Future Generation Computer Systems 141 (2023) 399–413

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Improving Density Peaks Clustering through GPU acceleration

Zhuojin Liu, Shufeng Gong, Yuxuan Su, Changyi Wan, Yanfeng Zhang *, Ge Yu

School of Computer Science and Engineering, Northeastern University, China

ARTICLE INFO

Article history:
Received 3 June 2022
Received in revised form 23 November 2022
Accepted 26 November 2022
Available online 29 November 2022

Keywords:
Density peaks clustering
GPU
VP-tree
Parallelization
Dynamic clustering
Streaming data

ABSTRACT

Density Peaks Clustering (DPC) is a recently proposed clustering algorithm that has distinct advantages over existing clustering algorithms, which has already been used in a wide range of applications. However, DPC requires computing the distance between every pair of input points, therefore incurring quadratic computation overhead, which is prohibitive for large data sets. To address this efficiency problem, we propose to use GPU to accelerate DPC. We exploit a spatial index structure VP-Tree to efficiently maintain the data points and propose a GPU-friendly parallel VP-Tree construction algorithm. Based on the constructed VP-Tree, we propose a GPU-Accelerated DPC algorithm *GDPC*, in which the all-pair computation in DPC is greatly accelerated. Furthermore, in order to process dynamic evolving datasets, we propose an incremental GDPC algorithm, *Incremental GDPC*. Our results show that GDPC can achieve over 5.3-148.9X acceleration compared to the state-of-the-art GPU-based, multicore-based, and distributed DPC implementations, a 2.3-40.5X acceleration compared to the state-of-the-art incremental DPC algorithm.

1. Introduction

Data clustering is one of the most fundamental problems in many real-world applications, such as recommender systems, social networks, image processing, and bioinformatics. Basically, it groups a set of objects based on the similarities of objects such that objects in the same group (i.e., cluster) are more similar to each other than to those in other groups. Many different clustering algorithms have been proposed in the literature such as Kmeans [1] and DBSCAN [2]. There are also many research efforts to improve the efficiency of clustering to handle massive data [3–5].

Density Peaks Clustering (DPC) [6] is a novel clustering algorithm proposed recently. Given a set of points, DPC computes two metrics for every point p: (i) the local density ρ, which is the number of points within a specified distance from p; and (ii) the dependent distance δ, which is the minimum distance from p to other points with higher densities. It is observed that the center of a cluster sees the highest local density among its neighboring points (i.e., density center), and has a relatively large distance from other points with higher densities (i.e., far away from other density centers). Thus, cluster centers can be identified as points with both large ρ and large δ. With these identified cluster centers and the density center dependency trees extracted during the dependent distance δ's computation, the point-to-center relationship, or in other words, the point-to-cluster assignment (clustering results), can be discovered.

Compared with previous clustering algorithms, DPC has many advantages. (1) Unlike Kmeans, DPC does not require a pre-specified number of clusters. (2) DPC does not assume the clusters to be "balls" in space and supports arbitrarily shaped clusters. (3) DPC is more deterministic, since the clustering results have been shown to be robust against the initial choice of algorithm parameters. (4) The extraction of (ρ, δ) provides a two-dimensional representation of the input data, which can be in very high dimensions, so that it is easier for users to gain new insights from the two-dimensional representation plot of the data. Due to its effectiveness and novelty, DPC has already been employed in a wide range of applications, such as neuroscience [7], geoscience [8], and computer vision [9].

While DPC is attractive for its effectiveness and its simplicity, the application of DPC is limited by its computational cost. In order to obtain the density values ρ, DPC computes the distance between every pair of points. That is, given N points in the input data set, its computational cost is $O(N^2)$. Moreover, in order to obtain the dependent distance values δ, a global sort operation on all points based on their density values (with computational cost $O(Nlog(N))$) and $\frac{N(N-1)}{2}$ compare operations are required. As a result, it can be very time consuming to perform DPC on large data sets.

In the past few years, several research efforts have been put on accelerating DPC. LSH-DDP [3], EDDPC [10] and FDDP [11] leverage distributed approaches to help DPC handle large scale datasets. EDMStream [12] improves DPC by efficiently maintaining a novel in-memory dependent-tree structure. Ex-DPC and

* Corresponding author.
E-mail address: zhangyf@mail.neu.edu.cn (Y. Zhang).

https://doi.org/10.1016/j.future.2022.11.033
0167-739X/

S-Approx-DPC [13] accelerate DPC efficiency by leveraging multi-core processing.

The recent advance in GPU technology is offering great prospects in parallel computation [14,15]. With up to 80 GB GPU memory size [16], it is possible to use GPU to process large-scale data. There exist several related work have been devoted to accelerate DPC using GPU's parallelization ability. Li et al. [17] propose a thread/block model and shared memory designs to accelerate the distance matrix computation. CUDA-DP [18] also integrates GPU's parallelization ability and improves data locality to increase performance. However, these methods only focus on employing GPU's many-core features to accelerate DPC, without paying attention to utilizing spatial index structures that can greatly filter out a large number of unnecessary all-pair computations.

In this paper, we exploit a spatial index structure vantage point tree (VP-Tree) [19] to help efficiently maintain clustering data. With VP-Tree, data points are partitioned into "hypershells" with decreasing radius. Comparing with other spatial index structures (such as KD-Tree [20] and Ball-Tree [21]), VP-Tree is more appropriate in DPC algorithm, because the decreasing-radius hypershell structure in VP-Tree can well support the point density computation (that obtains nearby points within a predefined radius of a point) and the dependent distance computation (that obtains the distance to the nearest neighbor with higher density). Besides, VP-Tree is more suitable for clustering high-dimensional data [22]. More importantly, the construction of VP-Tree and the search of VP-Tree can be well parallelized to adapt to GPU's many-core architecture. Based on the GPU-based VP-Tree, we propose *GDPC* algorithm, where the density ρ and the dependent distance δ can be efficiently calculated by querying the index structure and lots of unnecessary distance measurements (between faraway points) can be avoided.

On the other hand, new data are produced every day and the data distribution may evolve overtime. As a result, the clustering results should continuously evolve correspondingly. The dynamically evolving feature of data drives us to seek an incremental clustering approach. Rather than re-performing DPC on the whole updated data sets, the incremental clustering algorithm leverages previous clustering results and only updates the affected point-to-cluster assignments. Considering the massive size of data sets, the incremental processing approach is desirable especially in production usage. Therefore, we further propose Incremental GDPC to support incremental clustering, which extends GDPC in the following aspects. (1) We design a GPU-friendly dynamic VP-Tree index update scheme that reduces the number of tree traversals and eliminates the write-write conflicts in GPU's many core computations. (2) Based on this dynamic VP-Tree index, we propose Incremental GDPC which can efficiently update the clustering results.

To sum up, we list our contributions in the following.

- **GPU-Accelerated VP-Tree Construction.** We design a vectorized VP-Tree layout to adapt to GPU architecture and take full advantage of GPU parallelism to speed up the VP-Tree index construction.
- **GPU-Accelerated DPC Implementation.** We propose to use the VP-Tree index to improve the efficiency of all-pair computation and rely on this index to avoid unnecessary computations in DPC's density evaluation and dependent distance evaluation with GPU's parallel computation support.
- **GPU-Accelerated Incremental DPC Update.** We provide incremental clustering support for dynamically evolving data by designing the GPU-friendly incremental update methods of density and dependent distance based on the dynamic VP-Tree.

We perform experiments on various real-world datasets and compare with a state-of-the-art GPU-based DPC algorithm CUDA-DP [18], a multicore-based parallel DPC algorithm S-Approx-DPC [13], and a distributed DPC implementation LSH-DDP [3]. Our results show that our GDPC can achieve 5.3–17.8X speedup over CUDA-DP, 43–148.9X speedup over S-Approx-DPC and 44.8–78.8X speedup over LSH-DDP. We further perform experiments on evolving datasets and compare with the state-of-the-art incremental DPC algorithm EDMStream [12]. Our results show that our Incremental GDPC can achieve 2.3–40.5X speedup over EDMStream.

The remainder of the paper is organized as follows. Section 2 describes the background on DPC and GPU's architecture. Section 3 presents the GPU-accelerated VP-Tree construction and query methods. Section 4 proposes our GPU-accelerated DPC algorithm GDPC. Section 5 introduces how we maintain the dynamic VP-Tree on GPU and proposes Incremental GDPC. Section 6 reports the experimental results. Section 7 discusses related work and Section 8 concludes the paper.

2. Background and preliminaries

In this section, we first review the standard Density Peaks Clustering (DPC) algorithm. We then introduce the background of GPU architecture and memory hierarchy.

2.1. DP clustering

Density Peaks Clustering (DPC) [6] is proposed based on two observations: (i) cluster centers are surrounded by neighbors with lower local densities, and (ii) cluster centers are at a relatively large distance from any points with higher local densities. Correspondingly, DPC computes two metrics for every data point: (i) its local density ρ and (ii) its distance δ from other points with higher density. DPC uses these two values to identify density peaks, i.e., cluster centers.

The *local density* ρ_i of data point p_i is the number of points whose distance to p_i is smaller than d_c

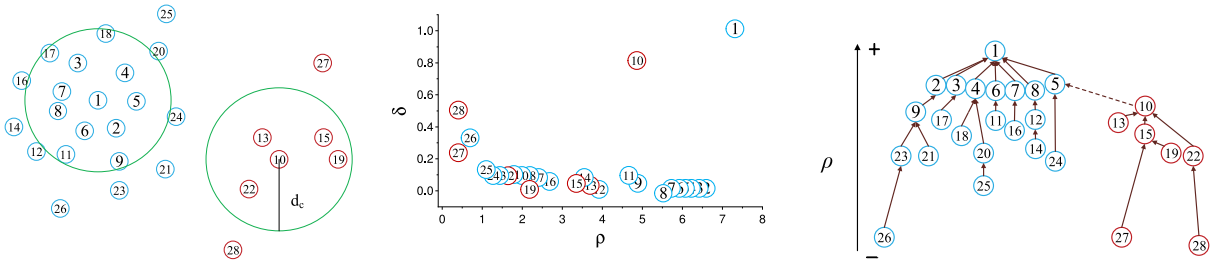
$$\rho_i = |\{p_j | d_{ij} < d_c\}| \tag{1}$$

where d_{ij} is the distance from point p_i to point p_j, and d_c is called the cutoff distance specified by users. The *dependent distance* δ_i of point p_i is computed as

$$\delta_i = \min_{j:\rho_j > \rho_i} (d_{ij}). \tag{2}$$

It is the minimum distance from point p_i to any other point whose local density is higher than that of point p_i. Suppose point p_j is point p_i's nearest neighbor with higher density, i.e., $p_j = argmin_{j:\rho_j>\rho_i}(d_{ij})$. We say that point p_i is *dependent* on point p_j and name point p_j as the *dependent point* of point p_i. If there are multiple nearest neighbors with higher density having the same distance to point p_i, we randomly pick one among them as the dependent point. If a point i has the highest density among all data points, i.e., $i = \arg\max_t \rho_t$, we set $\delta_i = \max_j(d_{ij})$ and name it as the *absolute density peak*. Then we can label every point with two values ρ and δ.

Based on these two sets of values, cluster centers can be identified as the points with both large ρ and large δ. The principle can be explained as follows. A point with small ρ_i is likely to be outliers or boundary points no matter how large its δ_i is. Next, we focus on the points with relatively large ρ_i to study the effect of δ_i. Small δ_i implies that point p_i is surrounded by at least one higher density neighbor. Anomalously large δ_i implies that point p_i is far from another dense area and point p_i itself could be the cluster center (density peak) of its own region, since there is no point



(a) Original data layout (b) Decision graph (c) Dependency relation

Fig. 1. An illustrative example of density peaks clustering.

with higher density around it. δ_i is much larger than the typical nearest neighbor distance only for points that are local or global maxima in density. Thus, the density peaks (i.e., cluster centers) are recognized as points for which the value of δ_i is anomalously large as well as large ρ_i.

Fig. 1 illustrates the process of DPC through a concrete example. Fig. 1(a) shows the distribution of a set of 2-D data points. Each point p_i is depicted on a *decision graph* by using (ρ_i, δ_i) as its x-y coordinate as shown in Fig. 1(b). By observing the decision graph, the *density peaks* can be identified in the top right region since they are with relatively large ρ_i and large δ_i. Since each point is only dependent on a single point, we can obtain a dependent tree [12] rooted by the absolute density peak as shown in Fig. 1(c). The height of each point implies the density. The length of each link implies the dependent distance. For each point there is a dependent chain ending at a density peak. After the density peaks (as cluster representatives) have been found, each remaining point is assigned to the same cluster as its dependent point.

2.2. General-purpose GPUs

Graphics Processing Units (GPUs) have been widely used in many fields for high-performance computing and big data processing. A GPU consists of many *stream multiprocessors* (SMs) and each has many CUDA cores with L1 cache. Several memory hierarchy layers such as the L2 cache and the DRAM *global memory* are shared by all SMs. The global memory is off-chip with the largest memory size (up to 24 GB) but with the slowest access speed. A group of 32 threads are organized into a *warp* and are executed in a single-instruction-multiple-data (SIMD) manner. Multiple warps compose a *thread block* that can be dispatched to a specific SM. The thread blocks further constitute a GPU kernel, which is a parallel function that can be invoked by the programmer and executed on all the SMs in a GPU.

GPUs provide powerful computation resources and high memory bandwidth. To fully utilize GPUs, three key points should be noticed during algorithm design: (1) Avoiding warp divergence. When different instructions among the threads are executed in a warp, *warp divergence* could occur, which is a major performance bottleneck that prevents data-intensive applications from gaining high performance in GPUs. An experienced programmer should avoid warp divergence and load imbalance across threads. (2) Avoiding memory divergence. *Coalesced memory access* should be used to make a batch of memory addresses requested by a warp fall within one GPU cache line, so that they can be served by a single memory transaction. Otherwise, multiple memory transactions will be required, which leads to memory divergence. Memory divergence reduces loading efficiency and throughput. (3) Reducing global memory access. Global memory also should be minimized by maximizing the use of shared memory on the device, because the latency of global memory access costs hundreds of clock cycles. Therefore, increasing on-chip data locality and reducing global memory access are critical for improving performance.

3. GPU-accelerated VP-Tree construction and query

VP-Tree is the key component in our proposed GPU-accelerated DPC algorithm. In this section, we first discuss why we prefer VP-Tree over other spatial index structures. We then describe the VP-Tree construction and query methods in detail.

3.1. Why VP-Tree?

In DP clustering, the calculations of the density value ρ and the dependent distance value δ for each data point are the two key steps, which take up most of the computation time. According to Eq. (1), the computation of the density values requires a huge amount of nearest neighbors (NN) search operations, especially for big data clustering. According to Eq. (2), the computation of a point's dependence value also requires to access the point's NNs since the point's dependent point is likely to be close (recall that the dependence value of a point is the distance to its dependent point, which is the nearest neighbor with higher density). A common approach for speeding up NN search is to exploit the spatial index. There exist several well-studied spatial index structures, including KD-Tree [20], Ball-Tree [21], VP-Tree [19], etc.

KD-Tree [20] is one of the most commonly used indexes for NN search problems. The KD-Tree is a binary tree in which each leaf node is associated with a k-dimensional point. Every internal node of the tree represents a splitting hyperplane that divides the space into two parts. Points to the left/right of this hyperplane are stored on the left/right subtree. The hyperplane direction is carefully chosen perpendicular to one of the axes in the k-dimensional space. In other words, KD-Tree leverages the dimensions to separate data points. However, in DPC, the computations of ρ and δ require to use the relative distances between points. The data points should be organized by their relative distances but not by their dimensions. This is also the reason why KD-Tree's performance degrades significantly when dealing with high-dimensional data.

Ball-Tree [21] can be used to index data points according to their relative distances, which is also a binary tree where each node defines a k-dimensional hypersphere (i.e., ball). Each internal node of the tree partitions the data points into two disjoint subsets which are associated with different balls. Each leaf node of the tree defines a ball and enumerates all data points inside that ball. While the balls themselves may intersect, each point could be assigned to more than one ball according to its distance from the ball's center. The performance of the Ball-Tree greatly depends on the placement of balls, and a good placement algorithm aims to minimize the overlap between balls. However, the ball placement algorithms [23] usually contain complex logic flow with many if-else branches, which is difficult to be parallelized with GPU architecture.

R-Tree [24] is a hierarchical data structure that groups nearby objects and represents them with their minimum bounding rectangles in the next higher level of the tree. It has been widely

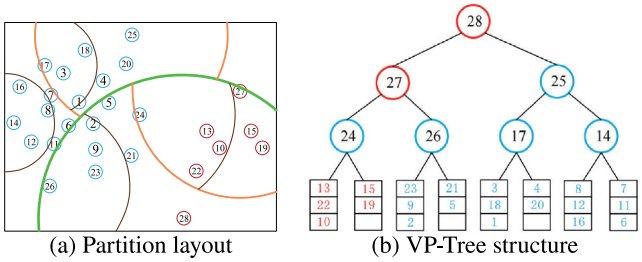


Fig. 2. An illustrative example of VP-Tree structure. Red points form a cluster, and blue points form another cluster.

adopted in many applications for indexing multi-dimensional spatial data. R-Tree requires a preprocessing step before construction to recursively divide the data space into small partitions, and then build a tree structure based on these partitions. While, VP-Tree directly builds the index according to the distance without the preprocessing. In addition, R-Tree allows sibling nodes to overlap each other, so query operations cannot guarantee a unique search path, which may lead to many useless queries and significantly impact query performance. Although there are a lot of work on building R-Tree based on GPUs [25–27], or using R-Tree to accelerate the density clustering algorithm [28]. However, compared with VP-Tree, R-tree requires more complex operations to reduce the overlap between nodes and keep balance, which is not suitable for GPU.

The Vantage Point Tree (VP-Tree) [19] is another spatial index similar to Ball-Tree. Each node of the tree contains one of the data points and a radius. Under the left child are all points that are closer to the node's point than the radius. The other child contains all of the points which are farther away. The construction of VP-Tree can be explained with an illustrative example. As shown in Fig. 2, point 28 is first chosen as the *vantage point* (vp) as it is far away from other points. Point 28 is also picked as the level-0 vp (root node) of the VP-Tree as shown in Fig. 2(b). We then draw a ball centered at point 28 with carefully computed radius r such that half of the points are in the ball while half is outside. All the points in the ball are placed in the root node's left subtree, while all the points outside are placed in the right subtree. The process is recursively applied for the inside-ball points and outside-ball points respectively. Finally, we will obtain such a VP-Tree as shown in Fig. 2(b).

The tree requires no other knowledge about the points in it but only a distance function that satisfies the properties of a metric space [29]. It does not need to find bounding shapes (hyperplanes or hyperspheres) or find points midway between them. These properties greatly match the needs of DPC. Furthermore, the construction and the search of VP-Tree can be efficiently parallelized with CUDA architecture since only a few data dependencies are required to handle. In the following subsections, we will present the details of the Vectorized VP-Tree structure and the GPU-accelerated VP-Tree construction methods.

3.2. Vectorized VP-Tree layout

The original VP-Tree structure is stored with many pointers that link different memory locations. A child node reference is a pointer referring to the location of the next level child. Since the memory locations of these tree nodes are randomly spread out in memory space, it is difficult to utilize the GPU memory hierarchy to explore the data locality and could result in memory divergence. Moreover, the next child location is obtained through the reference pointer, which will incur many indirect global memory accesses. As discussed in Section 2.2, a GPU-friendly design should avoid warp divergence, memory divergence, and reduce global memory accesses. The pointer-based tree structure does not adapt to GPU architecture. Therefore, a vectorized GPU-friendly VP-Tree structure is desired.

In our approach as shown in Fig. 3, the VP-Tree nodes are arranged in a breadth-first fashion in a one dimensional array (or vector) instead of pointers. The root node is stored at position 0 in the array. Suppose a node's position is i, we can obtain its left child position as $2i + 1$ and its right child position as $2i + 2$. Since a node's child position is known, there is no need to store pointers. This design requires less memory and provides higher search throughput due to coalesced memory access.

In the original VP-Tree index, each internal node stores a vantage point as well as its corresponding radius (which will be used during tree search). It is noticeable that, in our vectorized VP-Tree design, each array element does not store the point data but stores the point id and its corresponding radius, which only takes up 4+4=8 bytes. The point data are separately maintained in a long array where each array element stores a specific point data and can be accessed directly with a given point id. In addition, the parallel query performance of VP-Tree is greatly limited by the height of the tree. A higher tree implies more branches which are harmful to parallelization. In our design, we compact 32 point ids into a leaf node in order to reduce the height of the tree. Since the leaf nodes only store point ids but no radius, a leaf node takes up $32 \times 4 = 128$ bytes in order to exactly match the cache line size. This can also maximize the fanout and improve parallelism effectively. A number of 32 threads can be scheduled to execute the computation in parallel. Thus, we have another array design that stores leaf nodes where each array element takes 128 bytes. To sum up, our vectorized VP-Tree design is composed of three arrays. We refer to the array that maintains the VP-Tree vantage points' metadata as *vantage array*, the array that maintains leaf point ids as *leaf array*, and the array that maintains the point data as *data array*.

The typical tree construction methods involve many random insert operations and node split operations, e.g., B-tree, which incurs a large number of divergent branches. Branch divergence has a significant impact on the performance of GPU programs and limits parallelism. While VP-Tree is created from the root node to leaf nodes level by level in a top-down manner. More and more internal tree nodes are created and contained in the tree as the tree is created from the top level to lower levels. Since VP-Tree is a balanced binary tree and there is no dependency between these internal nodes, the construction of VP-Tree can be parallelized efficiently, which highly adapts to single-instruction-multiple-data (SIMD) architecture of GPUs.

3.3. Parallel construction of VP-Tree

The VP-Tree is recursively created as shown in Algorithm 1. As mentioned, a vectorized VP-Tree that adapts to GPU is composed of three arrays. Given the original data array *data*[] that stores the point data, our algorithm will output a vantage array that stores VP-Tree's internal nodes and a leaf array that stores VP-Tree's leaf nodes. Based on a heuristic method for choosing a vantage point [30], we first randomly pick a point p and make the farthest point from p as the level-0 vantage point $vp[0]$ (Line 4). Then we compute each point's distance from $vp[0]$ (Line(s) 9–10). The point ids are sorted in ascending order of these distance values (Line 11), where we use CUB [31] Library for high-performance sort operation. The medium of these distance values (i.e., $dist[sort\_ids[mid]]$) is used as level-0 vantage point's radius (Line 13). The points within this radius are arranged in the left sub-tree, while the points outside this radius are arranged in the right sub-tree. We then choose the in-radius vantage point id as

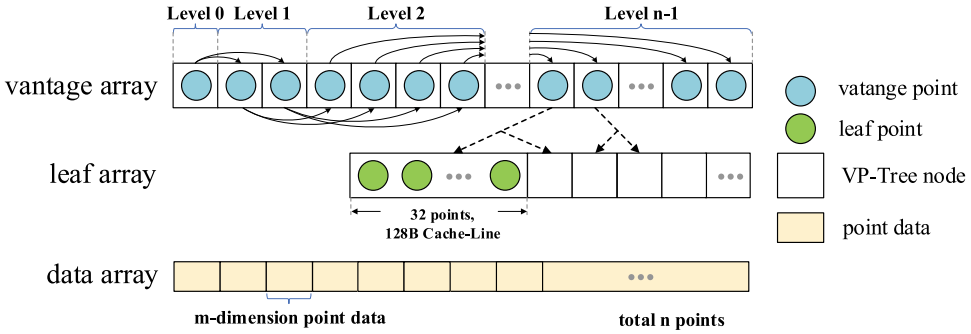


Fig. 3. Vectorized VP-Tree Structure on GPU.

Algorithm 1: Vectorized VP-Tree Construction

Input: point data array *data*[]
Output: vantage array *vp*[] and leaf array *leaf*[][]

```
1  ids[] ← initialize with point ids;
2  h ← ⌈log2(ids[].length/32)⌉; // tree height
3  p ← randomly select a point from data[];
4  vp[0].id ← find the farthest point id from p;
5  Recur_Build(ids[], 0);

   // ids[]: point ids; i: vantage point id
6  Function Recur_Build(ids[], i):
7      size ← ids[].length;
8      if size > 32 then
           // construct internal node
9          foreach id in ids[] do
10             ⟨id, dist[id]⟩ ← compute distance from data[id] to
                 vp[i]
11         sort_ids[] ← sort ids in ascending order of dist[id] in
             parallel;
12         mid ← ⌊size/2⌋;
13         vp[i].r ← dist[sort_ids[mid]];// vp's radius
           // recursively build left subtree
14         vp[2i + 1].id ← sort_ids[mid];
15         Recur_Build(data[], sort_ids[0, mid − 1], 2i + 1);
           // recursively build right subtree
16         vp[2i + 2].id ← sort_ids[size − 1];
17         Recur_Build(data[], sort_ids[mid, size − 1], 2i + 2);
18     else
           // construct leaf node
19         leaf[i − (2^h − 1)] ← ids[];
```

sort_ids[*mid*], which is likely to be far away from all other in-radius points, and store it in the left child node (Line 14). We also choose the out-radius vantage point id as *sort_ids*[*size* − 1], which is likely to be far away from all other out-radius points and store it in the right child node (Line 16). Given the in-radius points (i.e., with point ids in *sort_ids*[0, *mid* − 1]) and in-radius vantage point, we can apply the same process on the left sub-tree (Line 15). Similarly, given the out-radius points (i.e., with point ids in *sort_ids*[*mid*, *size*]) and out-radius vantage point, we can also apply the same process on the right sub-tree (Line 17). The recursive process will proceed if the number of in-radius points or out-radius points is larger than 32 (Line 8). Otherwise, we terminate the recursive construction process and make this node as a leaf node and accommodate all the point ids in the leaf node.

CUDA Multi-Stream Optimization. As we construct a balanced binary tree in a top-down manner, the degree of parallelism is exponentially increasing since there is no dependency when creating multiple same-level child nodes. We can issue concurrent GPU operations by placing them in multiple CUDA streams and achieve task-level parallelism (Line 15 and Line 17).

4. GDPC based on VP-Tree

In this section, we describe our proposed GPU-based DPC algorithm, GDPC that utilizes the constructed VP-Tree to accelerate DPC. The original DPC algorithm contains three steps, computing density values ρ, computing dependent distances δ, and assigning points to clusters. All three steps of GDPC are performed on GPU. In the following, we will describe these steps respectively.

4.1. Computing density values ρ

The computation of ρ requires a large number of nearest neighbors search operations. This requires $O(n^2)$ distance measurements in a naive implementation. Our basic idea is to utilize the VP-Tree index to avoid unnecessary distance measurements. We illustrate the use of the existing VP-Tree through an illustrative example as shown in Fig. 4. When computing a point's density value, it is required to access the points that are in point 21's d_c range. Let us compute point 21's density value (i.e., count the number of points within the gray circle) based on an existing VP-Tree's space partition result as shown in Fig. 4. We first evaluate the distance from point 21 to level-0 vantage point 28. Since the gray circle with radius d_c is completely inside the level-0 ball (with green arc line), i.e., $|p21, p28| + d_c \leq vp[0].r$ where $|\cdot, \cdot|$ is the distance between two points, it is enough to search the left child, where the vantage point is point 27. Vantage point 27's ball (with orange arc line) intersects with the gray circle, i.e., $|p21, p27| - d_c \leq vp[1].r$ (the gray circle has a part inside the orange ball) and $|p21, p27| + d_c \geq vp[1].r$ (the gray circle has a part outside the orange ball), so we need to search both the left child (with vantage point 24) and the right child (with vantage point 26). Similarly, we find the gray circle is completely inside vantage point 24's ball but intersecting with vantage point 26's ball, so we can locate the covered leaf nodes, i.e., vantage point 24's left leaf node (containing point 24, 13, 10, 22), vantage point 26's left leaf node (containing points 26, 23, 9, 2), and vantage point 26's right leaf node (containing point 5, 21, 28). These points are the candidate points for further distance calculations.

We describe the details more formally in Algorithm 2. Lines 1–10 depict how we determine the covered leaf nodes for point *p*, i.e., *cover_leaves*[*p*], which contain all the candidate points for distance calculation. This is exactly the same process as we illustrated in the above example. Line 11 is the ρ computation that is only based on the candidate points contained in *cover_leaves*[*p*] (a set of covered leaf nodes for point *p*).

To achieve high parallelism and to coalesced memory access during the point search process, we adopt several implementation optimizations as follows.

Calculation Order Arrangement (Line 3). During tree traversal, if multiple threads in a warp execute random queries, it is difficult to achieve coalesced memory access because they might traverse

Algorithm 2: GDPC Algorithm based on VP-Tree

Input: data array *data*[], vantage array *vp*[], leaf array *leaf*[][], and cut-off distance d_c
Output: density array ρ[], dependent distance array δ[], point-cluster assignment *cluster*[]

1 init *cover_leaves*[];// cover leaves for each point
2 $n \leftarrow vp[].length$; // number of internal nodes
/* compute density values */
3 **foreach** *point p* ***parallel*** **do**
4 Stack *S*.push(0) ; // push root into stack *S*
// search covered leaves within d_c range
5 **while** *S is not empty* **do**
6 $i \leftarrow S$.pop();
7 **if** $i \geq n$ **then**
// this is a covered leaf node
8 *cover_leaves*[*p*].append(*leaf*$[i-n]$);
9 **if** $|data[vp[i].id], data[p]| - d_c \leq vp[i].r$ **then**
// search left child node
10 *S*.push($2i+1$);
11 **if** $|data[vp[i].id], data[p]| + d_c \geq vp[i].r$ **then**
// search right child node
12 *S*.push($2i+2$);
13 $\rho[pid] \leftarrow$ evaluate *p*'s ρ based on *cover_leaves*[*p*];
/* compute dependent distances */
14 **foreach** *point p* ***parallel*** **do**
15 **if** *p has highest density among cover_leaves*[*p*] **then**
// compute dependent distances globally
16 {*dep*[*p*], δ[*p*]} $\leftarrow$ find *p*'s dep. neighbor and evaluate δ based on all points *data*[];
17 **else**
// compute dependent distances locally
18 {*dep*[*p*], δ[*p*]} $\leftarrow$ find *p*'s dep. neighbor and evaluate δ based on *cover_leaves*[*p*];
/* peak selection and cluster assignment */
19 *peak*[] $\leftarrow$ determine density peaks (large ρ and large δ);
20 *cluster*[] $\leftarrow$ determine each point's cluster assignment based on *peak*[] and *dep*[];

the tree along different paths. Diverse queries in a warp would lead to poor GPU performance due to memory divergence. If multiple queries share the same traversal path, the memory accesses coalesce when they are processed in a warp. We design our warp parallelism in terms of VP-Tree properties. Because the points assigned to the same leaf node share the same traversal path, we assign the threads in the same warp to process the points in the same leaf node. That is, we execute warp-parallelism between leaf nodes and execute thread-parallelism within each leaf node. In this way, the warp divergence is mitigated.

Ballot-Counting Optimization (Line 13). CUDA's *ballot* function takes a boolean expression and returns a 32-bit integer, where the bit at every position i is the boolean value of thread i within the current thread's warp. The ballot function performs a reduction-and-broadcast operation over a predicate, which is usually a comparison between a key (or a pivot) and each thread's key. The intrinsic operation can enable an efficient implementation of the per-block scan. By combining the `__ballot()` and `__popc()` intrinsics, we can efficiently count the number of points within d_c.

Fully Contained Leaf Nodes. In the original VP-Tree, the points in the vantage node might be not contained in leaf nodes. That means, we have to check internal vantage nodes in order not to miss d_c range points, which could result in memory divergence. In our implementation, the vantage points also have their copies in leaf nodes to avoid memory divergence.

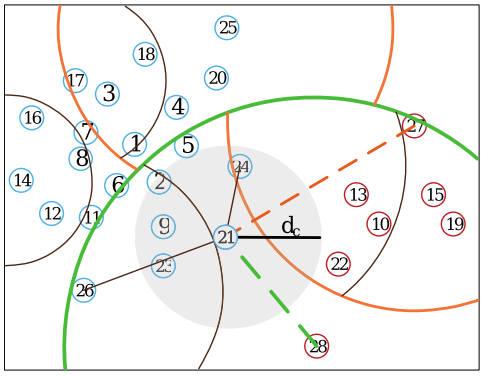


Fig. 4. An illustrative example of using VP-Tree (better with color).

4.2. Computing dependent distances δ

Given the computed density values, we can calculate the dependent distance values as shown in Lines 14–18 in Algorithm 2. Recall that the dependent distance of a point is its distance to the nearest neighbor with a higher density as shown in Eq. (2). We can again leverage the VP-Tree index. Since a point *p*'s nearest neighbor with a higher density is highly likely to be in its leaf nodes, we first locally search among its leaf nodes. If such a point does not exist (when the point itself has the highest density among its leaf nodes), we will search globally by checking all the other points with higher density (Line(s) 15–16). Otherwise, its dependent neighbor resides in the covered leaf nodes *cover_leaves*[*p*], so we can obtain its dependent neighbor and compute its dependent distance only considering the candidate points in its covered leaf nodes (Line(s) 17–18). The distance calculation results in the previous step are reused to find the closest points with higher density to save computation costs.

Similarly, we achieve coalesced memory access by arranging the calculation order (Line 18), making threads in a warp to process the points in the same warp. In order to avoid unnecessary distance calculations, we also sort the candidate points in descending order of their density values. Since we attempt to find the nearest neighbor with a higher density, we can skip the distance calculations with the points which have a lower density in terms of the density-ordered list (Line(s) 16,18).

4.3. Assigning points to clusters

Given the density values ρ and the dependent distance values δ, we then need to pick the density peaks, which have both large ρ and large δ (Line 19). Given a set of density peaks (each represents a cluster), we should assign each point to a certain density peak (i.e., a certain cluster). We perform this process by tracing the assignment chain till meeting a certain density peak. The assignment dependency relationship (as shown in Fig. 1(c)) is recorded when computing δ (Line(s) 16,18).

To adapt to GPU's parallel architecture, we need to build a reverse index of the assignment chain. Then, the point assignment is similar to a label propagation process starting from a number of density peaks in a top-down manner (Line 20), where the label is a certain density peak's id and the reversed dependencies can be regarded as the underlying graph edges. This label propagation process can be easily parallelized since the propagation processes on different sub-trees are totally independent.

5. Incremental GDPC

In this section, we present how to handle incremental updates of GDPC clustering results on evolving datasets. Suppose the original dataset is $\mathcal{D}$ and the newly added data is $\Delta\mathcal{D}$. In order to update clustering results in response to input changes, we should first update the VP-Tree efficiently with GPU parallelism. In Section 5.1, we propose an incremental update method that can adjust the GPU-based VP-Tree incrementally with newly added data $\Delta\mathcal{D}$. Furthermore, the addition of new points always results in changes of the ρ and δ values. The simplest way to update clustering results is to reperform GDPC based on the updated GPU-based VP-Tree. However, reperforming GDPC will result in a large amount of redundant computation, because most of the points' ρ and δ values are not changed, i.e., they are not affected by the newly added data. Thus we only compute the ρ and δ values for the new points in $\Delta\mathcal{D}$ and update the ρ and δ values that are affected by $\Delta\mathcal{D}$. We will introduce the details of the incremental update of ρ and δ values in Sections 5.2 and 5.3. Last, we introduce how to incrementally update clustering results in Section 5.4.

5.1. Incremental update of VP-Tree

The structure of VP-Tree is determined by the selected vantage points and their radius. When adjusting the VP-Tree to adapt to the newly added points, some points will be re-partitioned due to the re-selection of vantage points or the changes of their radius. In order to minimize the changing scope of VP-Tree, Fu et al. [30] have proposed a dynamic VP-tree structure, which processes the new insertions sequentially one by one. However, this strategy is designed based on CPU and involves upward backtracking, which incurs significant redundant reconstruction costs on GPU.

In this subsection, we propose a GPU-friendly incremental update method to update VP-Tree dynamically and efficiently in parallel. The basic idea is to group points in ΔD according to their target leaf nodes (data locations) and perform insertions/deletions within the same target leaf node together. This can adapt to GPU's SIMD execution model and greatly improve parallelism. To avoid leaf node overflow after insertion, we should ensure enough space for these leaf nodes. We introduce a new array *free*[] to maintain the free room of all leaf nodes rooted by vp[] and that of all leaf nodes *leaf*[]. Given the incremental input $\Delta\mathcal{D}$, if there is not enough free room to accommodate $\Delta\mathcal{D}$, the VP-Tree should be expanded. There are two ways to increase the capacity of GPU-based VP-Tree, one is to rebuild the GPU-based VP-Tree with all data points $\mathcal{D}\cup\Delta\mathcal{D}$, and the other is to split each leaf node and increase the tree height, i.e., the original leaf node becomes an internal node (vantage point). The first way requires updating the whole vp[] array while the second way only needs to append the new vantage points at the end of vp[]. To reduce update cost, we prefer the second one to increase the capacity of GPU-based VP-Tree.

Algorithm 3 presents the details of our incremental update method for GPU-based VP-Tree. We first check if the free room of the whole VP-Tree (i.e., maintained in *free*[0]) is enough to hold the new points (Line 2). If there is not enough space, an `Inc_Capacity()` function is first invoked to expand the tree (Line(s) 7–18). As discussed above, we choose to extend leaf nodes to increase capacity. We split each leaf by selecting a new vantage point among the points in the old leaf node and assigning the old points to the left/right new leaf nodes according to their distances to the new vantage point. As described in Section 3, the points in each leaf are sorted in ascending order of their distances to their vantage/parent. Within the leaf node, we select the last point as the new vantage point (Line 11) and compute the distances from each point to the new vantage point (Line(s) 12–13). We sort the point ids according to the distance values and use the medium value of these distances as the new vantage point's radius (Line(s) 14–16). The points within this radius are assigned to the new left leaf node, while the points outside this radius are assigned to the new right leaf node (Line(s) 17–18). Since there is no dependency between leaf nodes, the leaf nodes can be split in parallel. After the capacity of the GPU-based VP-Tree is increased, we update the free space array *free*[] and the height of the tree (Line(s) 4–5). Given the free space of all new leaf nodes, we update each internal node's free space based on its leaf nodes recursively in a bottom-up manner, i.e., $free[i] = free[2i+1] + free[2i+2]$.

As the VP-Tree has enough free space to accommodate the new points, the insertion of new points is similar to the construction of the VP-Tree. We first compute the distance from each point to the top-level vantage point vp[0]. The points within vp[0]'s radius are assigned to the left subtree, while the points outside the radius are assigned to the right subtree (Line(s) 21–25). The same process is applied on the left and right subtrees recursively until arriving at the leaf node (Line(s) 28–33). By this way, the new points targeted at the same leaf nodes are grouped and processed at the leaf node together, which achieves batched processing and adapts to parallel execution. If there is not enough free room to accommodate the points, we will reconstruct the subtree with all points of this subtree and the assigned new points (Line(s) 35–36). The incremental update approach is parallel-friendly since we arrange the point insertions into groups according to their locations and perform batch processing for each group.

Fig. 5 shows an illustrative example, in which ten new points (id 29–38) are added. Fig. 5(a) shows the initial partition layout, where the gray circles are new points. Fig. 5(b) shows the original VP-Tree structure. Before insertion, we find the free space is not enough to accommodate ten new points, so we first expand the leaf nodes to increase the capacity according to Algorithm 3. Then the new points are inserted into the expanded tree together in a batch and in a top-down manner as shown in Fig. 5(c). Specially, each point is determined to be assigned to the left subtree or the right subtree of the root vantage point (point 28) according to its distance to point 28 and root vantage point's radius. This assignment process proceeds recursively layer-by-layer based on the distance to the level-specific vantage point and the vantage point's radius, and finally we obtain the updated VP-Tree as shown in Fig. 5(c). Accordingly, we have the updated vantage point partition layout as shown in Fig. 5(d).

5.2. Incremental update of density values ρ

Based on the updated VP-Tree, we next perform the incremental update of the density values. It requires not only (i) evaluating ρ values of the newly added points but also (ii) updating the previously computed ρ of old points, since a new point p may change an old point q's density value as p falls into q's d_c range. The evaluation of the new points' ρ follows the same process as depicted in Section 4.1. Regarding the update of existing ρ, we provide two alternative approaches in the following.

Incidental Method. The idea of the incidental method is that, when calculating a new point q's ρ_q, the density values of the old points that are within q's d_c radius range are incremented by 1 incidentally. This can avoid another separate pass for updating old points' densities. However, since two threads compute the densities of new points q_1, q_2 may trigger the density update of the same old point at the same time, there exist write-write conflicts that should be resolved to ensure correctness. A locking

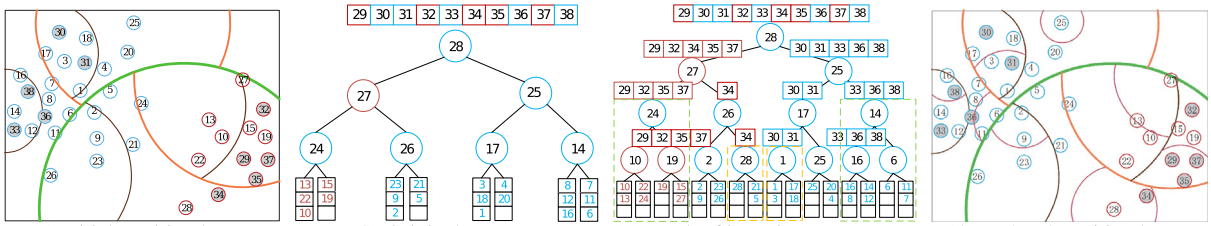


Fig. 5. An illustrative example of VP-Tree update.

mechanism that avoids concurrent updates is needed but this can hurt parallel execution performance.

Separate Method. An alternative is to separate new points' ρ computation from old points' ρ update. If an old point p's *cover_leafs* contain newly inserted points, p is marked as a point to be updated with a new density value. Because we have counted all old points within p's d_c radius, we only need to compute the distance from p to the newly inserted point in its *cover_leafs*. There is no write-write conflict in this method since multiple threads update different ρ_p independently. However, the separate method incurs redundant distance measurements performed by multiple threads.

Intuitively, when processing the small number of new points that leads to a small conflict probability, the incidental method is preferred. When processing large amounts of incremental data, the separate method is preferred. In Section 6.2.4, we will empirically show the effects of these two update methods.

5.3. Incremental update of dependent distances δ

For newly added points, we should calculate their dependent distances δ according to their density values. This can be done according to the GPU-friendly method proposed in Section 4.2. On the other hand, the insertion of new points and the changes of old points' ρ values will result in the changes of old points' δ values as well. Therefore, the δ values of old points that are affected should be updated accordingly. The δ value of an old point p will change only in the following two cases. (i) The density value of p is updated and becomes larger than that of its dependent point. (ii) The density of a point q that is closer to p than p's dependent point increases to be larger than that of p, i.e., q becomes p's new dependent point. For the first case, we have to recompute δ_p. For the second case, though the dependent distance of p may be updated, its old dependent distance, denoted as δ^*, can be treated as the upper bound of its new δ value. We can update δ_p by checking p's neighbors within δ^* range.

5.4. Incremental update of clustering results

After updating ρ and δ values, the clustering results can be updated immediately. Following the assignment chain that is updated during δ's incremental update, we rearrange the point-cluster assignment, i.e., some points originally belonging to cluster A might be merged to another cluster B. We also check whether a point p's updated δ and ρ are large enough to be density peaks (line 19 in Algorithm 2). If so, then p is treated as a new density peak (cluster center), and all the points in the assignment chain ended by p will form a new cluster centered at p, i.e., a cluster is split into two clusters.

5.5. Deleting

When updating the clustering results, the operation of point deletion is similar to that of addition.

For the updates of VP-Tree, as shown in Fig. 5(c), we divide the deletion points into different batches from top to bottom, locate the leaf nodes where they are, and then delete them.

For the updates of density values, similar to point insertion, we also provide two update methods, the Incidental Method and the Separate Method (Section 5.2). When deleting a point, the main idea of the Incidental Method is to update the density of points within the d_c range of the deletion point (density ρ is updated to be ρ-1). This method has write-write conflicts, we need to lock the points where the density is updated to ensure the accuracy of the density. The Separate Method first checks whether the leaf nodes within the d_c range of each point p_i (cover_leaves) contain the deletion points, if so, updates the density of point p_i by subtracting the number of deletion points contained in the cover_leaves from the original density. This method is suitable for a larger number of deletion points.

For the updates of the delta, the delta values need to be updated only when the following two conditions meet. (1) The density of p's dependent point becomes smaller than that of p. (2) There is a point q whose distance to p is less than the old delta value of p, and the density of q becomes larger than that of p due to the p's density value decreasing. For the first case, we have to recalculate the delta of point p. For the second case, we take the old delta of point p as the upper bound, and then look up the nearest neighbors with higher density within the upper bound. Finally, we update the clustering results based on the updated density and delta.

6. Experiments

This section evaluates GDPC and its incremental variant on real-world datasets to verify their benefits.

Machine Configuration. We conduct all experiments on two 8-core servers (Intel Xeon CPU Silver 4110 @ 2.1 GHz, 32 logical CPUs and 64GB host memory) with an NVIDIA RTX2080Ti GPU. It has 68 SMs and 11 GB GDDR6 memory with a peak bandwidth of 616 GB/s. Our implementation is compiled by CUDA 10 along with nvcc optimization flag -O3.

Dataset. Table 1 lists the data sets used in our experiments. These include two small and medium sized 2D data sets, and seven real world large high-dimensional data sets Facial, 3DSpatial, KDD, BigCross, Household, PAMAP, and Airpline. all of which are available online [32–35]. The data set, BigCross, contains 11.6 million data points, each with 57 dimensions. To obtain clustering result in a reasonable time period, we construct a smaller BigCross500K data set by randomly sampling 500,000 points from the original BigCross data set.

Algorithm 3: Incremental Update of VP-Tree

Input: data array *data*[], vantage array *vp*[], leaf array *leaf*[][], free space array *free*[], and new point data array *new_data*[]
Output: updated vantage array *vp*[] and updated leaf array *leaf*[][]

```
1  new_ids[] ← initialize with new point ids;
2  while new_ids[].length > free[0] do
3      Inc_Capacity();
4      update free space array free[];
5      h = h + 1; // tree height
6  Recur_Insert(new_ids[], 0);

7  Function Inc_Capacity():
8      n ← vp[].length;
9      oldleaf[] ← leaf[]; leaf[] ← empty;
10     foreach oldleaf i parallel do
           // the last point in leaf as new vp
11         vp[n + i] = oldleaf[i][oldleaf[i].length − 1];
12         foreach id in oldleaf[i] do
13             ⟨id, dist[id]⟩ ← compute distance from oldleaf[i][id]
                to vp[n + i];
14         sort_ids[] ← sort ids in ascending order of dist[id];
15         len ← sort_ids[].length; mid ← ⌊len/2⌋;
16         vp[n + i].r = dist[sort_ids[mid]];
           // extend leaf array
17         leaf[2i] ← sort_ids[0, mid];
18         leaf[2i + 1] ← sort_ids[mid + 1, len − 1];

19 Function Recur_Insert(new_ids[], i):
20     if new_ids[].length > 0 then
           // assign points to left/right subtree
21         foreach id in new_ids[] parallel do
22             if |data[id], data[vp[i].id]| ≤ vp[i].r then
23                 L[].append(id);
24             if |data[id], data[vp[i]]| > vp[i].r then
25                 R[].append(id);
26         l = 2i + 1; r = 2i + 2;
27         if L[].length <= free[l] and R[].length <= free[r] then
               // free space is enough
28             if l >= 2^h − 1 then
                   // append to leaf nodes
29                 leaf[l − (2^h − 1)].append(L[]);
30                 leaf[r − (2^h − 1)].append(R[]);
31             else
                   // this is an internal node
32                 Recur_Insert(L[], l);
33                 Recur_Insert(R[], r);
34         else
               // rebuild subtree with new points
35             S[] ← Combine all points of the subtree identified by
                    vp[i] and new_ids[];
36             Recur_Build(S[], i);
```

6.1. Results of GDPC

We evaluate GDPC by comparing it with the state-of-the-art DPC implementations, including GPU-based DPC (CUDA-DP [18]), distributed DPC (LSH-DDP [3]), and multicore-based DPC (S-Approx-DPC [13]).

6.1.1. Performance comparison with state-of-the-arts

We compare the runtime of our proposed GDPC with other DPC implementations as shown in Fig. 6. We first compare with CUDA-DP [18] on smaller datasets Aggregation, S2, and Facial. CUDA-DP returns out of memory error on larger datasets 3Dspatial, KDD, and BigCross500K. This is because CUDA-DP just optimizes the distance calculations without leveraging spatial index structure. We can see that our GDPC can achieve 5.3X–17.8X speedup over CUDA-DP attributed to our vectorized VP-Tree design and GPU-friendly parallel algorithm.

We then compare GDPC with S-Approx-DPC on low-dimensional datasets Aggregation, S2, and 3DSpatial. S-Approx-DPC uses KD-Tree to index data, which only work well on low-dimensional datasets (see Section 3.1), so it cannot run on high-dimensional datasets. Our results show that GDPC achieves a 43–148.9x speedup over S-Approx-DPC, since our GDPC can benefit from GPU's parallelism.

In addition, to evaluate our algorithm on larger and higher dimensional datasets (including Facial, 3Dspatial, KDD, and BigCross500K), we compare with a state-of-the-art distributed DP clustering algorithm LSH-DDP [3], which is implemented based on Hadoop MapReduce and utilizes locality-sensitive hashing index to improve the ρ and δ calculations. The distributed LSH-DDP experiments are performed on a cluster with 5 machines (1 master and 4 slaves), each equipped with an Intel I5-4690 3.3G 4-core CPU, and 4 GB memory. We can see that our GDPC can

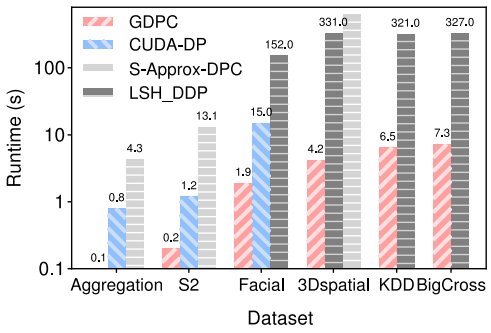


Fig. 6. Runtime comparison.

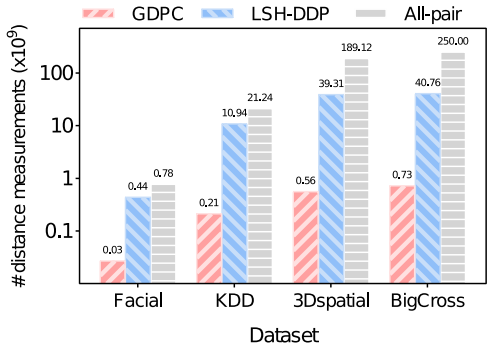


Fig. 7. Computational cost.

Table 1
Data sets.

Data set	No. instances	No. dimensions
Aggregation	788	2
S2	5,000	2
Facial	27,936	300
KDD	145,751	74
3Dspatial	434,874	3
BigCross500K	500,000	57
Household	2,075,257	4
PAMAP	3,252,226	12
Airpline	5,277,086	3

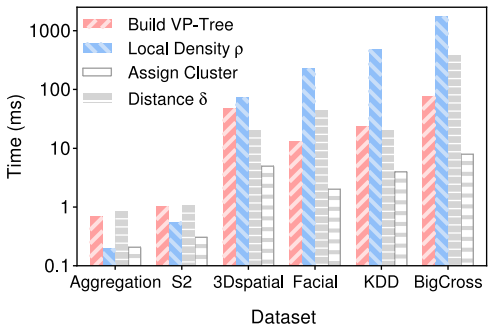


Fig. 8. Runtime breakdown.

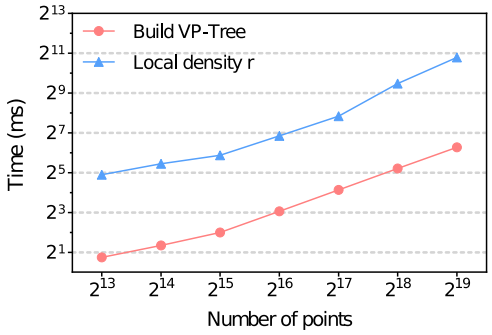


Fig. 9. Scaling performance.

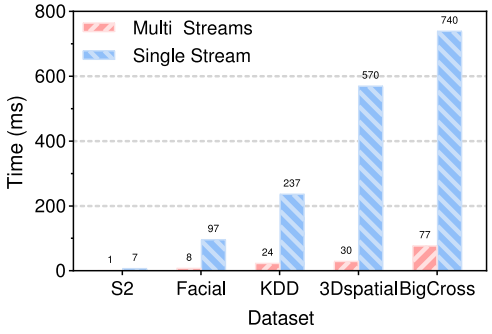


Fig. 10. Effect of multi-stream processing.

achieve 44.8–78.8X speedup. We found that LSH results in an unbalanced workload and suffers from extensive communication overhead and synchronization overhead.

6.1.2. Computational cost analysis

In DPC algorithm, the distance calculations for computing ρ and δ is the most expensive part, especially for large and high-dimensional data. The naive implementation requires to evaluate all-pair distances, which results in significant computational overhead. GDPC utilizes VP-Tree to avoid the large number of unnecessary distance calculations due to its excellent support for nearest neighbors search. Similarly, LSH-DDP also leverages LSH index to avoid unnecessary distance calculations. We evaluate the computational cost of naive all-pair computation, LSH-DDP, and GDPC by comparing their number of distance calculations during the clustering process and show the results in Fig. 7. We can see that our GDPC requires significantly fewer exact distance calculations than prior work, say only 1.4–6.8% of LSH-DDP and 0.3–3.8% of all-pair calculations.

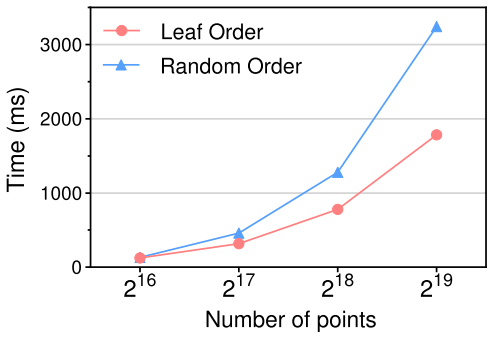


Fig. 11. Effect of coalesced memory access.

6.1.3. Runtime breakdown analysis

In the GDPC algorithm, there are four main steps to obtain the final clustering result, which is VP-Tree construction, density ρ calculation, dependent distance δ calculation, and point-to-cluster assignment. We break down the total runtime and see the runtime of each particular phase. The runtime of each phase is reported in Fig. 8. We can see that for the larger dataset, the ρ computation is always the most expensive part since it requires large number of distance measurements. While the point-to-cluster assignment is relatively fast.

6.1.4. Scaling performance

We also evaluate the scaling performance of GDPC when increasing data size. We first randomly choose $2^{13} - 2^{19}$ number of points from the BigCroos500K dataset to generate multiple same-distribution datasets with different sizes. We then record the runtime for the VP-Tree construction phase and the runtime for the ρ calculation phase when clustering different-size datasets. The results are reported in Fig. 9. The runtime exhibits linear growth when increasing the data size, while the runtime of all-pair distance calculations will exhibit quadratic growth. This experiment shows our GDPC algorithm can achieve great scaling performance.

6.1.5. Effect of CUDA multi-stream optimization

As presented in Section 3.3, during the VP-Tree construction process, we leverage CUDA multi-stream optimization to improve the parallelism when constructing the left child sub-tree and the right child sub-tree. We study the effect of multi-stream optimization empirically by comparing with single-stream implementation. The results of GDPC with and without multi-stream optimization are depicted in Fig. 10. We can see that the multi-stream optimization can significantly improve performance, say 7.73X–19.12X improvement.

6.1.6. Effect of coalesced memory access

Regarding GPU-based algorithm design, coalesced memory access is a key optimization technique that utilizes the properties of GPU memory hierarchy. In our GDPC implementation, a very important coalesced memory access optimization is the arrangement of point calculation order when locating the covered leaf nodes. As discussed in Section 4.1, we assign the threads in the same warp to process the points in the same leaf node. That is, processing points according to their belonging leaves. In this way, we can achieve coalesced memory access because the points in the same leaf node share the same traversal path. To understand the performance improvement of our design, we use the random processing order to see the advantages of using the leaf-based processing order. The results are shown in Fig. 11, we can see our approach shows better performance, say 1.07X–1.82X improvement.

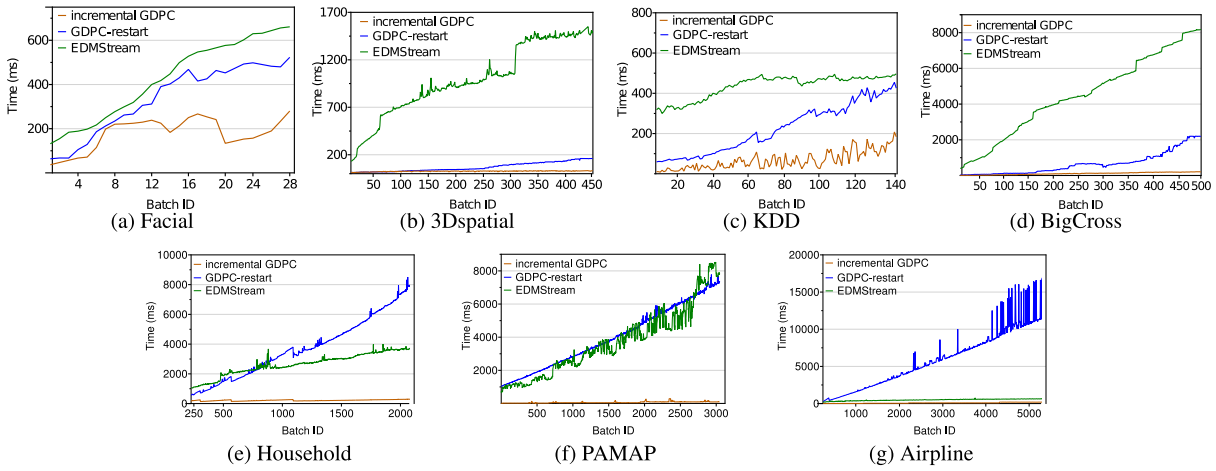


Fig. 12. The response time comparison of incremental clustering algorithms.

6.2. Results of incremental GDPC

We evaluate Incremental GDPC by comparing with performing GDPC from scratch (named as GDPC-restart). We also compare with another incremental DPC algorithm EDMStream [12]. For each dataset, we choose 10% of the original dataset as the base data and add data in batches with each batch containing 1000 random points from the original dataset.

6.2.1. Response time

With constantly incoming data, we separately run GDPC-restart, EDMStream, and Incremental GDPC to update clustering results and record their response time to cluster update for each incoming batch of data. Fig. 12 shows the response time of GDPC-restart, EDMStream, and Incremental GDPC for each incremental batch. As the base data size is larger and larger, all algorithms require a longer response time, which is under expectation. Compared with EDMStream, Incremental GDPC achieves significant speedup, say 2–61.3X speedup, thanks to our GPU-friendly algorithm design. Compared with GDPC-restart, Incremental GDPC achieves 1.2–81.5X speedup. This can be attributed to our incremental algorithm design that can effectively reduce the cost of rebuilding VP-Tree and avoid the redundant distance measurements when updating ρ and δ values.

6.2.2. Update time of each phase

We also compare Incremental GDPC with GDPC-restart on the update time of each phase to verify the efficiency of incremental updates. The δ update time in Incremental GDPC is similar to that in GDPC-restart, so we only show the runtime of tree update and ρ update in Incremental GDPC and GDPC-restart as shown in Fig. 13. We can see that the Incremental GDPC outperforms the GDPC-restart in both phases. Incremental GDPC performs incremental updates based on the previous clustering result rather than restarting clustering on the updated dataset. The incremental approach saves a large amount of redundant computation costs. Especially in the ρ update phase (the most time consuming phase as illustrated in Fig. 8), the runtime of the GDPC-restart approach increases steeply as data evolve and finally reaches up to 2,235 ms to complete the ρ update. While our incremental approach only needs at most 163 ms.

6.2.3. Incremental update of VP-Tree: GPU-friendly vs. CPU-friendly

In order to verify the efficiency of our proposed incremental VP-Tree update method, we compare our method with the incremental update method proposed in [30]. Rather than batching insertions, the method adopted in [30] processes insertions sequentially, which is CPU-friendly and not suitable for GPU architecture. For fairness, we have parallelized and implemented this method on GPU. As described in [30], we first insert the points located in leaf nodes with free space into VP-Tree in parallel. For the remaining points that are located in leaf nodes without free space, we adjust the VP-Tree by the method proposed in [30] in parallel. Then the remaining points can be inserted into VP-Tree in parallel. Fig. 14 shows the update time of our approach (GPU-friendly) and the parallelized sequential method (CPU-friendly) on two datasets, KDD and BigCross. Our batched method always spends less time updating the tree, because it does not need to backtrack to traverse the search path and search the nearest ancestor node that has enough space. It can be seen that our method is more simple and more effective.

6.2.4. Incremental update of ρ: Incidental vs. Separate

As discussed in Section 5.2, we provide two alternative ρ update methods. Each method has its own advantage and disadvantage (we use the incidental method by default). In this experiment, we compare them by varying different incremental data size $|\Delta\mathcal{D}|$ (from 1% to 16%) on KDD and BigCross datasets. The results are shown in Fig. 15. We can see that if the incremental data size $|\Delta\mathcal{D}|$ is smaller (e.g., $|\Delta\mathcal{D}| < 4\% \cdot |\mathcal{D}|$), the incidental method outperforms the separate method, otherwise the separate method performs better. This is because that when $|\Delta\mathcal{D}|$ is small, there are fewer write-write conflicts in the incidental method, and when $|\Delta\mathcal{D}|$ becomes larger, the lock-based conflict resolution occupies a very large computational overhead.

6.3. Bottleneck Analysis and Optimizations

We use the Nsight Compute CLI to analyze the utilization of GPU during the density computation (which is the most time consuming part, as shown Fig. 8). The density computation of point p mainly includes: (a) Obtaining the leaf nodes that may contain p's neighbors whose distance to p is smaller than d_c. (b) Traversing the points in these leaf nodes to find the points whose distance to p is smaller than d_c. (c) Calculating the density values by summing up these qualified points. In our naive implementation, each thread randomly processes a point in the data set, and these threads do not communicate with each other. For our naive GPU implementation, we show the profiling results on several metrics in Table 3, including the Achieved Occupancy, the Warp Execution Efficiency, the Branch Efficiency, and the Global Memory Load Efficiency. The detailed descriptions of these metrics are shown

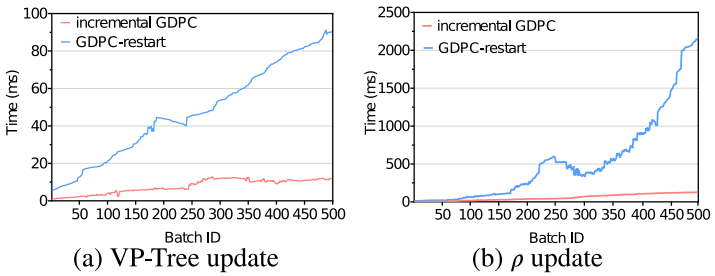


Fig. 13. Runtime comparison of VP-Tree update phase and ρ update phase.

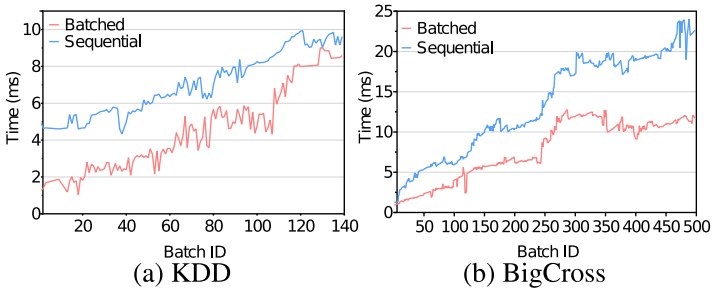


Fig. 14. Runtime comparison for updating VP-Tree: batched vs. sequential.

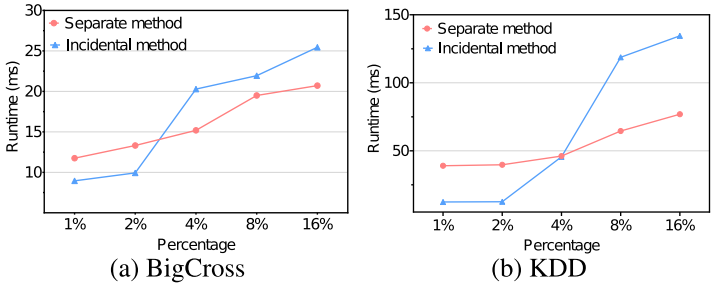


Fig. 15. Runtime comparison for updating ρ: incidental vs. separate.

in Table 2. From Table 3, we can see that the utilization of GPU is less than 55.5% without optimizations. Therefore, we focus on improving the utilization of GPU.

The reasons for low GPU utilization in naive implementation without optimizations in Table 3 are as follows:

- Warp divergence. Warp divergence is a major performance bottleneck that prevents data-intensive applications from achieving high performance in GPU. Our naive implementation is that one thread processes a random point in the data set, which causes multiple threads in a warp to perform random queries resulting in warp divergence as they may traverse the tree along different paths. Warp divergence can seriously impact GPU performance.
- Load imbalance. The number of points in different cover_leaves varies. When the size of the cover_leaves between threads differs greatly within the same warp, the load imbalance problem will arise and impact GPU performance.
- Low bandwidth and high latency. Since the data are all stored in global memory, the programs have to frequently access global memory to get the data. If all threads of a warp execute a load instruction, and these threads access contiguous memory units of global memory, the access efficiency is the highest at this time, which is close to the peak of global memory bandwidth. However, the naive implementation method uses multiple threads to execute random queries, which may cause memory addresses requested in the same warp to fall into different GPU cache lines. In this case, multiple memory transactions are required to process these requests, which seriously impacts the loading efficiency and throughput.

To address the above problems, we mainly propose the following optimizations:

- We let each leaf node contain 32 data points. A warp generally contains 32 threads that execute the same instructions. We use a warp to process a leaf node, that is, 32 threads process the corresponding 32 data points. This method ensures that threads in a warp execute the same instructions and alleviates warp divergence. This optimization plays an important role in computing (Section 4.1) and updating density values (Section 5.2).
- We use the Shuffle instruction to reduce the latency of memory accesses when calculating density values (Section 4.1), updating density values (Section 5.2), calculating delta values (Section 4.2), and updating delta values (Section 5.3). The Shuffle instruction allows the current thread to directly read the values in the registers of other threads, which reduces the communication latency between threads, and performs data exchange between threads without additional memory. We also use the shuffle instruction to achieve cooperation between threads to balance the loads.

Table 2
Description of performance parameters.

Parameters	Instructions
Achieved Occupancy	Refers to the ratio of active warps to the total warps, enough active warps can ensure full execution of parallelism (conducive to delay hiding)
Warp Execution Efficiency	Execution efficiency of threads in a warp
Branch Efficiency	The ratio of the number of non-divergent branches to the number of all branches, the higher the value, the stronger the parallel execution ability
Global Memory Load Efficiency	The ratio of the requested global load throughput to the desired global load throughput, a measure of the application's utilization of device memory bandwidth
Global Load Throughout	Check the memory read efficiency of the kernel
Number of Transactions	Evaluate the number of global memory requests

Table 3
Performance comparison with and without optimizations.

	Without optimizations	With optimizations
Achieved Occupancy	48.34%	96.8%
Warp Execution Efficiency	55.5%	99.84%
Branch Efficiency	48.3%	99.9%
Global Memory Load Efficiency	21.22%	75.23%
Global Load Throughout	162.08 GB/sec	527.42 GB/sec
Number of Transactions	$251.2x10^6$	$142.5x10^6$

- In addition, we also use the CUDA Stream to improve the performance of the building (Section 3.3) and updating VP-Tree (Section 5.1).

Table 3 shows the GPU utilization and performance comparison with and without optimizations. It can be found that the Achieved Occupancy, the Warp Execution Efficiency Global, and the Branch Efficiency have all reached more than 90% after optimizations, so the parallelism has been greatly improved. The Global Memory Load Efficiency has also reached more than 70%, which is a significant improvement compared to 21%. We also measured the Global load Throughout and the Number of Transactions, the Global Load Throughout is increased by 3.2 times, and the Number of Transactions is reduced by half. It means that the memory access efficiency and bandwidth have been greatly improved with optimizations.

7. Related work

Parallel clustering makes full use of the resources of multiple processors, makes the clustering algorithm run on multiple processors at the same time, greatly shortens the execution time of the clustering algorithm, and provides an effective solution for large-scale data clustering analysis.

In recent years, general-purpose graphics processing units (GPGPU) have been widely used to facilitate processing-intensive operations for their parallel processing ability. There exist a number of studies aiming to use GPU to accelerate data clustering. Cao et al. [36] exploit the high computational power and pipeline of GPUs for distance computing and comparison, and has sped up the k-means algorithm substantially. Farivar et al. [37] propose the GPU-accelerated k-means algorithm and has achieved 13x speedup over CPU-based k-means clustering. Lutz et al. [38] execute k-means in a single data pass per iteration and allows to perform the point assignment phase and the centroid update phase efficiently on GPUs. GDBSCAN [39] can be 100 times faster than the sequential version by using graphs to explore various parallelization opportunities. BPS-HDBSCAN [40] is a shared memory hybrid CPU/GPU approach that can perform clustering on a billion-point scale. Taylor et al. [41] propose a GPU accelerated Yinyang k-means [42], GPU-YYS, by exploiting shared memory and a centroid update scheme to accelerate Yinyang k-means.

There also exist several GPU-based approaches for accelerating DP clustering algorithms. Li et al. [17] accelerate the distance matrix computation in DPC with GPU. The implementation is based on JCuda and has only achieved an acceleration about 7 folds. CUDA-DP [18] has redesigned the data structure of the data point array on the basis of [17]. They do not use the traditional array of structure (AOS), but the structure of array (SOA) form. The results of the experiment show that CUDA-DP can achieve a 45-fold acceleration. But the cost of distance matrix calculation still degrades the performance. Also, it is limited by the memory size of a single GPU.

On the other hand, several research efforts have focused on improving DPC by exploring distributed computing power or introducing efficient data structures. EDDPC [10] leverages distributed machines to help DPC handle large scale datasets by filtering unnecessary distance computations. LSH-DDP [3] combines the power of locality sensitive hashing and the advantage of MapReduce distributed computing to support highly efficient but approximate DP clustering. FDDP [11] maps multi-dimensional data points into one-dimensional z-value and employs z-value index to filter invalid distance evaluation. Amagata et al. [13] propose three parallel DPC algorithms, Ex-DPC, Approx-DPC, and S-Approx-DPC. They outperform state-of-the-art DPC by employing the power of parallel computation. EDMStream [12] improves DPC by efficiently maintaining a novel in-memory dependent-tree structure and further supports stream clustering. It is designed based on CPU and processes new insertions sequentially one by one.

Comparison with the original paper. This paper is an extension of [43], in which we introduced GPU-accelerated Density Peak Clustering (GDPC). In the paper, we present the complete work of GDPC and propose a GDPC-based incremental clustering algorithm to deal with evolving data, providing a broader application scenario for GDPC.

8. Conclusion

In this paper, we propose a parallel density peaks algorithm named GDPC, which can fully utilize the powerful computation resources of GPU. It leverages a GPU-friendly spatial index VP-Tree to reduce unnecessary distance calculations. The VP-Tree construction process and the DP clustering process are greatly improved by utilizing GPU's parallel optimizations. Our results

show that GDPC can achieve 5.3–148.9X speedup over the state-of-the-art DPC implementations and our Incremental GDPC can achieve 2.3–40.5X speedup over another incremental clustering algorithm EDMStream.

CRediT authorship contribution statement

Zhuojin Liu: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Shufeng Gong:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing. **Yuxuan Su:** Methodology, Software, Writing – original draft. **Changyi Wan:** Methodology, Software, Writing – original draft. **Yanfeng Zhang:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Ge Yu:** Project administration, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (62072082, U2241212, U1811261, 62202088), the Key R&D Program of Liaoning Province (2020JH2/10100037), and the Fundamental Research Funds for the Central Universities (N2216015, N2216012).

References

[1] S. Lloyd, Least squares quantization in PCM, IEEE Trans. Inform. Theory 28 (2) (1982) 129–137.
[2] M. Ester, H.P. Kriegel, J. Sander, X. Xu, et al., A density-based algorithm for discovering clusters in large spatial databases with noise, in: Conference on Knowledge Discovery and Data Mining, 1996, pp. 226–231.
[3] Y. Zhang, S. Chen, G. Yu, Efficient distributed density peaks for clustering large data sets in MapReduce, IEEE Trans. Knowl. Data Eng. 28 (12) (2016) 3218–3230.
[4] Y. Wang, Y. Gu, J. Shun, Theoretically-efficient and practical parallel DBSCAN, in: 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 2555–2571.
[5] J. Gan, Y. Tao, DBSCAN revisited: Mis-claim, un-fixability, and approximation, in: 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 519–530.
[6] A. Rodriguez, A. Laio, Clustering by fast search and find of density peaks, Science 344 (6191) (2014) 1492–1496.
[7] D. Kobak, W. Brendel, C. Constantinidis, C.E. Feierstein, A. Kepecs, Z.F. Mainen, R. Romo, X.L. Qi, N. Uchida, C.K. Machens, Demixed principal component analysis of population activity in higher cortical areas reveals independent representation of task parameters, 2014, arXiv preprint arXiv:1410.6031.
[8] K. Sun, X. Geng, L. Ji, Exemplar component analysis: A fast band selection method for hyperspectral imagery, Geosci. Remote Sens. Lett. 12 (5) (2015) 998–1002.
[9] K.M. Dean, L.M. Davis, J.L. Lubbeck, P. Manna, P. Friis, A.E. Palmer, R. Jimenez, High-speed multiparameter photophysical analyses of fluorophore libraries, Anal. Chem. 87 (10) (2015) 5026–5030.
[10] S. Gong, Y. Zhang, EDDPC: An efficient distributed density peaks clustering algorithm, J. Comput. Res. Dev. 53 (6) (2016) 1400–1409.
[11] J. Lu, Y. Zhao, K. Tan, Z. Wang, Distributed density peaks clustering revisited, IEEE Trans. Knowl. Data Eng. 34 (8) (2022) 3714–3726.
[12] S. Gong, Y. Zhang, G. Yu, Clustering stream data by exploring the evolution of density mountain, VLDB Endow. 11 (4) (2017) 393–405.
[13] D. Amagata, T. Hara, Fast density-peaks clustering: Multicore-based parallelization approach, in: 2021 ACM SIGMOD International Conference on Management of Data, 2021, pp. 49–61.
[14] Q. Wang, X. Ai, Y. Zhang, J. Chen, G. Yu, HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management, CoRR (2022) abs/2208.14935.
[15] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, G. Yu, NeutronStar: Distributed GNN Training with Hybrid Dependency Management, in: 2020 ACM SIGMOD International Conference on Management of Data, 2022, pp. 1301–1315.
[16] Nvidia A100 Tensor Core GPU, URL https://www.nvidia.com/en-us/data-center/a100/.
[17] M. Li, J. Huang, J. Wang, Paralleled fast search and find of density peaks clustering algorithm on GPUs with CUDA, in: 17th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2016, pp. 313–318.
[18] K. Ge, H. Su, D. Li, X. Lu, Efficient parallel implementation of a density peaks clustering algorithm on graphics processing unit, Front. Inform. Technol. Electron. Eng. 18 (7) (2017) 915–927.
[19] P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 311–321.
[20] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.
[21] S.M. Omohundro, Five Balltree Construction Algorithms, International Computer Science Institute Berkeley, 1989.
[22] N. Kumar, L. Zhang, S.K. Nayar, What is a good nearest neighbors algorithm for finding similar patches in images? in: European Conference on Computer Vision, 2008, pp. 364–378.
[23] K. Fischer, B. Gärtner, M. Kutz, Fast smallest-enclosing-ball computation in high dimensions, in: European Symposium on Algorithms, 2003 pp. 630–641.
[24] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.
[25] S.K. Prasad, M. McDermott, X. He, S. Puri, GPU-based parallel R-tree construction and querying, in: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 2015, pp. 618–627.
[26] L. Luo, M.D. Wong, L. Leong, Parallel implementation of R-trees on the GPU, in: 17th Asia and South Pacific Design Automation Conference, 2012, pp. 353–358.
[27] S. You, J. Zhang, L. Gruenwald, Parallel spatial query processing on GPUs using R-trees, in: 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, 2013, pp. 23–31.
[28] Z. Rasool, R. Zhou, L. Chen, C. Liu, J. Xu, Index-based solutions for efficient density peak clustering, IEEE Trans. Knowl. Data Eng. 34 (5) (2022) 2212–2226.
[29] I. Kramosil, J. Michálek, Fuzzy metrics and statistical metric spaces, Kybernetika 11 (5) (1975) 336–344.
[30] A.W. Fu, P.M. Chan, Y. Cheung, Y.S. Moon, Dynamic VP-tree indexing for n-nearest neighbor search given pair-wise distances, VLDB J. 9 (2) (2000) 154–173.
[31] CUDA UnBound (CUB) library, URL https://nvlabs.github.io/cub/index.html.
[32] K-means properties on six clustering benchmark datasets, URL http://cs.uef.fi/sipu/datasets/.
[33] UCI machine learning repository, URL https://archive.ics.uci.edu/ml/datasets.php.
[34] BigCross500K & Facial, URL https://github.com/IGDPC/DataSets.git.
[35] Harvard Dataverse, http://dx.doi.org/10.7910/DVN/J0HC23.
[36] F. Cao, A.K.H. Tung, A. Zhou, Scalable clustering using graphics processors, in: International Conference on Advances in Web-Age Information Management, 2006, pp. 372–384.
[37] R. Farivar, D. Rebolledo, E. Chan, R.H. Campbell, A parallel implementation of K-means clustering on GPUs, in: International Conference on Parallel and Distributed Processing Techniques and Applications, 2008, pp. 340–345.
[38] C. Lutz, S. Breß, T. Rabl, S. Zeuch, V. Markl, Efficient K-means on GPUs, in: International Workshop on Data Management on New Hardware, 2018, pp. 1–3.
[39] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, L. Rocha, G-DBSCAN: A GPU accelerated algorithm for density-based clustering, Procedia Comput. Sci. 18 (2013) 369–378.
[40] M. Gowanlock, Hybrid CPU/GPU clustering in shared memory on the billion point scale, in: International Conference on Supercomputing, 2019 pp. 35–45.
[41] C. Taylor, M. Gowanlock, Accelerating the Yinyang K-means algorithm using the GPU, in: International Conference on Data Engineering, 2021, pp. 1835–1840.
[42] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, T. Mytkowicz, Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup, in: International Conference on Machine Learning, 2015, pp. 579–587.
[43] Y. Su, Y. Zhang, C. Wan, G. Yu, GDPC: A GPU-accelerated density peaks clustering algorithm, in: International Conference on Database Systems for Advanced Applications, 2020, pp. 305–313.

Zhuojin Liu is currently working towards a graduate degree in computer science at Northeastern University. Her research interests include parallel and high-performance computing. She works on designing efficient algorithms and data structures for processing large-scale data sets.

Shufeng Gong received the Ph.D. degree in computer science from Northeastern University, China, in 2021. He is currently a lecturer with Northeastern University, Key Laboratory of Intelligent Computing in Medical Image, Ministry of Education, China. His research interests include cloud computing, distributed graph processing, and data mining.

Yuxuan Su received the master degree in computer technology from Northeastern University, China, in 2021. Her research interests include database indexing and high-performance computing.

Changyi Wan received the master degree in computer technology from Northeastern University, China, in 2021. His main area of research includes high-performance computing and machine learning.

Yanfeng Zhang received the Ph.D. degree in computer science from Northeastern University, China, in 2012. He is currently a professor with Northeastern University, Key Laboratory of Intelligent Computing in Medical Image, Ministry of Education, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in SoCC 2011 was honored with 'Paper of Distinction'.

Ge Yu (Senior Member, IEEE) received the Ph.D. degree in computer science from Kyushu University, Japan, in 1996. He is currently a professor with Northeastern University, China. His current research interests include distributed and parallel systems, cloud computing and big data management, blockchain techniques and systems. He has published more than 200 papers in refereed journals and conferences. He is the ACM member, the IEEE senior member, and the CCF fellow.