Journal of Parallel and Distributed Computing 207 (2026) 105187

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

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

A General-Purpose K-Nearest Neighbor Method with an Efficient Pruning Strategy for GPUs

Jue Wang [a,*], Fumihiko Ino [a]

[a] *Graduate School of Information Science and Technology, The University of Osaka, 1-5 Yamadaoka, Suita, 565-0871, Osaka, Japan*

ARTICLE INFO

Keywords:
GPU
k-nearest neighbor
parallel computing.

ABSTRACT

K-nearest neighbor (kNN) search is widely applied to low- and high-dimensional tasks, as well as various data distributions and distance functions. However, its computational cost increases with the data volume, causing a bottleneck for many applications. The workload of the existing tree-based methods linearly increases with the neighbor count k in the worst case. In addition, some tree-based methods only apply to tasks with L2 distances and may have severe warp divergence when employed on GPUs. Our goal is to develop a general-purpose kNN method based on cluster sorting to achieve better pruning efficiency compared with tree-based approaches. We optimize the proposed method to achieve higher performance on tasks with different dimensionalities or distance functions. The proposed Sort, TraversE, and then Prune (STEP) algorithm is a kNN method that clusters the data points beforehand. With various 1) numbers of data points, 2) numbers of query points, 3) neighbor counts, 4) dimensions, and 5) distance metrics, the STEP method offers high performance because of the following aspects. First, our method prunes the data points efficiently by sorting the clusters for each query. Second, we exploit the single-instruction multiple-threads (SIMT) architecture of the GPU and utilize both coarse- and fine-grained parallelism to accelerate computation. The proposed method concurrently computes all queries and minimizes warp divergence by assigning a query to a GPU warp. Third, the STEP method rapidly updates the kNN results using bitonic operations. Fourth, we proposed an adaptive approach that automatically switches from the indexing approach to the exhaustive approach to achieve good scalability on high-dimensional data. Finally, we develop a variant of Gärtner's bounding sphere algorithm so that our indexing method can handle distance metrics other than the L2 distance. The STEP method achieves a 15.9 times speedup with L2 distances and a 36.7 times speedup with angular distances compared with other state-of-the-art methods.

1. Introduction

K-nearest neighbor (kNN) search is widely used in many applications such as image processing [1], data mining [2], and geographic information systems [3]. The implementation of kNN changes depending on the problem type. Tree-based indexing methods are often used for collision detection with 2D or 3D data [4]. Exhaustive kNN methods have been developed as classifiers for high-dimensional feature vectors [5,6]. In addition, they have been proposed for graph space to solve road network problems [7,8]. For tasks such as implicit surface definition [9,10], L2 distances between the points are calculated to perform the kNN methods. Angular distance is commonly used in computer vision tasks, such as content-based image retrieval [11]. In string space, the edit distance between two strings can be used to find the closest substrings from a database to a query string [12]. Researchers have developed several specific kNN methods for these applications. However, we attempt to develop a general-purpose kNN method that is broadly applicable to tasks with different conditions, such as data volume, dimensionality, and distance metrics.

The input of a kNN method contains a set of data points R, a set of query points Q, and the scalar of the neighbor count k. The kNN method then outputs the exact k nearest data points in R for each query $q \in Q$. We denote the size of Q and R by m and n, respectively. The elements of the data and queries are all d-dimensional vectors. The value of d changes depending on the task, e.g., $d = 2$ for the geographic data [13] and $d = 128$ in some computer vision problems [14]. Notice that this paper focuses on exact kNN algorithms, and the proposed method as well as the baselines are all exact kNN solutions.

As the data volume increases, the computational cost of the kNN computation increases, which becomes a bottleneck for intended applications. Several GPU-based parallel indexing methods have been proposed based on their sequential version [15–17]. An indexing method first performs a pre-processing step to build a data structure based on the data points. Then, the method transfers the queries and the newly built

[*] Corresponding author.
E-mail address: j-wang@ist.osaka-u.ac.jp (J. Wang).

https://doi.org/10.1016/j.jpdc.2025.105187
Received 12 November 2024; Received in revised form 2 April 2025; Accepted 15 October 2025
Available online 20 October 2025

data structure to the GPU and computes kNN results. Indexing methods offer reduced workloads owing to their pruning strategies. However, we must consider warp divergence [18] because of branching instructions in the indexing methods. On the other hand, exhaustive methods can achieve good performance using general matrix multiply (GEMM) accelerators on GPUs [6,19]. These methods divide the kNN computation into a distance matrix computation step and a top-k selection step. The distance matrix computation benefits from the GEMM accelerators, whereas the top-k selection limits the overall performance. Most exhaustive methods therefore focus on optimizing the performance of the top-k selection step.

GPU-based approximate nearest neighbor algorithms [20–23] have been proposed to achieve good scalability on large-scale data while sacrificing accuracy. However, exact kNN results are important and necessary in many practical applications, such as kNN searching on financial data [24] and nearest neighbor classifiers for time series [25]. We concentrate on exact kNN solutions in this research.

Existing kNN methods are suitable to specific situations and have limited performance in other conditions. When searching for one nearest neighbor, i.e., $k = 1$, tree-based methods exhibit logarithmic time complexity against the number of data points n. However, the time complexity linearly increases with the neighbor count k in the worst case. In addition, some tree-based methods only apply to tasks with specific distance functions and may have severe warp divergence when implemented on GPUs. Algorithms using space-partitioning data structures, e.g., k-d tree and octree, only handle tasks using the L2 distance. The similarity search tree (SS-tree) method relies on clustering instead of space partitioning, but its pruning efficiency worsens as the data volume increases [26]. Some indexing approaches adopt clustering methods to improve pruning efficiency [35,36]. Such methods have time-consuming pre-processing phases and are hard to achieve intra-query parallelism. Moreover, pruning strategies in indexing methods become less effective on high-dimensional data [27]. In contrast, exhaustive methods have poor performance on low-dimensional data. The curse of dimensionality has little effect on exhaustive methods because they work without pruning strategies.

Our goal is to develop a novel kNN method that yields better pruning efficiency than tree-based methods, especially in the presence of large k. We further optimize the proposed method to deal with both low- and high-dimensional data and with all types of distance functions. To achieve good performance with various (1) number of data points, (2) number of query points, (3) neighbor counts, (4) dimensions, and (5) distance metrics, our contributions are as follows:

1. We design a novel kNN algorithm with an efficient pruning strategy to reduce the workload. In the pre-processing step, the method divides the data points into clusters and computes the distances between each pair of query and cluster. The method sorts the clusters for each query beforehand and traverses all clusters. Once the pruning condition is satisfied, the search terminates, and all the remaining clusters are pruned. The proposed pruning strategy greatly reduces data access, which improves performance.
2. We exploit the SIMT architecture of the GPU and utilize both coarse- and fine-grained parallelism. The proposed method computes queries concurrently and assigns a query to a GPU warp of 32 threads. The main goal is to minimize the warp divergence with fine-grained parallelism. In addition, we developed an out-of-core approach of the STEP method to handle tasks that exceed the GPU memory capacity.
3. The proposed method further accelerates the kNN results update process. The STEP method batches candidate data points and merges them into the kNN results in parallel. A warp traverses the clusters and inserts candidate data points into a shared queue. Once the queue is full, the warp rapidly updates the kNN results using parallel bitonic sort and merge operations.
4. To improve the scalability of various dimensions, we design an adaptive workflow that switches between indexing and exhaustive approaches. In low-dimensional cases, the STEP method achieves high performance using k-means clustering and bounding sphere generation. For high-dimensional data, the STEP method naively clusters the data points and computes the exact minimal distances of each cluster to increase pruning efficiency. In addition, the STEP method speeds up the computation by initializing the kNN results with the exact minimal distances.
5. We develop a variant of Gärtner's algorithm [28] that generates exact bounding sphere results. The proposed method can compute bounding spheres with angular distance, leading to a noticeable speedup compared with approximate bounding sphere algorithms employed on high-dimensional data.

The proposed method yields better pruning efficiency than baseline methods with large k or in high-dimensional cases. On practical datasets, our method achieves a 15.9 times speedup with L2 distances and a 36.7 times speedup with angular distances compared with the state-of-the-art methods.

The remainder of this paper is organized as follows. In Section 2, we introduce related studies regarding GPU-based parallel kNN methods and sequential kNN algorithms using clustering as pre-processing. We explain the bounding sphere algorithms embedded in the evaluated methods in Section 3. We present the proposed parallel kNN methods in Section 4. We describe the implementation issues that must be solved to achieve complete GPU acceleration in Section 5. In Section 6, we show the experimental results of the proposed and baseline methods. We present the conclusions of our study in Section 7.

2. Related Work

Several existing sequential indexing methods for kNN search are parallelized and implemented on GPUs. The authors in [15] developed a parallel k-d tree approach. This method stores tree nodes in a single array and updates kNN results with on-chip shared memory to reduce memory traffic. In [16], an adaptive grid-partitioning method, which uses z-order curves to perform kNN search, was proposed. The authors in [29] proposed an indexing method based on a semiconvex hull tree. Each node is made from a set of hyperplanes and represents a semiconvex hull. The method proposed in [17] partitions the data points into equally sized bins. The search begins from the bin that contains the query and moves to adjacent bins until all nearest neighbors are found. The authors further extended the method to handle large datasets that exceed the GPU memory capacity [30]. These algorithms use coarse-grained parallelism, i.e., a GPU thread computes one query, which leads to high warp divergence, reducing computing efficiency of GPU computing. We chose the state-of-the-art method [15] using k-d tree as a baseline in our experiment.

Other indexing methods use fine-grained parallelism to relieve the warp divergence caused by branching and pruning. The buffer k-d tree method [31] builds a variant of the k-d tree from the data points. The method preserves good spatial locality with a small top tree in which each leaf node contains multiple data points. Once a query reaches a leaf node, it is stored in the leaf buffer. Threads in a thread block concurrently compute queries in the same leaf buffer. In [26], the authors developed a variant of the SS-tree method, which builds the tree using k-means clustering beforehand. First, the method finds the nearest leaf for each query and initializes the nearest neighbors. Then, the method searches the tree using in-order traversal and updates kNN results. A thread block is assigned to a query to parallelize distance computation. A drawback of this method is that pruning efficiency depends on the initialized kNN results. Notably, the pruning efficiency of indexing methods decreases as the high-dimensional data increases [27].

In addition to indexing methods, parallel exhaustive kNN approaches also achieve good performance owing to their concurrency mechanisms on the GPU. They mainly comprise two steps: distance matrix computation and top-k selection. The authors in [32] proposed a brute force kNN approach implemented on an NVIDIA GPU with two CUDA kernels. The first kernel computes the distances between all data points and query points. In the second kernel, each thread sorts the distances of one query and chooses the first k results as the nearest neighbors. In [6], the authors optimized the calculation of the distance matrix and improved the performance using the cuBLAS GEMM function. The cuBLAS library fully exploits the Tensor Cores on NVIDIA GPUs so that high-dimensional distance computation can be done efficiently. Therefore, the cuBLAS library is widely used in many exhaustive kNN solutions to compute distance matrices. The Sel-kNN method proposed in [19] assigns each query to a thread block to enable fine-grained parallelism when sorting distances. This method obtains kNN results efficiently with a parallel selection sort. The warp select method [33] is the state-of-the-art exhaustive kNN approach. Each query is assigned to a warp after the distance matrix computation. Each thread maintains a thread queue in registers to store kNN candidates. Once a thread queue is full, bitonic sort and merge operations are performed to update the kNN results. The authors in [34] improved the warp select method by replacing the thread queues with a shared queue. We compare the performance of our method with the Sel-kNN method [19] and warp select method [33]. The main advantage of exhaustive methods is that they compute kNN tasks directly without pre-processing. In addition, they can deal with various distance metrics. However, they may have relatively poor performance on low-dimensional data compared with indexing methods.

Besides GPU-oriented parallel kNN algorithms, the authors in [35] proposed a CPU-based sequential kNN solution that uses clustering methods as pre-processing. The method uses k-means clustering to split data points, computes the distances d_{rc} between data points and their centroids, and sorts the data points in descending order. In the searching process of one query, the method first calculates the distances d_{qc} from the query to the cluster centers and sorts the clusters in ascending order. It then traverses the data points in each cluster and prunes them using triangle inequality. In [36], the authors optimized pruning efficiency and memory consumption by selecting appropriate centroids without harming accuracy. Other methods [37,38] use clustering methods to compute approximate kNN results. These kNN methods using clustering methods have several drawbacks: (1) They lack an early stopping feature but traverse all clusters. (2) When using triangle inequality, the pruning efficiency drops significantly in high-dimensional cases. (3) Although enabling coarse-grained parallelism by assigning queries to different processors is trivial, it is hard to achieve fine-grained parallelism, i.e., intra-query parallelism. The reason is they perform pruning sequentially by verifying the triangle inequality of each data point. They thus require hundreds of seconds to compute kNN results for one query on 50 thousand data points [36]. We addressed these issues in the proposed method.

3. Bounding Sphere Algorithms

In this section, we briefly describe the bounding sphere algorithms. They are used to determine the boundaries of clusters in some indexing methods. Ritter's algorithm [39] and Gärtner's algorithm [28] generate approximate and exact bounding spheres, respectively. Notice that the proposed methods and the baselines compute exact kNN results, regardless of the quality of bounding spheres.

3.1. Ritter's Algorithm

Ritter's algorithm is an approximate approach to bounding sphere generation that outputs a 5%–20% larger bounding sphere than the optimum [39]. Given a set of data points, the algorithm determines the initial bounding sphere with two distant data points. Then, it iteratively adjusts the centroid and radius by traversing all data points and checking their distances. The algorithm is shown in Algorithm 1.

Algorithm 1 Ritter's Algorithm.

```
Input: data point array $R = \{r_0, r_1, \dots, r_{n-1}\}$.
Output: an approximate bounding sphere $b$.
1:  $r_u$ = the furthest data point to $r_0$
2:  $r_v$ = the furthest data point to $r_u$
3:  $r_w$ = the furthest data point to $r_v$
4:  $b_{\mathrm{o}}$ = the middle point of $r_v$ and $r_w$ // centroid
5:  $b_{\mathrm{r}} = DIST(r_v, r_w)/2$ // radius
6:  while true do
7:      $r_i$ = the furthest data point to $b_{\mathrm{o}}$
8:      if $DIST(r_i, b_{\mathrm{o}}) \leq b_{\mathrm{r}}$ then
9:          Break
10:     end if
11:     $\vec{v}$ = the unit vector from $b_{\mathrm{o}}$ to $r_i$
12:     $b_{\mathrm{o}} = b_{\mathrm{o}} + (DIST(r_i, b_{\mathrm{o}}) - b_{\mathrm{r}}) \cdot \vec{v}/2$
13:     $b_{\mathrm{r}} = (b_{\mathrm{r}} + DIST(r_i, b_{\mathrm{o}}))/2$
14: end while
15: return $b$
```

The SS-tree method employs Ritter's algorithm to generate bounding spheres for clusters because of its high computational efficiency. However, the workload of bounding sphere computation is significantly lower than that of k-means clustering. We can replace the approximate bounding sphere algorithm with the exact one to improve the quality of bounding spheres, which can enhance pruning efficiency.

3.2. Gärtner's Algorithm

Gärtner's algorithm generates an exact bounding sphere of the input data points [28]. The algorithm iteratively calculates the bounding sphere by adding points to a forced point list B, which contains all points on the boundary of the sphere. In each iteration, Gärtner's algorithm adds the point having the largest distance to the centroid to the beginning of the forced point list. Gärtner's algorithm is based on the move-to-front heuristic [40], as shown in Algorithm 2. The approach keeps the data points in an ordered list R, which gets updated during computation. Let R_j denote the length-j prefix of the list.

Algorithm 2 Move-To-Front (MTF) Heuristic.

```
Input: data point list $R = \{r_0, r_1, \dots, r_{s-1}\}$, forced point list $B$.
Output: the minimal bounding sphere $b$.
1:  $b = \mathrm{BS\_L2}(B)$ // compute the circumsphere of $B$
2:  // $b_{\mathrm{o}}$ and $b_{\mathrm{r}}$ are the centroid and radius of $b$
3:  // $d$ is the dimensionality of the points
4:  if $|B| = d + 1$ then
5:      Return $b$
6:  end if
7:  for $i = 0$ to $s - 1$ do
8:      if $DIST(r_i, b_{\mathrm{o}}) > b_{\mathrm{r}}$ then
9:          $b = \mathrm{MTF}(R_{i-1}, \{r_i\} \cup B)$
10:     end if
11: end for
12: Return $b$
```

This approach incrementally computes the bounding sphere by extending the forced point list B in which the points are affinely independent. The efficiency originates from the fact that the points far from the centroid are moved to the front. Therefore, these points are processed early in subsequent recursive calls. The move-to-front heuristic serves as a subroutine for small point sets in Gärtner's algorithm, as shown in Algorithm 3.

The authors further optimize the computation of the circumsphere in line 1 of Algorithm 2 using matrix operations. However, the matrix operation only works with the L2 distance. We develop a variant of the

Algorithm 3 Gärtner's Algorithm.

```
Input: data point list $R = \{r_0, r_1, \ldots, r_{n-1}\}$.
Output: the minimal bounding sphere $b$.
1:  $b_o = r_0, b_r = -1$ // the centroid and radius of $b$
2:  $s = 1$ // the number of forced points
3:  while true do
4:      $e = 0$ // check for excess
5:      foreach $r$ in $R$ do
6:          $e = \max(e, DIST(r, b_o) - b_r)$
7:          $r' = r$ if $e$ is updated
8:      end for
9:      if $e \le 0$ then
10:         Break // because no data point is outside the sphere
11:     end if
12:     $B = \{r'\}$ // $B$ gets updated in MTF
13:     $b$ = MTF($R_s$, $B$) // $R_s$ is the length-$s$ prefix of $R$
14:     Move $r'$ to the front of $R$
15:     $s = |B| + 1$
16: end while
17: Return $b$
```

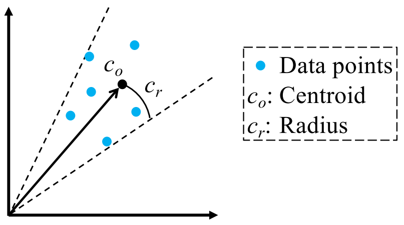


Fig. 1. 2D bounding sphere of the data points with angular distances. The bounding sphere is defined using the centroid c_o and the radius c_r.

circumsphere computation for angular distances, which is more suitable for practical high-dimensional tasks.

3.3. Bounding Spheres with Different Distance Functions

For the clusters calculated with angular distances, we also use the phrase "bounding sphere" for convenience, although the shape is more like a cone rather than a sphere. An example of a bounding sphere of the data points with angular distances is shown in Fig. 1. The centroid c_o is defined using either a point or a vector. The radius c_r is the angle between c_o and the boundary. A d-dimensional bounding sphere is determined using d data points on the boundary when using angular distances. For the L2 distance, it is trivial to calculate a bounding sphere of data points.

The distance between the query q and the cluster (or bounding sphere) c is calculated as follows:

$$DIST(q, c) = DIST(q, c_o) - c_r, \tag{1}$$

where c_o is the centroid and c_r is the radius of c. The distance function is represented using $DIST()$. This applies to any distance function that follows the triangle inequality. Examples of computing an L2 distance and an angular distance are shown in Fig. 2.

4. Method

The STEP algorithm is a general-purpose method proposed for kNN search. The computational workflow of the STEP method is shown in Fig. 3. The STEP method first clusters the data points during the pre-processing step. The access order of the clusters is resolved in Phase 1. Then, it loops over Phases 2–4 to compute kNN results. In this section, we first introduce the indexing STEP approach for a better understanding of the workflow. We bring out the exhaustive variant of the STEP method in Section 4.4.

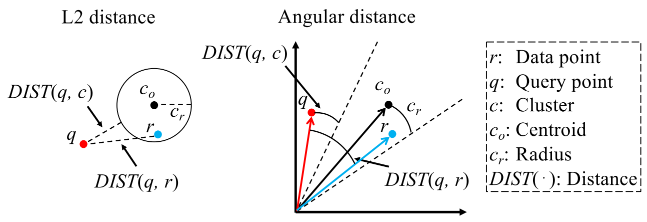


Fig. 2. Computation of the distance $DIST(q, r)$ from the query point q to the data point r and the distance $DIST(q, c)$ from q to the cluster c.

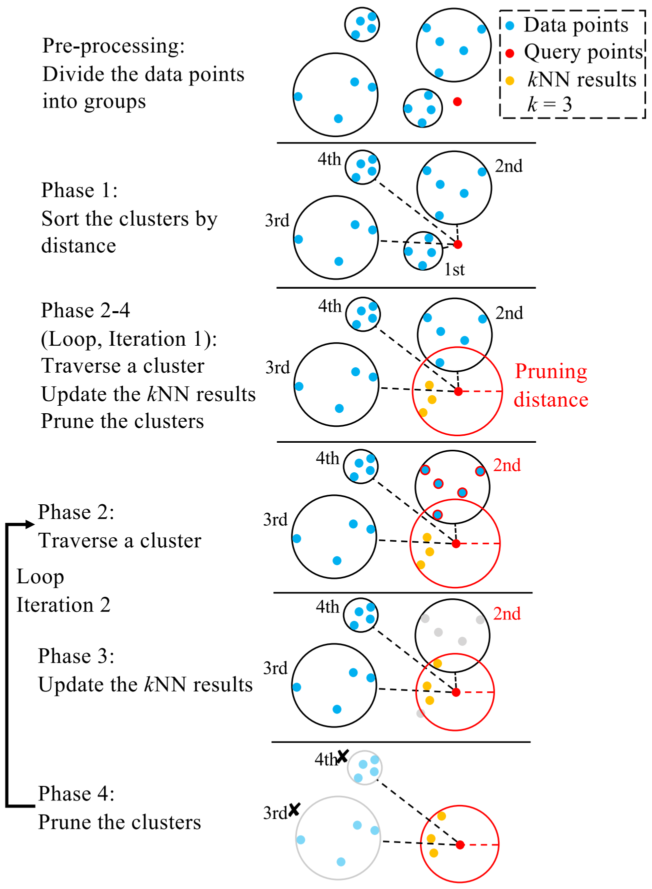


Fig. 3. Computation workflow of the STEP method for a single query. The STEP method concurrently computes kNN results for all queries.

In pre-processing, the STEP method divides data points into p groups with k-means clustering. The centroids of the groups are initialized with the k-means++ algorithm [41]. Then, the STEP method generates a bounding sphere for each cluster. In addition to the data points, the centroids and radii of clusters are transferred to the GPU after pre-processing.

4.1. Pruning Strategy

The STEP method maintains a pruning distance throughout the kNN computation, as shown in Fig. 3. The pruning distance is the distance between the query point q and the furthest data point in its kNN results. The STEP method prunes the data points in a cluster if the data points are further than the pruning distance. The relationship between the distance from q to the data point r and one to the cluster c is shown via the lemma below.

Algorithm 4 The Indexing STEP Method.

```
Input: data point set $R = \{r_0, r_1, \dots, r_{n-1}\}$, query set $Q = \{q_0, q_1, \dots, q_{m-1}\}$, neighbor count $k$, number of clusters $p$, size of candidate queue $n_c$.
Output: kNN search results $S[m][k] = \{\}$.
1:  $C = \{c_0, c_1, \dots, c_{p-1}\}$ = KMEANSCLUSTERING($R, p$)
2:  foreach query $q_i$ in $Q$ in parallel do
3:      $D[p] = \{\}$ // distances from $q_i$ to each cluster
4:      foreach cluster $c_j$ in $C$ in parallel do
5:          $(c_{j,o}, c_{j,r})$ = (centroid of $c_j$, radius of $c_j$)
6:          $D[j] = DIST(q_i, c_{j,o}) - c_{j,r}$
7:      end for
8:      SORT($D, C$)
9:      $S[i]$ = TRAVERSECLUSTERS($R, C, D, q_i, k, p, n_c$)
10: end for
11: return $S$
```

Lemma 1. $DIST(q, r) \geq DIST(q, c), \forall r \in c$.

Proof. When the query lies inside or on the sphere, i.e., $DIST(q, c) \leq 0$, the statement holds because $DIST(q, r)$ is nonnegative. When the query lies outside the cluster, i.e., $DIST(q, c) > 0$, we have the following relation because the distance function $DIST()$ follows the triangle inequality:

$$DIST(q, r) + DIST(r, c_o) \geq DIST(q, c_o) \tag{2}$$

Since $r \in c$, the distance between r and c_o is not larger than c_r. Consequently,

$$\begin{aligned} DIST(q, c) &= DIST(q, c_o) - c_r \\ &\leq DIST(q, c_o) - DIST(r, c_o) \\ &\leq DIST(q, r) \end{aligned} \tag{3}$$

□

The STEP method prunes the data points in the cluster c for the query q when the pruning distance is smaller than $DIST(q, c)$.

In Phase 1, the STEP method computes the distances between a query and all clusters. To determine the access order of the clusters, the STEP method then sorts the distances using a parallel bitonic sort adopted in [33]. The process corresponds to lines 3-8 in Algorithm 4. Sorting the clusters is critical in our pruning strategy. When the STEP method prunes a cluster c_j for a query q by comparing $DIST(q, c_j)$ and the pruning distance (explained later), the remaining clusters regarding the access order are all pruned because the data points in them cannot have a smaller distance than $DIST(q, c_j)$.

In Phase 2-4, the STEP method traverses all clusters based on the access order. The pruning strategy is shown as branching in Algorithm 5. The if-statement in line 3 prunes clusters, and the one in line 7 prunes data points. The array S', which is equivalent to the array $S[i]$ in Algorithm 4, maintains the kNN results of the query q_i in ascending order by distance. Therefore, the STEP method uses the data point in $S'[k-1]$, which stores the furthest data point to q_i, as the pruning distance. The STEP method compares the pruning distance with the distance between q_i and c_j before traversing the data points in c_j. If the pruning distance is smaller, all data points in c_j are further than the current kNN results according to Lemma 1. Therefore, the STEP method prunes c_j and all data points in it because they are irrelevant to the kNN results. All the remaining clusters are pruned simultaneously. It is unnecessary to check the pruning condition for the remaining clusters. The STEP method further reduces the workload by employing data point pruning (line 7). A data point r_t is inserted into the candidate queue only when the pruning distance is larger than the distance between q_i and r_t.

Notably, the STEP method determines the individual order of clusters for each query. Each query traverses the clusters in different orders, and the search process of each query is independent of the others. The STEP method computes the kNN results of all queries in parallel.

Algorithm 5 Traverse Clusters.

```
Input: data point set $R = \{r_0, r_1, \dots, r_{n-1}\}$, sorted cluster indices $C = \{c_0, c_1, \dots, c_{p-1}\}$, sorted cluster distances $D = \{d_0, d_1, \dots, d_{p-1}\}$, query $q_i$, neighbor count $k$, number of clusters $p$, size of candidate queue $n_c$.
Output: kNN search results $S'[k] = \{\}$.
1:  $R_c[n_c] = \{\}$ // candidate queue
2:  foreach cluster $c_j$ in $C$ do
3:      if $DIST(q_i, S'[k-1]) \leq D[j]$ then
4:          break
5:      end if
6:      foreach data point $r_t$ in $c_j$ do
7:          if $DIST(q_i, S'[k-1]) > DIST(q_i, r_t)$ then
8:              INSERT($R_c, r_t$)
9:          end if
10:         if $R_c$ is full then
11:             MERGE($S', R_c$)
12:         end if
13:     end for
14: end for
15: MERGE($S', R_c$) // Merge the remaining candidates
16: return $S'$
```

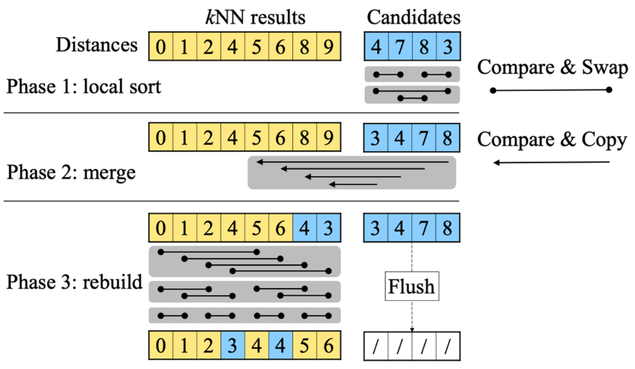


Fig. 4. Advanced results update process with bitonic sort and merge operations. The nearest neighbor count k is 8 and the candidate count n_c is 4. A grey block represents a parallel step.

4.2. Advanced Results Update Process

Updating kNN results is time-consuming for large k values. Moreover, it is nontrivial to exploit the parallelism of a max heap on a GPU. Our solution is to store n_c candidate data points in a shared queue and use bitonic sort and merge operations to update kNN results once the queue is full. The value n_c is a pre-defined constant, and the update process is fully parallelized with the threads in a warp to achieve high computational efficiency. We achieve parallel insert operations using the approach in [34].

Fig. 4 shows the results update process of the proposed method. The key idea is to batch n_c candidates and update the kNN results with a single bitonic merge operation. The update process sorts the candidate queue and merges it into the already sorted kNN results with three phases. In phase 1, the candidates are sorted with $\log n_c$ parallel steps. The candidates are merged into the kNN results with one parallel step in phase 2. The method rebuilds the kNN results in phase 3 with $\log k$ parallel steps and flushes the candidate queue. Notice that we only need to copy the data points instead of swapping them in phase 2 because it's unnecessary to maintain the candidate queue. There are $O(n_c \log n_c + k \log k)$ compare & swap/copy operations in total. However, the performance of this approach is limited when $k < n_c$ because some threads will become idle. In practice, we pad the size of the kNN result array to a multiple of n_c.

Designing an appropriate results update process is a trade-off between the workload and the update frequency of the pruning distance. When we update the kNN results once a candidate is found, the pruning distance is updated immediately so that more data points can be pruned. In contrast, the pruning distance is updated less frequently when we batch the candidates and merge them into the kNN results afterward. In our approach, the pruning distance is the last element of the sorted kNN result array. Even when a candidate with a smaller distance is inserted into the shared queue, the pruning distance will remain untouched. Because the STEP method prunes both clusters and data points with the pruning distance, the update frequency has a noticeable effect on the performance. We therefore set the value of n_c to 64 to fully utilize the threads in bitonic sort while making the shared queue small enough to increase the update frequency.

4.3. Variant of Gärtner's Algorithm

We developed a variant of Gärtner's algorithm [28] to compute bounding spheres with angular distances. To compute an exact bounding sphere of the data points in a cluster, we select a set of forced points B and calculate the unique circumsphere to obtain its centroid c_o and radius c_r. When the distances from all points to c_o are smaller than c_r, the circumsphere is the minimal bounding sphere $MB(B)$ of the cluster. The selection process is omitted from this section.

We assume that B contains affinely independent points with dimension d. In this case, the centroid c_o is restricted to the affine hull of B. Therefore, c_o and c_r satisfy the following equations, where $B = \{r_0, \ldots, r_{m-1}\}, m = d$:

$$\begin{aligned} &\frac{r_i^T c_o}{||r_i|| \cdot ||c_o||} = \cos(c_r), \quad i = 0, \ldots, m-1, \\ &\sum_{i=0}^{m-1} \lambda_i \frac{r_i}{||r_i||} = c_o, \text{and} \\ &\sum_{i=0}^{m-1} \lambda_i = 1. \end{aligned} \tag{4}$$

The points in B are constant during computation. We denote the normalized vectors of the points as follows:

$$\hat{r}_i = \frac{r_i}{||r_i||}, \quad i = 0, \ldots, m-1. \tag{5}$$

Defining $R_i := \hat{r}_i - \hat{r}_0$ for $i = 0, \ldots, m-1$ and $C_o := c_o - \hat{r}_0$, we derive that:

$$\hat{r}_0 c_o = \hat{r}_i c_o, \quad i = 1, \ldots, m-1, \tag{6}$$

$$R_i c_o = 0. \tag{7}$$

From Eq. (4) and Eq. (7), we obtain:

$$\sum_{i=1}^{m-1} \lambda_i R_i = C_o, \tag{8}$$

$$R_i^T C_o = -R_i^T \hat{r}_0, \quad i = 1, \ldots, m-1. \tag{9}$$

Substituting C_o with $\sum_{i=1}^{m-1} \lambda_i R_i$ in Eq. (9), we deduce a linear system in the variables $\lambda_1, \ldots, \lambda_{m-1}$, which can be written as

$$A_B \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_{m-1} \end{pmatrix} = \begin{pmatrix} -R_1^T \hat{r}_0 \\ \vdots \\ -R_{m-1}^T \hat{r}_0 \end{pmatrix}, \tag{10}$$

where $A_B[i][j] = R_{i+1}^T R_{j+1}$ because $R_s^T R_t = R_t^T R_s$ for $0 \le s, t \le m-1$:

$$A_B := \begin{pmatrix} R_1^T R_1 & \cdots & R_1^T R_{m-1} \\ \vdots & & \vdots \\ R_1^T R_{m-1} & \cdots & R_{m-1}^T R_{m-1} \end{pmatrix} \tag{11}$$

Eq. (10) presents a linear system with the format $Ax = b$. Therefore, we solve the linear system by computing $x = A^{-1}b$ to get the values of $\lambda_1, \ldots, \lambda_{m-1}$. Finally, we compute c_o and c_r as follows:

$$\begin{aligned} c_o &= \sum_{i=0}^{m-1} \lambda_i \hat{r}_i, \\ c_r &= \arccos(\hat{r}_0 \frac{c_o}{||c_o||}), \end{aligned} \tag{12}$$

where $\lambda_0 = 1 - \sum_{i=1}^{m-1} \lambda_i$.

4.4. Adaptive Workflow in High-Dimensional Cases

For indexing kNN solutions, the performance drops significantly in high-dimensional cases because the distance information between data points contributes little to the pruning efficiency due to the curse of dimensionality. Besides, on-demand distance computation during the search process harms the performance because of the global memory access latency on GPUs.

Algorithm 6 The Exhaustive STEP Method.

Input: data point set $R = \{r_0, r_1, \ldots, r_{n-1}\}$, query set $Q = \{q_0, q_1, \ldots, q_{m-1}\}$, neighbor count k, number of clusters p, size of candidate queue n_c.

Output: kNN search results $S[m][k] = \{\}$.

```
1:  $C = \{c_0, c_1, \ldots, c_{p-1}\} = \text{NaiveClustering}(R, p)$
2:  foreach query $q_i$ in $Q$ in parallel do
3:      $D[p] = \{\}$ // distances from $q_i$ to each cluster
4:      foreach cluster $c_j$ in $C$ in parallel do
5:          $D[j] = +\infty$
6:          foreach data point $r_t$ in $c_j$ do
7:              $D[j] = \min(DIST(q_i, r_t), D[j])$
8:          end for
9:      end for
10:     $\text{Sort}(D, C)$
11:     $S[i] = \text{TraverseClusters}(R, C, D, q_i, k, p, n_c)$
12: end for
13: return $S$
```

To achieve good scalability in high-dimensional cases, we propose an adaptive workflow of the STEP method that automatically switches from the indexing approach to the exhaustive approach. Algorithm 6 shows the exhaustive STEP method. Specifically, the STEP method alters the behavior in pre-processing and phase 1 shown in Fig. 3. In addition, all distances are computed beforehand in the pre-processing phase. The approach switch happens when the number of dimensions is not lower than a pre-defined threshold d_t. We suggest fixing d_t to 16 considering the GPU hardware features, but we evaluated the two approaches on both low- and high-dimensional datasets.

The major issue that weakens the pruning efficiency is the imprecise pruning thresholds of clusters, i.e., the radii of bounding spheres. The k-means clustering and bounding sphere generation offer good pruning efficiency in low dimensions. However, the bounding spheres expand rapidly as the number of dimensions increases. A query is often located inside most of the bounding spheres when the number of dimensions exceeds 16. Such clusters cannot be pruned because their distances to the query are 0. But for the data points in a bounding sphere, their distances to the query can be far more larger than the sphere's radius.

We replace the bounding sphere radii with exact minimal distances to improve the pruning efficiency. For each query, the STEP method computes its distances to all data points in a cluster and sets the minimal distance as the pruning threshold of the cluster. This approach has three advantages: (1) The meaningful pruning thresholds enable the STEP method to prune more clusters in high-dimensional cases. (2) The STEP method can directly initialize the kNN results with the exact minimal distances. (3) The STEP method can compute kNN results with distance functions that do not follow the triangle inequality. The drawback is obvious: the pre-processing becomes query-dependent, which makes the STEP method an exhaustive approach. Nevertheless, the indexing and exhaustive approach perform similar numbers of distance computations when the pruning efficiency is limited. They both compute $O(mn)$ distances but the indexing method requires additional branching operations, which leads to an inferior performance of the indexing method on high-dimensional data. Notice that the STEP method stores the computed distances in the GPU global memory and reuses them during the search process.

To further reduce the pre-processing time, we use a naive clustering strategy instead of the k-means clustering in the exhaustive STEP method. The exact minimal distances are more important than the clustering results regarding the pruning efficiency. The STEP method therefore performs the clustering most naively by equally dividing the data points into clusters based on the input order.

5. GPU-Based Implementation

In addition to the parallel algorithm, we performed several optimizations specific to the GPU architecture. These optimizations can be applied to other kNN methods. We consider Turing architecture [42] as the target GPU architecture. The STEP method is implemented using C++ and CUDA.

The details of reducing warp divergence and uncoalesced accesses are discussed in Section 5.1. In Section 5.2, we explain the kernel fission strategy that increases the occupancy of streaming multiprocessors in specific situations. We introduce the out-of-core STEP method to handle large-scale data that exceed the GPU memory capacity in Section 5.3.

Although most of the strategies in Section 5.1 are common techniques, they significantly enhanced the performance of the STEP method. Kernel fission in Section 5.2 optimized shared memory usage and slightly made the computation more efficient. The out-of-core strategy in Section 5.3 improves the applicability of the STEP method on large datasets.

5.1. Reducing Warp Divergence and Uncoalesced Accesses

Each query is assigned to a GPU warp of 32 threads in the STEP method to reduce warp divergence. When the threads in a warp compute kNN for multiple queries, they may execute different instructions according to the pruning results of the queries. In such cases, the threads sequentially execute instructions, which reduces performance. In contrast, the threads in a warp execute the same instructions when they compute for the same query.

Pruning conditions cause no warp divergence in the STEP method because the threads in a warp are assigned to the same query. Notice that the STEP method still computes all queries concurrently instead of dedicating all GPU resources to a single query. In addition, the insertion and bitonic merge operations in line 8, line 11, and line 15 are fully parallelized in the STEP method.

To eliminate most of the uncoalesced global memory access, we store the data in the structure-of-array pattern. The 32 threads in a warp access consecutive memory locations in the GPU global memory. In addition, the STEP method reduces shared memory usage by storing the kNN results with their indices and distances instead of the coordinates. Before returning the results in line 11 of Algorithm 4, the STEP method accesses the global memory with the indices and outputs the coordinates of kNN results. The uncoalesced global memory access in the STEP method is mainly caused by this operation.

With the structure-of-array pattern, the STEP method performs a parallel distance computation of the clusters without uncoalesced access in line 6 of Algorithm 4. We also optimized the computation of the loop in line 6 of Algorithm 5. Specifically, 32 data points in a cluster are processed in parallel. In the indexing STEP method, the statement $DIST(q_i, r_t)$ in line 7 is moved outside the loop and completed in advance.

5.2. Kernel Fission

Kernel fission means separating a CUDA kernel function into two or more kernels. While kernel fusion is a common optimization technique to reduce redundant global memory access and kernel launch overhead, kernel fission improves streaming multiprocessor (SM) occupancy in our implementation. The capacity of shared memory is fixed on an SM, and the amount of required shared memory limits the occupancy in the STEP method. More thread blocks fit into an SM when a block's shared memory requirement decreases.

Before applying kernel fission, the STEP method uses a single kernel to compute from phase 1 to phase 4. The frequently accessed data are stored in the shared memory because of its low access latency. The STEP method stores the cluster distances in the shared memory in phase 1. During phase 2 to phase 4, the kNN results and candidates are stored in the shared memory. Even when we reuse the shared memory space after phase 1 with pointer reinterpretation, the occupancy is still limited by the amount of shared memory. The motivation for kernel fission is that sorting clusters in phase 1 has a relatively low workload but requires a large amount of shared memory, especially when the cluster count is larger than k.

We therefore separate the STEP kernel into two kernels: The first kernel sorts the clusters and stores the distances in ascending order together with their cluster indices into the global memory. The second kernel loads the cluster information to determine the cluster access order and performs the remaining parts of STEP computation.

The merit of kernel fission in the STEP method is increasing the occupancy when the value of k is smaller than the cluster count. Assuming the required shared memory amount of cluster sorting is twice as large as that of the remaining STEP computation, kernel fission doubles the occupancy of the second kernel when the amount of shared memory is the bottleneck. The major disadvantage of kernel fission is the kernel launch overhead. The global memory reading and writing are not redundant because we also need to store the sorted clusters in global memory before we apply kernel fission.

5.3. Out-of-Core Approach

When implementing the STEP method on a single GPU, the scalability is limited by the GPU memory capacity. We therefore proposed an out-of-core method to improve the scalability on datasets with a large number of data points and query points.

To compute tasks with a large number of data points, the key idea is to divide the data points into chunks that fit into the GPU memory and compute local kNN results. After the STEP method finds the local results in a chunk, it launches a lightweight GPU kernel to merge them into the global kNN results. In the lightweight merge kernel, we use the bitonic operations similar to that in Section 4.2. We pipeline the our-of-core STEP method with three CUDA streams to overlap the GPU computation with CPU-GPU data transfer.

In the exhaustive STEP method, the large matrix containing the distances between query points and data points limits the scalability. While dividing data points introduces extra overhead because of the additional merge kernel, splitting independent queries into chunks is sometimes a reasonable strategy to handle large-scale datasets. However, it is necessary to compute over about 500 queries in parallel to fully utilize GPU resources when we assign each query to a GPU warp. To this end, the exhaustive STEP method automatically divides both the query points and data points into chunks to achieve better performance.

Notice that this strategy can also be utilized to extend the STEP method to multi-GPU platforms when the problem size exceeds the capacity of a single GPU. Each GPU node holds a chunk of data points and computes the local kNN results. Then, a single GPU node collects all local results and merges them into the global kNN results. Similarly, when the query count is too small to fill the GPU resources, we can distribute the chunks to different warp to increase parallelism. Multiple warps compute the local kNN results for the same query, and one warps merges the local results into the global kNN results.

6. Experiments

We evaluated the proposed STEP method in terms of GPU computation time, number of accessed (data) points, and number of retained (data) points. We used the latter two criteria originating from the average results of all queries to evaluate pruning efficiency. The accessed points were the ones read by the warp of its corresponding query. The retained points were the ones that were not pruned during the search process. Specifically, the number of retained points in the STEP method represents the number of data points inserted into the candidate queue. Our experimental machine had an AMD Ryzen 5 3600 with 32 GB RAM and an NVIDIA GeForce RTX 3070 GPU with 8 GB VRAM. The size of GPU shared memory per thread block was 48 KB. We used CUDA 11.7 and GPU Driver 525.147.05 on Ubuntu 22.04.

We compared the proposed method with seven baselines: Sel-kNN method [19], GPU-based Faiss library [33], CPU-based Faiss library [33], GPU-based k-d tree method [15], CPU-based k-d tree method [43], buffer k-d tree method [31], and SS-tree method [26]. The Sel-kNN method is an exhaustive approach that first computes a distance matrix with GEMM and then obtains the kNN results with a parallel selection sort. The GPU-based Faiss library is a state-of-the-art exhaustive approach that uses the warp select method. The threads in a warp store candidates in thread queues and merge them into the kNN results with bitonic operations. The CPU-based Faiss library, on the other hand, optimizes the computation with the BLAS library and SIMD vectorization. The GPU-based k-d tree method builds a binary tree of the data points and searches the tree with coarse-grained parallelism. The CPU-based k-d tree method also enables coarse-grained parallelism with multi-threading. The buffer k-d tree method constructs a small top k-d tree in which each leaf contains multiple data points. The SS-tree method builds the tree with k-means clustering and searches the tree using in-order traversal. The buffer k-d tree method and the SS-tree method both use fine-grained parallelism.

The pre-processing time of indexing methods and CPU-GPU data transfer time were excluded from our experiments. Instead, we provide qualitative results of the pre-processing time. The k-d tree method and buffer k-d tree method have relatively less pre-processing times because they build the index structures with $O(n)$ workloads. In contrast, the indexing STEP method and SS-tree method require a large amount of time to perform pre-processing. The workload of k-means clustering is $O(np\alpha)$, where p is the number of clusters and α is the iteration count of k-means computation. The pre-processing time of these two methods is longer than the kNN searching time in many cases, even though we used a GPU-based k-means library. For instance, the index building time reaches tens of seconds whereas the searching time is a few seconds.

Because there is no indexing building phase in the exhaustive methods, all their GPU computation time is included in the experimental results. Notably, the pre-processing of the exhaustive STEP method does not involve any clustering progress, except equally dividing the input data points into clusters. Compared to the time-consuming k-means clustering in the indexing STEP method, clustering in the exhaustive STEP method is omittable regarding the computation time. The computation time of the exhaustive STEP method includes all-to-all distance computation, exact minimal distance search, cluster sorting, and all the following phases.

The experimental setups are described in Section 6.1. We compare the performances of the proposed methods with those of the baseline methods on random datasets in Section 6.2. We evaluate the methods in various aspects using different number of query points m, number of data points n, neighbor counts k, and dimensions d and test the L2 distance and the angular distance in each case. Importantly, the k-d tree method and buffer k-d tree method can only perform kNN search with L2 distances. Therefore, we exclude these two methods when conducting experiments with angular distances. This is followed by the performance comparison of the practical datasets in Section 6.3. An ablation study on the STEP method is conducted in Section 6.4. Then, we analyze the warp divergence of the methods in Section 6.5. Finally, we evaluate the sensitivity of the STEP method to the bounding spheres in Section 6.6.

6.1. Experimental Setups

We tuned the parameters of the STEP method using preliminary experiments. The number of clusters p was set to 512 and 2048 in the indexing and exhaustive STEP method, respectively. We observed a considerable drop in pruning efficiency when $p < 512$. When setting $p \geq 1024$ in the indexing STEP method, the pruning efficiency improves little, and the occupancy decreases due to a large amount of the required shared memory. For the exhaustive STEP method, the pruning efficiency benefits from a larger value of p because the kNN results are initialized with the exact minimal distance of each cluster. However, sorting clusters becomes time-consuming when $p > 2048$. The number of threads in each thread block (or block size) was fixed at 32 because the required shared memory increases with the block size. Gärtner's algorithm was used to generate bounding spheres only for tasks with angular distances because few differences exist among the bounding sphere algorithms with L2 distances. The parameters of the baseline methods were well chosen for the experimental GPU.

Randomly generated and practical datasets were used in our evaluations. We prepared random datasets using the KISS algorithm [44], which generates query points and data points with normal distribution. Six practical datasets were used in our experiments. Open Street Map (OSM) datasets [13] include construction information from all over the world uploaded by contributors. We used the two-dimensional coordinates of 72,276 schools and 353,962 shops in the U.S. as the query points and data points, respectively. ANNSIFT1M [14] is a computer vision dataset that includes scale-invariant feature transform descriptors to identify objects in images. The dataset contains 10,000 query points and 1,000,000 data points of 128 dimensions. The KDDCUP-Bio dataset [45] includes various scores that describe protein sequences. The goal of the original task was to predict homologous proteins. The dataset contains 10,000 query points and 139,658 data points of 74 dimensions. KDDCUP-Phy [45] is a particle physics dataset. The data were collected in collision experiments with high-energy particle beams. A major problem in the experiments is to classify particle tracks. The dataset contains 10,000 query points and 100,000 data points of 78 dimensions. All values in the datasets are nonnegative. GIST [46] is also a computer vision dataset containing the global image structure tensor descriptors. The dataset contains 1000 query points and 1,000,000 data points of 960 dimensions. The DEEP10M dataset [47] consists of image embeddings produced as the outputs from the GoogLeNet model. The dataset contains 10,000 query points and 10,000,000 data points of 96 dimensions.

In all the evaluated methods, L2 distances were stored in float-type variables. However, the methods store angular distances in double-type variables because the values are too small to maintain precision with the float type. The details of the data types in each dataset are listed in Table 1. We use L2 distances when evaluating the OSM dataset for convenience, although computing the distance between two coordinates with the Haversine formula offers a better assessment. Notably, the experimental GPU supports single-precision instructions far more than double-precision instructions. We show the average results of 10 executions in all experiments.

6.2. Performance on Random Datasets

We evaluated the methods based on varying numbers of query points m, number of data points n, neighbor counts k, and dimensions d. When one of these four values changes, the other three are fixed to the default values as shown in Table 2.

Table 1
Data types in each dataset.

Datasets	Coordinates	Distances
Random values (L2)	Short	Float
Random values (angular)	Short	Double
OSM (L2)	Float	Float
ANNSIFT (L2)	Float	Float
KDDCUP-Bio (angular)	Float	Double
KDDCUP-Phy (angular)	Float	Double
GIST (L2)	Float	Float
DEEP10M (L2)	Float	Float

Table 2
Default values of each parameter.

Query Points	Data points	Neighbors	Dimensions
65536	65536	128	2

Fig. 5 shows the performances of the six methods on randomly generated datasets with L2 distances. The computation time against the query count is shown in Fig. 5a. The workload of all the methods increases linearly with the number of queries. The numbers of accessed and retained data points against the query count are demonstrated in Fig. 5b. The increasing query count has little effect on the accessed and retained points. The number of accessed points in the indexing STEP method is larger than that in the GPU-based k-d tree method. However, the computation time of the STEP method is smaller because the GPU-based k-d tree method shows a severe warp divergence. A similar result was observed for the SS-tree method in Fig. 5a and 5b. The STEP method achieves a 12.9 times speedup compared with the SS-tree method when the query count is 2^{16}.

The results against various numbers of data points are shown in Fig. 5c and 5d. The slopes of the GPU-based k-d tree method and the buffer k-d tree method are smaller than the other methods regarding the computation time because the height of the k-d tree increases by 1 when the number of data points doubles itself. The workload increases logarithmically for these two methods. In contrast, the workload of the other methods shows a larger increasing rate. Regardless, the accessed point count of the STEP method increases sublinearly with the number of data points, and the slope of the retained point count is smaller than 1.3. The STEP method is 13.9 times faster than the GPU-based k-d tree method when the number of data points is 2^{16}. When the number of data points surpasses 2^{24}, the computation time of the STEP method exceeds that of the k-d tree method. The exhaustive STEP method shows limited performance in Fig. 5a and Fig. 5c because its naive clustering strategy leads to a low pruning efficiency on 2D data.

Fig. 5e and 5f demonstrate the performance with varying neighbor counts k. For indexing methods, the numbers of accessed and retained points increase with the neighbor count. In contrast, the exhaustive methods, Sel-kNN and exhaustive STEP, always compute the distances for all pairs of query points and data points. The workload of updating kNN results increases in all methods. Importantly, the number of accessed points in the STEP method becomes less than that in the GPU-based k-d tree method when k is larger than 1024. Although the two k-d tree-based methods show good performance when the neighbor count is less than 10, their computation time increases rapidly with the neighbor count. The computation time of the buffer k-d tree method increases distinctly when k increases from 256 to 512 because the shared memory required for kNN results exceeds the GPU threshold. Thus, the buffer k-d tree method stores the kNN results in the global memory, which humbles the performance. The two STEP methods show good scalability regarding the neighbor count. The indexing STEP method attains a 28.9 times speedup compared with the SS-tree method when $k = 2048$.

The results with different dimensions d are illustrated in Fig. 5g and 5h. For the Sel-kNN method and exhaustive STEP method, the value of d is only related to the distance matrix computation step, of which the performance is limited by the number of query points and data points. In addition, the cuBLAS GEMM function has sufficient parallelism on the dimension. Thus, the value of d has little effect on the computation time of the exhaustive methods. The pruning efficiency degrades rapidly for the indexing methods owing to the curse of dimensionality [27]. The accessed point counts of the four indexing methods almost reach the maximal value of 65536 when $d = 16$ and remain the same when $d = 32$. The slopes of the computation time reduce accordingly when d increases from 16 to 32. Note that the buffer k-d tree method stores the kNN results in the global memory because of the restricted shared memory capacity. The indexing STEP method achieves a 29.6 times speedup compared with the SS-tree method when $d = 8$. When $d = 32$, the exhaustive STEP method shows better performance and achieves a 6.1 speedup compared with the SS-tree method.

For the angular distance, all the evaluated methods exhibit similar behaviors with different numbers of query points, numbers of data points, and neighbor counts. We observe some differences from the results with varying dimensions. Fig. 6 shows the performances for angular distances with different numbers of data points and dimensions. The workload of computing angular distances is larger than that of computing L2 distances. This has very little effect on the Sel-kNN method and exhaustive STEP method because the distance computation is optimized using GEMM. Generally, the retained points of the SS-tree and indexing STEP methods slightly increase compared with the ones using L2 distances. However, the accessed points of the indexing STEP method have a smaller slope with angular distance than that with L2 distance because of the tighter bounding sphere generated by Gärtner's algorithm. In other words, a tighter bounding sphere increases the number of pruned clusters in the indexing STEP method using angular distance. As stated in Section 6.1, Gärtner's algorithm only leads to a performance improvement with angular distance and high-dimensional data. The exhaustive STEP method achieves a 55.6 times speedup when $d = 32$ compared with the SS-tree method.

The workflow of the exhaustive STEP method and the Sel-kNN method can be divided into (a) query-dataset distance computation and (b) top-k selection. In Fig. 6c, both methods spend about 160 ms computing the distances with GEMM. For top-k selection, the exhaustive STEP method spends about 50 ms and approximately achieves a 200 times speedup over the Sel-kNN method. Besides the pruning strategy and the advanced results update process, the exhaustive STEP method greatly reduces the number of retained points by initializing the kNN results with the exact minimal distances of the clusters. Consequently, the exhaustive STEP method updates the kNN results less frequently and achieves high performance.

6.3. Performance on Practical Datasets

We evaluated the performance of the proposed method on six practical datasets. We only changed the value of the neighbor count k in the experiments using practical datasets because the other three parameters were fixed. In the two-dimensional OSM datasets, the query points and data points are the latitudes and longitudes of constructions in the U.S. The computation time as well as accessed points and retained points are shown in Fig. 7a and 7b. Similar to the case on the random dataset, the GPU-based k-d tree method is the fastest approach among the evaluated methods. However, its performance becomes inferior to that of the indexing STEP method when the neighbor count increases to 32. The STEP method achieves a 15.9 times speedup compared to the GPU-based k-d tree method when $k = 128$. Unlike the results on random datasets when $k = 128$, the k-d tree method offers better performance than the SS-tree method. The STEP method achieves a 20.8 times speedup compared

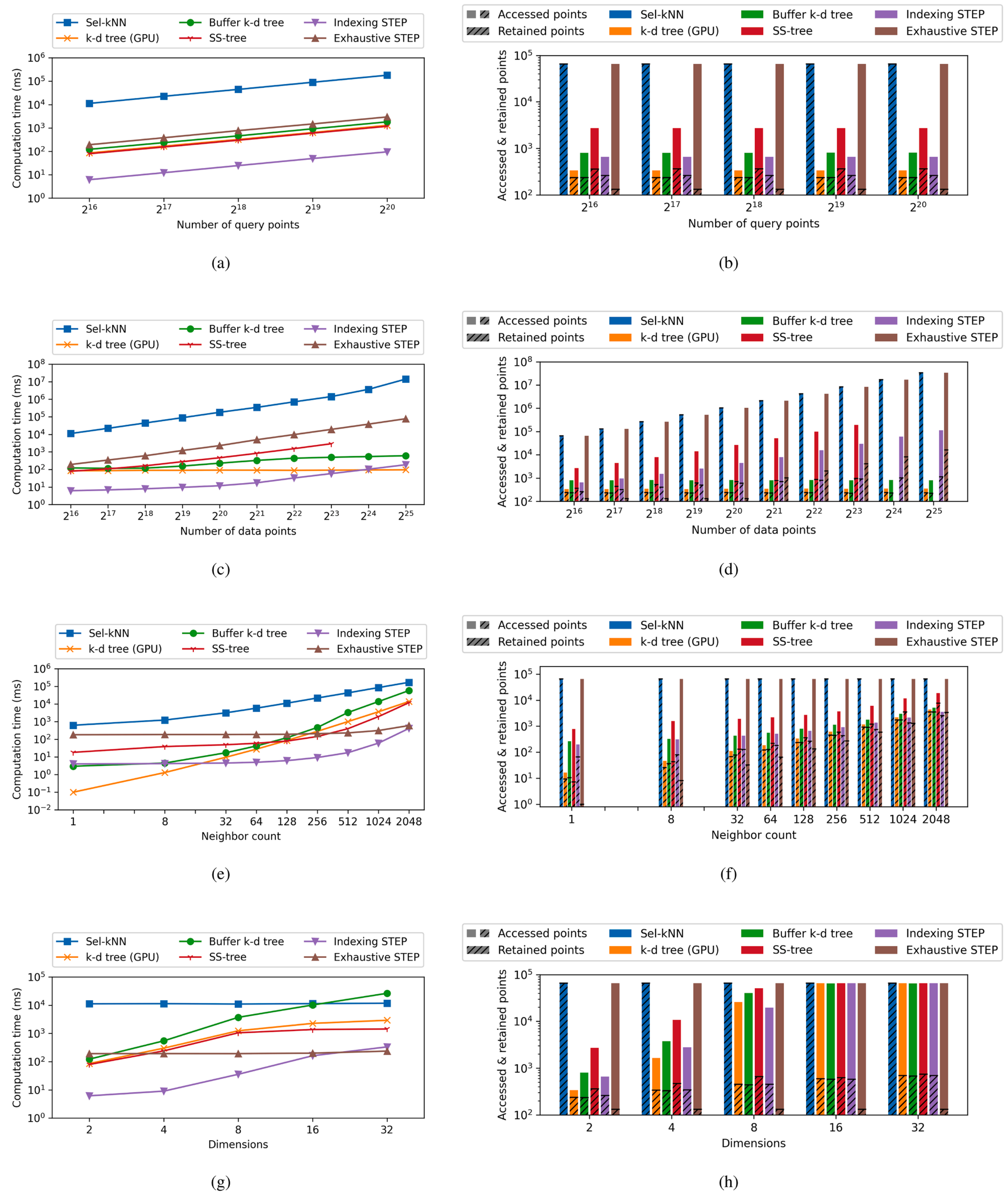


Fig. 5. Computation time and accessed and retained points on randomly generated datasets with L2 distances.

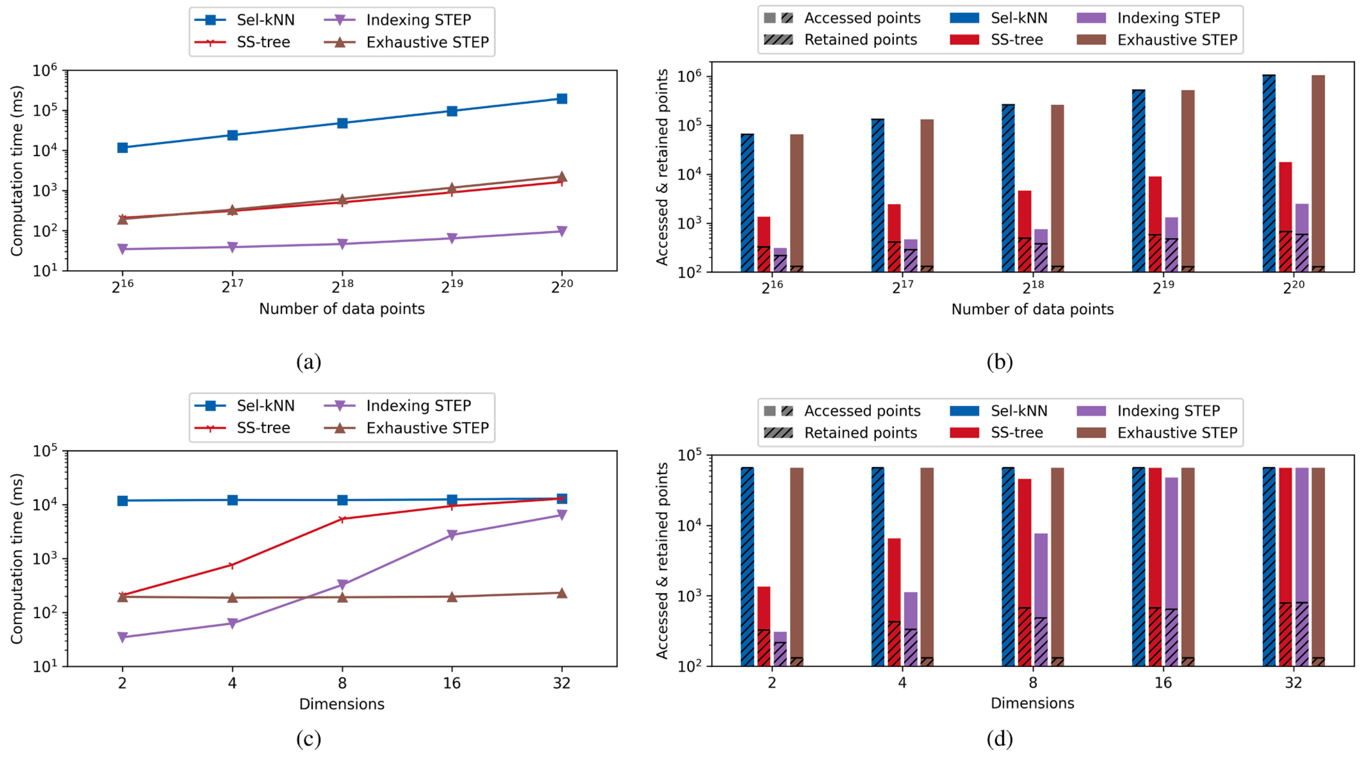


Fig. 6. Computation time and accessed and retained points on randomly generated datasets with angular distances.

to the SS-tree method when $k = 2048$. This speedup is less than the one achieved on random datasets because the pruning efficiency of the STEP method decreases. On random datasets, the retained point count of the STEP method reduces by 67% compared with the value of the SS-tree method when $k = 2048$. This ratio becomes 53% on OSM datasets. Different from the normal distribution in the randomly generated datasets, irregular distributions in practical datasets lead to lower pruning efficiency and potential load imbalance among clusters. This harms the performance of the indexing STEP method, and similar phenomena are found when evaluating the other practical datasets.

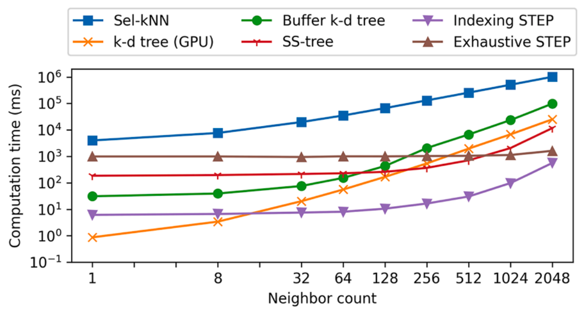
Fig. 7c and 7d show the results with varying values of k on the ANNSIFT dataset. The ANNSIFT dataset contains vectors of 128 dimensions and uses the L2 distance metric. Pruning strategies in the indexing methods hardly work on such high-dimensional data. The accessed point count of the indexing STEP method increases from 87% to 93% when k increases. In contrast, the exhaustive STEP method shows good performance on such high-dimensional data. The exhaustive STEP method has a smaller increasing rate of computation time than the Sel-kNN method because the kNN results are updated efficiently with bitonic operations. The buffer k-d tree method and SS-tree method are out of memory when $k \geq 1024$ and $k \geq 2048$, respectively. The exhaustive STEP method achieves a 29.0 times speedup compared with the SS-tree method when $k = 1024$.

On the KDDCUP-Bio dataset containing 74-dimensional data, the pruning efficiency of the indexing methods is higher than that obtained on the ANNSIFT datasets. Fig. 7e and 7f show the results with varying values of k on the KDDCUP-Bio datasets. The accessed point count of the indexing STEP method is 15% of the total number of data points when $k = 128$. The ratio increases to 23% when $k = 1024$, and the increasing rate is larger than that of the SS-tree method. The exhaustive STEP method achieves 36.7 times and 24.4 times speedups compared with the SS-tree method when $k = 128$ and $k = 1024$, respectively. The speedup decreases because the amount of required shared memory increases with the value of k, which limits the occupancy of the exhaustive STEP method. Fig. 7g and 7h demonstrate the results on the KDDCUP-Phy dataset with 74-dimensional data. The results show a similar trend although the slopes of the computation time of the two indexing methods are larger because of the higher increasing rate of the retained points with the value of k. The speedups of the exhaustive STEP method are 24.7 and 27.1 compared with the SS-tree method when $k = 128$ and $k = 1024$, respectively.

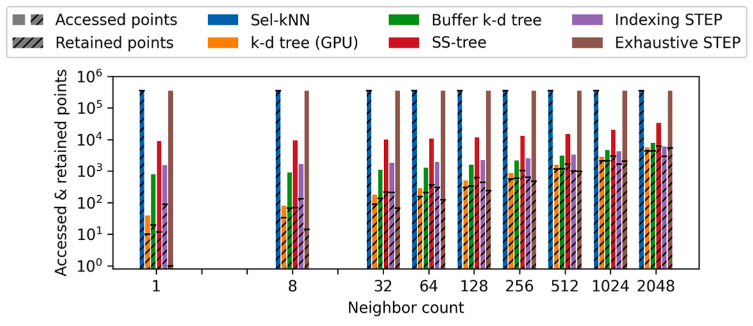
From Fig. 7, we conclude that the indexing STEP method is the fastest approach on low-dimensional datasets when the neighbor count is larger than 32. For tasks with a small neighbor count using L2 distances, the GPU-based k-d tree method offers good performance. In high-dimensional tasks, the exhaustive STEP method demonstrated significantly better performance compared to other indexing methods and the Sel-kNN method.

We compare the performance of the exhaustive STEP method and the Faiss library on large-scale datasets in Fig. 8. The tree-based baseline methods (k-d tree, buffer k-d tree, and SS-tree) required too much GPU memory to store the built tree, leading to out-of-memory issues when computing on the DEEP10M and GIST datasets. We show the comparison in a separate figure because the differences are too small to show in Section 6.3 which has a wide range of computation time. Notably, the Faiss library is devoted to high-dimensional datasets. With the default parameter setting in Table 2, the computation time of the GPU-based Faiss library is over 200 ms, whereas the indexing STEP method completes the computation in 10 ms. We only show the computation time of the Faiss library because the numbers of accessed and retained points were unavailable.

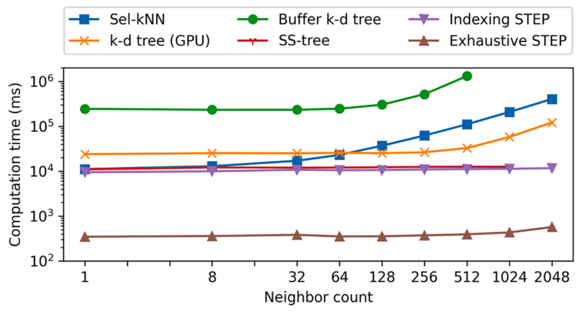
In Fig. 8a, we illustrate the computation on the ANNSIFT dataset. When the neighbor count $k = 1$, the performance of the GPU-based Faiss library is slightly better than that of the STEP method. However, the speedup of the STEP method over the Faiss library gradually increases with the neighbor count. The speedups are 1.2 and 2.1 when $k = 128$ and $k = 2048$, respectively. The neighbor count greatly affects the streaming multiprocessor's occupancy in these two methods. In the STEP method, the occupancy is limited by the size of the required shared memory because the candidates and kNN results are stored in the shared



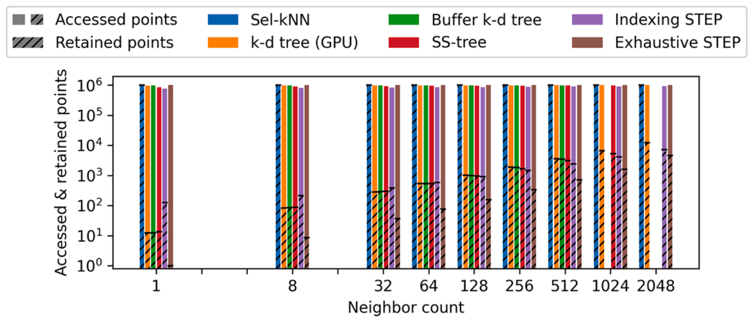
(a)



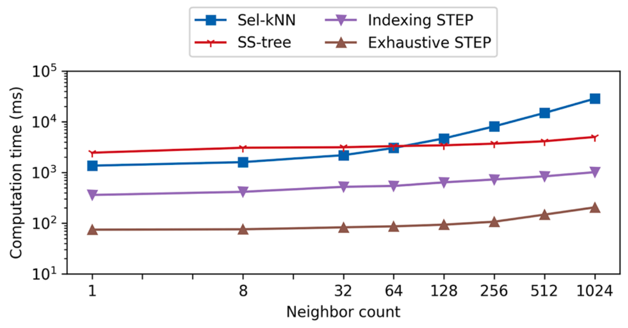
(b)



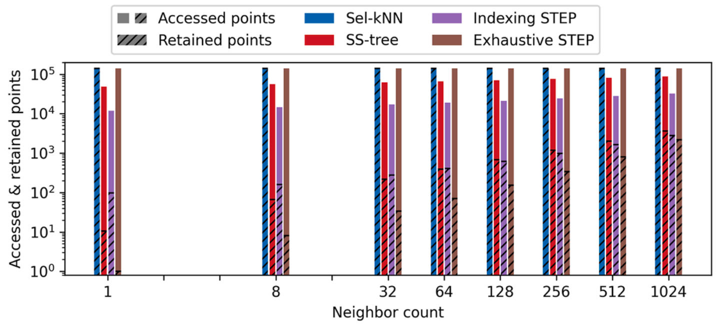
(c)



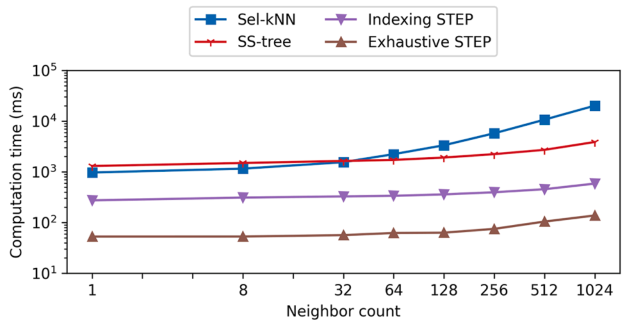
(d)



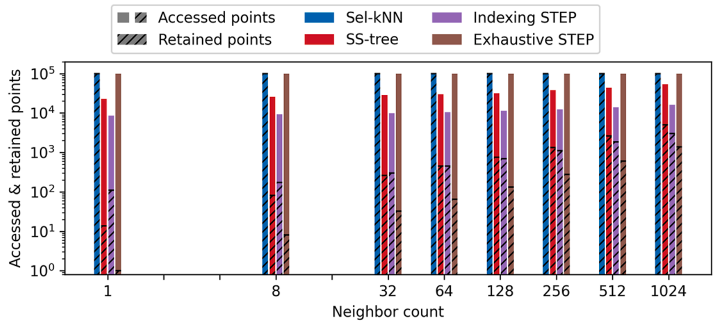
(e)



(f)

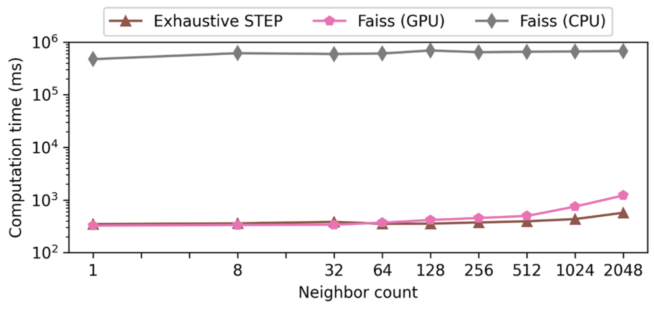


(g)

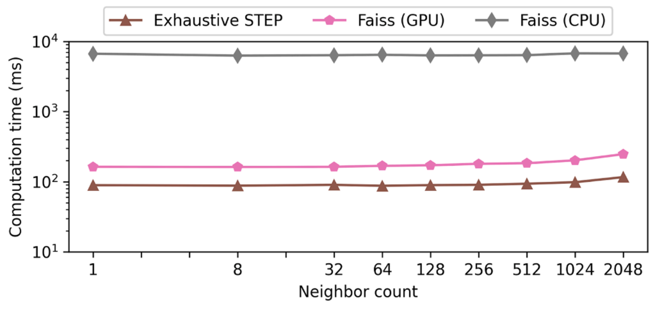


(h)

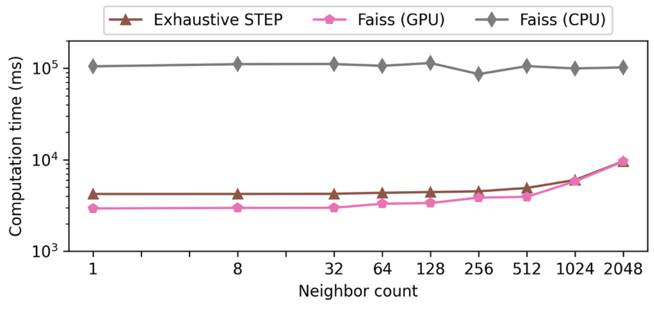
Fig. 7. Computation time and accessed and retained points on (a) and (b) OSM, (c) and (d) ANNSIFT, (e) and (f) KDDCUP-Bio, and (g) and (h) KDDCUP-Phy datasets. We use L2 distances on OSM and ANNSIFT datasets and angular distances on the other datasets.



(a)



(b)



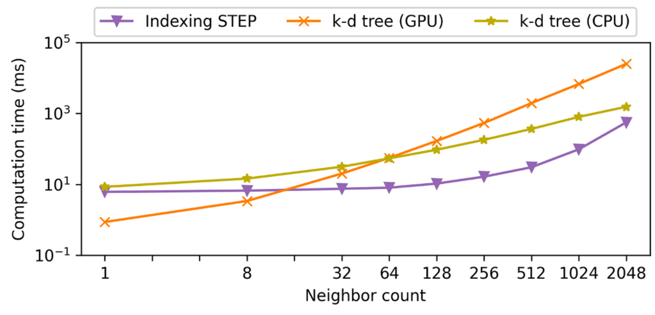
(c)

Fig. 8. Performance comparison between the exhaustive STEP method and the Faiss library. We show the computation time on (a) ANNSIFT, (b) GIST, and (c) DEEP10M datasets with L2 distances.

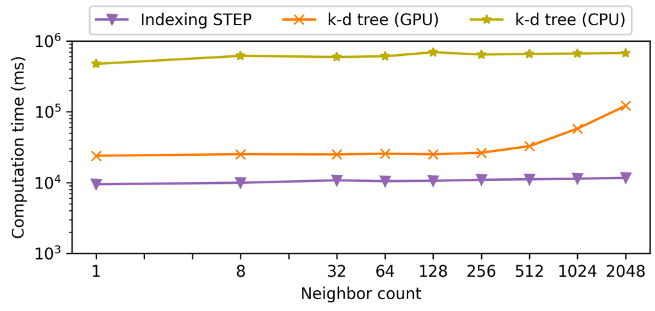
memory. In contrast, the GPU-based Faiss library stores the candidates and results in the registers when updating the kNN results. This strategy makes the number of registers the limiter of the occupancy. The STEP method achieves better scalability on various neighbor counts because of the efficient results update process. The CPU-based Faiss library is three orders of magnitude slower than the GPU-based methods, which shows that it is better to leverage the parallelism of the GPU to process such a large number of queries with the exhaustive methods.

For the performance on the GIST dataset with 960-dimensional data in Fig. 8b, the exhaustive STEP method demonstrates a similar behavior to that on the ANNSIFT dataset. The STEP method shows good scalability regarding the neighbor count, whereas the performance of the Faiss library drops with a low value of k. The speedups are 1.9 and 2.1 when $k = 128$ and $k = 2048$, respectively. This shows the robustness of the exhaustive STEP method on high-dimensional datasets.

We use the out-of-core STEP method on the DEEP10M dataset with 96-dimensional data in Fig. 8c. The dataset contains 10 thousand query points and 10 million data points, which makes the distance matrix size



(a)



(b)

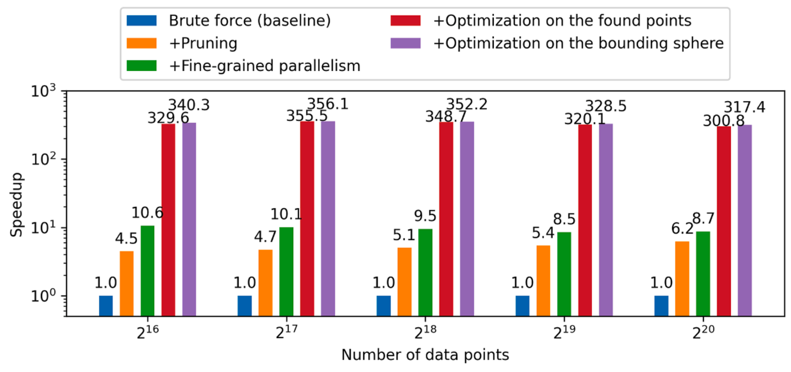
Fig. 9. Performance comparison between the indexing STEP method, the GPU-based k-d tree method, and the CPU-based k-d tree method. We show the computation time on (a) OSM and (b) ANNSIFT datasets with L2 distances.

too large to fit into the GPU memory capacity. The speedups are 0.8 and 1.0 when $k = 128$ and $k = 2048$, respectively. The speedup is low due to the extra local result merging operations in the STEP method. However, the STEP method achieves similar performance to that of the GPU-based Faiss library because of its better scalability regarding the neighbor count. We only evaluated the performance with up to 10 million data points because of the limited CPU memory capacity. Nevertheless, our out-of-core extension offers linear scalability regarding the number of data points. A machine with a large CPU memory capacity can run the proposed method efficiently.

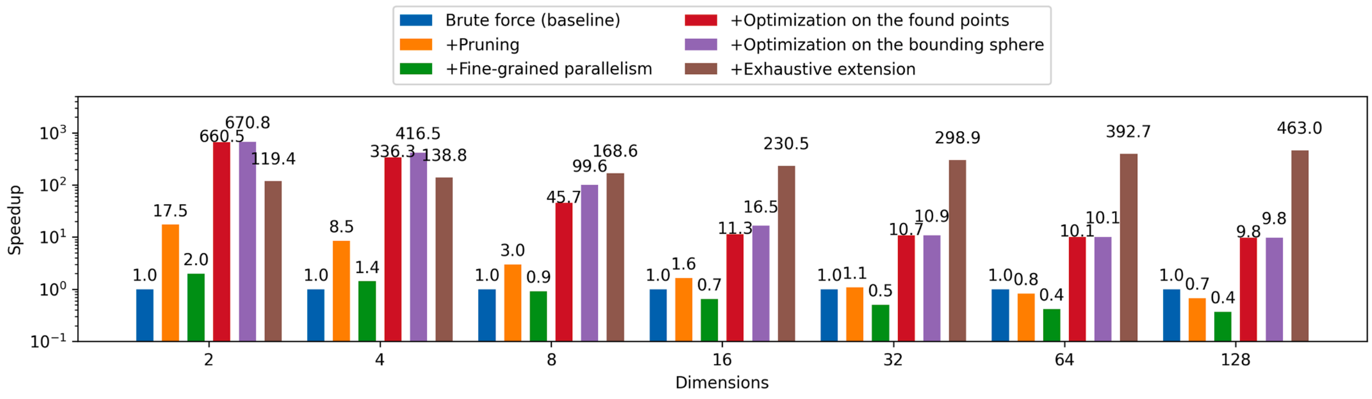
We also compare the performance of the indexing STEP method, the GPU-based k-d tree method, and the CPU-based k-d tree method in Fig. 9. In Fig. 9a, we show the computation time on the OSM dataset. Both CPU- and GPU-based k-d tree methods are significantly fast when the neighbor count is 1 because they rapidly reach the nearest neighbor with the binary search. When the neighbor count increases to 128, the CPU-based k-d tree method achieves better performance than that of the GPU version. A possible reason is that the GPU version stores the candidates in the shared memory. When the neighbor count increases, the amount of the shared memory becomes the bottleneck that limits the performance. The same issue is also observed in the indexing STEP method. Fig. 9b demonstrates the computation time on the ANNSIFT dataset with 128-dimensional data. The tree structure becomes less efficient in high-dimensional cases. In conclusion, the CPU-based k-d tree method shows comparable performance with the indexing STEP method on low-dimensional tasks. The indexing STEP method is more efficient when the number of dimensions increases.

6.4. Ablation Study

We show the results of the ablation study of the STEP method in Fig. 10. We compare the STEP method with its five variants. The first variant is a brute force method, and we show the speedups of the



(a)



(b)

Fig. 10. Ablation study of the STEP method with (a) L2 and (b) angular distances. The evaluations are conducted on randomly generated datasets using the default parameters in Table 2.

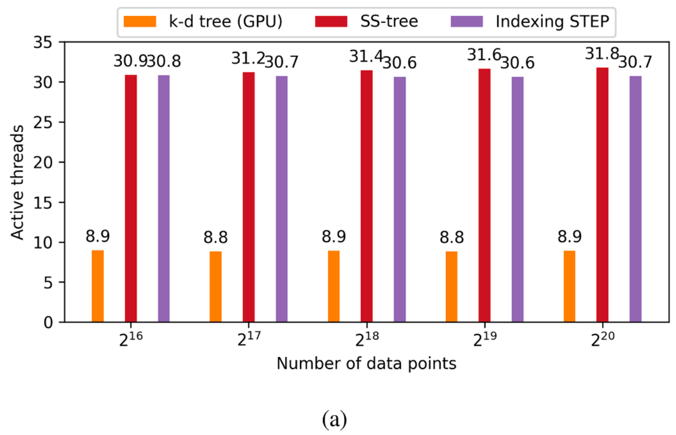
other variants against this method. Notably, the brute force method is different from the Sel-kNN method in that it replaces the GEMM with a kernel function that assigns each query to a thread and traverses the data points. The remaining five variants correspond to our five contributions. The second variant uses the proposed pruning strategy with k-means clustering and a naive bounding sphere algorithm. A query is assigned to a thread, and the kNN results are stored in the global memory. We compare the bounding sphere algorithms in Section 6.6. The third variant applies fine-grained parallelism that computes kNN for a query with 32 threads in a warp. However, the kNN update is still computed using one thread. The fourth variant stores the kNN results in shared memory and merges the candidates in the shared queue with kNN results using bitonic operations. The fifth variant optimizes the bounding sphere algorithm to generate the tightest bounding spheres for clusters. The sixth variant, which is only evaluated with various dimensions, switches from the indexing approach to the exhaustive approach to achieve good scalability on high-dimensional data. We evaluate the performance with different numbers of data points n using L2 distances and that with different dimensions d using angular distances on random datasets. We set the other parameters based on Table 2.

When the number of data points increases in tasks using L2 distances, the speedup of the second variant, which is optimized only with the pruning strategy, also increases. The pruning efficiency is enhanced with a larger number of data points. The fine-grained parallelism accelerates the computation a step further but reduces the slope of speedup, which also affects the last two variants. The optimization of the update process leads to an order of magnitude speedup, whereas the optimization of the bounding sphere algorithm slightly improves the performance. The indexing STEP method achieves a 317.4 times speedup compared with the brute force method.

The situation varies in tasks using angular distances with different dimensions. The pruning efficiency worsens when the dimension increases. The second variant is 17.5 times faster than the brute force method when $d = 2$. However, the speedup decreases to 0.7 when $d = 128$ owing to the curse of dimensionality [27]. The fine-grained parallelism makes the computation even slower because heavy angular distances are computed repeatedly when updating kNN results. The optimization of the update process greatly improves performance by reducing the workload of distance computation and memory traffic. The optimization of the bounding sphere algorithm also shows considerable improvement with high-dimensional data. The accessed point count reaches the maximal value, which is the number of data points, when $d = 32$. Thus, the speedups change insignificantly when $d \geq 32$. The results of the accessed points obtained using the STEP method are shown in Fig. 6d. After switching from the indexing approach to the exhaustive approach, the performance decreases when $d \leq 4$ due to the unnecessary distance computations between all pairs of query points and data points. However, the exhaustive extension leads to a performance improvement when $d \geq 8$ because the exact minimal distances of the clusters greatly enhance the pruning efficiency. The exhaustive STEP method achieves a 463.0 times speedup compared with the brute force method.

6.5. Analysis of Warp Divergence

We evaluated the warp divergence of the proposed method using the average active threads in a warp reported by NVIDIA Nsight Compute [48]. The number of active threads was measured when profiling a kernel function. Most of the computation in the Sel-kNN method was performed by calling a GEMM function instead of a kernel function. The buffer k-d tree method iterates with multiple kernel functions with



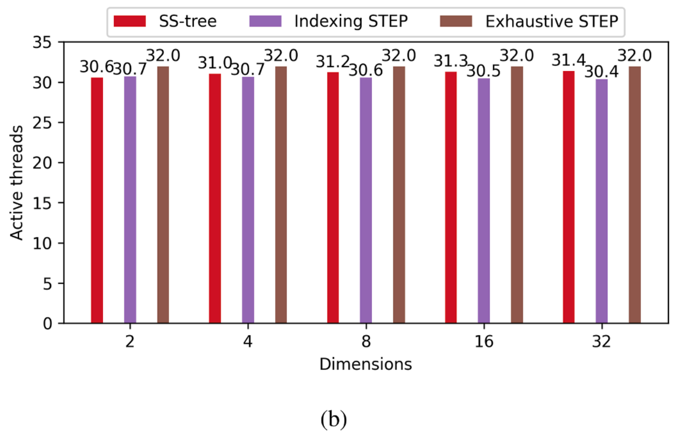


Fig. 11. Average active threads in a warp obtained using the evaluated methods on random datasets with (a) L2 and (b) angular distances.

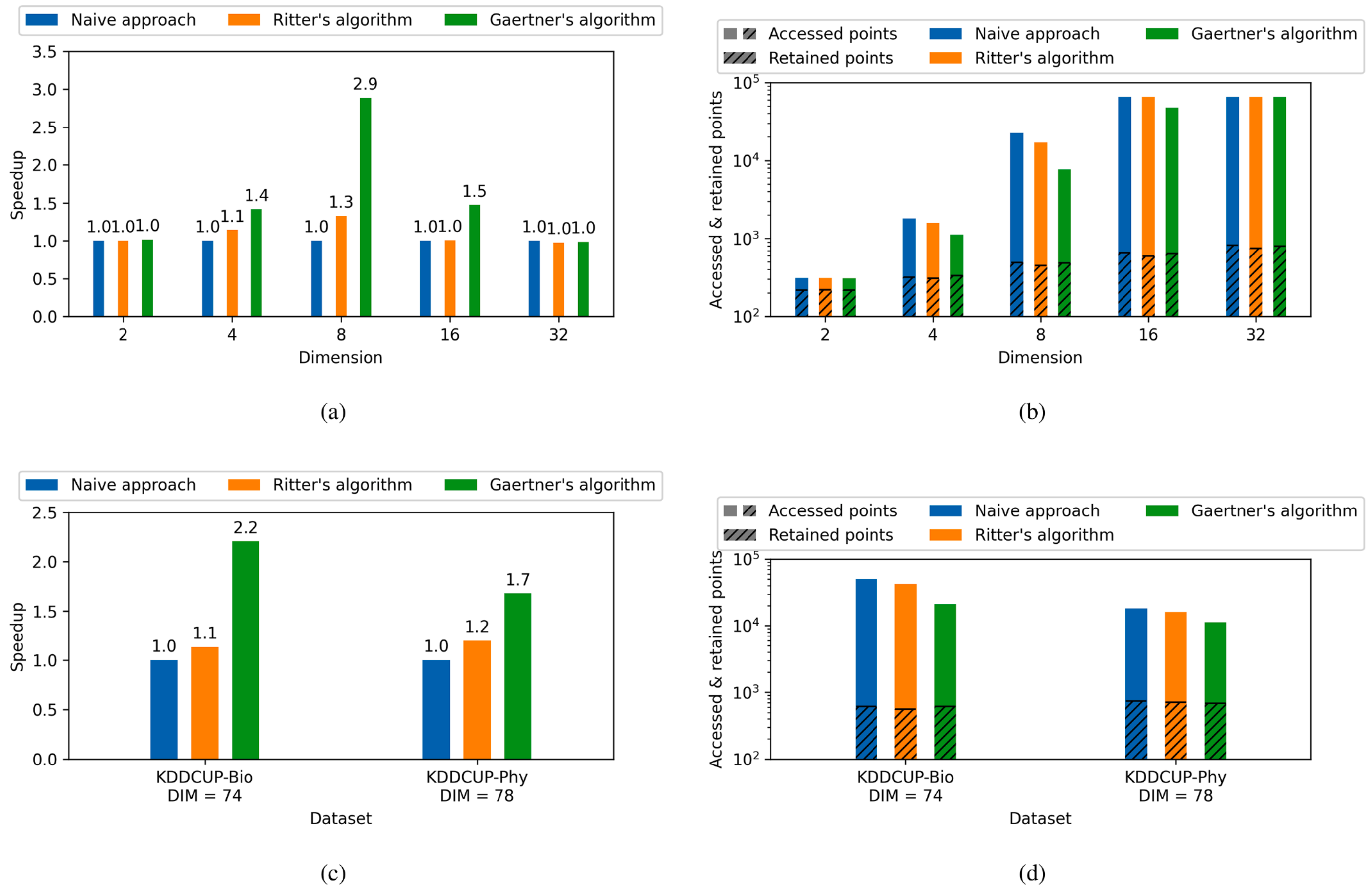


Fig. 12. Comparison of different bounding sphere algorithms in the STEP method. These figures show the speedups as well as accessed and retained points obtained on (a) and (b) random datasets and (c) and (d) practical datasets with angular distances. We use the naive approach as the baseline of speedup.

different workloads. Thus, we excluded these two methods from our experiments. The methods are evaluated with different numbers of data points n using L2 distances and with different dimensions d using angular distances. We set the other parameters based on Table 2.

Fig. 11a shows the average active threads of the three methods with L2 distances. We exclude the exhaustive STEP method because it is devoted to high-dimensional cases. The GPU-based k-d tree method has less than 9 active threads in a warp in all test cases. In contrast, the indexing STEP method and SS-tree methods have more than 30 active threads. When $n = 2^{20}$, the STEP method has 30.7 active threads, which is 96% of the total threads in a warp. Therefore, the STEP method computes the kNN in less time compared to the GPU-based k-d tree method when $n = 2^{20}$, as shown in Fig. 5c and 5d. Although the numbers of accessed points and retained points of the STEP method are larger than those of the GPU-based k-d tree method, the STEP method achieves better performance because of more active threads in a warp. For angular distance, similar behavior is observed, as shown in Fig. 11b. The two indexing methods maintain a large number of average active threads with varying dimensions. We eliminate most of the warp divergence in the exhaustive STEP method by setting the number of data points in each cluster to a multiple of 32. Consequently, all threads are active when traversing clusters. However, it is unachievable with k-means clustering in the indexing STEP method because the number of data points in each cluster is dependent on the centroid locations.

6.6. Sensitivity to the Bounding Spheres

The analysis of the bounding sphere algorithms is demonstrated in this section. We only show the results of the STEP method using angular distances because the bounding sphere algorithms have little effect on tasks using L2 distances. The results are computed using the indexing STEP method because the exhaustive approach replaces the bounding spheres with exact minimal distances. We evaluate the sensitivity to the bounding spheres using three bounding sphere algorithms. The naive approach chooses the initial centroid using the same pattern as that in Ritter's algorithm. Then, it adjusts the bounding sphere by only enlarging the radius but not the centroid. Ritter's algorithm is employed by the SS-tree method and generates an approximate result that is 5% to 20% larger than the exact bounding sphere. We implement a variant of Gärtner's algorithm, which generates the exact bounding spheres with angular distances.

Fig. 12a and 12b illustrate the performance of the STEP method using the three bounding sphere algorithms employed on random datasets. We used the results of the naive approach as the baseline. When the number of dimensions d increases, the STEP method accesses fewer clusters (or data points) with the exact bounding spheres. A 2.9 times speedup is achieved with Gärtner's algorithm when $d = 8$. The speedup drops when $d \geq 16$ because the number of accessed points reaches the maximal value of 2^{16} for the naive approach and Ritter's algorithm. We observe the same results even with a larger number of data points owing to the normal distribution of the randomly generated datasets. Besides, Gärtner's and Ritter's algorithms generate the same bounding spheres when $d = 2$. The radii computed by Gärtner's algorithm become smaller when $d > 2$.

The results of the three bounding sphere algorithms on the practical datasets are shown in Fig. 12c and 12d. Different from the results on random datasets, the pruning strategy using the naive approach still works well on high-dimensional practical datasets. The STEP method with Gärtner's algorithm achieves a 2.2 times speedup compared with that obtained using the naive approach. The STEP method is more sensitive to the quality of bounding spheres on practical datasets than on randomly generated datasets.

Notably, Gärtner's algorithm sometimes produces more retained points than Ritter's algorithm. For instance, the number of retained points of Gärtner's algorithm is 9% larger than that of Ritter's algorithm on the KDDCUP-Bio dataset. The reason is that the candidate count in each cluster does not always decrease with the cluster order. Sorting exact bounding spheres may generate suboptimal cluster access orders. Nevertheless, Gärtner's algorithm offers a higher performance because of fewer accessed points. In other words, the STEP method prunes more clusters with the tightest bounding spheres.

7. Conclusion

In this paper, we presented STEP, a general-purpose method for kNN computation on GPU. We designed a novel clustering-based pruning strategy as well as its GPU-oriented optimization. To obtain good performance in both low- and high-dimensional cases, the STEP method switches between the indexing and exhaustive approaches to improve pruning efficiency. We further accelerate the STEP method by enabling fine-grained parallelism and integrating bitonic operations into the updating procedure of kNN candidates. Our method achieves a 15.9 times speedup with L2 distances and a 36.7 times speedup with angular distances compared with the state-of-the-art methods. The proposed method achieves high performance on various randomly generated datasets and practical datasets, showing its generalizability across different distance functions and dimensionalities.

The pre-processing time of the indexing STEP method is sometimes comparable to or even exceeds the computation time. The workload of k-means clustering in pre-processing is $O(ndp\alpha)$, where n is the number of data points, d is the number of dimensions, p is the number of clusters, and α is the iteration count of k-means computation. We set $p = 512$ and $\alpha = 50$ in our experiments. On the other hand, the workload of searching is $O(mndk)$, where m and k are the query count and neighbor count. The searching time of our default test case, where $m = 65536$ and $k = 128$, is similar to the pre-processing time. This indicates that pre-processing cannot be disregarded when applying the indexing STEP method to practical cases. The suitable use-cases include (a) multiple STEP calls using the same dataset and different queries and (b) tasks with relatively larger m and k. In contrast, the exhaustive STEP method can be applied to all use-cases thanks to the naive clustering strategy.

Although we focused on exact kNN approaches, there are some potential extensions to achieve efficient approximate nearest neighbor searching with the STEP method. For instance, the k-means clustering can be replaced with a space-filling curve approach to reduce preprocessing time. We can further improve the STEP method's efficiency by limiting the number of clusters it accesses. In our future work, we will extend the STEP method to multi-GPU platforms to improve the scalability on large-scale datasets. Besides, we will develop a CPU-based STEP method that is useful for tasks with a small number of queries.

CRediT authorship contribution statement

Jue Wang: Writing – original draft, Methodology, Investigation, Visualization, Writing – review & editing, Software, Formal analysis; **Fumihiko Ino:** Validation, Writing – review & editing, Funding acquisition, Supervision, Resources, Project administration.

Data availability

Data will be made available on request.

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.

Acknowledgments

This research was in part supported by the Japan Society for the Promotion of Science KAKENHI grant number 23H03371.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jpdc.2025.105187

References

[1] F. Bajramovic, F. Mattern, N. Butko, J. Denzler, A Comparison of Nearest Neighbor Search Algorithms for Generic Object Recognition, Proceedings of the 8th Advanced Concepts for Intelligent Vision Systems (ACIVS) 4179 (2006) 1186–1197.
[2] Q. Huang, J. Feng, Q. Fang, W. Ng, W. Wang, Query-Aware Locality-Sensitive Hashing Scheme for Lp Norm, The VLDB Journal 26 (5) (2017) 683–708.
[3] K. Ahmadian, M. Gavrilova, D. Taniar, Multi-criteria Optimization in GIS: Continuous K-Nearest Neighbor Search in Mobile Navigation, Proceedings of the International Conference on Computational Science and Its Applications (ICCSA) 6016 (2010) 574–589.
[4] J. Pan, D. Manocha, GPU-Based Parallel Collision Detection for Fast Motion Planning, The International Journal of Robotics Research 31 (2) (2012) 187–200.
[5] M.N. A.H. Sha'abani, N. Fuad, N. Jamal, M.F. Ismail, P. Svm, Classification for EEG: A Review, Proceedings of the 5th International Conference on Electrical 632 (2019) 555–565. LNEE.
[6] V. Garcia, E. Debreuve, F. Nielsen, M. Barlaud, K-Nearest Neighbor Search: Fast GPU-Based Implementations and Application to High-Dimensional Feature Matching, in: Proceedings of the 17th IEEE International Conference on Image Processing, 2010, pp. 3757–3760.

[7] M.L. Yiu, N. Mamoulis, D. Papadias, Aggregate Nearest Neighbor Queries in Road Networks, IEEE Transactions on Knowledge and Data Engineering 17 (6) (2005) 820–833.
[8] H. Hu, D.L. Lee, J. Xu, Fast Nearest Neighbor Search on Road Networks, Proceedings of the 10th International Conference on Extending Database Technology (EDBT) 2896 (2006) 186–203.
[9] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C.T. Silva, Point Set Surfaces, in: Proceedings of the 14th IEEE Visualization Conference (VIS), 2001, pp. 21–28.
[10] M. Alexa, M. Gross, M. Pauly, H. Pfister, M. Stamminger, M. Zwicker, Point-Based Computer Graphics, in: Proceedings of the ACM SIGGRAPH Course Notes, 2004, p. 7.
[11] G. Qian, S. Sural, Y. Gu, S. Pramanik, Similarity Between Euclidean and Cosine Angle Distance for Nearest Neighbor Queries, in: Proceedings of the 19th ACM Symposium on Applied Computing (SAC), 2004, pp. 1232–1237.
[12] T. Kahveci, A.K. Singh, Efficient Index Structures for String Databases, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 351–360.
[13] M. Haklay, P. Weber, OpenStreetMap: User-Generated Street Maps, IEEE Pervasive Computing 7 (4) (2008) 12–18.
[14] H. Jégou, R. Tavenard, M. Douze, L. Amsaleg, Searching in One Billion Vectors: Re-rank with Source Coding, in: Proceedings of the 36th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2011, pp. 861–864.
[15] J. Wang, F. Ino, J. Ke, PRF: A Fast Parallel Relaxed Flooding Algorithm for Voronoi Diagram Generation on GPU, in: Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2023, pp. 713–723.
[16] F. Lettich, S. Orlando, C. Silvestri, Processing Streams of Spatial k-NN Queries and Position Updates on Manycore GPUs, in: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL), 2015, pp. 1–10.
[17] P. Velentzas, M. Vassilakopoulos, A. Corral, A Partitioning GPU-based Algorithm for Processing the k Nearest-Neighbor Query, in: Proceedings of the 12th International Conference on Management of Digital EcoSystems (MEDES), 2020, pp. 2–9.
[18] P. Xiang, Y. Yang, H. Zhou, Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation, in: Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 284–295.
[19] R.J. Barrientos, F. Millaguir, J.L. Sánchez, E. Arias, GPU-Based Exhaustive Algorithms Processing kNN Queries, Journal of the Supercomputing 73 (10) (2017) 4611–4634.
[20] W. Zhao, S. Tan, P. Li, SONG: Approximate Nearest Neighbor Search on GPU, in: Proceedings of the 36th International Conference on Data Engineering (ICDE), 2020, pp. 1033–1044.
[21] F. Groh, L. Ruppert, P. Wieschollek, H.P.A. Lensch, GGNN: Graph-Based GPU Nearest Neighbor Search, IEEE Transactions on Big Data 9 (1) (2023) 267–279.
[22] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, Y. Wang, CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs, in: Proceedings of the 40th International Conference on Data Engineering (ICDE), 2024, pp. 4236–4247.
[23] V. K, S. Khan, S. Singh, H.V. Simhadri, J. Vedurada, BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU, 2024. arXiv:2401.11324.
[24] Y. Zheng, R. Lu, S. Zhang, J. Shao, H. Zhu, Achieving Practical and Privacy-Preserving kNN Query over Encrypted Data, IEEE Transactions on Dependable and Secure Computing, 2024, pp. 1–13.
[25] H. Zhang, Y. Dong, J. Li, D. Xu, An Efficient Method for Time Series Similarity Search Using Binary Code Representation and Hamming Distance, Intelligent Data Analysis 25 (2021) 439–461.
[26] M. Nam, J. Kim, B. Nam, Parallel Tree Traversal for Nearest Neighbor Query on the GPU, in: Proceedings of the 45th International Conference on Parallel Processing (ICPP), 2016, pp. 113–122.
[27] A. Zimek, E. Schubert, H.-P. Kriegel, A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data, Statistical Analysis and Data Mining: The 5 (2012) 363–387.
[28] B. Gärtner, Fast and Robust Smallest Enclosing Balls, Proceedings of the 7th Annual European Symposium on Algorithms (ESA) 1643 (1999) 325–338.
[29] Y. Chen, L. Zhou, N. Bouguila, B. Zhong, F. Wu, Z. Lei, J. Du, H. Li, Semi-Convex Hull Tree: Fast Nearest Neighbor Queries for Large Scale Data on GPUs, in: Proceedings of the IEEE International Conference on Data Mining (ICDM), 2018, pp. 911–916.
[30] P. Velentzas, M. Vassilakopoulos, A. Corral, GPU-Based Algorithms for Processing the k Nearest-Neighbor Query on Disk-Resident Data, Proceedings of the 10th International Conference on Model and Data Engineering (MEDI) 12732 (2021) 264–278.
[31] F. Gieseke, J. Heinermann, C. Oancea, C. Igel, Proceedings of the 31st International Conference on Machine Learning (ICML), 2014. Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs.
[32] A.K. usek, W. Dzwinel, Multi-GPU k-Nearest Neighbor Search in the Context of Data Embedding, in: Parallel computing is everywhere, IOS Press, 2018, pp. 359–368.
[33] J. Johnson, M. Douze, H. Jégou, Billion-Scale Similarity Search with GPUs, IEEE Transactions on Big Data 7 (3) (2021) 535–547.
[34] J. Zhang, A. Naruse, X. Li, Y. Wang, Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2023, pp. 1–13.
[35] X. Wang, A Fast Exact K-Nearest Neighbors Algorithm for High Dimensional Search Using K-means Clustering and Triangle Inequality, in: Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN), 2011, pp. 1293–1299.
[36] Y. Pan, Z. Pan, Y. Wang, W. Wang, A New Fast Search Algorithm for Exact K-nearest Neighbors Based on Optimal Triangle-Inequality-Based Check Strategy, Knowledge-Based Systems 189 (2020) 105088.
[37] M. Muja, D.G. Lowe, Scalable Nearest Neighbor Algorithms for High Dimensional Data, IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (11) (2014) 2227–2240.
[38] Z. Deng, X. Zhu, D. Cheng, M. Zong, S. Zhang, Efficient KNN Classification Algorithm for Big Data, Neurocomputing 195 (2016) 143–148.
[39] J. Ritter, An Efficient Bounding Sphere, in: Graphics gems, USA, Academic Press Professional, Inc, 1990, pp. 301–303.
[40] E. Welzl, Smallest Enclosing Disks (Balls and Ellipsoids), in: H. Maurer (Ed.), New Results and New Trends in Computer Science, Berlin, Heidelberg, Springer, 1991, pp. 359–370.
[41] D. Arthur, S. Vassilvitskii, Means, The Advantages of Careful Seeding, in: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007, pp. 1027–1035.
[42] Nvidia, NVIDIA Turing Architecture Whitepaper, 2023. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.
[43] P. Virtanen, et al., SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nature Methods 17 (3) (2020) 261–272.
[44] G. Marsaglia, A. Zaman, The KISS Generator, Technical Report, Department of Statistics, University of Florida, 1993.
[45] R. Caruana, T. Joachims, L. Backstrom, KDD-Cup 2004: Results and Analysis, SIGKDD Explorations Newsletter 6 (2004) 95–108.
[46] L. Amsaleg, H. Jégou, Datasets for Approximate Nearest Neighbor Search, 2024. http://corpus-texmex.irisa.fr.
[47] A. Babenko, L. Victor, Efficient Indexing of Billion-Scale Datasets of Deep Descriptors, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2055–2063.
[48] Nvidia, Nsight Compute Documentation, 2023. https://docs.nvidia.com/nsight-compute/index.html.

Jue Wang received the BE degree in microelectronics science and engineering from Shanghai Jiao Tong University, Shanghai, China, in 2021, and the ME degree in information and computer sciences from Osaka University, Osaka, Japan, in 2024. He is currently working toward the PhD degree in information and computer sciences from Osaka University. His major research interests include high-performance computing and parallel programming.

Fumihiko Ino received the BE, ME, and PhD degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently a Professor with the Graduate School of Information Science and Technology, Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.