PARS: Optimizing High-Dimensional Vector Search with Dimensional Reduction and Ray Tracing Core

Yiling Ma[†], Mengbai Xiao[†,‡,*], Zhixiong Xiao[†], Yuan Yuan[†], Dongxiao Yu[†], Guanghui Zhang[†], Feng Li[†]
[†]Shandong University, China
[‡]Shandong Provincial Key Laboratory of Computing-Network Integration, China
ma10@mail.sdu.edu.cn, {xiaomb, xiaozx, yyuan, dxyu, gh.zhang, fli}@sdu.edu.cn

***Abstract*—Approximate nearest neighbor (ANN) search in high-dimensional spaces is a fundamental but challenging problem. GPU-based graph indexes have recently emerged as a promising solution. However, such methods have a large memory footprint and are memory-bandwidth bound when operating on high-dimensional vectors, which limits their scalability on commodity GPUs. In this paper, we propose PARS, a GPU-based ANN search method that improves both efficiency and scalability through dimensionality reduction. PARS projects data points into a low-dimensional space using principal component analysis (PCA), where distance relationships are sufficiently preserved for computation. We further design a sampling-based method to determine the minimum projection dimension required to meet a target recall. To improve the search performance, we leverage the emerging RT cores to identify high-quality entry points. Additionally, we introduce an optional GPU buffer that dynamically supplies full-precision vectors for each query batch, improving search accuracy under tight memory constraints. Experiments on a wide range of datasets demonstrate the effectiveness of PARS. At 90%–95% recall, PARS achieves up to 9.3× speedup over CAGRA, the current GPU-based state of the art, while reducing GPU memory usage by 23.1%–73.0%, enabling ANN search on 100M-scale datasets using a single NVIDIA L40 GPU.**

***Index Terms*—Approximate Nearest Neighbor Search, Dimension Reduction, Ray Tracing Core, GPU**

I. Introduction

With the rapid development of AI applications, such as large language models and multi-modal systems, massive vector representations are generated to encode semantic information. Efficient nearest neighbor (NN) search in high-dimensional spaces has become a fundamental building block for modern data-intensive applications like retrieval-augmented generation (RAG), vision-language alignment, and recommendation systems [1]–[3]. However, in high-dimensional space, the computational cost of exact NN search converges to that of a brute-force solution due to the curse of dimensionality. To address this, Approximate nearest neighbor (ANN) search has gained widespread attention as a more efficient alternative. ANN search aims to achieve high search throughput by tolerating accuracy loss that remains acceptable for practical applications. In recent years, both academia and industry have proposed a wide range of approaches toward ANN search [4], including graph-based ones [5]–[14], quantization-based ones [15]–[22], hash-based ones [23]–[26], and tree-based ones [27]–[29].

* Corresponding author.

Among these methods, the graph-based ones have demonstrated their superior performance, achieving both promising search throughput and accuracy. The graph-based methods construct a proximity graph where nodes represent high-dimensional points and edges connect neighboring points. Searching the graph for the nearest neighbors is conducted via a best-first search that prunes distant vectors. By iteratively expanding nodes that are closer to the query, the algorithm efficiently reaches the neighborhood of the query.

Enabled by the massive parallelism of GPUs, an increasing number of graph-based methods have begun to leverage GPUs for efficient ANN search [11], [13], [14], [30]. They integrate GPU architecture features, including execution model and memory hierarchy, into both the graph structure and search design, achieving superior performance. Despite the massive parallelism offered by GPUs, graph-based ANN search on GPUs is fundamentally memory-bound. During query processing, the execution time is dominated by fetching high-dimensional vectors from global memory, rather than arithmetic distance computations. Meanwhile, the limited memory of GPUs further restricts scalability, it either fails to contain the full-precision vectors in device memory or transfers vectors through PCIe channels at inferior rates. These challenges indicate that improving GPU-based ANN search requires not only faster computation, but also a careful reduction of memory traffic and memory footprint, without significantly sacrificing search accuracy.

Reducing vector dimensions could be an effective solution to the aforementioned challenges. First, lower-dimensional vectors require less data transfer from global memory during distance computations, directly alleviating the memory bandwidth bottleneck. Meanwhile, prior work has demonstrated that partial-dimension distance calculations can still yield reliable results [31], [32]. Second, reduced dimensions can significantly decrease the memory footprint of vectors, which constitute the dominant portion of GPU memory usage. For instance, in a typical NSW-built index graph for 100 million 128-dimensional vectors, 47.68 GB out of 59.60 GB are occupied by full-precision vectors, while the remaining space stores graph topology maintaining 32 neighbors per point.

In this paper, we propose PARS, a scalable graph-based ANN search method featuring dimension reduction on GPU. To reduce memory accesses and footprint, we employ Principal Component Analysis (PCA) to project full-precision

vectors into a low-dimensional space prior to graph search. This design decouples dimension reduction from graph search, enabling the GPU to perform PCA operations in batches for incoming queries. Furthermore, we introduce a sampling-based method to determine the minimum projection dimension that guarantees a target recall. Since the leading PCA dimensions capture a large fraction of the variance, they provide a compact spatial embedding that is well suited for efficient entry point selection. This observation naturally enables the use of GPU RT cores [33], which are highly optimized for spatial queries, to locate high-quality entry points with negligible overhead. These entry points enhance search efficiency by initializing the process in close proximity to the query. Finally, we devise an optional enhancement module called *twin-buffer* to bridge the accuracy gap under limited GPU memory. *Twin-buffer* does not alter the core search pipeline of PARS, but dynamically stores full-precision vectors of points within spatially adjacent clusters for the current query batch. This improves search precision without requiring the entire dataset to be stored in its original high-dimensional form.

Extensive experiments across diverse datasets show that, at 90%–95% recall, PARS achieves a 2.6–9.3× speedup over CAGRA, the current GPU-based state of the art, while reducing memory usage by 23.12%–73.01%. Additionally, under bounded error, the real-world datasets SIFT1M and GIST can be reduced to 29.7% and 7.6% of their original dimensionality, respectively. We further validate the effectiveness of using the top three PCA dimensions to locate entry points. On a graph with degree 32, the selected entry points are 18%–30% closer to the true nearest neighbors in Euclidean distance than randomly selected ones, and reduce graph traversal hops by 27.3%–47.3%, while accounting for only 0.05%–2.13% of the total search time. Our pluggable module, *twin-buffer*, significantly improves the memory–recall trade-off, raising recall@1000 on DEEP100M from 56% to 82%. These results demonstrate that PARS exhibits strong scalability to large datasets on a single GPU.

Our contributions in this work are as follows:

- We optimize graph-based ANN search on GPU through dimension reduction and propose a sampling-based method to identify the most appropriate projection dimension.
- We exploit the emerging RT cores to locate high-quality entry points for graph search. It effectively compensates the degraded accuracy introduced by dimension reduction.
- We design an optional *twin-buffer* that caches full-precision vectors near the query, enabling high-precision refinement without storing the full dataset in GPU memory.
- We conduct extensive experiments to evaluate our method. At the recall range of 90–95%, PARS achieves a 2.6–9.3x speedup over CAGRA, and the memory footprint of index is reduced by up to 73%.

II. Background

A. Approximate Nearest Neighbor Search

Approximate nearest neighbors (ANN) search is the problem of finding data points that are close to the query point. Formally, given a dataset $X = \{x_1, x_2, \ldots, x_N\} \subset \mathbb{R}^D$ and a distance metric $\text{dist}(\cdot, \cdot)$, for a query $q \in \mathbb{R}^D$, the ANN search algorithm returns a point $x' \in X$ such that

$$\text{dist}(q, x') \leq (1 + \epsilon) \cdot \text{dist}(q, x^*), \tag{1}$$

where x^* is the true nearest neighbor and $\epsilon > 0$ controls the approximation error.

Existing ANN methods seek to strike a critical trade-off between maintaining high search accuracy and achieving fast query performance. This trade-off is typically characterized by two metrics: recall and queries per second (QPS). The recall is defined as $\frac{|A \cap G|}{|G|}$, where A represents the result set returned by an ANN search algorithm, and G represents the ground-truth k-NN set. The QPS measures the number of queries that can be processed per second by the ANN system, reflecting its query throughput.

B. GPU-based ANN Search

ANN search on GPUs has attracted increasing attention in academia [11], [13], [14], [20], [30]. As the high computational throughput of GPUs significantly improves search efficiency, the pioneering work, Faiss [15], adopts inverted file index (IVF) based strategies, while most of the following studies [11], [13], [14], [30] explore graph indexes due to their higher accuracy. The search on a graph index starts from a set of entry points and iteratively explores neighboring nodes using a best-first strategy until convergence. To utilize the massive parallelism of GPU architecture, a parameter called *search width*, defined as w, is introduced [14]. The search simultaneously evaluates w unvisited nodes from the candidate list and adds their neighbors to the list if they meet the criteria.

To further optimize the GPU-based search, several optimization techniques could be employed [14]. For top-m selection, bitonic sort is well-suited to GPU architecture. Additionally, the most significant bit (MSB) of a node index could be used to flag parent-child relationships, avoiding extra hash table lookups and improving performance without sacrificing correctness.

C. Ray Tracing Core

RT cores are shipped with NVIDIA GPUs for accelerating ray tracing operations in graphics rendering [34]. They are specialized hardware units designed to accelerate two key tasks: traversing bounding volume hierarchies (BVHs) and testing ray-triangle intersections.

BVH traversal. The ray tracing pipeline employs BVHs composed of axis-aligned bounding boxes (AABBs) to index geometric primitives. The left side of Figure 1 illustrates a 2-dimensional (2D) example with four circles in a scene. The innermost AABBs (3, 4, and 5) enclose one or more geometric primitives and are recursively enclosed by the outer AABBs

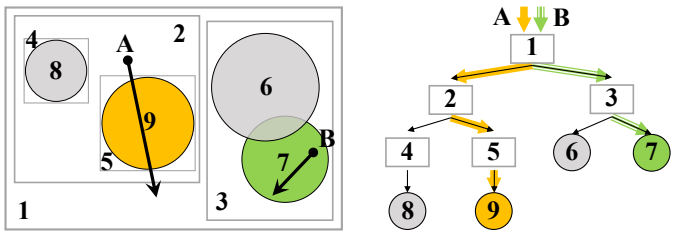


Fig. 1: An example of BVH traversal. The left shows a 2-dimensional (2D) scene with four circle primitives, and the right shows the corresponding BVH. The traversal paths taken by rays A and B are marked with yellow and green, respectively.

(1 and 2). Such a hierarchical organization allows BVH to efficiently prune unnecessary branches. A ray begins traversal from the root node of the BVH. If it intersects the bounding box of the current node, the traversal continues to its children; otherwise, the branch is pruned. When the traversal reaches a leaf node, intersection tests are performed between the ray and the enclosed primitives. The process terminates after all potentially intersecting nodes have been visited. The right side of Figure 1 highlights the traversal path taken by the ray using bold lines.

Ray tracing pipeline. To enable high-performance ray tracing applications on RT cores, NVIDIA provides the OptiX framework [35], which offers high-level APIs for BVH construction, BVH traversal, intersection testing, and programmable shaders. An OptiX programming model is shown in Figure 2. The OptiX execution pipeline begins with creating a context to manage GPU resources, followed by setting up a pipeline and binding program modules. A BVH is constructed using the `optixAccelBuild` function by providing geometric primitives such as triangles, curves, or AABBs. Once configured, the ray tracing process launches ray generation in function `optixLaunch`, then performs BVH traversal and intersection tests by `optixTrace`, and invokes the corresponding custom shaders. Each ray is mapped to a separate CUDA thread, and the shaders are executed on CUDA cores. The BVH traversal process and the ray-AABB intersection tests at each node are efficiently accelerated by RT cores.

RT cores for NN search. As RT cores have the capacity of efficiently detecting intersections between rays and geometric primitives, the NN search problem in 3-dimensional (3D) space can be mapped onto this accelerator. RTNN [33] is the first to formulate NN search in 3D space as a ray tracing problem, where queries and data points are transformed to rays and geometric primitives, respectively. RT-KNNS [36] incrementally adjusts the radius of each query during the search process, alleviating the limitations of using a fixed radius. JUNO [37] accelerates Inverted File with Product Quantization (IVFPQ) lookup by reformulating product quantization (PQ) distance computation as a ray tracing problem, encoding PQ codewords as spatial primitives to leverage RT cores for efficient distance estimation.

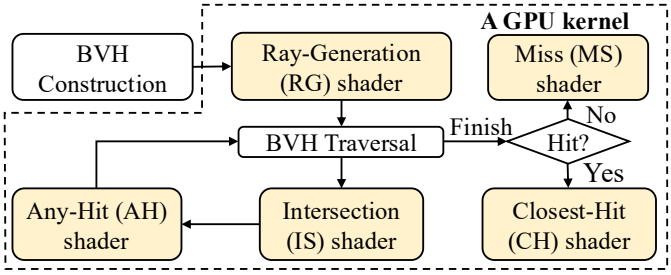


Fig. 2: The execution pipeline of OptiX. The steps marked with yellow are programmable. The RG shader emits rays, then each ray simultaneously executes BVH traversal and intersection tests. At leaf nodes, the IS shader tests for primitive intersections. If a hit is found, the AH shader records it before traversal continues. Once traversal completes, the MS or CH shader is invoked.

III. Design

In this section, we introduce the design of PARS, a graph-based ANN search method on GPU that features dimension reduction and RT-based entry point selection. We project the original vectors into d dimensions via PCA, and perform graph search on these reduced representations. To mitigate the resulting accuracy loss, the emerging RT cores are employed to locate high-quality entry points. For scenarios with high recall requirements, PARS further provides an optional GPU buffer, termed *twin-buffer*, to cache full-precision vectors near queries. This buffer enables the search engine to perform exact distance computation surrounding queries, thereby achieving higher recalls. The detailed workflow is illustrated in Figure 3.

A. Hybrid Index Structure

The index in PARS can be viewed as a two-level structure. At the low level, PARS leverages existing graph indexes while operating on dimension-reduced data points, enabling broad compatibility with arbitrary graph-based indexes. At the high level, we construct a BVH over the first three dimensions of the dimension-reduced vectors to identify suitable entry points, which are then used to initiate graph search.

*1) **Graph Index with Dimension Reduction**:* Dimension reduction accelerates the graph-based ANN search by reducing the number of dimensions involved in the distance computation [31], [32]. ADsampling is one such approach, which first applies a transformation to the dataset before search and then incrementally computes the distance between a query and a data point over several dimensions at a time during the search. The distance computation terminates once the scaled partial distance exceeds the current k-nearest-neighbor threshold. However, this inherently sequential computation does not fit the SIMT execution model of modern GPUs.

We propose to decouple dimension reduction from graph search. Specifically, we reduce the dimensionality of data points in advance from D to d ($d < D$) via linear transformation, and then load the resulting d-dimensional vectors onto the GPU for coarse distance computation during search.

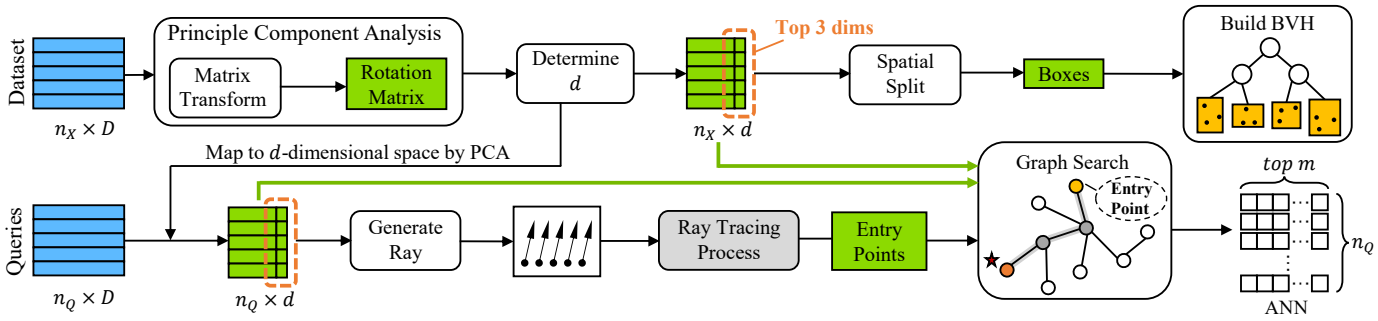


Fig. 3: Framework overview. Blue and green denote data stored on the CPU and GPU, respectively. Gray blocks indicate operations executed on the RT core. PCA is applied to reduce the data to d dimensions for distance computation. The top 3 dimensions are used for box partitioning and BVH construction. Query rays are generated from top-3 dimensions, traced to intersected boxes, whose points serve as entry nodes for graph search, yielding m approximate nearest neighbors.

Compared to the ADsampling, this method better exploits GPU parallelism for dimension reduction by eliminating the sequential dimension-by-dimension reduction process. Additionally, by performing dimension reduction in advance, we avoid computing dimensions that contribute little to nearest neighbor search, further improving GPU efficiency.

To make the coarse distance computation more accurate, we aim to preserve as much information from the original D-dimensional space as possible in the reduced d-dimensional representation. In other words, the d-dimensional vector is constructed to capture the maximum possible variance of the original D dimensions. This goal aligns with PCA, which seeks a low-dimensional subspace that maximizes the variance of the projected data. We first compute the eigenvectors of the covariance matrix of the dataset X and select the top d eigenvectors to form the transformation matrix $W \in \mathbb{R}^{D\times d}$. Using W, each data point $x_i \in \mathbb{R}^D$ is projected as $y_i = W^\top x_i$, yielding a d-dimensional vector $y_i \in \mathbb{R}^d$. All of these steps are performed before launching the search.

Selection of d: The dimensionality d is a key factor in balancing accuracy and efficiency. A smaller d results in reduced precision in distance computation, whereas a larger d increases computational cost. We hope to find the smallest d that meets the accuracy requirement.

We design a sampling-based iterative method to determine d. First, we apply a PCA transformation to both the data set and the training set, which maps the original D-dimensional vectors to a new D-dimensional representation whose dimensions are ordered by decreasing variance. Then, we retain the first d dimensions of the training set to perform distance computation against the dataset. For the i-th query in the training set, we obtain a corresponding recall value r_i^d based on the resulting distance matrix. Finally, we estimate the average recall of the training set as:

$$\hat{R}_d = \frac{1}{t}\sum_{i=1}^{t} r_i^d. \tag{2}$$

In this way, we can compute the recall estimates for increasing values of d at each iteration and return the smallest d that satisfies $\hat{R}_d \geq \tau$, where τ denotes the target recall. However, since $\hat{R}_d$ is calculated from samples, it is subject to estimation error. To ensure the reliability of the training results on real queries, we need to choose a theoretically guaranteed minimum training set size t. To account for this, we introduce a tolerance parameter ϵ, and adopt a stricter criterion: d is acceptable only when $\hat{R}_d \geq \tau + \epsilon$. According to Hoeffding's inequality [38], for t independent samples we have

$$\Pr\left[|\hat{R}_d - R_d| \geq \epsilon\right] \leq 2\exp(-2t\epsilon^2). \tag{3}$$

Let the desired confidence level be α, and the allowable failure probability be $\delta = 1-\alpha$. To ensure that the estimation error is within tolerance with probability at least α, we require:

$$2\exp(-2t\epsilon^2) \leq \delta \quad \Rightarrow \quad t \geq \frac{1}{2\epsilon^2}\ln\left(\frac{2}{\delta}\right). \tag{4}$$

Therefore, given a confidence level α and a tolerance ϵ, we have a minimum t, ensuring that the average recall estimated from the samples can be reliably generalized to the entire dataset.

It is expected that the average recall estimate monotonically increases as d increases if the latter dimensions are not dominated by noise. Therefore, we could evaluate the recalls in subspaces with increasing dimensions. Once the recall satisfies the target τ, the current dimensionality is returned as d. Algorithm 1 illustrates how we find the minimum d that meets the target recall τ.

Compatibility with graph-based indexes: Our method is compatible with any arbitrary graph-based index since we only reduce the dimensions of data so that they can be loaded to GPU for coarse distance computation.

2) ***Entry Point Selection Using BVH****:* Although we aim to preserve distance fidelity after projection, the search accuracy is inevitably degraded due to dimension reduction. To compensate for this, we propose selecting entry points near queries to search the graph index, thereby increasing the likelihood of reaching the true nearest neighbors.

After PCA transformation, data points have been concentrated to d dimensions, and more importantly, the first three

Algorithm 1: Determine the Minimum Acceptable d

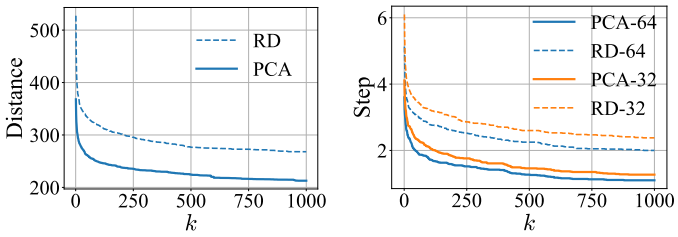
Input : Dataset $X \in \mathbb{R}^{n_X \times D}$, training set $A \in \mathbb{R}^{n_A \times D}$, target recall threshold τ, tolerance ϵ, confidence level α, recall parameters k, m

Output: Smallest d such that $\hat{R}_d \geq \tau + \epsilon$

$\delta \leftarrow 1 - \alpha$;
$t \leftarrow \lceil \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right) \rceil$;
$X_{\text{pca}} \leftarrow \text{PCA}(X)$; ▷ shape: $n_X \times D$
$A_{\text{pca}} \leftarrow \text{PCA}(A)$; ▷ shape: $n_A \times D$
Randomly sampled query set $T \subset A_{\text{pca}}$, $|T| = t$;
Initialize Dis as a $t \times n_X$ zero matrix;
for $d = 1$ **to** D **do**
  foreach $i \in T$, $j \in X_{pca}$ **do**
    $Dis_{ij} \leftarrow Dis_{ij} + (T[i][d] - X_{\text{pca}}[j][d])^2$;
  foreach $i \in T$ **do**
    $recall_i \leftarrow \text{recall}(Dis_{i*})$;
  $\hat{R}_d \leftarrow \frac{1}{t} \sum_{i=1}^{t} recall_i$;
  if $\hat{R}_d \geq \tau + \epsilon$ **then**
    return d;
return D;

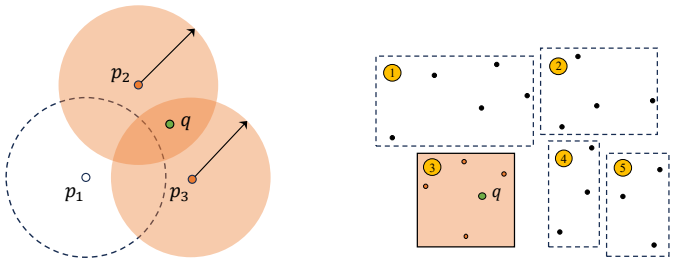
dimensions are expected to capture a significant portion of variance. As a result, we can search for nearest neighbors using these three-dimensional vectors, which can be effectively accelerated by RT cores due to their hardware characteristics. While these results may not be the exact nearest neighbors in the original D-dimensional space, they still serve as effective entry points for initiating the search in the graph index. Preliminary experiments on SIFT1M demonstrate the effectiveness of using the first three dimensions after PCA transformation for entry point selection. As shown in Figure 4, compared to random sampling, points selected via PCA proximity are 18%–30% closer to the true nearest neighbors in the original space. Furthermore, starting search from these points reduces the hops to the nearest neighbor by 26%–47% across different graph degrees (32 and 64). A similar phenomenon is observed on other datasets as well. These results confirm that the first three dimensions provide high-quality entry points that significantly accelerate the graph traversal.

Existing work leverages RT cores for efficient low-dimensional nearest neighbor search by formulating it as a geometric ray-tracing task [33]. In this approach, data points are modeled as spheres organized within a BVH, as shown in Figure 5a, and queries act as short rays to identify candidate neighbors through intersection tests. However, this per-point primitive strategy leads to a substantial and increasingly costly memory footprint as the dataset grows (e.g., 3.6–4.8 GB for 100 million points). To improve memory-scaling efficiency, we propose a solution that constructs a BVH representing spatial regions rather than individual points, significantly reducing the index size.



(a) (b)

Fig. 4: Comparison of entry point quality between random selection (RD) and PCA using the top 3 dimensions. Each method selects k candidates. (a) shows the Euclidean distance from the closest candidate to the true nearest neighbor (top-1), and (b) shows the minimum hops to it.



(a) sphere primitive (b) box primitive

Fig. 5: Two different methods for modeling graphical objects.

We choose AABBs as primitives for building the BVH since we need to split and organize the three-dimensional spatial regions, as shown in Figure 5b. These AABBs enclose at most a fixed number of data points but vary in size depending on the data distribution. To determine these AABBs, we recursively partition the entire space, whose boundaries are defined by $\{y_1^{\min}, y_2^{\min}, y_3^{\min}, y_1^{\max}, y_2^{\max}, y_3^{\max}\}$, in a manner similar to the KD-tree construction. We subdivide the current box at the midpoint of its longest axis and repeat this process until all boxes contain no more than α points. The resulting boxes are inherently non-overlapping AABBs, which are then used to build the BVH.

When a ray representing a query is issued, it is expected to intersect the closest AABB by traversing the BVH, and all points within that AABB are selected as the entry points. Since these entry points populate the candidate list that is used when searching the graph-based index, we always set α to match the size of the candidate list. When deploying our method, we observe that rays occasionally issued at the boundaries of AABBs intersect no boxes at all. To address this, we expand our AABBs by a small margin.

*3) **Structure of twin-buffer**:* The *twin-buffer* is designed to store the exact vectors that are close to the queries in the current batch. This design is based on the observation that points far away from a query can be effectively filtered out using partial computations in a small number of dimensions, while points close to the query require more precise vector comparisons to determine whether they are true nearest neigh-

Algorithm 2: ANN search in PARS

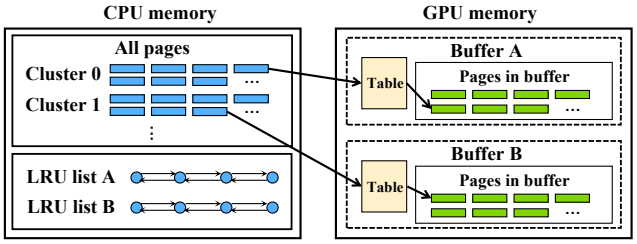
Input : Query vector $q \in \mathbb{R}^D$, PCA matrix $W \in \mathbb{R}^{D \times d}$, graph index G, BVH, result set size m

Output: Top-m nearest neighbors R

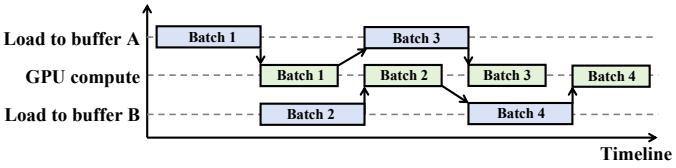
1 $q_d \leftarrow W^\top q$; ▷ $q_d \in \mathbb{R}^d$
2 $E \leftarrow$ FINDENTRYPOINTS(q_d, BVH);
3 $R \leftarrow$ SEARCH(q_d, E, G, m); ▷ R is result set
4 **return** R;
5 **Function** FINDENTRYPOINTS(q_d, *BVH*)
6 $o \leftarrow q_d[0{:}3]$; ▷ Using the first 3 dims as ray origin
7 $\vec{r} \leftarrow (o, 10^{-6})$; ▷ Ray from o with a short length
8 $AABB_{hit} \leftarrow$ BVHTRAVERSAL(BVH, $\vec{r}$);
9 $E \leftarrow$ RANDOMSELECT($AABB_{hit}$, L);
10 **return** E;
11 **Function** SEARCH(q_d, E, G, m)
12 $\forall v \in E$, $v.dis \leftarrow \text{dist}(v, q_d)$;
13 $C \leftarrow E$; $R \leftarrow \emptyset$; ▷ C is candidate list
14 $H_v \leftarrow E$; ▷ H_v is visited list
15 $H_e \leftarrow \emptyset$; ▷ H_e is extended list
16 MERGE(C, R); ▷ Merge top-m points to R
17 **while** $R \setminus H_e \neq \emptyset$ **do**
18 $P \leftarrow$ Top-w points in $R \setminus H_e$ by distance to q_d;
19 $H_e \leftarrow H_e \cup P$;
20 $C \leftarrow \bigcup_{p \in P} \mathcal{N}(p)$;
21 **foreach** $v \in C \setminus H_v$ **do**
22 $v.dis \leftarrow \text{dist}(v, q_d)$;
23 $H_v \leftarrow H_v \cup \{v\}$;
24 MERGE(C, R);
25 **return** R;

bors accurately [31], [32]. As shown in Figure 6a, all full-precision vectors of the dataset are maintained on the CPU and partitioned into multiple clusters using k-means. Within each cluster, vectors are organized into fixed-size pages, which serve as the smallest operable unit. A doubly linked list is used to track the residency status of pages in *twin-buffer*, and page replacement follows a least recently used (LRU) policy.

The *twin-buffer* acts as a cache between CPU and GPU. For a batch of queries, pages from clusters near the queries are loaded into *twin-buffer* in advance to avoid direct access to CPU memory. When switching to a new batch, pages in clusters relevant to the new queries are fetched from the CPU, while the least recently accessed pages are evicted from *twin-buffer* according to the LRU policy. To overlap the computation of the current batch with data transfer for the next batch, *twin-buffer* is divided into two separate buffers, each assigned to an individual stream. This design allows the GPU to perform the search using one buffer while the other is being populated with new pages, effectively mitigating the latency of transferring vectors from the CPU to the GPU. An



(a) Structure of *twin-buffer*.



(b) Pipeline of *twin-buffer*.

Fig. 6: Overview of *twin-buffer*. (a) shows the structure of *twin-buffer*. It consists of two buffers, A and B, each managed by an LRU list on the CPU for page replacement. On the GPU, a table maps global page IDs to buffer page IDs. (b) illustrates the processing pipeline. A query batch is executed only after its data has been loaded into the buffer. The execution of one batch is overlapped with the data loading for another batch.

example of this asynchronous operation is shown in Figure 6b. It is worth noting that whether the PCIe transfer and the GPU computation can be overlapped is determined by their time ratio, and increasing the buffer number does not help. If one of them requires more time than the other, the pipeline is throttled by the one with higher overhead. We observe that the computation requires more time as batch size is small since the GPU parallelism is not well exploited. Thus, the batch size should be carefully selected.

In addition, *twin-buffer* maintains a mapping from points to their corresponding pages, along with the offsets within each page. This can help to efficiently determine whether a point resides in *twin-buffer* during the search process.

B. Search in PARS

The search algorithm is shown in Algorithm 2. When a query arrives, it is first projected into the d-dimensional space using the same PCA transformation matrix as the dataset (Line 1). The subsequent search has two stages: (1) Utilizing BVH to locate nearby entry points to initialize the candidate list (Line 2), and (2) a graph-based search is performed in the reduced space (Line 3).

In the first stage, we convert the projected queries into rays originating at their coordinates in the first three dimensions, with the ray lengths set to be extremely short (Line 6-7). We then randomly select L points from the hit AABBs after BVH traversal to serve as entry points (Line 8-9). In the second stage, we follow the standard best-first strategy to search on the graph index with a candidate list initialized by the selected entry points (Line 11-25). It is worth noting that all distance computations are performed in the d-dimensional space, and only the projected vectors are stored on the GPU.

To further reduce processing latency, the PCA transformation of the query set is accelerated using Tensor Cores.

Refining with *twin-buffer*. To improve the utilization of *twin-buffer*, queries within the same batch should be spatially close. To construct such batches, we first find the nearest clusters for each query, and then sort the queries using their cluster IDs as keys. Since the original cluster IDs do not capture spatial relationships, we remap these IDs using the Hilbert curve.[1] Finally, we partition the sorted queries into batches according to a certain batch size. This design increases the spatial coherence within each query batch.

During graph search, the distance between a query q and each candidate point p is evaluated. When *twin-buffer* is enabled, the algorithm first checks if the full-precision vector of point p is available in *twin-buffer*. If a hit occurs, the vector is retrieved from *twin-buffer* to compute the exact distance. Otherwise, the distance is approximated using the dimension-reduced representation.

As a result, two types of distances coexist during the search process. Some are computed exactly from full-precision vectors, while others are estimated from low-precision vectors. To ensure a fair comparison and consistent ranking, the approximate distances must be calibrated. We adopt a linear model to correct the approximate distance, which is defined as

$$\hat{d} \approx \alpha \times d' + \beta, \tag{5}$$

where d' denotes the approximate distance, $\hat{d}$ is the calibrated distance, and α and β are learnable parameters. The values of α and β can be learned from a training set. Let d_i^* be the exact distance and d_i' be the corresponding approximate distance for the i-th training sample. The optimization objective is to minimize the squared error:

$$\min_{\alpha,\beta} \sum_{i=1}^{N} \left(d_i^* - (\alpha \times d_i' + \beta)\right)^2. \tag{6}$$

This problem corresponds to a standard least-squares regression, whose closed-form solution can be obtained by setting the partial derivatives with respect to α and β to zero. The resulting parameters are then used during search to refine approximate distances, enabling consistent comparison with exact distances.

IV. Evaluation

A. Experimental Settings

Environment. All experiments are conducted on a server running Ubuntu 22.04.5 LTS equipped with NVIDIA L40 GPUs (48 GB memory). The ray tracing component of PARS utilizes NVIDIA OptiX 7.6.

Dataset. We evaluate PARS on a set of widely used real-world datasets, including SIFT [15], DEEP [20], GIST [15], Crawl[2], and COCO-I2I[3]. These datasets exhibit diverse distributions, as quantified by the cumulative explained variance ratio of the top 10% principal components ($V_{10\%}$) reported in Table I. For each dataset, we perform dimensionality reduction to obtain a projected dimension d and construct a graph with a specified degree deg. The degree is selected from commonly used values, i.e., 16, 32, and 64. We choose the smallest value that preserves graph connectivity in practice. During the search, the result set size is set to m, and the candidate list size is configured as $w \times deg$. The detailed settings are summarized in Table I. Notably, though all of our evaluations are carried out on Euclidean distances, the experimental results also hold for cosine similarity. The cosine similarity θ satisfies $dist^2 = 2 - 2\cos\theta$ when the Euclidean distance is normalized to $dist \in [0, 1]$.

TABLE I: Datasets

Dataset	D	d	Size	Degree	m	$V_{10\%}$
SIFT1M	128	64	1M	32	128	66.8%
SIFT10M	128	64	10M	32	128	57.5%
SIFT100M	128	48	100M	64	1024	57.4%
DEEP1M	96	64	1M	32	128	36.7%
DEEP10M	96	64	10M	64	128	36.7%
DEEP100M	96	48	100M	64	1024	26.9%
GIST	960	128	1M	64	256	82.7%
Crawl	300	150	1,989,995	64	256	28.9%
COCO-I2I	512	192	113,287	16	64	62.2%

Baselines. To evaluate the effectiveness of PARS, we select baselines based on dataset scale and hardware feasibility. For the 1M and 10M datasets, we compare PARS with three state-of-the-art GPU-based graph ANN search methods: CAGRA [14], GGNN [13], and GANNS [11]. These methods require the entire graph structure to reside in GPU memory during querying. For the 100M dataset, such GPU-based graph methods are inapplicable because the total size of the graph index and necessary data exceeds the memory capacity of the L40 GPU. Consequently, we employ three scalable baselines: IVFPQ [15], DiskANN [9], and DDC [32]. IVFPQ is a quantization-based GPU method that maintains low memory consumption while producing high-quality candidate sets. DiskANN is a graph-based method that we run in in-memory mode. DDC is a CPU-based method designed to improve distance computation efficiency through PCA transformation and linear distance correction.

Implementation details. Regarding the baseline configurations, we ensure fair comparison on the datasets by having PARS, CAGRA, and GANNS share the same single-layer graph index constructed by CAGRA, while GGNN uses its official graph construction algorithm. For large-scale experiments, PARS, DiskANN, and DDC all use the same graph index constructed by DiskANN. DiskANN uses 64 threads in its default multithreaded setup, whereas DDC is executed in single-thread mode as it currently lacks multi-threading support. Unless otherwise specified, all experiments are conducted in a single-batch setting with 10,000 queries per batch.

To analyze the impact of different components in our design, we implement and evaluate three distinct variants.

[1] https://github.com/galtay/hilbertcurve

[2] https://commoncrawl.org

[3] https://cocodataset.org

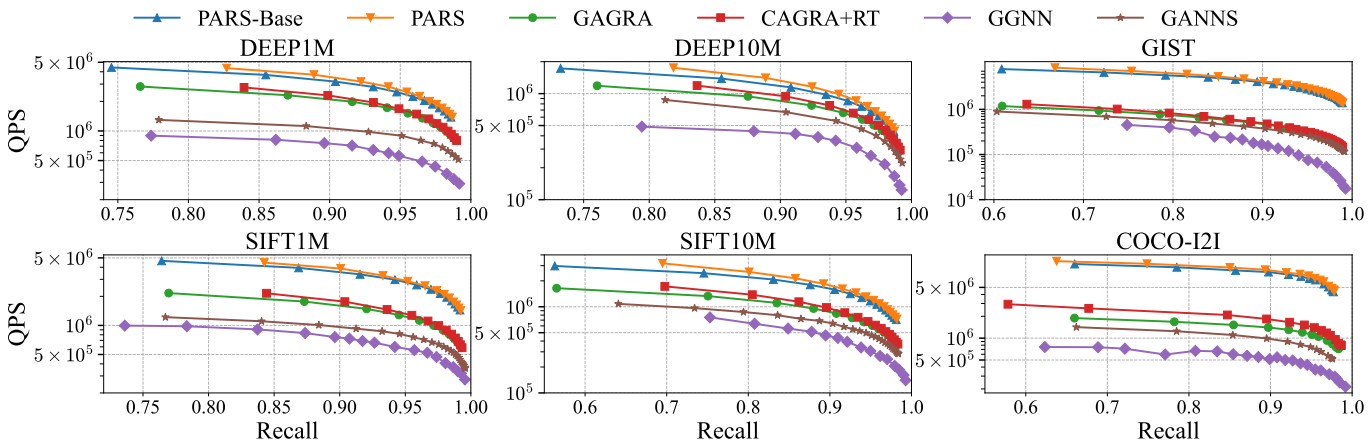


Fig. 7: Comparison of recall and QPS of various methods.

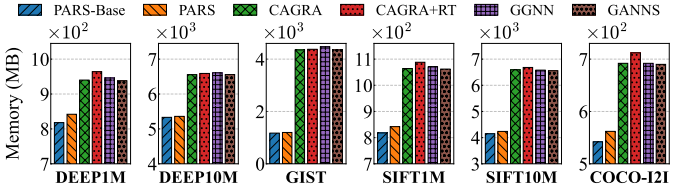


Fig. 8: Memory usage of various methods.

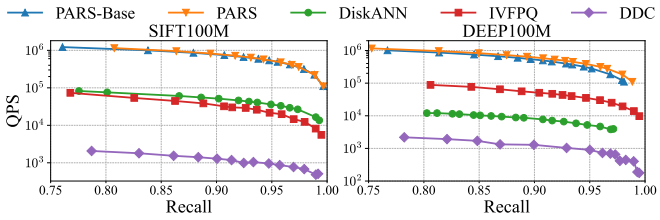


Fig. 9: Experimental Results on SIFT100M and DEEP100M.

PARS represents our full method, featuring projected-distance graph search coupled with ray tracing-assisted entry points. **PARS-Base** serves as a baseline variant that performs graph search using only projected distances, excluding the ray tracing component. Finally, **CAGRA+RT** is a hybrid variant that incorporates our ray tracing-based entry point selection into the standard CAGRA framework. The index construction is performed once per dataset and is independent of the online query workload, so its optimization is considered orthogonal and beyond the scope of this work.

Metrics. We evaluate the performance of PARS and the baseline methods in terms of throughput, recall, and memory footprint. Throughput is measured by QPS. Recall is evaluated using recall@$k(m)$, where m (the length of the result list) is predefined for each dataset as shown in Table I. Formally, for a set of n_q queries, let G_i denote the set of ground-truth top-k nearest neighbors for the i-th query, and A_i denote the set of m retrieved neighbors. Then, recall@$k(m)$ is computed as $\frac{1}{n_q}\sum_{i=1}^{n_q}\frac{|A_i \cap G_i|}{|G_i|}$. For GPU-based methods, the memory footprint is measured as the total GPU memory consumption, including both the index and the auxiliary data structures used during search.

B. Experimental Results

*1) **Search Performance**:* As shown in Figure 7, PARS and PARS-Base consistently outperform all baselines in terms of both recall and QPS across different datasets and graph sparsity levels. At around 98% recall, PARS is 1.99×, 4.34×, and 2.43× faster than CAGRA, GGNN, and GANNS on SIFT10M, respectively. On DEEP10M, where variance is more evenly distributed, PARS still achieves speedups of 1.37×, 3.25×, and 1.77×, demonstrating robust advantages across different data distributions.

Meanwhile, RT-based entry point selection brings noticeable accuracy gains with negligible overhead, even though the top 3 PCA dimensions preserve only 45%, 16%, 34%, and 8% of the variance on SIFT, DEEP, GIST, and COCO-I2I, respectively. We validate the effect of RT-based entry points by comparing CAGRA with and without RT-based entry selection. CAGRA+RT accelerates CAGRA by 1.06–1.10× on SIFT and DEEP, by 1.02× on GIST, and by up to 1.19× on COCO-I2I. These results show that even when very little variance is retained, using the top three dimensions for entry point selection remains effective, and under sparse graph settings, search quality is more sensitive to the choice of entry points.

Figure 8 shows that PARS significantly reduces GPU memory consumption by storing only a subset of feature dimensions. Compared with CAGRA, PARS achieves 10.43%–72.46% memory savings, and this advantage grows with dataset size and dimensions, demonstrating its suitability for large-scale and high-dimensional ANN search.

*2) **Results on larger datasets**:* As shown in Figure 9, we evaluate PARS on the SIFT100M and DEEP100M datasets and compare it with DiskANN, IVFPQ, and DDC in terms of recall-QPS trade-offs. At comparable recall levels, PARS consistently achieves the highest throughput among all meth-

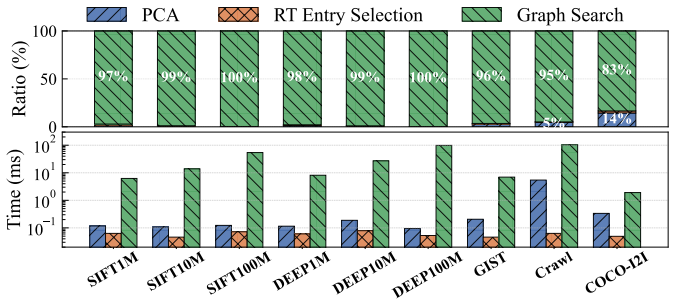


Fig. 10: Runtime breakdown of PARS.

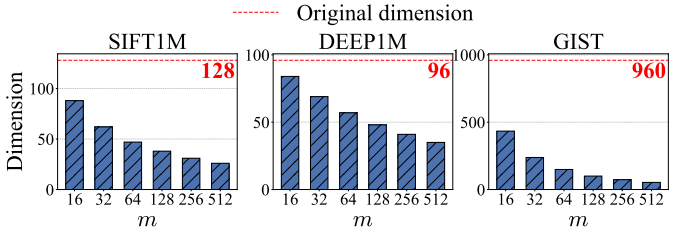


Fig. 11: Minimum dimensions after PCA required to achieve a target recall $\tau = 0.97$ under different m. A lower value indicates more effective dimensionality reduction.

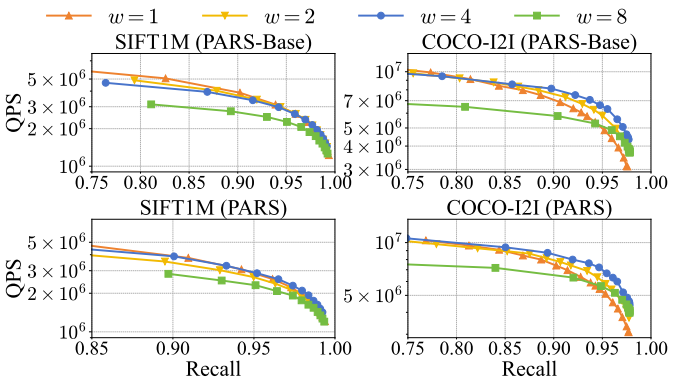


Fig. 12: Varying the search width.

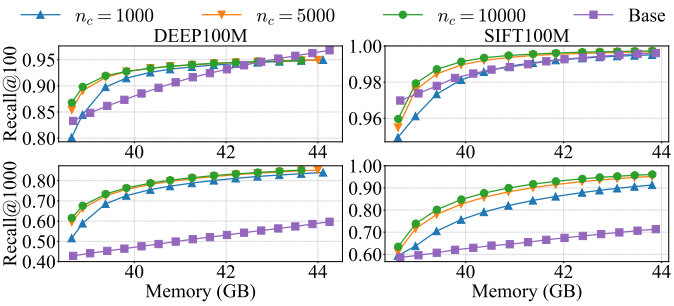


Fig. 13: Memory and recall with and without *twin-buffer*. n_c indicates cluster counts used for partitioning, 'Base' is the baseline without *twin-buffer*. The x-axis denotes the total memory budget determined by the combination of the static index and variable buffer sizes.

ods. Specifically, at around 99% recall on SIFT100M, PARS is 11.17× faster than DiskANN, 23.07× faster than IVFPQ, and 344.51× faster than DDC; similar speedups of 27.86×, 7.40×, and 273.97× are observed on DEEP100M. These results confirm that PARS scales effectively to 100M-scale datasets while maintaining both high recall and high throughput.

*3) **Runtime Breakdown of PARS**:* The query latency of PARS consists of three stages: projecting queries using PCA, selecting entry points with the RT core, and performing ANN search on the graph index. Figure 10 reports the detailed time breakdown of these stages at 99% recall across different datasets. We observe that, on most datasets, both PCA projection and RT-based entry point selection take less than 1ms, while nearly 99% of the total time is spent on graph search. This indicates that integrating RT-based entry point selection into other graph-based methods introduces negligible overhead but offers a highly cost-effective performance improvement.

*4) **Dimensionality Reduction Validation**:* We adopt the incremental search algorithm described in Section III-A1 to determine d. We set the target recall to $\tau = 0.97$, the confidence level to $\alpha = 95\%$, and the error tolerance to $\epsilon = 0.02$. According to Hoeffding's inequality, at least 4,616 samples are required to estimate recall with the desired reliability. With $k = 10$, we evaluate different results for list size m. Figure 11 reports the results on SIFT1M, DEEP1M, and GIST, showing that PCA can substantially reduce dimensionality under theoretical guarantees on recall. The achievable reduction varies across datasets because their variance distributions are different. The above analysis is based on an exact brute-force search in the reduced d-dimensional space, whereas a graph index introduces additional approximation errors in practice. Therefore, the reduced dimension d needs to be further adjusted to balance accuracy and memory efficiency. Based on this trade-off, we choose the final values of d reported in Table I for all subsequent experiments.

*5) **Varying Search Width**:* The search width w determines the number of nodes expanded simultaneously during each exploration step. Increasing w ($w > 1$) facilitates escaping local optima. We evaluate PARS and PARS-Base with $w \in \{1, 2, 4, 8\}$. As shown in Figure 12, increasing w improves recall but decreases QPS. We observe that when w is set to 4, it offers the best balance. Thus, we select this value as the default search width.

*6) **Memory-Recall Trade-off with twin-buffer**:* We evaluate the effectiveness of *twin-buffer* in improving ANN search accuracy under different GPU memory usage, as shown in Figure 13. The batch size is set to 5. For PARS with *twin-buffer*, the projection dimension is set to 32, while for PARS without *twin-buffer*, we vary the projection dimension from 37 to 52 to achieve different memory usages. We fix the result list size returned by PARS to 1024 and report accuracy on both low-recall (recall@100) and high-recall (recall@1000) settings. It can be observed that, without *twin-buffer*, the method already achieves good recall under low-recall setting,

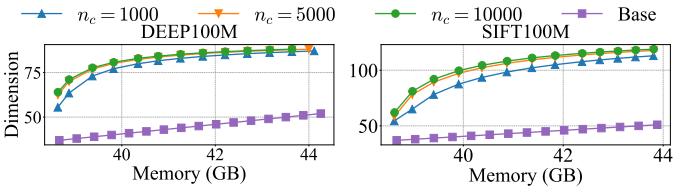


Fig. 14: Average dimensions per distance calculation.

but becomes uncompetitive when high-recall setting is set. This is because distances in low-dimensional space are less accurate, requiring a longer result list to capture a sufficient number of true neighbors. By selectively computing exact distances for critical candidates, *twin-buffer* effectively addresses this issue. Under the same GPU memory budget, *twin-buffer* consistently delivers higher recall than the method without *twin-buffer*. For example, at 43 GB of memory, recall@1000 increases from 56% without *twin-buffer* to 85% with *twin-buffer* on DEEP100M, showing that *twin-buffer* can achieve higher search accuracy with limited GPU memory. We also find that the number of clusters has an impact on search accuracy. Using more clusters generally leads to higher recall, because finer partitioning of the data space reduces boundary effects and makes it easier to cover true neighbors around the query.

To further validate that *twin-buffer* plays a crucial role in improving search accuracy in the previous comparison, we measure the average dimension involved in each distance computation. As shown in Figure 14, under the same memory usage, enabling *twin-buffer* leads to a much higher average dimension than the baseline. For instance, at a GPU memory usage of 43 GB, the average dimension is 49 for both DEEP100M and SIFT100M without *twin-buffer*, but increases to 88 on DEEP100M and 117 on SIFT100M when *twin-buffer* is enabled. This indicates that important candidates are evaluated using full-precision and high-dimensional representations, confirming that *twin-buffer* effectively reallocates limited memory to where it matters most.

V. Related Work

Approximate Nearest Neighbor Search. ANN search has been extensively studied on both CPU and GPU platforms. On CPU, graph-based methods, e.g., NSW [6], HNSW [7], NSG [8], HCNNG [10], and Vamana [9], achieve high recall and fast search, while quantization-based methods such as IVFPQ [15], OPQ [16], and IMI [17] provide compact indexes and efficient distance computation. With increasing demand for large-scale, high-throughput applications, many ANN methods have been adapted to GPU. Quantization-based approaches leverage GPU parallelism naturally, and graph-based GPU methods, e.g., SONG [30], GANNS [11], GGNN [13], and CAGRA [14], address indexing and memory challenges on modern hardware. Among numerous ANNS methods, PARS explores how to accelerate distance computation in a CPU-GPU architecture.

Distance Computation Optimization. One of the effective means that accelerates ANN search is optimizing the distance computation by reducing the dimensions used. For example, ADSampling [31] computes only a subset of vector dimensions with early stopping, whereas VSAG [32] adopts PCA to approximate distances efficiently. PARS extends this paradigm to a CPU-GPU architecture, which is commonly throttled by limited memory capacity. By performing dimension reduction in advance, both the computation overhead and memory footprint are effectively reduced by PARS.

General-Purpose Computation on RT Cores. Recent research explores RT cores for general-purpose computing, demonstrating their efficacy in non-graphical problems [39]–[44]. For ANN search, RTNN [33] and TrueKNN [36] introduced RT cores for accelerating spatial queries, while JUNO [37] recasts PQ table lookup as a geometric ray-primitive intersection problem. Additionally, Arkade [45] extends RT-based kNN search to support non-Euclidean distance metrics. Among these methods, PARS is the first to employ RT hardware for entry point selection, offering a scalable and practical integration of RT cores into graph-based ANN systems.

VI. Conclusion

In this paper, we propose PARS, a GPU-based ANN search method using dimension reduction. We project the data points into a d-dimensional space using PCA and design a sampling method to minimize d while achieving a target recall. RT cores are further used to locate high-quality entry points. This makes PARS superior to other GPU-based methods. With the pluggable *twin-buffer* module, PARS further improves distance computation accuracy by using full-precision vectors for critical candidates. Experiments show that PARS significantly outperforms GPU baselines in speed and memory efficiency, enabling large-scale ANN search on a single L40 GPU.

Acknowledgment

This work is supported in part by the Shandong Provincial Natural Science Foundation (Project No. ZR2024LZH001), the National Natural Science Foundation of China (No. 62572279, 62472253), and the Taishan Scholars Program under Grant tsqn202408025.

References

[1] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 conference on empirical methods in natural language processing*, 2020, pp. 6769–6781.

[2] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *International conference on machine learning*, 2021, pp. 8748–8763.

[3] L. Wang, E.-P. Lim, Z. Liu, and T. Zhao, "Explanation guided contrastive learning for sequential recommendation," in *Proceedings of the 31st ACM international conference on information & knowledge management*, 2022, pp. 2017–2027.

[4] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, vol. 87, p. 101374, 2020.

[5] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 577–586.

[6] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[7] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[8] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 461–474, 2019.

[9] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, 2019.

[10] J. V. Munoz, M. A. Gonçalves, Z. Dias, and R. d. S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition*, vol. 96, p. 106970, 2019.

[11] Y. Yu, D. Wen, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Gpu-accelerated proximity graph approximate nearest neighbor search and construction," in *2022 IEEE 38th International Conference on Data Engineering*, 2022, pp. 552–564.

[12] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4139–4150, 2022.

[13] F. Groh, L. Ruppert, P. Wieschollek, and H. P. Lensch, "Ggnn: Graph-based gpu nearest neighbor search," *IEEE Transactions on Big Data*, vol. 9, no. 1, pp. 267–279, 2022.

[14] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus," in *2024 IEEE 40th International Conference on Data Engineering*, 2024, pp. 4236–4247.

[15] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[16] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 2946–2953.

[17] A. Babenko and V. Lempitsky, "The inverted multi-index," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 6, pp. 1247–1260, 2014.

[18] A. Babenko and V. Lempitsky, "Additive quantization for extreme vector compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 931–938.

[19] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.

[20] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[21] T. Zhang, C. Du, and J. Wang, "Composite quantization for approximate nearest neighbor search," in *International Conference on Machine Learning*, 2014, pp. 838–846.

[22] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 2321–2328.

[23] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 518–529.

[24] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.

[25] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 541–552.

[26] X. Zhao, Y. Tian, K. Huang, B. Zheng, and X. Zhou, "Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces," *Proceedings of the VLDB Endowment*, vol. 16, no. 8, pp. 1979–1991, 2023.

[27] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 97–104.

[28] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International conference on computer vision theory and applications*, 2009, pp. 331–340.

[29] S. Dasgupta and K. Sinha, "Randomized partition trees for exact nearest neighbor search," in *Conference on learning theory*, 2013, pp. 317–337.

[30] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering*, 2020, pp. 1033–1044.

[31] J. Gao and C. Long, "High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023.

[32] M. Yang, W. Li, J. Jin, X. Zhong, X. Wang, Z. Shen, W. Jia, and W. Wang, "Effective and general distance computation for approximate nearest neighbor search," in *41st IEEE International Conference on Data Engineering*, 2025, pp. 1098–1110.

[33] Y. Zhu, "Rtnn: accelerating neighbor search using hardware ray tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 76–89.

[34] J. Burgess, "Rtx on—the nvidia turing gpu," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020.

[35] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "Optix: a general purpose ray tracing engine," *Acm transactions on graphics*, vol. 29, no. 4, pp. 1–13, 2010.

[36] V. Nagarajan, D. Mandarapu, and M. Kulkarni, "Rt-knns unbound: Using rt cores to accelerate unrestricted neighbor search," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 289–300.

[37] Z. Liu, W. Ni, J. Leng, Y. Feng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Juno: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024, pp. 549–565.

[38] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *The collected works of Wassily Hoeffding*, pp. 409–426, 1994.

[39] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location," *High Performance Graphics (Short Papers)*, vol. 7, p. 13, 2019.

[40] N. Morrical, W. Usher, I. Wald, and V. Pascucci, "Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing," in *2019 IEEE Visualization Conference*, 2019, pp. 256–260.

[41] Y. Lv, K. Zhang, Z. Wang, X. Zhang, R. Lee, Z. He, Y. Jing, and X. S. Wang, "Rtscan: Efficient scan with ray tracing cores," *Proceedings of the VLDB Endowment*, vol. 17, no. 6, pp. 1460–1472, 2024.

[42] Z. Xiao, M. Xiao, Y. Yuan, D. Yu, R. Lee, and X. Zhang, "A case study for ray tracing cores: Performance insights with breadth-first search and triangle counting in graphs," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 2, pp. 1–25, 2025.

[43] L. Geng, R. Lee, and X. Zhang, "Rayjoin: Fast and precise spatial join," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 124–136.

[44] L. Geng, R. Lee, and X. Zhang, "Librts: A spatial indexing library by ray tracing," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 396–411.

[45] D. K. Mandarapu, V. Nagarajan, A. Pelenitsyn, and M. Kulkarni, "Arkade: k-nearest neighbor search with non-euclidean distances using gpu ray tracing," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 14–25.