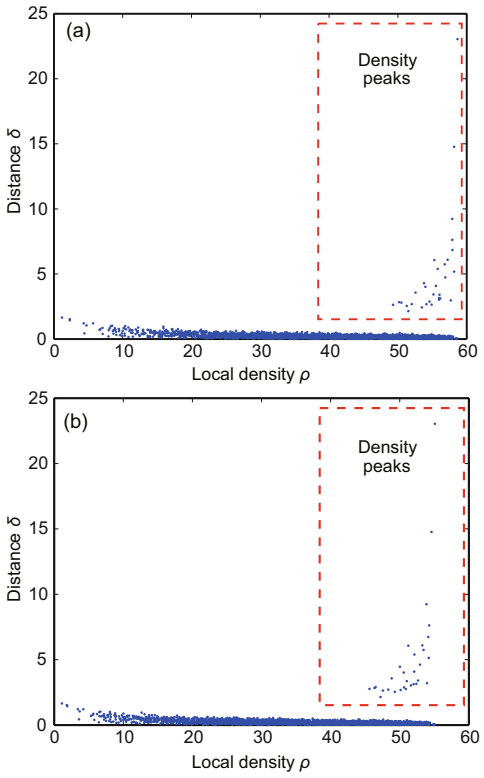

Fig. 9 Decision graphs obtained by the binary search method (a) and the sampling method (b) on D31

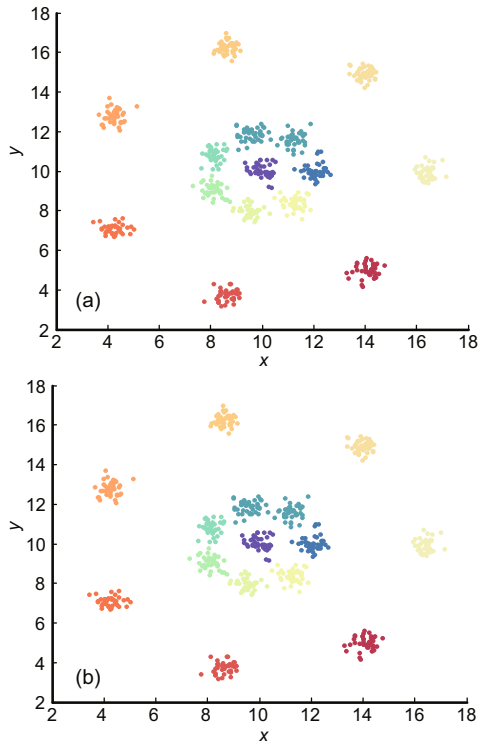

Fig. 10 Clustering results obtained by the binary search method (a) and the sampling method (b) on R15

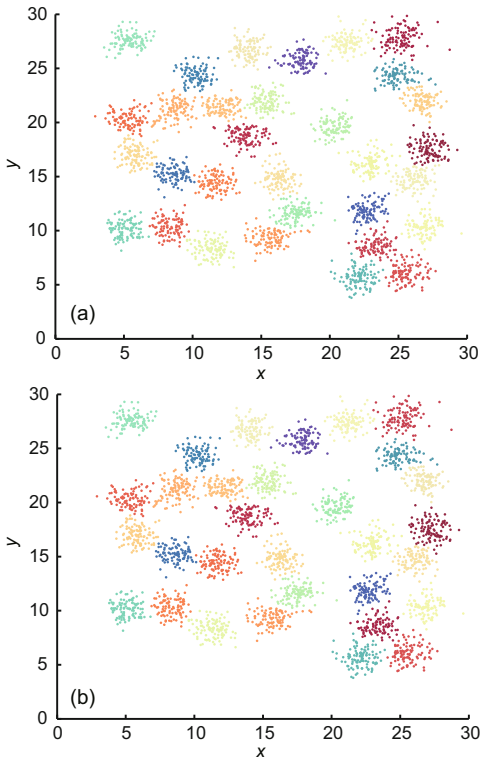

Fig. 11 Clustering results obtained by the binary search method (a) and the sampling method (b) on D31

5 Conclusions

In this paper, we propose a parallel implementation DP algorithm CUDA-DP, which is based on a GPU platform to reduce the time overhead associated with the DP algorithm. Based on the theoretical analysis of DP, we designed a parallel implementation scheme that makes full use of GPU hardware characteristics. In addition, in CUDA-DP, we reorganized the data structure in the GPU global memory. It significantly reduces the memory access overhead. The validity and efficiency of the threshold d_c were also compared between the binary search-and-sampling methods. In the experiment, compared with the sequential program, CUDA-DP achieves an acceleration factor larger than 45 folds.

CUDA-DP is limited by the memory size of a single GPU. The next step is focusing on extending CUDA-DP to multiple GPUs. In the future work, it can be used in resource allocation (Li *et al.*, 2015), text clustering, image processing, and other specific areas to accelerate these applications.

References

Andrade, G., Ramos, G., Madeira, D., *et al.*, 2013. G-DBSCAN: a GPU accelerated algorithm for density-based clustering. *Proc. Comput. Sci.*, **18**:369-378. http://dx.doi.org/10.1016/j.procs.2013.05.200

Arlia, D., Coppola, M., 2001. Experiments in parallel clustering with DBSCAN. European Conf. on Parallel Processing, p.326-331. http://dx.doi.org/10.1007/3-540-44681-8_46

Begum, N., Ulanova, L., Wang, J., *et al.*, 2015. Accelerating dynamic time warping clustering with a novel admissible pruning strategy. Proc. 21st ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, p.49-58. http://dx.doi.org/10.1145/2783258.2783286

Cheng, Y., 1995. Mean shift, mode seeking, and clustering. *IEEE Trans. Patt. Anal. Mach. Intell.*, **17**(8):790-799. http://dx.doi.org/10.1109/34.400568

Dean, K.M., Davis, L.M., Lubbeck, J.L., *et al.*, 2015. High-speed multiparameter photophysical analyses of fluorophore libraries. *Anal. Chem.*, **87**(10):5026-5030. http://dx.doi.org/10.1021/acs.analchem.5b00607

Ester, M., Kriegel, H.P., Sander, J., *et al.*, 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. Proc. Int. Conf. on Knowledge Discovery and Data Mining, p.226-231.

Franti, P., Virmajoki, O., Hautamaki, V., 2006. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Trans. Patt. Anal. Mach. Intell.*, **28**(11):1875-1881. http://dx.doi.org/10.1109/TPAMI.2006.227

Fukunaga, K., Hostetler, L., 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Trans. Inform. Theory*, **21**(1):32-40. http://dx.doi.org/10.1109/TIT.1975.1055330

Garg, A., Mangla, A., Gupta, N., *et al.*, 2006. PBIRCH: a scalable parallel clustering algorithm for incremental data. 10th Int. Database Engineering and Applications Symp., p.315-316. http://dx.doi.org/10.1109/IDEAS.2006.36

He, Y., Zhang, F., Li, Y., *et al.*, 2016. Multiple routes recommendation system on massive taxi trajectories. *Tsinghua Sci. Technol.*, **21**(5):510-520. http://dx.doi.org/10.1109/TST.2016.7590320

Kohonen, T., 1990. The self-organizing map. *Neurocomputing*, **78**(9):1464-1480. http://dx.doi.org/10.1016/S0925-2312(98)00030-7

Li, J., Li, D., Ye, Y., *et al.*, 2015. Efficient multi-tenant virtual machine allocation in cloud data centers. *Tsinghua Sci. Technol.*, **20**(1):81-89. http://dx.doi.org/10.1109/TST.2015.7040517

Li, M., Huang, J., Wang, J., 2016. Paralleled fast search and find of density peaks clustering algorithm on GPUs with CUDA. 17th IEEE/ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, p.313-318. http://dx.doi.org/10.1109/SNPD.2016.7515918

MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. on Mathematical Statistics and Probability, p.281-297.

Meng, X., Bradley, J., Yavuz, B., *et al.*, 2016. MLlib: machine learning in Apache Spark. arXiv:1505.06807v1. https://arxiv.org/abs/1505.06807

NVIDIA, 2016. CUDA C Best Practices Guide v8.0. http://developer.nvidia.com [Accessed on Nov. 22, 2016].

Park, H.S., Jun, C.H., 2009. A simple and fast algorithm for K-medoids clustering. *Exp. Syst. Appl.*, **36**(2):3336-3341. http://dx.doi.org/10.1016/j.eswa.2008.01.039

Rasmussen, C.E., 2000. The infinite Gaussian mixture model. *Adv. Neur. Inform. Proc. Syst.*, **12**:554-560.

Rodriguez, A., Laio, A., 2014. Clustering by fast search and find of density peaks. *Science*, **344**(6191):1492-1496. http://dx.doi.org/10.1126/science.1242072

Sarazin, T., Azzag, H., Lebbah, M., 2014. SOM clustering using Spark-MapReduce. IEEE Int. Parallel and Distributed Processing Symp. Workshops, p.1727-1734. http://dx.doi.org/10.1109/IPDPSW.2014.192

Shalom, S.A., Dash, M., Tue, M., 2008. Efficient k-means clustering using accelerated graphics processors. Int. Conf. on Data Warehousing and Knowledge Discovery, p.166-175. http://dx.doi.org/10.1007/978-3-540-85836-2_16

Veenman, C.J., Reinders, M.J.T., Backer, E., 2002. A maximum variance cluster algorithm. *IEEE Trans. Patt. Anal. Mach. Intell.*, **24**(9):1273-1280. http://dx.doi.org/10.1109/TPAMI.2002.1033218

Xu, R., Wunsch, D., 2005. Survey of clustering algorithms. *IEEE Trans. Neur. Netw.*, **16**(3):645-678. http://dx.doi.org/10.1109/TNN.2005.845141

Xu, X., Jäger, J., Kriegel, H.P., 2002. A fast parallel clustering algorithm for large spatial databases. *In:* Guo, Y., Grossman, R. (Eds.), High Performance Data Mining: Scaling Algorithms, Applications and Systems. Springer US, Boston, p.263-290. http://dx.doi.org/10.1007/0-306-47011-X_3

Zamuner, S., Rodriguez, A., Seno, F., *et al.*, 2015. An efficient algorithm to perform local concerted movements of a chain molecule. *PLOS ONE*, **10**(3):1-27. http://dx.doi.org/10.1371/journal.pone.0118342

Zhang, T., Ramakrishnan, R., Livny, M., 1997. BIRCH: a new data clustering algorithm and its applications. *Data Min. Knowl. Discov.*, **1**(2):141-182. http://dx.doi.org/10.1023/A:1009783824328

Zhang, Y., Chen, S., Yu, G., 2016. Efficient distributed density peaks for clustering large data sets in MapReduce. *IEEE Trans. Knowl. Data Eng.*, **28**(12):3218-3230. http://dx.doi.org/10.1109/TKDE.2016.2609423

Zhao, W., Ma, H., He, Q., 2009. Parallel k-means clustering based on MapReduce. IEEE Int. Conf. on Cloud Computing, p.674-679. http://dx.doi.org/10.1007/978-3-642-10665-1_71