GDPC: A GPU-Accelerated Density Peaks Clustering Algorithm

Yuxuan Su, Yanfeng Zhang(✉), Changyi Wan, and Ge Yu

Northeastern University, Shenyang, China
{yuxuansu,wanchangyi}@stumail.neu.edu.cn, {zhangyf,yuge}@mail.neu.edu.cn

Abstract. Density Peaks Clustering (DPC) is a recently proposed clustering algorithm that has distinctive advantages over existing clustering algorithms. 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 the efficiency problem of DPC, we propose to use GPU to accelerate DPC. We exploit a spatial index structure VP-Tree to help efficiently maintain the data points. We first propose a vectorized GPU-friendly VP-Tree structure, based on which we propose GDPC algorithm, where the density ρ and the dependent distance δ can be efficiently computed by using GPU. Our results show that GDPC can achieve over 5.3–78.8× acceleration compared to the state-of-the-art DPC implementations.

1 Introduction

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 and (ii) the dependent distance δ. 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.

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

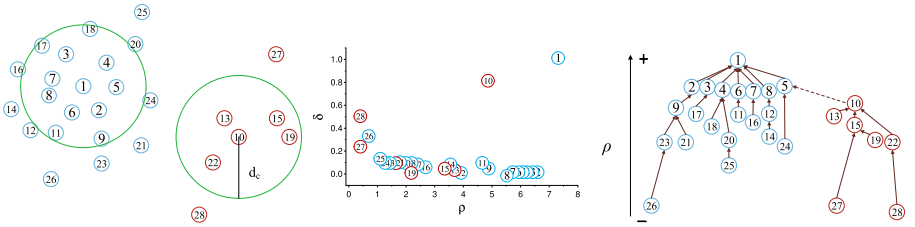
where d_{ij} is the distance from point p_i to point p_j, and d_c is called the cutoff distance. The *dependent distance* δ_i of point p_i is computed as

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

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 the 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.

Figure 1 illustrates the process of DPC through a concrete example. Figure 1a 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. 1b. 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

Y. Nah et al. (Eds.): DASFAA 2020, LNCS 12112, pp. 305–313, 2020.
https://doi.org/10.1007/978-3-030-59410-7_21



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

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

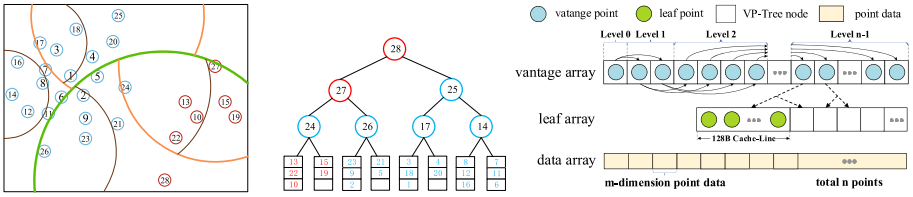
is only dependent on a single point, we can obtain a dependent tree [2] rooted by the absolute density peak as shown in Fig. 1c. The height of each point implies the density. The length of each link implies the dependent distance. For each point there is a dependent chain ending at a density peak. Then each remaining point is assigned to the same cluster as its dependent point.

Compared with previous clustering algorithms [5], 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.

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 the points based on their density values (with computational cost $O(Nlog(N))$) and $\frac{N(N-1)}{2}$ comparison operations are required. As a result, it can be very time consuming to perform DPC for large data sets. The recent advance of GPU technology is offering great prospects in parallel computation. There exist several related works [1,4] having been devoted to accelerate DPC using GPU's parallelization ability. However, these methods only consider utilizing GPU's hardware features to accelerate DPC without paying attention to parallelizable index structures that can maximize GPU performance.

In this paper, we exploit a spatial index structure vantage point tree (VP-Tree) [7] to help efficiently maintain clustering data. With VP-Tree, data points are partitioned into "hypershells" with decreasing radius. Comparing with other spatial index structures, VP-Tree is more appropriate in DPC algorithm, because the decreasing-radius hypershell structure can well support the point density computation (that obtains a point's nearby points within a pre-defined radius) and the dependent distance computation (that obtains the distance to a nearest neighbor with higher density). More importantly, the construction and the search

of VP-Tree can be well parallelized to adapt to GPU's structure. Based on the GPU-based VP-Tree, we propose *GDPC* algorithm, where the density ρ and the dependent distance δ can be efficiently calculated. Our results show that GDPC can achieve over 5.3–78.8× acceleration compared to the state-of-the-art DPC implementations.



(a) Partition layout (b) VP-Tree index (c) Vectorized VP-Tree

Fig. 2. VP-Tree

2 Vectorized VP-Tree Layout

In DP clustering, the calculations of the density value ρ and the dependence 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. A common approach for speeding up NN search is to exploit spatial index.

Based on our observation and analysis, Vantage Point Tree (VP-Tree) [7] is the best spatial index candidate. Each node of the tree contains one of the data points, and a radius. Under the left child are all points which 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 firstly 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. 2b. 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 are 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. 2b. The tree only requires a distance function that satisfies the properties of a metric space [3]. It does not need to find bounding shapes (hyperplanes or hyperspheres) or find points midway between them. Furthermore, the construction and the search of VP-Tree can be efficiently parallelized with CUDA since only a few data dependencies are required to handle.

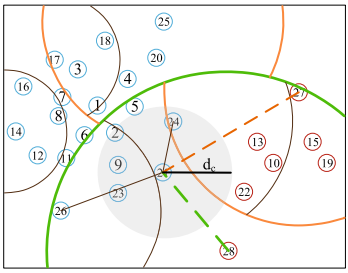


Fig. 3. An illustrative example of using VP-Tree (better with color) (Color figure online)

In the original VP-Tree, a child node reference is a pointer referring to the location of 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. Therefore, a vectorized GPU-friendly VP-Tree structure is desired instead of the pointer-based tree structure. In our approach as shown in Fig. 2c, 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.

3 GDPC Based on VPTree

3.1 Computing Density Values ρ

Our basic idea is to utilize the VP-Tree index to avoid unnecessary distance measurements. We illustrate the use of existing VP-Tree through an illustrative example. When computing a point's density value, it is required to access the points in point 21's d_c range. Let us compute point 21's density value (i.e., count the number of points within the grey circle) based on an existing VP-Tree's space partition result as shown in Fig. 3. We first evaluate the distance from point 21 to the level-0 vantage point 28. Since the grey circle with radius d_c is totally inside the level-0 ball (with green arc line), i.e., $|p21, p28| + d_c \leq vantage[0].radius$ where $|\cdot|$ is distance measurement, 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 grey circle, i.e., $|p21, p27| - d_c \leq vantage[1].radius$ (the grey circle has a part inside the orange ball) and $|p21, p27| + d_c \geq vantage[1].radius$ (the grey 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 grey circle is totally inside vantage point 24's ball

Algorithm 1: GDPC Algorithm based on VP-Tree

Input: cut-off distance dc, data array $data[]$, vantage array $vantage[]$, vatange array length n, and leaf array $leaf[]$
Output: density $\rho[]$, dependent distance $\delta[]$, point-cluster assignment $cluster[]$

1 **foreach** *point pid* ***parallel*** **do**
2 Stack S.push(0) ; `// push root node id into stack`
3 **while** S *is not empty* **do**
4 $i \leftarrow S$.pop();
5 **if** $i \geq n$ **then**
6 $cover\_leafs$.append$(i - n)$; `// this is a covered leaf node`
7 **if** $\left|data[vantage[i].id], data[pid]\right| - d_c \leq vantage[i].radius$ **then**
8 S.push$(2i + 1)$; `// search left child node`
9 **if** $\left|data[vantage[i].id], data[pid]\right| + d_c \geq vantage[i].radius$ **then**
10 S.push$(2i + 2)$; `// search right child node`
11 $\rho[pid] \leftarrow$ count the number of points in all $leaf[l] \in cover\_leafs$ whose distance to *pid* is less than d_c
12 $peak\_candidates \leftarrow \emptyset$; `// initialize the density peak candidates set`
13 **foreach** *point pid* ***parallel*** **do**
14 $dep[pid], \delta[pid] \leftarrow$ find the nearest neighbor point that has higher density than $\rho[pid]$ in all $leaf[l] \in cover\_leafs$ and compute its distance ;
15 **if** *point pid has the highest density among the covered leaf nodes* **then**
16 add *pid* into $peak\_candidates$;
17 **foreach** *point pid* $\in peak\_candidates$ ***parallel*** **do**
18 $dep[pid], \delta[pid] \leftarrow$ find the nearest neighbor point that has higher density than $\rho[pid]$ among all points and compute its distance ;
19 $peak[] \leftarrow$ determine the density peaks which have both larger ρ and larger δ;
20 $cluster[] \leftarrow$ assign points to clusters based on $peak[]$ and $dep[]$;

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 point 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 1. Line 1–11 depicts the ρ computation process.

We design a GPU-friendly search algorithm on the vectorized VP-Tree to achieve high parallelism and coalesced memory access. Specifically, we use several parallel optimizations: **1) Arrange calculation order** (Line 1). During tree traversal, if multiple threads in a warp execute random queries, it is difficult to achieve a coalesced memory access because they might traverse the tree along different paths. If multiple queries share the same traversal path, the memory accesses can be coalesced 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. By this way, we can mitigate warp divergence. **2) Ballot-Counting optimization** (Line 11). 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 intrinsic operation can enable an efficient implement of the per-block scan. By combining the `__ballot()` and `__popc()` intrinsics, we can efficiently count the number of points within d_c. **3) Fully contained leaf nodes**. In original VP-Tree, the points in vantange 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. In our implementation, the vantage points also have their copies in leaf nodes to avoid memory divergence.

3.2 Computing Dependent Distances δ

Given the computed density values, we can calculate the dependent distance values as shown in Line 12–18 in Algorithm 1. Recall that the dependent distance of a point is its distance to the nearest neighbor with higher density as shown in Eq. (2). We can again leverage the VP-Tree index. Since a point's nearest neighbor with higher density is highly likely to be in its leaf nodes, we first locally search among its leaf nodes (Line 14). If such a point does not exist (when the point has the highest density among its leaf nodes), we sort the candidate points in the descending order of their density values and then search globally by checking all the other points with higher density (Line 18).

3.3 Assigning Points to Clusters

After picking a set of density peaks (Line 19), we should assign each point to a certain cluster. We perform this by tracing the assignment chain till meeting a certain density peak. The assignment dependency relationship (as shown in Fig. 1c) is recorded when computing δ (Line 14, 18). To adapt to GPU's parallelization ability, 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 propagations on different sub-trees are totally independent.

4 Experiments

Preparation. We conduct all experiments on an 8-core server with an NVIDIA RTX2080Ti GPU. Our implementation is compiled by CUDA 10 along with nvcc optimization flag -O3. Table 1 lists the data sets used in our experiments. These include two small sized 2D data sets, and four real world large high-dimensional data sets, all of which are available online. To obtain clustering

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

result in a reasonable time period, we construct a smaller BigCross500K data set by randomly sampling 500,000 points from the original data set.

Performance Comparison with State-of-the-Art DPC Methods. We first compare with the state-of-the-art GPU-based DPC algorithm CUDA-DP [1] on smaller datasets, which returns out-of-memory error on larger datasets 3Dspatial, KDD, and BigCross500K. CUDA-DP just optimizes the distance calculations without leveraging spatial index structure. In Fig. 4, we can see that our GDPC can achieve a 5.3×–17.8× speedup attributed to our vectorized VP-Tree design and GPU-friendly parallel algorithm. In addition, in order to evaluate our algorithm on larger datasets, we compare with a state-of-the-art distributed DP clustering algorithm LSH-DDP [8], which is implemented based on Hadoop

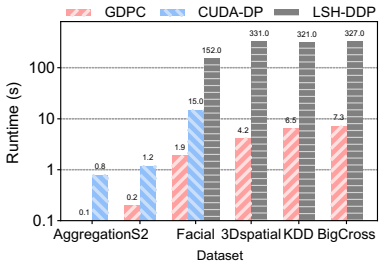


Fig. 4. Runtime comparison

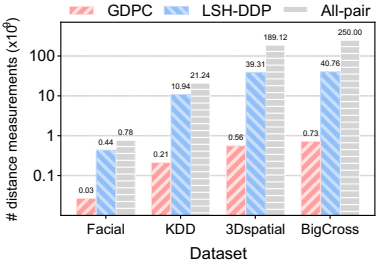


Fig. 5. Computational cost

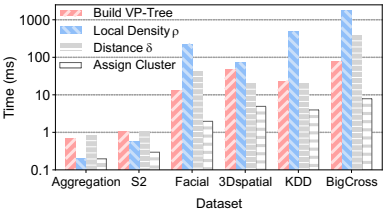


Fig. 6. Runtime breakdown

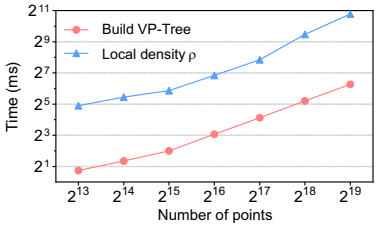


Fig. 7. Scaling performance

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, each equipped with an Intel I5-4690 3.3G 4-core CPU, 4 GB memory. We can see that our GDPC can achieve 44.8–78.8× speedup.

Computational Cost Analysis. The distance calculations for computing ρ and δ is the most expensive part, especially for large and high-dimensional data. GDPC utilizes VP-Tree to avoid 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. 5. Our GDPC requires significantly fewer exact distance calculations than prior works, say only 1.4–6.8% of LSH-DDP and 0.3–3.8% of all-pair calculations.

Runtime Breakdown Analysis. 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. In Fig. 6, we can see that for larger dataset, the ρ computation is always the most expensive part since it requires large number of distance measurements.

Scaling Performance. We randomly choose 2^{13}–2^{19} number of points from the BigCroos500K dataset to generate multiple datasets with different sizes. The runtime in Fig. 7 exhibits linearly growth when increasing the data size, while the all-pair distance calculations will exhibit quadratic growth. This experiment shows our GDPC algorithm can achieve great scaling performance.

Effect of Multi-stream Construction. We leverage CUDA's multi-stream optimization to improve the parallelism when constructing left child sub-tree and right child sub-tree. We can see the results in Fig. 8 that multi-stream optimization can significantly improve the performance by an order of magnitude.

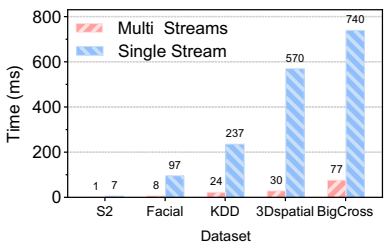


Fig. 8. Effect of multi-stream processing

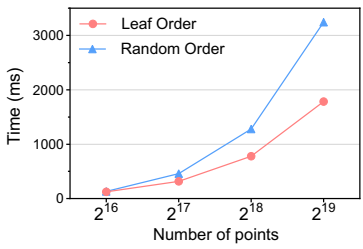


Fig. 9. Effect of memory access

Effect of Coalesced Memory Access. To understand the performance improvement by our optimization, we use the random processing order to see the advantages by using the leaf-based processing order. In Fig. 9, our approach shows significant better performance especially when increasing the data size.

5 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 the unnecessary distance calculations. The VP-Tree construction process and the DP clustering process are greatly improved by utilizing GPU's parallel optimizations. Our result show that GDPC can achieve 5.3–78.8× speedup over the state-of-the-art DPC implementations.

Acknowledgements. This work was partially supported by National Key R&D Program of China (2018YFB1003404), National Natural Science Foundation of China (61672141), and Fundamental Research Funds for the Central Universities (N181605017, N181604016).

References

1. Ge, K., Su, H., Li, D., Lu, X.: Efficient parallel implementation of a density peaks clustering algorithm on graphics processing unit. Front. Inf. Technol. Electron. Eng. **18**(7), 915–927 (2017). https://doi.org/10.1631/FITEE.1601786
2. Gong, S., Zhang, Y., Yu, G.: Clustering stream data by exploring the evolution of density mountain. Proc. VLDB Endow. **11**(4), 393–405 (2017)
3. Kramosil, I., Michálek, J.: Fuzzy metrics and statistical metric spaces. Kybernetika **11**(5), 336–344 (1975)
4. Li, M., Huang, J., Wang, J.: Paralleled fast search and find of density peaks clustering algorithm on gpus with CUDA. In: SNPD '2016. pp. 313–318 (2016)
5. Patil, C., Baidari, I.: Estimating the optimal number of clusters k in a dataset using data depth. Data Sci. Eng. **4**(2), 132–140 (2019)
6. Rodriguez, A., Laio, A.: Clustering by fast search and find of density peaks. Science **344**(6191), 1492–1496 (2014)
7. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA '93. pp. 311–321 (1993)
8. Zhang, Y., Chen, S., Yu, G.: Efficient distributed density peaks for clustering large data sets in mapreduce. IEEE Trans. on Knowl. Data Eng. **28**(12), 3218–3230 (2016)