软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(3):1037−1057 [doi: 10.13328/j.cnki.jos.007512] [CSTR: 32375.14.jos.007512]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

GPU 加速的高维向量聚类算法*

李忠根[1], 龚盛豪[2], 于浩然[1], 朱轶凡[2,3], 柳 晴[1,3], 高云君[1,2,3]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](浙江大学 软件学院, 浙江 宁波 315048)

[3](全省大数据智能计算重点实验室 (浙江大学), 浙江 杭州 310027)

通信作者: 朱轶凡, E-mail: xtf_z@zju.edu.cn

摘 要: 聚类是大规模高维向量数据分析的关键技术之一. 近年来, 基于密度的聚类算法 DBSCAN (density-based spatial clustering of applications with noise) 因其无须预先指定聚类数量、能够发现复杂聚类结构并有效识别噪声点的特性, 在数据分析领域得到了广泛应用. 然而, 现有的基于密度的聚类算法在处理高维向量数据时将产生极高的时间代价且面临维度灾难等问题, 难以在实际场景中部署应用. 此外, 随着信息技术的发展, 高维向量数据规模急剧增加, 使用 CPU 进行高维向量聚类在时间代价和可扩展性等方面将面临更大的挑战. 为此, 提出一种 GPU 加速的高维向量聚类算法, 通过引入 K 近邻 (K-nearest neighbor, KNN) 图索引加速 DBSCAN 的计算. 首先, 设计了 GPU 加速的并行 K 近邻图构建算法, 显著降低了 K 近邻图索引的构建开销. 其次, 提出了基于层间并行的 K-means 树分区算法及基于广度优先搜索和核心近邻图的并行聚类算法, 改进了 DBSCAN 算法的计算流程, 实现了高并发向量聚类. 最后, 在真实向量数据集上进行了大量实验, 并将所提出的方法与现有方法进行了性能对比. 实验结果表明, 所提方法在保证聚类精度的前提下, 将大规模向量聚类的效率提高了 5.7–2 822.5 倍.

关键词: 基于密度的聚类; 高维向量; GPU 加速; 并行计算; K 近邻图

中图法分类号: TP311

中文引用格式: 李忠根, 龚盛豪, 于浩然, 朱轶凡, 柳晴, 高云君. GPU加速的高维向量聚类算法. 软件学报, 2026, 37(3): 1037–1057. http://www.jos.org.cn/1000-9825/7512.htm

英文引用格式: Li ZG, Gong SH, Yu HR, Zhu YF, Liu Q, Gao YJ. GPU-accelerated Clustering Algorithm for High-dimensional Vectors. Ruan Jian Xue Bao/Journal of Software, 2026, 37(3): 1037–1057 (in Chinese). http://www.jos.org.cn/1000-9825/7512.htm

GPU-accelerated Clustering Algorithm for High-dimensional Vectors

LI Zhong-Gen[1], GONG Sheng-Hao[2], YU Hao-Ran[1], ZHU Yi-Fan[2,3], LIU Qing[1,3], GAO Yun-Jun[1,2,3]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](School of Software Technology, Zhejiang University, Ningbo 315048, China)

[3](Zhejiang Key Laboratory of Big Data Intelligent Computing (Zhejiang University), Hangzhou 310027, China)

Abstract: Clustering serves as one of the critical technologies for large-scale, high-dimensional vector data analysis. Recently, a density-based clustering algorithm DBSCAN (density-based spatial clustering of applications with noise) has been widely adopted in data analysis due to their advantages of not requiring pre-specified cluster numbers, discovering complex cluster structures, and identifying noise points. However, existing density-based clustering algorithms suffer from high computational costs when processing high-dimensional vectors. Meanwhile, these methods also face challenges like the “curse of dimensionality”, restricting their practical applications. With the rapid growth of high-dimensional vector data in the era of information technology, CPU-based clustering approaches encounter increasing

* 基金项目: 国家自然科学基金 (62025206, U23A20296, 62302444); 浙江省尖兵领雁项目 (2024C01259, 2025C01195)
李忠根和龚盛豪为共同第一作者.
本文由“向量数据库及 DB4LLM 技术”专题特约编辑高宏教授、李国良教授、张蓉教授推荐.
收稿时间: 2025-05-01; 修改时间: 2025-06-30; 采用时间: 2025-08-20; jos 在线出版时间: 2025-09-02
CNKI 网络首发时间: 2025-12-04

challenges in time efficiency and scalability. To address these issues, this study proposes a GPU-accelerated clustering algorithm for high-dimensional vector data, introducing the K-nearest neighbor (KNN) graph index to accelerate DBSCAN. First, a GPU-accelerated parallel KNN graph construction algorithm is developed, significantly reducing the index construction overhead. Furthermore, to enhance the pipeline of DBSCAN and achieve highly concurrent vector clustering, a K-means tree partitioning algorithm with inter-layer parallelism and a parallel clustering algorithm based on breadth-first search and a core KNN graph are designed. Finally, extensive experiments are conducted on real-world datasets, and the proposed method is compared against existing approaches. Experimental results show that the proposed algorithm improves the efficiency of large-scale vector clustering by 5.7–2 822.5 times while maintaining clustering accuracy.

Key words: density-based clustering; high-dimensional vector; GPU acceleration; parallel computing; K-nearest neighbor (KNN) graph

高维向量聚类是大规模向量数据分析的关键技术之一, 其通过将具有相似特征的向量划分至同一类别来揭示数据分布的内在规律, 为异常检测、群体偏好识别和个性化推荐等数据分析任务提供技术支撑[1]. 基于密度的空间聚类算法 DBSCAN (density-based spatial clustering of applications with noise)[2]因其无须预先设定聚类数量、可识别复杂簇结构及噪声点的优势, 已在自然语言处理[3,4]、图像处理[5,6]、音频分析[7]、生物数据分析[8,9]等领域获得广泛应用. 随着人工智能与大数据技术的迅速发展, 文本、图像等数据向量化后的维度普遍达到数百维[10]. 同时, 向量数据集的数据规模亦普遍达到百万以上[11]. 然而, DBSCAN 作为基于密度的聚类算法, 在面临大规模高维向量时存在计算效率低下的问题, 难以满足实际应用场景中的高效聚类需求[12]. 因此, 提高 DBSCAN 算法在面对大规模高维向量时的计算效率成为亟待解决的重要课题.

DBSCAN 算法的计算开销主要来源于对每个数据对象的邻域计算[13,14]. 为了降低计算代价, 现有的方法多采用基于树或网格的索引加速 DBSCAN 的邻域计算[15–18]. 然而, 基于树或网格的索引结构在处理高维向量数据时易受维度灾难的影响, 导致近邻查询性能显著下降[19,20]. 为了克服维度灾难带来的性能退化问题, 将近邻图结构(如 ε 近邻图[21]、K 近邻 (K-nearest neighbor, KNN) 图[22]、HNSW (hierarchical navigable small world graph, HNSW)[20]等) 作为 DBSCAN 计算的索引结构已成为主流解决方案. 然而, 现有研究普遍基于 CPU 架构进行近邻图构建以及 DBSCAN 聚类. 由于 CPU 少量核心导致的有限并行计算能力以及图索引构建较高的计算复杂度[20–22], 使其难以满足大规模高维向量的高效聚类需求[23].

为了提升大规模向量数据的聚类效率, 研究者们开始利用 GPU 的强大并行能力加速 DBSCAN 算法[24–27]. 将数据点的邻域计算限定在子区域内部, 并行计算各数据点的近邻距离[28,29], 从而避免了各数据点相对于数据集内所有数据点的距离计算, 将时间复杂度降低至 $O(n^2)$ 以下. 然而, 在子区域划分过程中, 已有的基于 GPU 的 DBSCAN 算法受到高维空间维度灾难的影响. 此外, 现有的基于 GPU 的 DBSCAN 算法并不支持基于图的索引结构, 对高维向量数据的聚类效率低下.

因此, 为了突破 GPU 加速的高维向量聚类的技术瓶颈, 需要解决以下 3 个核心挑战.

(1) 如何高效构建近邻图索引. 引入近邻图索引可显著减小 DBSCAN 算法中的邻域计算代价, 且其对高维数据的查询可保持较高的精度. 然而近邻图的构建需处理大规模距离计算与邻居列表的动态频繁更新. 现有的基于 CPU 的方法计算效率低, 而现有的基于 GPU 的方法仅将距离计算等操作转移至 GPU 设备, 邻居列表更新仍依赖 CPU 处理, 导致频繁的 PCIe 数据传输与同步开销.

(2) 如何高效划分大规模高维向量数据. K-means 树由于其简洁而高效的实现及不受维度灾难影响的特性而成为对高维数据分区的流行方法[30,31]. 然而, 随着分区粒度细化, 参与并行计算的数据点数量呈指数衰减, 致使 GPU 资源利用率降低, 造成资源利用与计算效率的双重恶化.

(3) 如何实现大规模高维向量的并行聚类. 在各分区的局部聚类阶段, 需设计基于 K 近邻图的高效并行遍历策略, 以实现局部簇计算. 此外, 在局部簇合并阶段, 现有方法依赖于重叠分区检测并涉及大量串行操作, 严重制约了 GPU 的并行吞吐量, 产生较高的合并开销.

为了解决上述挑战, 本文针对高维向量提出了一种基于 GPU 架构和 K 近邻图索引加速的 DBSCAN 算法 (KNN graph-based and GPU-accelerated DBSCAN, KG-DBSCAN). 该算法首先利用 GPU 加速 K 近邻图索引的构建, 将 K 近邻图构建的全流程转移至 GPU, 并设计了偏好采样策略以加速近邻节点更新, 同时利用 GPU 加速高维向量

间的距离计算; 其次, 为了高效地划分大规模数据, 设计了层间并行计算模式, 利用 GPU 中的 Tensor 核心加速 K-means 树的计算, 使得 GPU 能够对每一层的树节点进行并行计算, 避免了数据点规模的指数衰减对并行计算效率产生的影响; 最后, 基于 K 近邻图索引进行并行广度优先遍历, 加速分区局部 DBSCAN 计算, 利用不同分区的核心点之间的密度直达关系构建核心近邻图, 基于核心近邻图合并局部计算结果, 减少了遍历所有数据点带来的计算开销.

本文的主要贡献总结为以下 4 点.

(1) 设计了一种 GPU 加速的 K 近邻图构建算法. 该算法利用 GPU 和偏好采样策略协同加速向量间距离计算与近邻更新, 显著提高了计算效率.

(2) 设计了一种 Tensor 核心增强的层间并行 K-means 树分区算法. 层间并行计算模式避免了数据点规模的指数衰减对并行计算产生的影响, 保证在生成 K-means 树的过程中对 GPU 资源的充分利用.

(3) 提出了基于广度优先遍历策略和核心近邻图的高效并行聚类方法. 并行广度优先遍历显著加速了分区局部 DBSCAN 计算. 同时, 核心近邻图减少了遍历所有数据点带来的计算开销, 极大地提升了聚类效率.

(4) 在 4 个真实数据集上开展了详尽的实验评估, 与现有方法进行了对比. 结果表明, 相比于现有的基于树索引和图索引及 GPU 加速的方法, 本文提出的方法在保证精度的同时, 显著提升了高维向量聚类的计算效率.

本文第 1 节介绍基于近邻图和 GPU 加速的 DBSCAN 的相关工作与研究现状. 第 2 节介绍 DBSCAN 算法及基于 K 近邻图的 DBSCAN 算法的相关基础知识. 第 3 节介绍 GPU 加速的高维向量聚类算法. 第 4 节进行实验分析, 验证并分析算法的性能. 第 5 节总结全文.

1 相关工作

本节总结与本文相关的研究工作, 第 1.1 节介绍基于近邻图的 DBSCAN 的相关工作, 第 1.2 节总结 GPU 加速的 DBSCAN 的相关工作.

1.1 基于近邻图的 DBSCAN 算法

Gan 等人[32]指出, 当数据维度大于 3 时, DBSCAN 的计算开销将显著增加, 这使得高维场景下的近似 DBSCAN 算法设计成为必然选择. 为了加速 DBSCAN 中开销最高的近邻查询, 现有研究使用多种数据结构优化近邻查询, 例如局部敏感哈希 (locality sensitive hashing, LSH)[33,34]、KD 树[15,16]、覆盖树[35]等. 然而在高维场景下, 基于哈希或树等数据结构的方法将面临维度灾难, 且存在效率低下的问题. 为了进一步加速高维空间向量聚类计算, PARDICLE[36]构建了 ε 近邻图, 并设计了一种密度估计方法, 在密集区域采样计算最近邻, 而在稀疏区域执行精确最近邻, 以降低 DBSCAN 的计算量. NG-DBSCAN[21]提出了一种快速构建 ε 近邻图的方法, 即随机初始化图中节点的近邻, 每次迭代时判断节点及其二阶邻居的距离是否小于 ε, 并根据判断结果更新节点的邻居列表, 直至达到既定迭代次数. 为了进一步减小 ε 近邻图的构造代价, SNG-DBSCAN[37]提出一种采样方法, 基于该方法构造了 ε 近邻图. KNN-DBSCAN[22]指出, 上述基于 ε 近邻图的方法在高维场景下对 ε 的取值十分敏感, ε 取值发生变动时即需重新构建近邻图结构, 而这将产生极高的索引构建开销. 为此, h-DBSCAN[19,20]使用 HNSW 图[38]作为索引, 基于该索引进行高效的近邻查询以加速 DBSCAN 算法. 为了避免额外的索引查询开销, KNN-DBSCAN[22]使用 K 近邻图作为索引结构, 引入了新的核心点、可达等 DBSCAN 相关概念的定义方式, 重新定义了基于密度的聚类, 基于最小生成树计算聚类. 然而, 这类方法在高维场景下仍面临索引构建低效的问题, 制约了聚类效率的提升.

1.2 GPU 加速的 DBSCAN 算法

CPU 架构受限于有限运算能力, 难以满足高效 DBSCAN 计算的需求. 由于 DBSCAN 的开销主要来源于近邻点搜索, 其中涉及大量的距离计算, 这促使研究者转向利用 GPU 的并行架构寻求加速. 现有的 GPU 加速 DBSCAN 的算法分为 3 类: 全局聚类计算、分区聚类计算和基于索引的聚类计算. 全局聚类计算利用 GPU 强大的计算能力遍历数据集为各数据点计算近邻点, 如 CUDA-DClust[39]和 Cal-DBSCAN[40]将每个数据点划分至不同的线程并行计算近邻点; G-DBSCAN[41]则遍历所有数据点以构造密度连通图, 而后在该近邻图上开展并行广度优先搜索, 以扩展簇的范围. 上述方法对所有数据点的遍历造成了大量的计算开销. 分区聚类计算首先将数据进行分区, 将近邻

点的计算限制在分区内, 最终合并各分区聚类结果. 例如, Mr. Scan[29]使用 MRNet 树将数据划分至各 GPU 节点, 各分区需在分区边界节点的基础上进一步扩展 ε 的范围, 称为 ε 邻域, 以保证准确识别核心点. 各 GPU 节点使用改进后的 CUDA-DClust 进行局部 DBSCAN 的计算, 最终通过判断分区间重叠数据点的性质进行簇归并; GSCAN[42]则使用网格划分数据, 计算簇时仅计算当前单元格及其邻近单元格点的距离; GPU Multi-grid[43]扩展了网格的定义, 实现了多层网格划分, 进一步限制搜索范围, 减小计算开销; Hybrid-DBSCAN[44]同样采用基于网格的方式, 不同的是其设计了一种 GPU-CPU 异构执行策略, 使用 GPU 计算近邻点, 而使用 CPU 进行聚类的计算; 为了提高分区效率, CudaSCAN[45]采用并行 KD 树划分数据集, 划分时与 Mr. Scan 采用相同的扩展 ε 邻域的策略, 合并簇时仅需检查各分区间的重叠数据. 分区聚类计算虽限制了近邻计算范围, 但其仍需遍历分区内所有数据以计算各数据点的近邻. 为了进一步减小近邻计算代价, 基于索引的聚类计算通过构建基于 GPU 的索引实现近邻查询的加速, 进而加速 DBSCAN 的计算. 例如, FDBSCAN 与 FDBSCAN-DenseBox[46]使用层次包装盒树 (bounding volume hierarchy based on tree, BVH) 作为索引结构, 每个线程分别负责一个数据点的近邻搜索; cuML-DBSACN[47]则采用一种更适合 GPU 并行计算的索引——随机球形覆盖 (random ball cover), 进一步提高了计算效率. 尽管上述方法通过不同路径提升性能, 其对于高维向量的聚类仍存在效率低下的问题.

2 基础知识

本节将在第 2.1 节介绍基于密度的聚类算法的相关概念和基础知识, 在第 2.2 节介绍利用 K 近邻图作为索引的 DBSCAN 算法的相关概念.

2.1 基于密度的聚类算法

DBSCAN 采用基于密度的思想, 将簇定义为密度相连点的最大集合, 能够发现任意形状的聚类, 并识别噪声点. DBSCAN 算法基于以下重要概念实现聚类.

定义 1 (**ε 邻域**). 给定数据点 p, 一个距离参数 ε, 数据点 p 的 ε 邻域为所有与 p 的距离小于等于 ε 的数据点组成的集合, 即 $N_\varepsilon(p)=\{q|d(p,q)\leqslant\varepsilon\}$, 其中 d 为距离函数.

定义 2 (**核心点**). 给定整数 $minPts$, 对于数据点 p, 如果 $N_\varepsilon(p)\geqslant minPts$, 则称数据点 p 为核心点; 反之, 如果 $N_\varepsilon(p)<minPts$, 则称 p 为非核心点.

定义 3 (**边界点**). 对于非核心点 q, 若存在一个核心点 p, 使得 q 位于 p 的 ε 邻域内, 即 $q\in N_\varepsilon(p)$, 则称数据点 q 为边界点.

定义 4 (**噪声点**). 对于非核心点 q, 若其不在任意一个核心点的 ε 邻域内, 则称该点为噪声点.

定义 5 (**密度直达**). 若 p 为核心点, q 位于 p 的 ε 邻域内, 则称 q 可由 p 密度直达.

定义 6 (**密度可达**). 若存在核心点序列 $\{p_1,p_2,\ldots,p_n\}$, 其中, $p_1=p$, $p_n=q$, 序列内任意一个点 p_i+1 都可由 p_i 密度直达, 则称 q 由 p 密度可达.

定义 7 (**密度相连**). 若存在数据点 o, 使得点 p 和点 q 都可由 o 密度可达, 则称 p 与 q 密度相连.

定义 8 (**基于密度的聚类**). 设集合 C 是所有数据点构成的集合的一个子集, 称集合 C 是一个基于密度的聚类簇, 当且仅当 C 满足以下 3 个条件: (1) C 中包含至少一个核心点 p; (2) C 中任意两个数据点均密度相连; (3) 不存在数据点 $s\notin C$, 且 s 与 C 内的数据点密度相连.

后文图 1 对上述定义进行展示, 该示例将 $minPts$ 设为 3, 共形成 2 个簇 C_1 与 C_2. 例如, 因为 p_1 位于 p_2 的 ε 邻域内, 所以 p_1 可由 p_2 密度直达. 而由于存在核心点序列 $\{p_3,p_2,p_1\}$, 其中, p_2 可由 p_3 密度直达, p_1 可由 p_2 密度直达, 所以 p_1 由 p_3 密度可达. 类似地, p_5 由 p_3 密度可达. 由于点 p_1 和点 p_5 都可由 p_3 密度可达, 因此 p_1 与 p_5 密度相连.

2.2 基于 K 近邻图的 DBSCAN 算法

为了实现基于 K 近邻图的 DBSCAN 算法, 本文在不改变结果的前提下对第 2.1 节的核心点定义进行了修改.

定义 9 (**K 近邻图核心点**). 给定正整数 $minPts<k$, 对于任意数据 p, 其在 K 近邻图中的邻居列表 $N_k(p)$ 按距离升序排序, 若 p 与其邻居列表中第 $minPts$ 个点 $N_k^{minPts}(p)$ 的距离 $d(p,N_k^{minPts}(p))\leqslant\varepsilon$, 则 p 为 K 近邻图核心点.

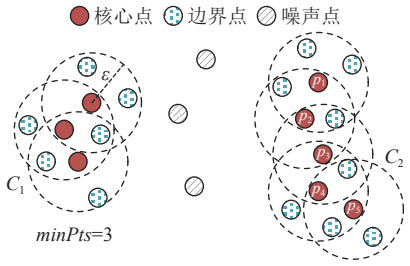


图 1 基于密度的聚类示例

定义 9 和定义 2 是等效的. 一方面, 根据定义 9, K 近邻图核心点 p 与其邻居列表中最近的 $minPts$ 个点的距离小于 ε, 则说明与数据 p 的距离小于 ε 的数据点的数量必定大于等于 $minPts$, 即 $N_\varepsilon(p) \geqslant minPts$, 因此符合定义 2. 另一方面, 根据核心点的定义 (定义 2), 与数据 p 的距离小于 ε 的数据点的数量大于等于 $minPts$, 则说明数据 p 的第 $minPts$ 近的邻居与 p 的距离必定小于 ε, 因此满足定义 9. 所以, 定义 2 与定义 9 是等效的.

根据定义 9, 核心点的判定从计算 ε 邻域内点的数量, 转换为判断与第 $minPts$ 个近邻点的距离是否在 ε 的范围内. 若点 p 为核心点, 对于其在 K 近邻图中的非核心点邻居 q, 若 q 满足 $q \in N_k(p) \wedge d(p,q) \leqslant \varepsilon$, 则 q 为边界点. 然而, 当点 p 的距离小于 ε 的近邻点数量大于 k, 即 $N_\varepsilon(p) > k$ 时, 其超出 k 的边界点将被识别为噪声点, 导致聚类精度降低. 为了进一步提高精度, 本文重新定义边界点 (定义 10), 以对噪声点进行后处理.

定义 10 (K 近邻图边界点). 给定非核心点 p, 若 p 在 K 近邻图的邻居 $N(p)$ 中存在核心点 q, 且两者的距离 $d(p,q) \leqslant \varepsilon$, 则点 p 为 K 近邻图边界点.

定义 10 与定义 3 是等效的. 一方面, 对于非核心点 p, 根据定义 10, 其在 K 近邻图的邻居中存在核心点 q, 且距离小于等于 ε, 则说明数据点 p 位于核心点 q 的 ε 邻域内, 即 $p \in N_\varepsilon(q)$, 因此满足定义 3. 另一方面, 根据定义 3, 边界点 p 位于核心点 q 的 ε 邻域内, 则 q 必定出现在点 p 在 K 近邻图的邻居中, 否则, 说明点 p 的 K 近邻均小于 $d(p,q) < \varepsilon$. 此时, 由于 $minPts < k$, 与点 p 的距离小于 ε 的点的数量大于 $minPts$, 因此点 p 为核心点, 这与 p 为边界点的前提矛盾. 故定义 3 中的边界点必定满足定义 10.

根据上述讨论, 修改后的定义 9 和定义 10 对于聚类结果而言, 与定义 2 和定义 3 是等效的.

2.3 GPU 架构

GPU 通常配备数十个流式多处理器 (streaming multiprocessor, SM), 每个流式多处理器作为独立处理单元, 其包含数百个计算核心以及独立的共享内存和寄存器结构. 在编程层面, 统一计算架构 (compute unified device architecture, CUDA) 对 GPU 硬件架构进行抽象化建模, 充当应用程序与 GPU 之间的桥梁. CUDA 编程模型将 32 个线程组织为线程组 (warp), 采用单指令多线程 (single instruction multiple thread, SIMT) 执行机制. 在 CUDA 架构中, 线程块 (block) 由多个线程组构成, 各线程块分别被分配至特定流式多处理器并行执行.

现代 GPU 通常包含两类常用的计算核心: CUDA 核心与 Tensor 核心. 其中, CUDA 核心作为通用计算任务的主要执行单元, 而 Tensor 核心则是为支持高效矩阵运算专门设计的, 其能在单个时钟周期内完成固定尺寸矩阵的乘法运算. GPU 存储层次由全局内存 (global memory)、共享内存 (shared memory) 及寄存器 (register) 构成: 全局内存虽具备 GPU 中最大存储容量, 但其读写带宽相对较低; 共享内存可供同一线程块内所有线程访问, 具有更高的带宽特性; 寄存器作为存储结构中访问速度最快的类型, 其存储空间一经声明即私有于各个线程.

3 GPU 加速的高维向量聚类算法

本节介绍 GPU 加速的高维向量数据聚类算法. 该算法分为 3 个模块, 分别进行 K 近邻图索引构建、数据分区、并行聚类计算. 算法流程如图 2 所示. 首先, 基于向量数据集进行 GPU 加速的 K 近邻图索引构建, 以利用 K 近邻图加速后续 DBSCAN 算法的近邻计算 (第 3.1 节). 同时, 为了充分利用 GPU 的并行性, 采用 K-means 树分区算法将向量数据集进行分区, 各分区分配给 GPU 中不同的线程块并行独立计算 (第 3.2 节). 在各分区内独立进行

广度优先搜索以扩展聚类簇范围, 形成分区内的局部聚类结果 (第 3.3.1 节). 最后, 基于核心近邻图进行局部聚类簇合并, 形成最终的聚类结果 (第 3.3.2 节).

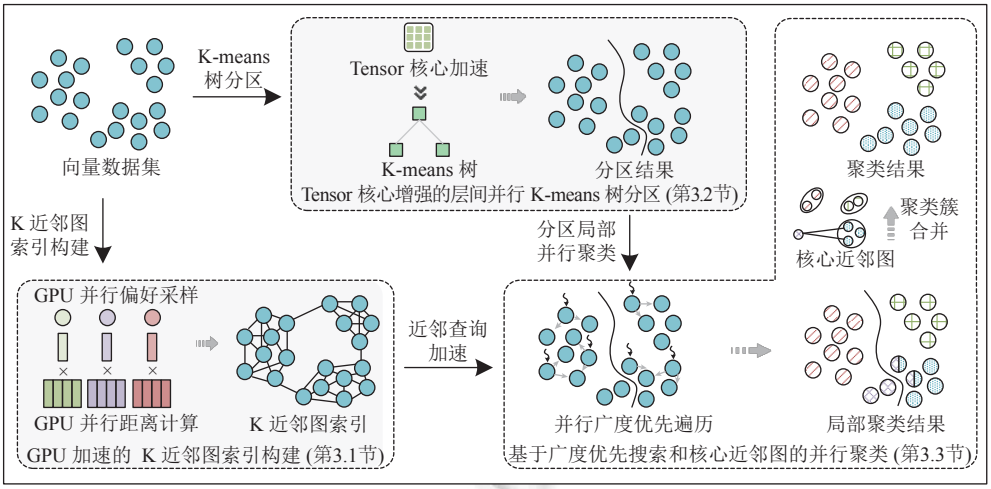


图 2 GPU 加速的高维向量聚类算法

3.1 GPU 加速的 K 近邻图索引构建

根据第 2.2 节的讨论, K 近邻图可作为 DBSCAN 算法的索引, 并显著加速聚类过程中开销最高的近邻计算. 为此, 本节首先进行 K 近邻图的构建. 然而, 现有的 K 近邻图构建算法普遍基于 CPU, 其在处理大规模高维向量时面临严重的效率瓶颈[22,48]. 为了突破 K 近邻图构建的效率瓶颈, 本文基于 NN-Descent[48]算法提出 GPU 加速的 K 近邻图并行计算优化框架, 以适配 GPU 高并发的计算特性.

NN-Descent 算法采用“邻居的邻居也可能是邻居”的思想, 首先随机初始化 K 近邻图, 而后图中每个节点对其二阶邻居进行随机采样, 并以相同的方式对该节点反向邻居的二阶邻居也进行随机采样, 计算与其采样得到的二阶邻居的距离, 而后将计算得到的距离和该节点与当前邻居的距离进行比较, 将距离更小的二阶邻居更新为其直接邻居. 同时, 按照相同的方式根据距离由小到大更新该节点的反向邻居. NN-Descent 算法基于 CPU 架构设计, 未考虑大规模并行计算. 为了使用 GPU 加速 NN-Descent 算法, 如图 3 所示, 本文提出的 GPU 加速的 K 近邻图索引构建算法将 NN-Descent 组织为 4 个阶段: 采样、去重、距离计算和邻居更新. 算法以近邻图节点为独立计算单元, 将复杂邻居计算解耦, 各节点分配至 GPU 的各线程块内计算, 消除了同步代价.

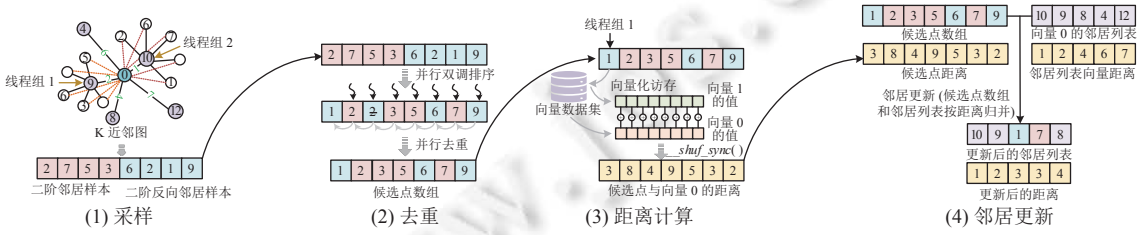


图 3 GPU 加速的 K 近邻图索引构建算法流程

(1) 采样. 与 NN-Descent 根据“邻居的邻居也有可能是邻居”的思想而采用的随机采样方法不同, 本文基于“最近邻的邻居更有可能是邻居”的思想设计了最近邻偏好采样策略, 以提高采样质量并减小计算代价. 线程块内的线程首先以线程组的形式并行对各节点的前 M 个最近邻的邻居节点实施采样, 而后采用同样的方式对反向邻居的二阶邻居实施采样, 最终合并采样结果构建采样节点集. 为了节省 GPU 显存开销, 并保证后续对采样节点访问的

高效性, 采样节点集将被存储于 GPU 线程块的共享内存中.

(2) 去重. 由于采样所得节点不可避免地存在重复, 故需对采样节点进行去重, 以保证最终邻居列表各节点的唯一性. 为了充分发挥 GPU 的大规模并行计算能力, 本文采用线程块内的并行双调排序算法[49]. 该算法通过若干次并行的两两元素比较与交换, 最终输出有序序列, 其时间复杂度为 $O(\log n)$. 形成有序序列后, 为序列中各元素分配一个线程, 各线程判断该元素与其前一个元素是否相同, 若不同则写入位于共享内存中的候选点数组.

(3) 距离计算. NN-Descent 原算法中节点间距离计算由单线程串行累加各维度距离完成, 这种计算方式难以充分利用 GPU 中的全部线程. 为了加速距离计算操作, 本文设计了面向 GPU 的高并发距离计算函数. 线程块中的线程并行计算候选点与该线程块所在点的距离. 针对高维向量特性, 本文采用线程组协作计算每个点对的距离. 在计算过程中, 为了加速数据访问, 采用向量化访存技术, 各线程一次加载多个操作数, 线程块所在点的数值加载至共享内存, 为所有线程组共享, 各线程组负责的候选点数值则加载至寄存器. 线程组中各线程计算得到局部维度的距离后, 使用 CUDA 的线程间通信函数 *__shuf_sync*() 实现线程间的数据交换, 进而将各线程计算结果归并得到最终距离. 线程间通信函数的使用避免了额外的内存访问, 进而提高了距离归约的效率. 最后, 为了进一步减小后续邻居更新的开销, 将计算得到的距离与线程块所在点当前邻居列表中的最大距离进行比较, 由于大于最大距离的候选点必定不会在更新阶段作为所在点的邻居, 因此仅保留小于最大距离的候选点.

(4) 邻居更新. 首先采用并行双调排序算法将候选点按距离进行排序, 随后将现有邻居列表与候选点数组按照距离进行归并, 保留前 k 个数据点, 并确保最终形成的邻居列表的有序性. 在邻居列表更新后, 需同步更新反向邻居, 以提高 K 近邻图的构建精度. 在 NN-Descent 原算法中, 可动态申请存放反向邻居的内存, 最后将所有更新的反向邻居与当前反向邻居进行完全排序. 而在 GPU 中, 由于 GPU 难以实现动态内存分配, 反向邻居的数量无法预先确定, 为了降低 GPU 的内存开销, 本文将各节点的反向邻居数量固定为 k, 当反向邻居数量超过 k 时, 通过模运算确定替换位置, 比较新的候选点距离与对应位置的反向邻居距离, 保留距离较小的节点. 该策略在降低内存开销的同时提升了近邻图的精度.

算法 1 描述了 GPU 加速的 K 近邻图构建算法. 该算法根据指定的迭代次数 it 进行计算 (第 1 行), 其中每个线程块负责一个节点的计算 (第 2 行). 首先进行最近邻偏好采样 (第 3、4 行), 并对采样后的节点排序后去重 (第 5 行), 该过程的时间复杂度为 $O(\log|S|)$. 接着, 对采样得到的候选节点计算与节点 u 的距离 (第 6–9 行), 其时间复杂度为 $O(|S|\times\log d/Warp)$, 其中, d 为向量维度, $Warp$ 表示线程组数量. 最后对 S 中的节点按距离排序后与 $N(u)$ 进行归并, 并更新 u 的反向邻居, 其时间复杂度为 $O(\log|S|)$. 综上, 算法 1 的时间复杂度为 $O(it\times|S|\times\log d/Warp)$.

算法 1. GPU 加速的 K 近邻图索引构建算法.

输入: 向量数据集 D, 迭代次数 it, 近邻图度数 k;
输出: 构建的 K 近邻图索引.

1. **for** ($i = 0$; $i < it$; i++) **do**
2. **for each** $u \in D$ in parallel at thread block level **do**
3. $S_1 \leftarrow$ 对 $N(u)$ 中前 M 个节点的邻居节点进行采样;
4. $S_2 \leftarrow$ 对 $\{v|u \in N(v)\}$ 中前 M 个节点的邻居节点进行采样;
5. $S \leftarrow$ 对 $S_1 \cup S_2$ 进行去重;
6. **for each** $v \in S$ in parallel at the warp level **do**
7. $v.dis = dis(v,u)$; /*warp 内线程并行计算距离*/
8. **if** $v.dis >$ $N(u)$ 中的最大距离 **do**
9. 在 S 中删除 v;
10. 对 S 中的节点按距离排序;
11. $N(u) \leftarrow$ 将 S 与 $N(u)$ 中的点按距离归并;

12. 更新反向邻居 $\{v|u \in N(v)\}$ 的邻居列表;
13. **return** K 近邻图;

3.2 Tensor 核心增强的层间并行 K-means 树分区

为了充分利用 GPU 线程块间的并行性, 本文采用分而治之的思想, 对数据集进行分区处理, 各分区由不同线程块并行执行 DBSCAN 聚类. 同时, 需确保分区内数据的相似性, 以减小后续簇合并的计算开销. 传统方法在低维场景中采用基于 KD 树或基于网格划分的方式实现对数据集的分区, 但此类方法在高维场景下将造成较高的时间开销. 为此, 本文采用 K-means 树对数据集进行高效分区. K-means 树通过递归聚类生成层级结构: 首先对所有数据点进行 K-means 聚类, 将得到的 k 个聚类簇作为树的 k 个子节点, 各子节点内部再次执行 K-means 聚类生成下一层节点. 与传统方法不同, K-means 树在计算时无须按维度划分数据, 且具有较高的效率. 值得注意的是, 分区过程无须精确地聚类, 仅需通过少量迭代将相似的数据分至不同分区.

尽管 K-means 树各节点内部的聚类计算适合使用 GPU 并行化, 但随着树深度的增加, 高层节点包含的数据数量呈指数级衰减. 顺序生成节点将导致 GPU 资源利用率不足, 进而引起计算性能的下降. 为了应对这一问题, 本文提出了 Tensor 核心增强的层间并行 K-means 树分区算法. 该算法通过并行生成同一层的所有节点, 而非逐层顺序生成, 确保 GPU 资源始终高效利用. 其核心在于, K-means 树各层总数据量不变, 因此层间并行可最大化并行度. 同时, 算法将 K-means 距离计算转换为矩阵乘法计算, 利用 Tensor 核心加速计算过程.

图 4 展示了 Tensor 核心增强的层间并行 K-means 树分区算法的一个示例. 在根节点, 随机初始化 k 个聚类中心, 通过矩阵乘法计算数据点与聚类中心的距离, 并使用 GPU 的 Tensor 核心加速该计算过程. 由于 GPU 内部每个线程块均可调用 Tensor 核心进行矩阵乘法运算, 为了充分发挥 GPU 的并行性, 将数据点均分至所有线程块, 各线程块将聚类中心点载入其共享内存, 以加快访存速度. 各线程块分别调用不同的 Tensor 核心计算数据点与聚类中心的距离, 将计算得到的距离写入共享内存. 此时, 各数据点与聚类中心的距离保存在对应线程块的共享内存中, 为了确定与数据点最近的聚类中心, 对于每个数据点, 线程块内的线程分别取该数据点与各聚类中心的距离, 首先使用 CUDA 线程间通信函数 *__shuf_sync*() 在线程组内 (32 个线程) 选出局部距离最小的聚类中心, 各线程组将其结果写入共享内存, 线程块内的线程再进行线程块级别的归约操作, 最终选出全局距离最小的聚类中心. 完成所有数据点的聚类中心分配后, GPU 并行更新聚类中心值. 按照上述流程迭代固定次数, 即可得到 k 个子节点包含的数据点.

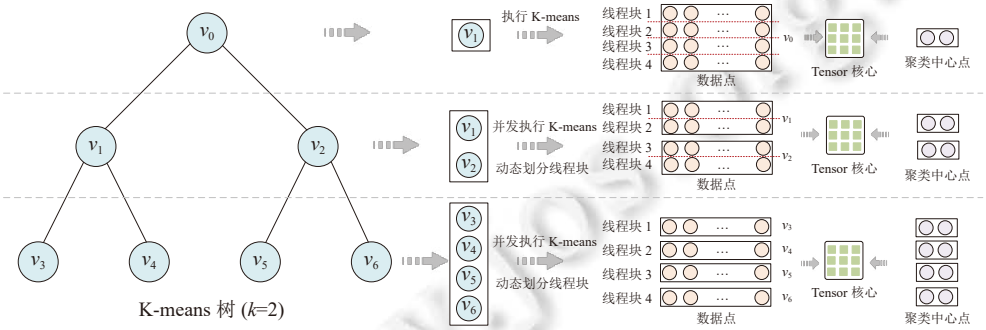


图 4 Tensor 核心增强的层间并行 K-means 树分区算法

第 1 层根节点聚类结束后, 即可得到第 2 层的 k 个子节点, 此时 k 个子节点需独立进行 K-means 聚类. 本文通过动态线程块分配实现层间并行: 按照 k 个子节点包含的数据点数量重新分配线程块, 各线程块内部执行与根节点 K-means 流程相同的计算. 例如, 在图 4 示例中, 第 2 层得到 2 个子节点. 由于 2 个子节点所包含的数据量相近, 因此分别分配 2 个线程块负责每个子节点的距离计算. 线程块为这 2 个子节点随机初始化 2 组聚类中心点, 线程块 1 和 2 将子节点 v_1 包含的数据点读入共享内存, 并将第 1 组聚类中心点读入共享内存, 调用 Tensor 核心进行距

离计算, 并采用上述归约流程得到各数据点的距离最小聚类中心. 同理, 线程块 3 和 4 按照相同的流程计算 v_2 包含的数据点. 聚类中心点分别更新后, 迭代上述流程至指定次数, 即可形成第 3 层节点. 第 3 层节点的并行计算模式与前两层相同, 基于此流程进行并行 K-means 树划分直至达到预设的树深度.

为了保证分区内数据的邻近性, 算法未添加分区均衡性的约束. 此外, 由于 GPU 共享内存的容量限制, 线程块中一次计算能处理的分区大小有限, 算法输出的较大分区将被均分为能够被线程块一次计算处理的大小. 为了均衡线程块间的负载, 在第 3.3.1 节分区局部聚类时, 本文将分区由大到小排序, 均衡分配给不同线程块, 以减小负载不均衡对后续聚类效率的影响.

算法 2 描述了 Tensor 核心增强的层间并行 K-means 树分区算法流程. 首先初始化聚类中心, 并指定每个线程块负责的分区序号 (第 1 行). 接着对 K-means 树的每一层 (第 2 行) 迭代计算 K-means 聚类 (第 3 行). 在计算过程中, 每个线程块负责一部分数据与对应聚类中心的距离计算, 并按距离更新数据点对应的聚类中心 (第 4、5 行), 该过程时间复杂度为 $O(|D_s|\times k)$. 最后更新聚类中心的值 (第 7 行). 当 K-means 迭代计算结束后, 需按聚类中心编号由小到大重排数据点 (第 8 行), 并为线程块动态分配数据分区, 同时更新 *BlockPar* (第 9 行), 以便下一层计算. 由于第 7–9 行操作可并行执行, 因此时间复杂度由第 3–6 行决定, 整个算法的时间复杂度为 $O(h\times it\times k\times |D_s|)$.

算法 2. Tensor 核心增强的层间并行 K-means 树分区算法.

输入: 向量数据集 D, 树高度 h, 迭代次数 it, 子节点数量 k;
输出: K-means 树分区结果.

1. 随机初始化聚类中心 $C[0]$, $BlockPar \leftarrow \{0,\ldots,0\}$; /*初始化每个线程块负责的分区序号为 0*/
2. **for** ($i = 0$; $i < h$; i++) **do**
3. **for** ($j = 0$; $j < it$; j++) **do**
4. **for each** $D_s \subset D$ in parallel at the thread block level **do** /*各线程块计算部分数据*/
5. $dis(D_s, C[BlockPar[blockID]]) \leftarrow$ Tensor 核心计算数据点与聚类中心距离;
6. 根据 $dis(D_s, C[BlockPar[blockID]])$ 为 D_s 包含的数据点选择聚类中心;
7. 更新聚类中心值;
8. 按聚类中心编号重排数据集 D;
9. 为线程块分配分区并更新 *BlockPar*;
10. **return** K-means 树分区结果;

3.3 基于广度优先搜索和核心近邻图的并行聚类

完成 K 近邻图构建以及数据集分区后, 下面基于构建好的 K 近邻图在各分区内进行 DBSCAN 局部聚类计算, 最后将局部聚类结果合并为全局聚类结果. 为此, 本节分为两部分: 首先介绍基于广度优先遍历策略的分区局部并行 DBSCAN 算法, 而后介绍基于核心近邻图的簇合并算法.

3.3.1 分区局部并行 DBSCAN 算法

通过 K-means 树对数据集分区后, 为各分区分配一个 GPU 线程块, 负责该分区的局部 DBSCAN 计算. 为了充分利用 GPU 线程块内的大量线程, 局部并行 DBSCAN 算法采用广度优先的方式对聚类簇进行扩展. 算法依次以一个核心点为起点, 并根据近邻关系收集核心点, 根据定义 8, 该过程中访问到的核心点密度与起点之间相连, 因此这些核心点及其密度邻域内的边界点与起点同属于一个聚类簇. 为此, 在每个线程块的共享内存中建立队列, 在线程块中设置一个主线程负责管理队列. 首先, 主线程选择一个核心点作为初始点入队列. 而后, 线程块中所有线程并行判断该核心点在 K 近邻图中距离小于等于 ε 的所有邻居节点. 若邻居节点为核心点, 则标记后利用 CUDA 原子操作加入队列, 若为非核心点, 则仅做标记. 此外, 由于第 3.1 节中构造的 K 近邻图为全局 K 近邻, 为了将搜索范围控制在分区内部, 需进一步核查遍历的点是否为当前分区的数据点, 若不位于当前分区, 则不进行任

何操作. 之后, 由主线程取出队列头部节点, 进一步执行上述流程.

算法 3 展示了分区局部并行 DBSCAN 算法的具体流程. 该算法的输入为各个分区的点集合 S、簇编号表 Cid、核心点近邻表 $core\_neighbors$. 簇编号表 Cid 中存放每个点所属的簇编号, 以并查集的形式存储, 簇编号为 −1 的点代表噪声点. 核心点近邻表 $core\_neighbors$ 为二维数组, 用于存放所有核心点的距离小于等于 ε 的近邻点. 该算法首先初始化队列 *queue* 和集合 *visited*, 其中, *queue* 用于控制广度优先搜索过程中数据点的访问顺序, *visited* 用于标记元素以避免重复访问 (第 1、2 行). 随后, 启动多个线程块对每个分区独立并行聚类, 对每个分区而言, 算法遍历分区内的所有点, 每次将未被访问的核心点入队, 并对其邻居进行扩展 (第 3–7 行). 接着, 算法持续迭代, 取出队首元素 $\overline{p}$, 并采用线程块中的所有线程对邻居列表 $N(\overline{p})$ 进行并行扩展 (第 8–10 行). 每个线程处理一个近邻 q, 若 q 为非核心点, 则 q 可被分配给任何与 q 密度可达的簇, 因此可将 q 分配给 $\overline{p}$ 所在的簇. 若 q 为核心点, 则需判断 q 和 $\overline{p}$ 是否属于同一分区, 若在同一分区, 则把 q 合并到 $\overline{p}$ 所在的簇, 并将 q 入队 (第 11–20 行). 此外, 为了防止节点的重复访问, 在处理过程中, 所有和 $\overline{p}$ 属于相同分区的点均被标记为 *visited*. 同时, 为了避免并发引起的竞争问题, q 的入队采用可以避免并发写数据的原子操作完成. 最后, 算法返回完成单分区聚类的聚类结果 Cid (第 21 行). 由上述流程可见, 对于分区 S_i 中的点, 若为非核心点, 则仅访问并标记, 若为核心点, 则需访问其邻居节点. 对于核心点, 其邻居节点至多为 k (近邻图度数), 设每个线程块中线程数量为 *threadNum*, 则多线程并行访问邻居节点的操作时间复杂度为 $O(k/threadNum)$. 设 S_i 中核心点数量为 *coreNum*, 则算法 3 的时间复杂度为 $O((|S_i|-coreNum)+(coreNum\times k/threadNum))$, 即为 $O(|S_i|+coreNum\times(k/threadNum-1))$.

算法 3. 分区局部并行 DBSCAN 算法.

输入: 各分区点集 S, 簇编号表 Cid, 核心点所有距离小于等于 ε 的近邻 $core\_neighbors$;
输出: 分区局部 DBSCAN 聚类结果.

1. $queue \leftarrow \varnothing$; /*队列, 位于共享内存, 用于 DBSCAN 过程的广度优先搜索*/
2. $visited \leftarrow \varnothing$; /*集合, 位于共享内存, 用于标记某点是否访问过*/
3. **for each** 分区 S_i in parallel at thread block level **do**
4. **for each** $p \in S_i$ **do** /*遍历分区内所有点*/
5. **if** $p \in visited$ 或 p 为非核心点 **then** /*若当前点已访问过或为非核心节点, 跳过该点*/
6. **continue**;
7. $thread_0$ **do**: p 入队列, $visited \leftarrow visited \cup \{p\}$; /*主线程将 p 入队列*/
8. **while** (*queue* 非空) **do**
9. $thread_0$ **do**: 队首节点 $\overline{p}$ 出队列;
10. $N(\overline{p}) \leftarrow core\_neighbors[\overline{p}]$; /*入队节点必为本分区核心点, 遍历其近邻进行簇扩展*/
11. **for each** $q \in N(\overline{p})$ in parallel at thread level **do**
12. **if** q 为非核心点 **then** /*此时的簇从 p 开始扩展, 故将 p 设置为簇编号*/
13. $Cid[q] \leftarrow p$;
14. **if** q 位于当前分区 **then**
15. $visited \leftarrow visited \cup \{q\}$;
16. **else**
17. **if** q 位于当前分区 **then** /*如果为当前分区的核心节点, 则入队*/
18. $Cid[q] \leftarrow p$;
19. $visited \leftarrow visited \cup \{q\}$;
20. q 入队列;
21. **return** 分区聚类结果 Cid;

由上述流程可见, 本算法将 DBSCAN 算法扩展聚类簇过程中开销最大的在线近邻查询转换为离线的 K 近邻图构建, 在扩展聚类簇时仅需遍历 K 近邻图中的邻居节点. 此方法具有两个优势: (1) DBSCAN 算法中各节点触发在线近邻查询的时间不同. 对于 GPU 而言, 只有当多个查询并发执行时才能充分利用其高并行度. DBSCAN 触发查询的时间差异使其难以有效利用 GPU 加速. 而通过将在线过程转换为离线过程, 构建 K 近邻图时存在大量并发的相似度比较, 此时可使用 GPU 高效并行处理, 从而能够充分发挥 GPU 的高并发特性; (2) 由于 DBSCAN 算法的结果依赖于 ε 和 *minPts* 两个参数的设置, 在线近邻查询需在不同参数设置下重新进行近邻查询, 无法复用之前的查询结果, 这导致了较高的计算开销. 通过将在线近邻查询转换为离线 K 近邻图构建, 只要近邻图邻居数量 $k > minPts$, 则在 ε 变化时仍可重复利用原先构建的索引, 无须重新构建 K 近邻图索引, 显著降低了近邻查询的成本; 当 *minPts* 增大时, 若 $k > minPts$ 仍成立, 则可重复利用原先构建的索引, 但 DBSCAN 精度将有所下降, 因此在实际使用时需在精度与效率之间进行权衡, 依据实际需求选择是否重新构建索引.

3.3.2 基于核心近邻图的簇合并算法

在各分区内的局部 DBSCAN 计算完成后, 每个分区已经形成局部聚类簇. 此时, 需要将各分区的局部聚类簇进行合并, 以输出最终的聚类结果.

为了高效地合并聚类簇, 本文提出了基于核心近邻图的簇合并算法. 该算法利用位于不同分区的核心点之间的关系来表示不同簇之间的关系. 若位于分区 S_i 中的核心点 p 与位于分区 S_j 中的核心点 q 密度直达, 则根据基于密度的簇定义, 在完整数据集上执行 DBSCAN 时, 无论由 p 还是由 q 出发, 最终 p 与 q 都将归属于同一个聚类簇. 因此在合并簇的过程中, p 与 q 所属的簇最终也应合并为同一个簇.

基于上述观察, 如图 5 所示, 首先使用 K 近邻图的近邻信息在分区间的核心点之间构建一个以核心点为节点、分区间的核心点间密度直达关系为边的核心近邻图. 核心近邻图是 K 近邻图的一个子图, 当分区间的两个核心点为彼此的近邻时, 则在核心点之间建立一条边. 有边相连的核心点及其 ε 邻域内的边界点应合并为同一个簇. 由此可将局部簇合并问题转换为计算核心近邻图的连通分量问题. 为了减小内存开销并避免重复合并, 将分区内的核心点视为核心近邻图中的一个节点, 分区内的核心点近邻不建立边, 因为分区内已在第 3.3 节的局部 DBSCAN 算法中进行了局部簇合并. 图 5 的示例包含 4 个不同分区的核心点与边界点, 来自不同分区的核心点之间组成了一个连通分量, 因此图中核心点及其 ε 邻域内的边界点均属于同一个聚类簇. 最终利用并查集将连通分量中的所有节点归并至同一个簇.

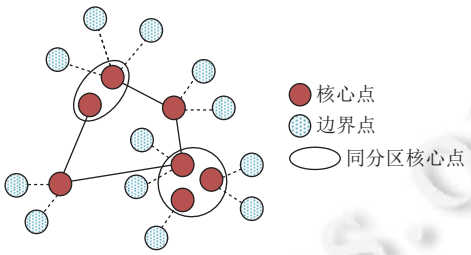


图 5 核心近邻图示例

核心近邻图的构建采用以节点为中心的并行模式, 每个 GPU 线程块负责一个节点的边的建立, 以提高并行效率. 在线程块内部, 采用类似第 3.3 节的广度优先搜索中的并行方式, 所有线程分别处理当前节点的不同邻居节点, 判断两节点间是否建立边. 这种分层并行策略有利于 GPU 线程资源的充分利用, 进一步提高了计算效率.

算法 4 展示了对分区聚类结果的合并过程, 其输入为簇编号表 *Cid*、核心点集 *C* 和核心点近邻 *core_neighbors*. 其中, *Cid* 和 *core_neighbors* 的含义与算法 3 相同, 核心点集 *C* 中存放着数据集中的所有核心点. 算法首先初始化边集 *edges*. 随后, 对于每一个核心点 p, 遍历其所有距离小于等于 ε 的近邻数据 q, 若 q 为核心点, 且与核心点 p 位于不同的分区, 则将边 (p, q) 加入边集 (第 2–6 行), 设每个线程块中线程数量为 *threadNum*, 由于核心点邻居数量至多为 k, 故该过程的时间复杂度为 $O(k/threadNum)$. 接着, 遍历 *edges* 中的所有边, 并合并边的两个顶点所代表的簇 (第 7、8 行), 该过程的时间复杂度为 $O(|edges|)$. 在所有边合并完成后, 算法形成了最终的聚类结果, 并将合并

后的结果返回 (第 9 行). 由于 *edges* 中边的数量远大于 *k*, 因此算法 4 整体的时间复杂度为 O(|*edges*|).

算法 4. 基于核心近邻图的簇合并算法.

输入: 簇编号表 *Cid*, 核心点集 *C*, 核心点所有距离小于等于 ε 的近邻 *core_neighbors*;
输出: DBSCAN 聚类结果.

1. *edges* ← ∅; /*集合, 位于 GPU 全局内存, 用于存放图中的边*/
2. **for each** $p \in C$ in parallel at the thread block level **do** /*遍历所有核心点的邻居, 在节点间建立边*/
3. $N(p) \leftarrow core\_neighbors[p]$;
4. **for each** $q \in N(p)$ in parallel at the thread level **do** /*遍历 p 的所有近邻*/
5. **if** q 为核心点且 q 与 p 位于不同分区 **then** /*若为其他分区的核心点, 则建立边*/
6. $edges \leftarrow edges \cup \{(p,q)\}$;
7. **for each** $(p,q) \in edges$ **do**
8. union(*Cid*, *p*, *q*); /*合并 p 和 q 所代表的集合*/
9. **return** 聚类结果 *Cid*;

4 实验分析

本节使用 4 个真实数据集对所提出的算法进行评估, 并与相关算法进行对比. 第 4.1 节介绍实验数据集的相关信息. 第 4.2 节介绍实验评估指标及基线对比算法. 第 4.3 节介绍实验参数设置. 第 4.4 节分别改变 ε 和 *minPts* 两个参数, 详细分析不同算法的聚类性能. 第 4.5 节对算法的可扩展性进行评估. 第 4.6 节对第 3 节中提出的 3 个模块分别进行分析, 评估模块的有效性.

4.1 实验数据

本文使用 4 个真实数据集 (DEEP1M[50]、SIFT1M[51]、MSong (Million Song)[52]、GIST[51]), 数据集规模达到百万级, 数据集的相关信息如表 1 所示.

表 1 数据集汇总

数据集名称	数据量	维度	数据大小 (MB)	默认 ε	默认*minPts*
DEEP1M	1 000 000	96	366.2	0.7	50
SIFT1M	1 000 000	128	488.3	200	50
MSong	992 272	420	1 589.8	20	10
GIST	1 000 000	960	3 662.1	1	50

4.2 基线算法与评估指标

● 基线算法. 本文将所提出的 KG-DBSCAN 算法与 5 种现有的 DBSCAN 算法进行了对比. 为了充分评估所提出的基于 K 近邻图索引加速的聚类算法, 本文所对比的基线算法分别选择了采用树结构、HNSW 图结构以及 K 近邻图结构作为索引的 CPU 算法. 对于 GPU 算法, 现有的支持高维数据 DBSCAN 聚类的算法均未采用图索引, 而分区聚类计算的 GPU 算法均采用网格或树结构进行分区, 面临维度灾难的问题 (第 1.2 节), 本文在可选的 GPU 加速的聚类算法中选择最新算法作为基线算法展开对比. 本文的基线算法具体包括: (1) BLOCK-DBSCAN[35]: CPU 方法, 采用基于覆盖树的索引加速近邻查询; (2) *h*-DBSCAN[19]: CPU 方法, 采用基于 HNSW 的索引加速近邻查询; (3) KNN-DBSCAN[22]: CPU 方法, 采用基于 K 近邻图的索引加速近邻查询, 使用最小生成树计算 DBSCAN 聚类; (4) Cal-DBSCAN[40]: GPU 方法, 利用 GPU 直接计算各节点的近邻而不使用任何索引; (5) cuML-DBSCAN[47]: GPU 方法, 提供了随机球形覆盖索引和直接计算区域内数据点两种近邻搜索选项, 由于前者索引构建时间远大于后者 DBSCAN 的计算时间, 因此在本文的实验中选择后者.

在上述方法中, BLOCK-DBSCAN 使用单个线程进行 DBSCAN 计算, 为了加速计算, 本文使用 OpenMP 并行计算向量距离, 线程数为 10. *h*-DBSCAN 在索引构建及查询时支持多线程计算, 线程数设为 10. KNN-DBSCAN 使用 MPI 进行并行计算, 本文将 MPI 线程设为 10, 并部署 10 个 MPI 任务进行 DBSCAN 的计算. 本文所有实验均在配备了 Ubuntu 24.04 系统、Intel(R) Core(TM) i9-10900K CPU @ 3.70 GHz、125 GB 内存和 NVIDIA GeForce RTX 3090 GPU 的服务器上进行.

● 评估指标. 为了全面评估算法效率与精度, 本文测量以下指标: (1) 运行时间: 包含索引构建、DBSCAN 聚类计算等全流程的端到端运行时间; (2) 准确率 (*precision*): 正确聚类的样本数占非噪声点数量的比例; (3) 召回率 (*recall*): 正确聚类的样本数占实际聚类样本数的比例; (4) *F*1 值: 该指标计算依赖准确率和召回率, 计算公式为 $\frac{2\times precision\times recall}{precision+recall}$. 与相关工作类似[21], 本文采用 Scikit-learn 库中的精确 DBSCAN 算法的输出结果作为聚类参考结果, 计算不同算法聚类准确率与召回率. 以上指标为外部指标, 为了进一步评估聚类结果, 本文同时计算聚类内部指标: 轮廓系数 (silhouette coefficient, SC)[53,54]. 内部指标不依赖于已有标签, 而是根据簇内与簇间的度量来评估聚类本身的质量.

4.3 参数设置

本文涉及的参数包括 K 近邻图索引的度数 k、K-means 树的节点度数及分区数 k'和 P 以及 DBSCAN 的参数 ε 和 *minPts*. 在实验中, k 的取值经第 4.6.1 节的评估设置为 *minPts* 的 2 倍, 使得算法在较低计算开销下取得较高聚类精度. 此外, 考虑到 GPU 的共享内存容量限制, k'和 P 固定为 16 和 256, 使得算法在 GPU 内存限制下最大化分区大小. DEEP1M、SIFT1M 和 GIST 数据集的 *minPts* 取值为{50, 60, 70, 80, 90}, MSong 数据集由于在 *minPts* 大于 50 时仅有一个簇, 故其 *minPts* 取值为{10, 20, 30, 40, 50}. ε 根据数据集内部距离取适当的值. 各数据集默认参数取值如表 1 所示.

4.4 聚类性能实验结果与分析

本文在第 4.4.1 和 4.4.2 节分别改变 ε 和 *minPts* 两个参数, 在 4 个数据集上对算法的运行时间和精度进行评估和对比; 在第 4.4.3 节利用内部指标对各算法的聚类质量进行评估.

4.4.1 查询半径 ε 变化

本节将所提出的 KG-DBSCAN 算法与第 4.2 节中列出的 5 个基线算法在 4 个数据集上进行对比, 各数据集 *minPts* 保持不变, 变化查询半径 ε, 对第 4.2 节中的指标进行评估. 实验结果如图 6 所示. BLOCK-DBSCAN 在 4 个数据集的运行时间均超过 5 h, 故将其时间报告为 INF. KG-DBSCAN 的运行时间显著小于其余 5 个基线算法, 其中, 超过 *h*-DBSCAN 算法 240.0–2 822.5 倍, 超过 KNN-DBSCAN 算法 16.2–28.2 倍, 超过 Cal-DBSCAN 算法 101.8–966.8 倍, 超过 cuML-DBSCAN 算法 6.2–47.7 倍. 由此可见, 即使相比于基于 GPU 的基线算法, 本文所提出的 KG-DBSCAN 也表现出更快的聚类速度. 这是由于 KG-DBSCAN 将 DBSCAN 中开销最高的近邻查询转换为 K 近邻图的构建, 并设计了高效的 K 近邻图构建算法以充分利用 GPU 的高并发特性. K 近邻图更适合高维场景下的快速构建与近邻查询. 此外, 对于 K 近邻图构建后的分区和聚类, KG-DBSCAN 同样进行了算法优化, 以保证聚类效率. 与之相比, Cal-DBSCAN 和 cuML-DBSCAN 需遍历数据集中的所有数据为数据点计算近邻, 其中存在大量冗余计算, G-DBSCAN[41]存在同样的问题. 在聚类精度方面, Cal-DBSCAN 和 cuML-DBSCAN 均为精确 DBSCAN 算法, 故其 *F*1 分数、准确率和召回率始终为 1. 在 DEEP1M、SIFT1M 和 GIST 数据集上, KG-DBSCAN 的 *F*1 分数、准确率和召回率始终在 0.98 以上. 在 MSong 数据集上, KG-DBSCAN 的精度有所下降, 这是由于 MSong 中的近邻数据点距离十分接近, KG-DBSCAN 未能准确识别部分核心点. 此外, *h*-DBSCAN 在 GIST 数据集上精度较低, 这是由于在聚类时产生了较多的只有一个核心点的簇, 该核心点在 ε 邻域内的点由于竞争均被其余核心点标记为其各自的 ε 近邻, 进而被划分至其余核心点所在的簇内.

当查询半径 ε 增加时, 除 Cal-DBSCAN 算法外, 其余各算法的运行时间均呈增加的趋势. 这是由于查询半径 ε 增加时, 邻域内数据点的数量增加, 因此距离计算操作增加. 此外, 簇扩展以及簇合并的过程中需遍历的数据点数

量随查询半径 ε 的增加而增加, 造成了算法总运行时间的增加. 值得指出的是, 图 6 虽然将 KG-DBSCAN 算法的 K 近邻图构建时间包含在内, 但在实际执行时无须重新构建 K 近邻图, 因为 *minPts* 的数值未发生改变. K 近邻图在查询半径 ε 变化时可复用的特性将显著降低 KG-DBSCAN 算法的计算开销. 另外, 对于 KG-DBSCAN 算法, 由于簇扩展和簇合并过程开销较小, 因此查询半径 ε 的增加对其影响较小, 故而总运行时间增加趋势不明显.

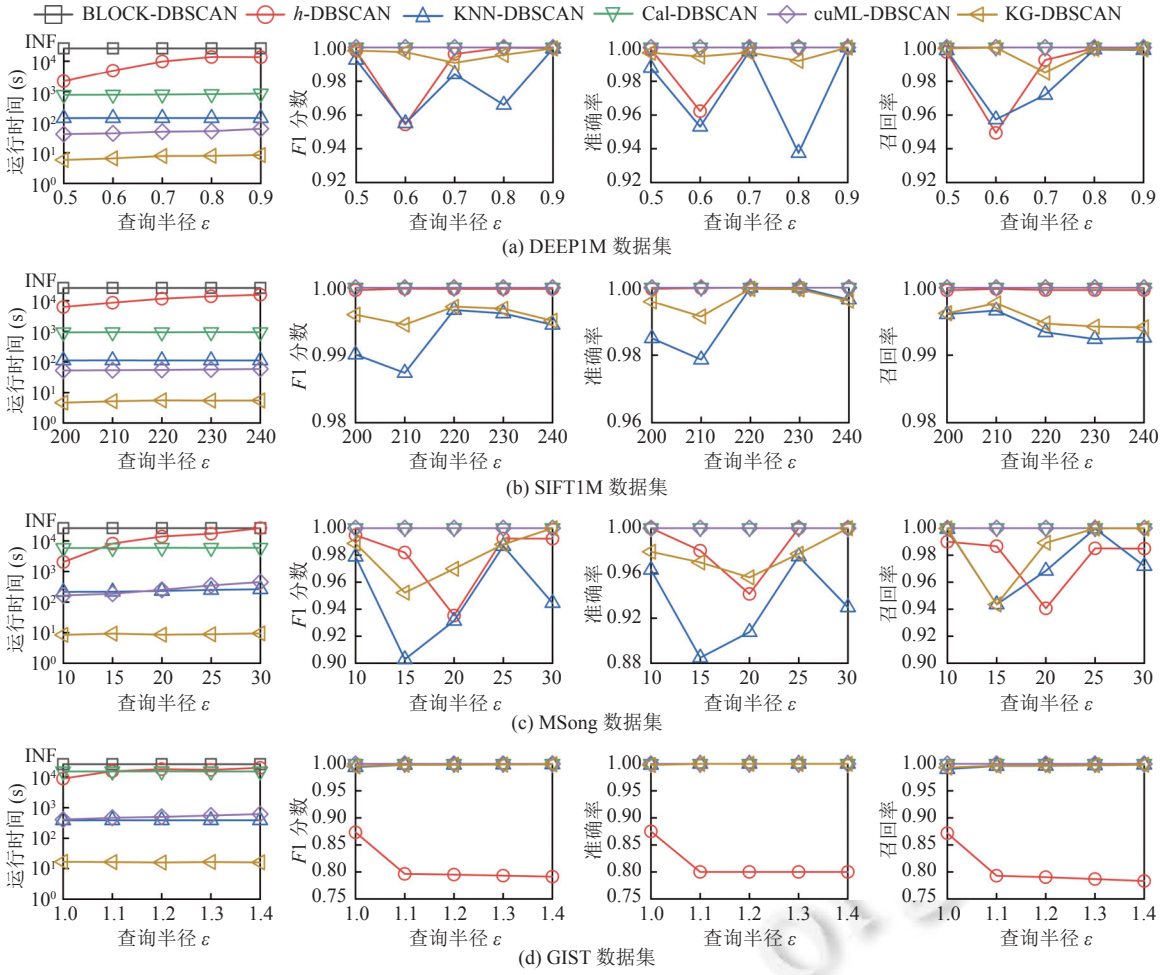


图 6 查询半径 ε 变化时的算法对比

4.4.2 查询参数 *minPts* 变化

进一步地, 本节将查询半径 ε 固定为表 1 中各数据集对应的默认值, 改变查询参数 *minPts*, 评估各算法在运行时间和聚类精度两方面的表现, 实验结果如图 7 所示. 结果表明, 本文所提出的 KG-DBSCAN 算法的计算效率在 *minPts* 变化时依然远超其余基线算法. 具体地, 相较于 *h*-DBSCAN、KNN-DBSCAN、Cal-DBSCAN 和 cuML-DBSCAN 算法分别快 530.4–1 617.2 倍、15.4–27.0 倍、93.0–942.6 倍和 5.7–29.3 倍. 由于本文实验过程中将 K 近邻图的参数 *k* 设置为 *minPts* 的 2 倍, 因此基于 K 近邻图的算法 (KG-DBSCAN 与 KNN-DBSCAN) 的运行时间会随查询参数 *minPts* 的增加而增加, *h*-DBSCAN 查询的近邻数量依赖于 *minPts* 的取值, 故其运行时间亦随着查询参数的增加而增加. 其余算法的运行时间不受 *minPts* 的影响. 即使如此, KG-DBSCAN 算法相比于其他算法仍具有绝对优势. 此外, KG-DBSCAN 的精度, 即 *F*1 分数、准确率和召回率与基线算法中的近似算法精度相似, 在部分数据集上超过了基线近似算法的精度.

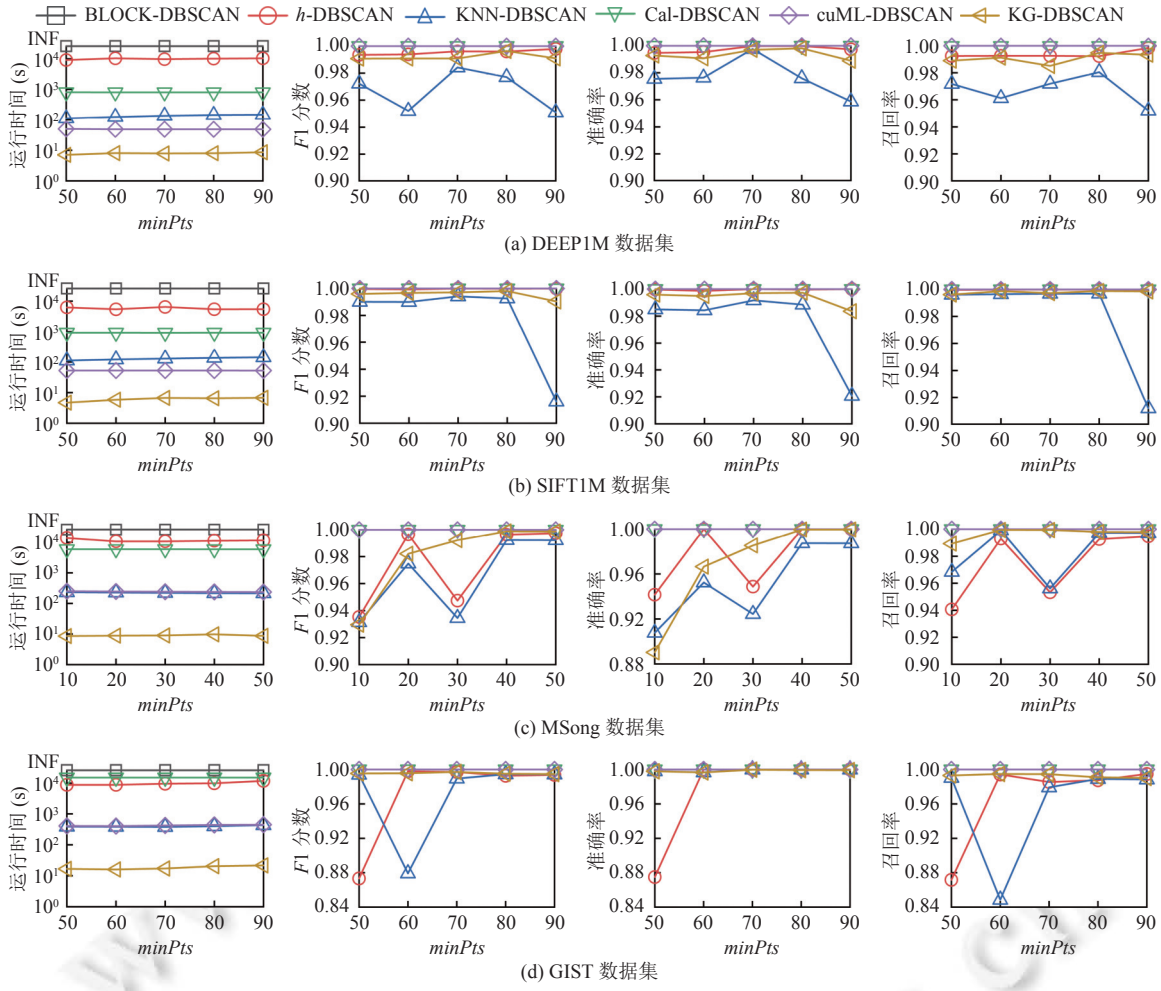


图 7 查询参数 *minPts* 变化时的算法对比

4.4.3 聚类质量评估

本节使用默认参数对 4 个数据集上各算法的轮廓系数进行了对比, 以评估各算法的聚类质量. 由于 BLOCK-DBSCAN 计算超时, 因此未对该算法进行评估. 结果如表 2 所示.

表 2 轮廓系数对比

算法	DEEP1M	SIFT1M	MSong	GIST
h-DBSCAN	0.0716	0.1035	0.0734	0.2245
KNN-DBSCAN	0.1924	0.1508	0.2472	0.3320
Cal-DBSCAN	0.1803	0.1516	0.2319	0.3270
cuML-DBSCAN	0.1799	0.1510	0.2446	0.3234
KG-DBSCAN	0.1692	0.1505	0.2611	0.3328

KG-DBSCAN 与 4 个基线算法得到了较为接近的轮廓系数 (SC), 且各算法的轮廓系数均大于 0, 说明各算法均取得了较高的聚类质量. *h*-DBSCAN 具有较低的轮廓系数值, 这是由于该算法产生了较多的单点簇, 导致了与其他算法指标的偏差. 虽然 KG-DBSCAN 被设计为近似算法, 其聚类质量仍可与 cuML-DBSCAN 和 Cal-DBSCAN 两个精确算法相媲美, 这得益于 KG-DBSCAN 设计了高精度的 K 近邻图构建以及基于核心近邻图的聚类簇合并

算法. K 近邻图构建通过偏好采样和 GPU 加速距离计算在提高效率的同时保证了精度, 聚类簇合并算法基于核心近邻图将簇合并问题转换为连通分量计算, 进一步保证了聚类簇的质量.

4.5 可扩展性分析

本节在 4 个数据集上对各算法的可扩展性进行评估, 使各数据集规模分别为原规模的 20%、40%、60%、80% 和 100%, 测量各算法的运行时间及精度. 实验结果如图 8 所示. 当数据集规模增加时, 各算法的运行时间随之增加. 除了 DEEP1M 数据集在 20% 规模时 KG-DBSCAN 略慢于 cuML-DBSCAN 之外, 其余规模下 KG-DBSCAN 的运行时间均显著优于其他算法. 另一方面, 随着数据集规模的增加, KG-DBSCAN 的 $F1$ 分数、准确率和召回率虽略有下降, 但在 DEEP1M、SIFT1M 和 GIST 数据集上始终高于 0.98, 这表明本文提出的 KG-DBSCAN 算法具有良好的可扩展性; 其在 MSong 数据集上较大的精度变化是由于 MSong 中的数据点距离十分接近, 查询参数的微小变化将显著影响聚类结果, 最终影响精度测量指标. 此外, BLOCK-DBSCAN 算法在数据规模较小时运行时间小于 5 h, 但其精度较低, 这是由于基于树的方法在高维空间中面临维度灾难的问题, 难以获得高精度的聚类结果.

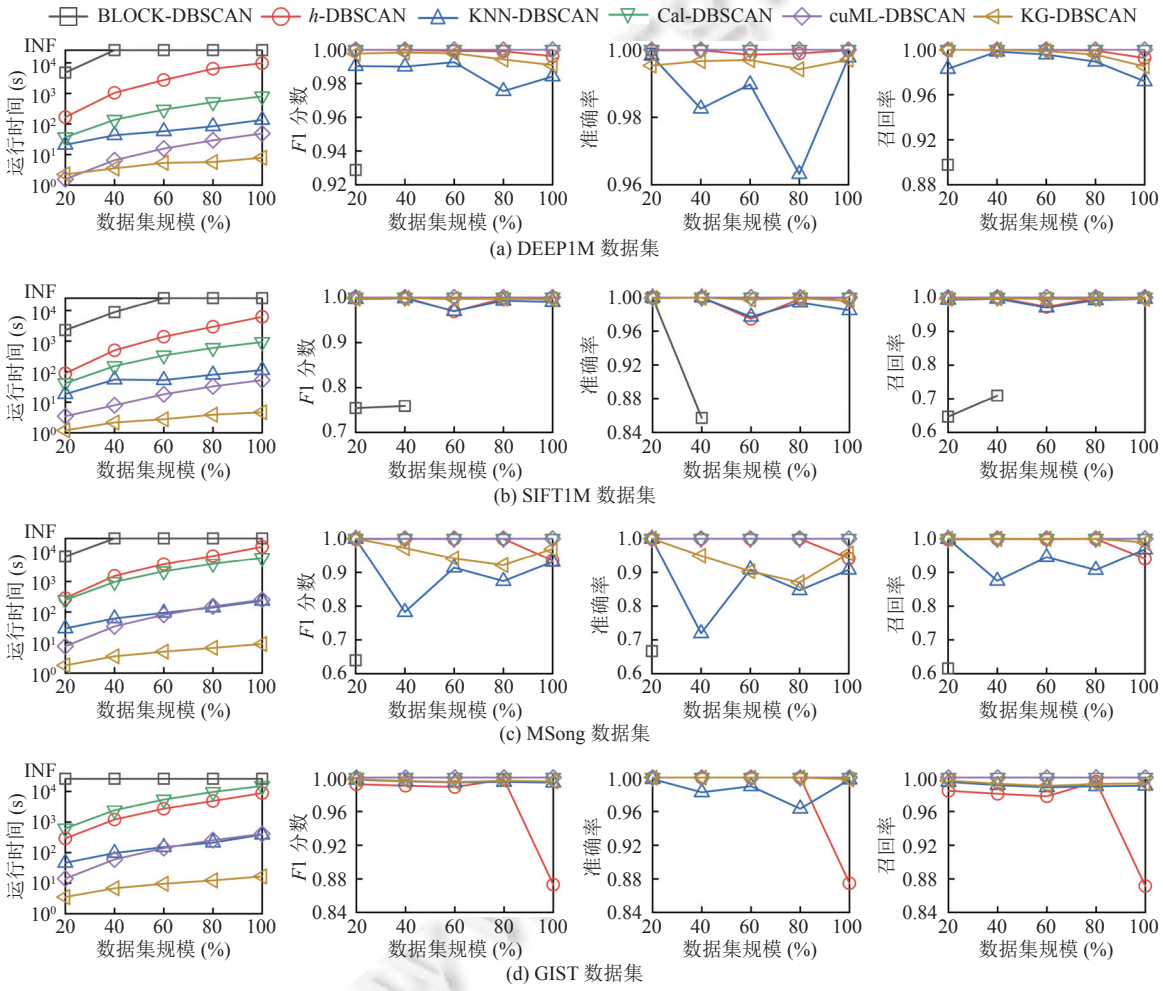


图 8 数据集规模变化时的算法对比

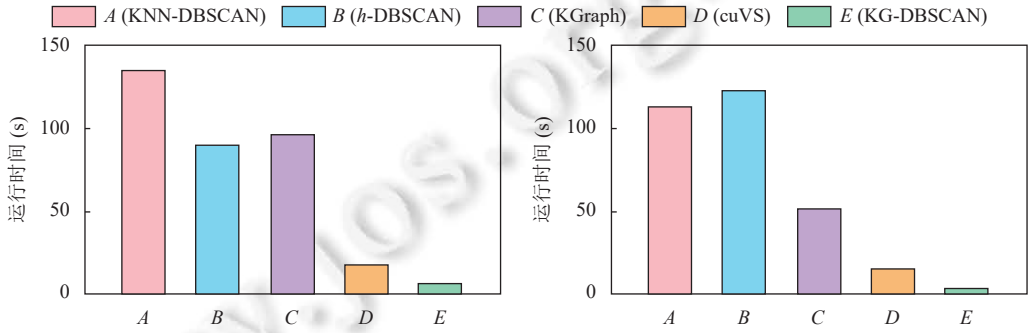
4.6 算法模块分析

本节对算法所包含的近邻图索引构建、K-means 树分区以及并行聚类这 3 个模块分别进行实验分析, 验证其

有效性, 并与基线算法对应模块及同类方法进行对比.

4.6.1 近邻图索引构建

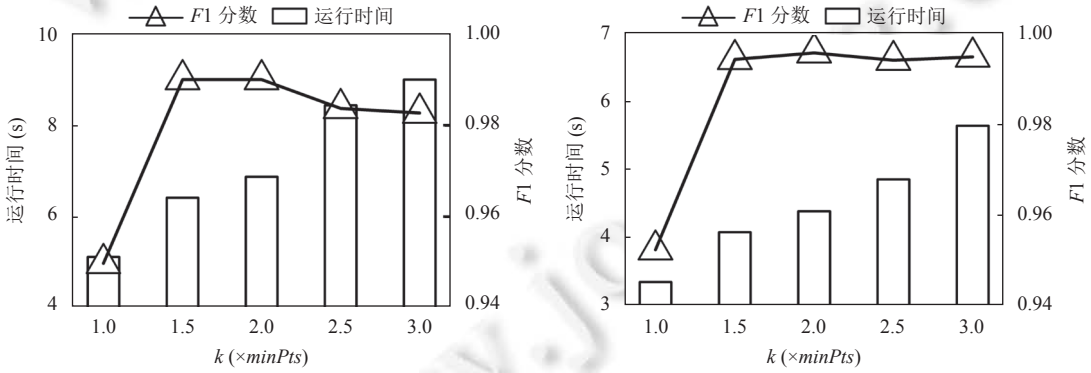
本节使用表 1 中的两个代表性数据集 DEEP1M 和 SIFT1M 及其各自的默认参数, 对 5 种近邻图索引构建方法进行评估. 其中, KNN-DBSCAN 使用开源库 GOFMM[55]中的随机投影树构建 K 近邻图, *h*-DBSCAN 使用开源库 HNSWlib[56]构建 HNSW 图索引, KGraph[57]、cuVS[58]和本文所提出的 KG-DBSCAN 均使用 NN-Descent 构建 K 近邻图, 不同的是, KGraph 在 CPU 上运行、cuVS 使用 GPU 加速距离计算并使用 CPU 进行邻居更新、KG-DBSCAN 进一步改进了 NN-Descent 算法并使用 GPU 进行全流程加速. 5 种近邻图索引构建的运行时间如图 9 所示, KG-DBSCAN 在两个数据集上的近邻图索引构建平均效率相比于 KNN-DBSCAN、*h*-DBSCAN、KGraph 和 cuVS 分别快 25.2 倍、23.4 倍、14.2 倍和 3.4 倍, 实现了高效的近邻图索引构建. 这是由于 KG-DBSCAN 充分利用了 GPU 的高并发特性, 基于其特性优化距离计算与邻居更新操作, 并将各数据点的计算解耦, 避免了同步开销. 相比于 cuVS, KG-DBSCAN 将计算全流程卸载至 GPU, 避免了 CPU 与 GPU 之间频繁的数据传输和同步带来的额外开销, 进一步提高了近邻图索引的构建效率.



(a) DEEP1M 数据集 (b) SIFT1M 数据集

图 9 近邻图索引构建时间对比

此外, 本节在两个代表性数据集上将近邻图度数 k 对聚类精度的影响进行了评估, k 在 [$minPts$, 3×$minPts$] 区间范围内均匀取 5 个值, 并固定 K 近邻图构造算法的迭代次数. 实验结果如图 10 所示.



(a) DEEP1M 数据集 (b) SIFT1M 数据集

图 10 K 近邻图度数对聚类精度的影响

聚类算法的运行时间随 K 近邻图度数的增加而增加. 这是因为 K 近邻图的构建具有较高的计算复杂度, 其运行时间占据了聚类算法 80% 以上的开销, K 近邻图度数的增加将导致 K 近邻图构造算法的开销增加. 此外, 在两个数据集上都可以观察到, 随着近邻图度数的增加, $F1$ 分数呈先增后减的趋势. 这是由于当 k 小于等于 2×$minPts$ 时, 近邻图度数的增加使得各数据点的 ε 邻域内的点更精确, 而当 k 超过 2×$minPts$ 时, 由于各数据点的近邻数量增

加, 固定的迭代次数不足以使得各数据点获得较高质量的邻居, 进而使得 $F1$ 分数略有下降. 为此, 本文选择 $k = 2\times minPts$ 作为聚类算法的输入参数.

4.6.2 K-means 树分区

为了验证本文第 3.2 节提出的层间并行策略的有效性, 本节首先在两个代表性数据集上对基于层间并行的 K-means 树分区和不采用层间并行策略的 K-means 树分区进行对比. 在 DEEP1M 数据集上, 分区时间分别为 0.64 s 和 0.98 s, 层间并行策略将分区效率提升了 47%; 在 SIFT1M 数据集上, 分区时间分别为 0.79 s 和 1.52 s, 层间并行策略将分区效率提升了 92%. 由此可见, 层间并行策略进一步提升了分区效率, 这是因为该策略通过合理分配 GPU 线程块, 并行计算各层所有数据点, 提高了 GPU 资源利用率, 进而提升了分区效率.

此外, 为了验证 K-means 树分区的有效性, 本节将 K-means 树分区与随机分区和 KD 树分区进行对比, 采用以上 3 种方式得到的分区进行聚类. 在达到相近精度的前提下, 随机分区、KD 树分区和 K-means 树分区所得分区结果在 DEEP1M 数据集上的并行聚类时间分别为 0.50 s、0.49 s 和 0.34 s, 在 SIFT1M 数据集上为 0.21 s、0.21 s 和 0.17 s, 可见采用 K-means 树分区后的并行聚类相较于随机分区和 KD 树分区后聚类的效率分别高 35.3% 和 33.8%. 这是由于 K-means 树分区不受到维度灾难的影响, 将距离相近的节点分至同一区域, 减小了合并聚类簇的代价, 进而提高了并行聚类的效率.

4.6.3 并行聚类

本节在不同参数设置下, 在两个代表性数据集上对第 3.3 节提出的 KG-DBSCAN 的并行聚类模块进行评估. 由于 cuML-DBSCAN 的近邻计算与聚类是耦合的, 无法单独测量聚类时间, 而 h-DBSCAN 的聚类是串行计算, 因此只与 KNN-DBSCAN 和 Cal-DBSCAN 的聚类模块进行对比, 结果如表 3 所示. 结果表明, KG-DBSCAN 的聚类模块效率显著优于基线算法的聚类模块, 平均超过 KNN-DBSCAN 算法 8.1 倍、Cal-DBSCAN 算法 10.1 倍. 这是由于本文在第 3.3 节中提出的基于广度优先搜索的分区局部并行 DBSCAN 算法对 GPU 并行遍历数据点进行了优化, 利用共享内存的高速访存速度以及多线程并行访问的特性加速局部 DBSCAN 的效率. 此外, 所提出的基于核心近邻图的簇合并利用 GPU 的并行能力高效构建了核心近邻图, 基于并查集进行了高效的簇合并. 以上技术共同促进了聚类的计算. Cal-DBSCAN 算法在 ε 固定、$minPts$ 变化时聚类时间变化幅度较小, 这是由于该算法聚类策略比较简单, 仅涉及广度优先搜索扩展聚类簇, 当 $minPts$ 变化时其计算代价所受影响较小.

表 3 并行聚类时间对比 (s)

数据集	参数 [ε, $minPts$]	KNN-DBSCAN	Cal-DBSCAN	KG-DBSCAN
DEEP1M	[0.5, 50]	1.39	0.48	0.10
	[0.5, 70]	1.73	0.47	0.15
	[0.7, 50]	1.99	4.77	0.31
	[0.7, 70]	2.00	4.74	0.34
SIFT1M	[200, 50]	1.63	1.85	0.17
	[200, 70]	1.75	1.86	0.24
	[220, 50]	1.76	4.01	0.29
	[220, 70]	1.99	4.04	0.36

5 总结与展望

大规模高维向量聚类旨在揭示数据分布的内在规律, 为异常检测等数据分析任务提供基础支撑. 针对现有的基于密度的聚类方法在高维场景下效率低下的问题, 本文提出了一种 GPU 加速的高维向量聚类算法. 首先, 本文采用 K 近邻图作为 DBSCAN 算法的索引, 设计了一种 GPU 加速的 K 近邻图索引构建算法, 利用偏好采样技术进一步提高 K 近邻图的构建精度并提高构建速度, 显著减小了索引构建开销. 其次, 本文提出 Tensor 核心增强的层间并行 K-means 树分区算法, 并设计了层间并行策略高效利用 GPU 资源. 此外, 本文提出基于广度优先搜索和核心近邻图的并行聚类算法, 设计了并行广度优先策略扩展局部簇, 并构造核心近邻图, 基于核心近邻图实现了有效

的局部簇合并. 最后, 本文在 4 个真实向量数据集上对所提出的算法进行评估并与现有算法进行比较, 实验结果表明本文所提出的算法在保证聚类质量的同时大幅提升了聚类效率. 然而, GPU 加速基于 K 近邻图的高维向量聚类依然存在一些问题: (1) K 近邻图的构建代价随着 k 的增加而呈线性增长, 在算法设定中 k 需大于 $minPts$, 因此当 $minPts$ 较大时将导致较高的索引构建开销; (2) 如第 2.2 节所述, 当节点间距离接近时, 算法存在将边界点识别为噪声点的现象, 进而导致聚类精度降低, 需对聚类结果进行后处理以保证高精度的聚类. 未来将围绕上述问题展开进一步研究.

References

[1] Mortensen KO, Zardbani F, Haque MA, Agustsson SY, Mottin D, Hofmann P, Karras P. Marigold: Efficient K-means clustering in high dimensions. Proc. of the VLDB Endowment, 2023, 16(7): 1740–1748. [doi: 10.14778/3587136.3587147]

[2] Ester M, Kriegel HP, Sander J, Xu XW. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. of the 2nd Int'l Conf. on Knowledge Discovery and Data Mining. Portland: AAAI Press, 1996. 226–231.

[3] Gong SF, Zhang YF, Yu G. Clustering stream data by exploring the evolution of density mountain. Proc. of the VLDB Endowment, 2017, 11(4): 393–405. [doi: 10.1145/3186728.3164136]

[4] Abdulsahib AK, Balafar MA, Baradarani A. DGBPSO-DBSCAN: An optimized clustering technique based on supervised/unsupervised text representation. IEEE Access, 2024, 12: 110798–110812. [doi: 10.1109/ACCESS.2024.3440518]

[5] Yang KY, Gao YJ, Ma R, Chen L, Wu S, Chen G. DBSCAN-MS: Distributed density-based clustering in metric spaces. In: Proc. of the 35th IEEE Int'l Conf. on Data Engineering. Macao: IEEE, 2019. 1346–1357. [doi: 10.1109/ICDE.2019.00122]

[6] Wang Z, Zhang R, Qi JZ, Yuan B. DBSVEC: Density-based clustering using support vector expansion. In: Proc. of the 35th IEEE Int'l Conf. on Data Engineering. Macao: IEEE, 2019. 280–291. [doi: 10.1109/ICDE.2019.00033]

[7] Ma SQ, Kim C, Nam Y. Noise handling techniques in DBSCAN clustering for adult and child voice diarization. In: Proc. of the 2024 Int'l Conf. on Platform Technology and Service. Jeju: IEEE, 2024. 23–26. [doi: 10.1109/PlatCon63925.2024.10830734]

[8] Gohil SH, Iorgulescu JB, Braun DA, Keskin DB, Livak KJ. Applying high-dimensional single-cell technologies to the analysis of cancer immunotherapy. Nature Reviews Clinical Oncology, 2021, 18(4): 244–256. [doi: 10.1038/s41571-020-00449-x]

[9] Wang YQ, Gu Y, Shun JL. Theoretically-efficient and practical parallel DBSCAN. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2555–2571. [doi: 10.1145/3318464.3380582]

[10] Wang MZ, Xu XL, Yue Q, Wang YX. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. Proc. of the VLDB Endowment, 2021, 14(11): 1964–1978. [doi: 10.14778/3476249.3476255]

[11] Du MY, Li QM, Zhang M, Chen X, Li XM, Yin QJ, Ji SL. Constructing benchmark datasets for privacy-protected user comments and evaluating the reasoning capabilities of large model. Chinese Journal of Computers, 2025, 48(7): 1529–1550 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2025.01529]

[12] Mo GL, Song SH, Ding H. Towards metric DBSCAN: Exact, approximate, and streaming algorithms. Proc. of the ACM on Management of Data, 2024, 2(3): 178. [doi: 10.1145/3654981]

[13] Ding H, Yang F, Wang MY. On metric DBSCAN with low doubling dimension. In: Proc. of the 29th Int'l Joint Conf. on Artificial Intelligence. 2021. 426.

[14] Schubert E, Sander J, Ester M, Kriegel HP, Xu XW. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Trans. on Database Systems, 2017, 42(3): 19. [doi: 10.1145/3068335]

[15] Jang J, Jiang H. DBSCAN++: Towards fast and scalable density clustering. In: Proc. of the 36th Int'l Conf. on Machine Learning. 2019. 3019–3029.

[16] Song H, Lee JG. RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 1173–1187. [doi: 10.1145/3183713.3196887]

[17] Boonchoo T, Ao X, Liu Y, Zhao WZ, Zhuang FZ, He Q. Grid-based DBSCAN: Indexing and inference. Pattern Recognition, 2019, 90: 271–284. [doi: 10.1016/j.patcog.2019.01.034]

[18] Huang XG, Ma TF, Liu C, Liu SZ. GriT-DBSCAN: A spatial clustering algorithm for very large databases. Pattern Recognition, 2023, 142: 109658. [doi: 10.1016/j.patcog.2023.109658]

[19] Weng SY, Gou J, Fan ZW. *h*-DBSCAN: A simple fast DBSCAN algorithm for big data. In: Proc. of the 13th Asian Conf. on Machine Learning. 2021. 81–96.

[20] Weng SY, Fan ZW, Gou J. A fast DBSCAN algorithm using a bi-directional HNSW index structure for big data. Int'l Journal of Machine Learning and Cybernetics, 2024, 15(8): 3471–3494. [doi: 10.1007/s13042-024-02104-8]

[21] Lulli A, Dell'Amico M, Michiardi P, Ricci L. NG-DBSCAN: Scalable density-based clustering for arbitrary data. Proc. of the VLDB Endowment, 2016, 10(3): 157–168. [doi: 10.14778/3021924.3021932]

[22] Chen YG, Ruys W, Biros G. KNN-DBSCAN: A DBSCAN in high dimensions. ACM Trans. on Parallel Computing, 2025, 12(1): 3. [doi: 10.1145/3701624]

[23] Mustafa H, Leal E, Gruenwald L. An experimental comparison of GPU techniques for DBSCAN clustering. In: Proc. of the 2009 IEEE Int'l Conf. on Big Data. Los Angeles: IEEE, 2019. 3701–3710. [doi: 10.1109/BigData47090.2019.9006169]

[24] Nagarajan V, Kulkarni M. RT-DBSCAN: Accelerating DBSCAN using ray tracing hardware. In: Proc. of the 2023 IEEE Int'l Parallel and Distributed Processing Symp. St. Petersburg: IEEE, 2023. 963–973. [doi: 10.1109/IPDPS54959.2023.00100]

[25] Wu GQ, Cao LQ, Tian HY, Wang W. HY-DBSCAN: A hybrid parallel DBSCAN clustering algorithm scalable on distributed-memory computers. Journal of Parallel and Distributed Computing, 2022, 168: 57–69. [doi: 10.1016/j.jpdc.2022.06.005]

[26] Loke SC, MacDonald BA, Parsons M, Wünsche BC. Accelerated superpixel image segmentation with a parallelized DBSCAN algorithm. Journal of Real-time Image Processing, 2021, 18(6): 2361–2376. [doi: 10.1007/s11554-021-01128-5]

[27] Cayton L. Accelerating nearest neighbor search on manycore systems. In: Proc. of the 26th IEEE Int'l Parallel and Distributed Processing Symp. Shanghai: IEEE, 2012. 402–413. [doi: 10.1109/IPDPS.2012.45]

[28] Poudel M, Gowanlock M. CUDA-DClust+: Revisiting early GPU-accelerated DBSCAN clustering designs. In: Proc. of the 28th IEEE Int' l Conf. on High Performance Computing, Data, and Analytics. Bengaluru: IEEE, 2021. 354–363. [doi: 10.1109/HiPC53243.2021.00049]

[29] Welton B, Samanas E, Miller BP. Mr. Scan: Extreme scale density-based clustering using a tree-based network of GPGPU nodes. In: Proc. of the 2013 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis. Denver: IEEE, 2013. 1–11. [doi: 10.1145/2503210.2503262]

[30] Chen YW, Zhou LD, Pei SW, Yu ZW, Chen Y, Liu X, Du JX, Xiong NX. KNN-BLOCK DBSCAN: Fast clustering for large-scale data. IEEE Trans. on Systems, Man, and Cybernetics: Systems, 2021, 51(6): 3939–3953. [doi: 10.1109/TSMC.2019.2956527]

[31] Chen Q, Zhao B, Wang HD, Li MQ, Liu CJ, Li ZZ, Yang M, Wang JD. SPANN: Highly-efficient billion-scale approximate nearest neighbor search. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 398.

[32] Gan JH, Tao YF. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 519–530. [doi: 10.1145/2723372.2737792]

[33] Wu YP, Guo JJ, Zhang XJ. A linear DBSCAN algorithm based on LSH. In: Proc. of the 2007 Int'l Conf. on Machine Learning and Cybernetics. Hong Kong: IEEE, 2007. 2608–2614. [doi: 10.1109/ICMLC.2007.4370588]

[34] Keramatian A, Gulisano V, Papatriantafilou M, Tsigas P. IP.LSH.DBSCAN: Integrated parallel density-based clustering through locality-sensitive hashing. In: Proc. of the 28th European Conf. on Parallel and Distributed Computing (Euro-Par 2022). Glasgow: Springer, 2022. 268–284. [doi: 10.1007/978-3-031-12597-3_17]

[35] Chen YW, Zhou LD, Bouguila N, Wang C, Chen Y, Du JX. BLOCK-DBSCAN: Fast clustering for large scale data. Pattern Recognition, 2021, 109: 107624. [doi: 10.1016/j.patcog.2020.107624]

[36] Patwary MMA, Satish N, Sundaram N, Manne F, Habib S, Dubey P. Pardicle: Parallel approximate density-based clustering. In: Proc. of the 2014 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. New Orleans: IEEE, 2014. 560–571. [doi: 10.1109/SC.2014.51]

[37] Jiang H, Jang J, Łącki J. Faster DBSCAN via subsampled similarity queries. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1879.

[38] Malkov YA, Yashunin DA. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2020, 42(4): 824–836. [doi: 10.1109/TPAMI.2018.2889473]

[39] Böhm C, Noll R, Plant C, Wackersreuther B. Density-based clustering using graphics processors. In: Proc. of the 2009 ACM Conf. on Information and Knowledge Management. Hong Kong: ACM, 2009. 661–670. [doi: 10.1145/1645953.1646038]

[40] Cal P, Woźniak M. Data preprocessing with GPU for DBSCAN algorithm. In: Burduk R, Jackowski K, Kurzynski M, Wozniak M, Zolnierek A, eds. Proc. of the 8th Int'l Conf. on Computer Recognition Systems (CORES 2013). Heidelberg: Springer, 2013. 793–801. [doi: 10.1007/978-3-319-00969-8_78]

[41] Andrade G, Ramos G, Madeira D, Sachetto R, Ferreira R, Rocha L. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. Procedia Computer Science, 2013, 18: 369–378. [doi: 10.1016/j.procs.2013.05.200]

[42] Loh WK, Moon YS, Park YH. Fast density-based clustering using graphics processing units. IEICE Trans. on Information and Systems, 2014, E97(5): 1349–1352. [doi: 10.1587/transinf.E97.D.1349]

[43] Qian Q, Zhao S, Xiao CJ, Hung CL. Multi-level grid based clustering and GPU parallel implementations. In: Proc. of the 14th Int'l Symp. on Pervasive Systems, Algorithms and Networks & the 11th Int'l Conf. on Frontier of Computer Science and Technology & the 3rd Int'l

Symp. of Creative Computing (ISPAN-FCST-ISCC). Exeter: IEEE, 2017. 397–402. [doi: 10.1109/ISPAN-FCST-ISCC.2017.75]

[44] Gowanlock M, Rude CM, Blair DM, Li JD, Pankratius V. A hybrid approach for optimizing parallel clustering throughput using the GPU. IEEE Trans. on Parallel and Distributed Systems, 2019, 30(4): 766–777. [doi: 10.1109/TPDS.2018.2869777]

[45] Loh WK, Yu H. Fast density-based clustering through dataset partition using graphics processing units. Information Sciences, 2015, 308: 94–112. [doi: 10.1016/j.ins.2014.10.023]

[46] Prokopenko A, Lebrun-Grandie D, Arndt D. Fast tree-based algorithms for DBSCAN for low-dimensional data on GPUs. In: Proc. of the 52nd Int'l Conf. on Parallel Processing. Salt Lake City: ACM, 2023. 503–512. [doi: 10.1145/3605573.3605594]

[47] Raschka S, Patterson J, Nolet C. Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. Information, 2020, 11(4): 193. [doi: 10.3390/info11040193]

[48] Dong W, Moses C, Li K. Efficient K-nearest neighbor graph construction for generic similarity measures. In: Proc. of the 20th Int'l Conf. on World Wide Web. Hyderabad: ACM, 2011. 577–586. [doi: 10.1145/1963405.1963487]

[49] Lee JD, Batcher KE. Minimizing communication in the bitonic sort. IEEE Trans. on Parallel and Distributed Systems, 2000, 11(5): 459–474. [doi: 10.1109/71.852399]

[50] Yandex BA, Lempitsky V. Efficient indexing of billion-scale datasets of deep descriptors. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 2055–2063. [doi: 10.1109/CVPR.2016.226]

[51] SIFT and GIST. 2023. http://corpus-texmex.irisa.fr

[52] Million Song dataset benchmarks. 2012. http://www.ifs.tuwien.ac.at/mir/msd/

[53] Zhu YF, Luo CY, Ma RY, Chen L, Mao YR, Gao YJ. Density-based data clustering algorithm in multi-metric spaces. Ruan Jian Xue Bao/Journal of Software, 2025, 36(2): 851–873 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7177.htm [doi: 10.13328/j.cnki.jos.007177]

[54] Liu JX, Zhang X. STK: Clustering method based on contrastive learning embedding. Computer Science, 2024, 51(11A): 240400011 (in Chinese with English abstract). [doi: 10.11896/jsjkx.240400011]

[55] GOFMM. 2025. https://github.com/ChenhanYu/hmlp

[56] HNSWlib. 2025. https://github.com/nmslib/hnswlib

[57] KGraph. 2025. https://github.com/aaalgo/kgraph

[58] cuVS. 2025. https://github.com/rapidsai/cuvs

附中文参考文献

[11] 杜梦瑶, 李清明, 张淼, 陈曦, 李新梦, 尹全军, 纪守领. 面向隐私保护的用户评论基准数据集构建与大模型推理能力评估. 计算机学报, 2025, 48(7): 1529–1550. [doi: 10.11897/SP.J.1016.2025.01529]

[53] 朱轶凡, 罗程阳, 马瑞遥, 陈璐, 毛玉仁, 高云君. 基于密度的多度量空间数据聚类算法. 软件学报, 2025, 36(2): 851–873. http://www.jos.org.cn/1000-9825/7177.htm [doi: 10.13328/j.cnki.jos.007177]

[54] 刘晋霞, 张曦. STK: 基于对比学习嵌入的聚类方法. 计算机科学, 2024, 51(11A): 240400011. [doi: 10.11896/jsjkx.240400011]

作者简介

李忠根, 博士生, CCF 学生会员, 主要研究领域为新型硬件加速.

龚盛豪, 博士生, 主要研究领域为流式大数据管理与分析.

于浩然, 硕士生, 主要研究领域为向量数据库.

朱轶凡, 博士, 研究员, CCF 专业会员, 主要研究领域为大数据管理与分析.

柳晴, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为数据库可用性, 图数据分析, 大数据.

高云君, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 大数据管理与分析, DB 与 AI 融合.