ALGAS: A Low-latency GPU-Based Approximate Nearest Neighbor Search System

Yuanhui Chen, Lixiao Cui, Zebin Yao, Hao Zhou, Gang Wang* and Xiaoguang Liu*

College of Computer Science, TMCC, SysNet,DISSec, GTIISC, Nankai University, Tianjin, China

{chenyh, cuilx, yaozb, zhouh, wgzwp, liuxg}@nbjl.nankai.edu.cn

***Abstract*—Approximate Nearest Neighbor Search (ANNS) is widely used in various fields, including database systems, recommendation engines, and large language models. As data dimension and size continue to expand, many studies explore GPU acceleration for graph-based ANNS. While previous methods use large batch to maximize throughput, they often lead to increased latency. In contrast, small batch is more effective for online low-latency applications, as it minimizes batch accumulation time. However, employing small batch on GPU presents challenges. First, the query bubble issue in batch processing negatively impacts both latency and GPU utilization. Second, current GPU search methods incur excessive sorting overhead, and introduce additional TopK-merging overhead on GPU. To address these challenges, we propose ALGAS, a low-latency GPU search system designed for small batch. ALGAS employs dynamic batching based on persistent GPU kernel function to optimize query bubble. Additionally, it employs beam extend to reduce sorting overhead, especially effective at high recall rate. It also eliminates TopK-merging overhead via GPU-CPU cooperation. Furthermore, it employs an adaptive GPU tuning scheme to optimize resource utilization. We compare ALGAS with the state-of-the-art graph-based works. ALGAS reduces latency by up to 21.9%-35.4% and increases throughput by up to 27.8%-55.2% under various real-world datasets.**

***Index Terms*—approximate nearest neighbors, GPU, information retrieval, vector similarity search**

I. Introduction

Vector retrieval plays a critical role in various domains, including databases [1], [2], recommendation systems [3], [4], and large language models [5], [6]. It is a k-nearest neighbor (k-NN) search problem, where the objective is to search the k closest vectors to a given query vector. To reduce the computational cost of k-NN search, researchers propose approximate nearest neighbor search (ANNS), which allows precision loss while significantly improving search efficiency.

Current research on ANNS classifies methods into hashing-based [7], [8], tree-based [9]–[11], quantization-based [12]–[14], and graph-based methods [15]–[20]. Graph-based methods perform excellently in terms of performance and recall. The increasing computational costs caused by the growing data dimensions and data size drive researchers to leverage GPU for accelerating ANNS [21]–[25], which achieve significant throughput, especially in graph-based ANNS.

Unfortunately, most GPU-based graph ANNS focus on throughput under large batch, with limited discussion on small batch. While using large batch can significantly enhance throughput, it comes at the cost of increased latency. First, waiting for enough requests to accumulate large batch is impractical in some real-world scenarios (e.g., online vector retrieval). Second, scheduling a large number of queries on limited GPU cores can significantly increase GPU utilization, but it reduces the resources available for each query, leading to increased latency. Conversely, small batch can be accumulated more quickly, allowing for faster processing of queries. With fixed resources, each query in small batch accesses more resources, balancing the performance and latency.

TABLE I: Performance in Graph-based GPU search works

	batch size	**Throughput**	**Latency**
CAGRA	single query	moderate	good
CAGRA	large batch	good	bad
ALGAS	**small batch**	**good**	**good**
GANNS	large batch	moderate	bad

However, we identify several bottlenecks limiting the effectiveness of employing small batch. First, due to different search deep of queries, there is a ***query bubble*** within the batch. Different search times require the batch to wait for the slowest query to finish, negatively impacting the latency of shorter queries. Furthermore, the more frequent processing of small batch leads to increased idle time. Unlike large batch, small batch cannot enhance GPU utilization by scheduling a significant number of queries. As a result, idle cores occur during wait times, ultimately causing decreased GPU utilization. Compared to the average latency of active queries, the waste rate ranges from 22.9% to 33.7%.

Second, ***limitations of parallel algorithm***: In graph based greedy search, it is necessary to sort the candidate list to obtain closest unvisited points. Current GPU algorithms exhibit significant sorting overhead, with parallel sorting methods resulting in an overhead of 19.9% to 33.9%. This directly constrains latency and throughput. In small batch, to enhance GPU utilization, queries should be allocated to numerous GPU threads. However, GPU Cooperative Thread Array (CTA) has a limit on the number of threads, requiring queries to span multiple CTAs to utilize additional threads. The Multi-CTA approach introduces an additional TopK merge operation once the search completes. This operation requires additional synchronization, extending the critical path and increasing latency during the search process.

In this paper, we introduce ALGAS, a low-latency GPU-accelerated graph search system optimized for small batch. Table I presents a performance comparison between ALGAS and state-of-the-art graph-based GPU methods. Compared with other works, ALGAS reduces latency by up to 21.9%-35.4% and increases throughput by up to 27.8% to 55.2% across different datasets.

The main contributions of our paper are as follows:

- Dynamic Batching Mechanism: We introduce a dynamic batching mechanism. It modifies batch to fixed slots and introduces state management for each independent slot, eliminating the bubble problem associated without batch synchronization. By employing a persistent kernel function to fix slots within the GPU, we eliminate the overhead associated with frequently launching kernel functions in small batch processing.
- Optimized parallel algorithm: We propose a optimized parallel algorithm to reduce the sorting overhead and the synchronization overhead associated with merging TopK results. We employ the beam extend method to reduce the number of sorting operations in the later phase of the search, lowering query latency. Additionally, we migrate the TopK merge operation to the CPU, allowing the GPU to focus on more efficient distance calculation and reducing the communication overhead associated with multiple CTAs on the GPU.
- Adaptive GPU parameter tuning scheme: We implement an adaptive tuning scheme for GPU resources and persistent kernel, allowing users to determine the best tuning strategy based on GPU hardware and software parameters, maximizing GPU resource utilization.

II. Background

A. ANNS

Approximate Nearest Neighbor Search (ANNS) is an approximate solution to the k-nearest neighbor (k-NN) problem. The k-NN problem involves a set of n-dimensional vectors P and, given a point $p \in P$ where $p \in \mathbb{R}^n$, the goal is to find k vectors $(p_1, p_2, \ldots, p_k) \in P$ that are closest to a query point $q \in \mathbb{R}^n$. The similarity between vectors is computed using a distance function $F(a, b)$, with common distance functions including Euclidean distance and cosine similarity. In Approximate Nearest Neighbor Search, to balance search performance and accuracy, the output consists of k approximate nearest neighbor vectors $\mathrm{K}_{\mathrm{approximate}}$ rather than the true k nearest neighbors $\mathrm{K}_{\mathrm{truth}}$. We evaluate the approximation quality using the recall rate defined as recall $= \frac{|\mathrm{K}_{\mathrm{approximate}} \cap \mathrm{K}_{\mathrm{truth}}|}{|\mathrm{K}_{\mathrm{truth}}|}$. A higher recall rate indicates a stronger recall capability of the algorithm.

Graph-based ANNS constructs an approximate nearest neighbor graph index from P, achieving excellent performance in both retrieval speed and recall. The approximate nearest neighbor graph $G = (V, E)$ connects the vectors in P according to specific graph construction rules, and during retrieval. The search process utilizes a greedy exploration approach to gather candidate points. Once a sufficient number of candidates are collected, the k nearest points are selected as the TopK results. Algorithm 1 describes a basic GPU parallel search process.

Algorithm 1 Parallel Greedy Search Algorithm

Input: Graph G; query q; enter point p; candidate size l
Output: k nearest neighbors of query point q
function GreedySearch(G, q, p, l)
    $i \leftarrow 0,\ S \leftarrow \emptyset$
    S.add(p)
    while $i < l$ **do**
        $i \leftarrow$ first unchecked index in S
        mark p_i as checked
        for each neighbor n of p_i in G **do**
            for each $n_{subvec} \in n$ **in parallel do**
                $d(s_{subvec}) \leftarrow$ Distance(n_{subvec}, q_{subvec})
            end for
            $d(s) \leftarrow$ Aggregate($d(s_{subvec})$ for all n_{subvec})
            S.add(n)
        end for
        $S \leftarrow$ parallel sort S by distance to q
        if size of $S > l$ **then**
            S.resize(l)
        end if
    end while
    return first k elements in S
end function

B. GPU Architecture

NVIDIA GPU contains multiple Streaming Multiprocessors (SMs). Each SM has several stream processors (SPs), also known as CUDA cores, responsible for executing specific arithmetic and logical operations. Multiple CUDA cores within SM are scheduled in a unit called warp, where the cores in a warp execute the same instruction at the same time and can use warp shuffle instruction to speed up. The storage architecture of GPUs is hierarchical. Global Memory, accessible to all cores, offers the largest capacity but slower speeds. Each SM has a portion of Shared Memory that can only be accessed by cores within the same SM. For external memory, NVIDIA's Unified Memory supports memory over-subscription, enabling programs to operate beyond the GPU memory limit.

CUDA [26] is the mainstream general-purpose parallel programming model for NVIDIA GPU. The GPU performs parallel computation in the form of kernel functions and abstracts a three-tier organizational structure of threads, blocks, and grids at the software level. Threads correspond to CUDA cores. Multiple warps combine to form Cooperative Thread Array (CTA), which makes up a block; The grid defines the number of blocks that can run a kernel function.

III. Motivation

A. Query bubble between batches

In Graph-based ANNS, combining multiple queries into a single batch can effectively utilize GPU resources. Prior works, including SONG [22] and GANNS [23], focus on employing a single search kernel function to process a large batch, saturating the GPU resources and achieving great throughput. However, in most online scenarios, gathering such a large batch for searching is impractical, as it would result in significantly high end-to-end latency. Small batch processing is a more reasonable choice, as it can achieve low query latency while maintaining high throughput.

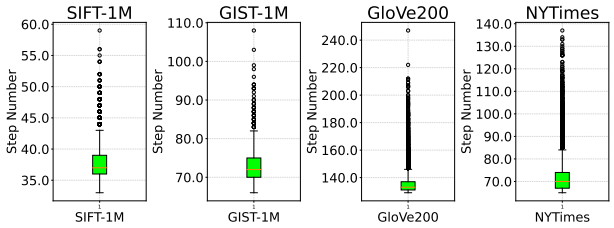

Fig. 1: The distribution of query steps of the whole dataset.

Batch processing requires synchronization on GPU; as a result, the completion time of a batch is determined by the slowest query. Unfortunately, we observe that the expected search times of queries are different, and some queries have longer tail latency. To verify this phenomenon, we count the number of steps in the query process. Each step consists of selecting the best unvisited point, expanding it, and resorting the candidate list, as shown in Algorithm 1 (line 7 to line 19). Fig. 1 shows the relevant results of different datasets. We observe significant variation in the number of steps across queries. For each dataset, we construct a graph index and collect steps of queries in its corresponding query set. To provide a clearer representation of the overall distribution, we excluded certain outliers from the dataset with excessively low step values. For queries with the highest number of steps, their step counts can reach 147.9%-190.2% of the average step count. These slow queries with long steps limit the overall improvements of latency.

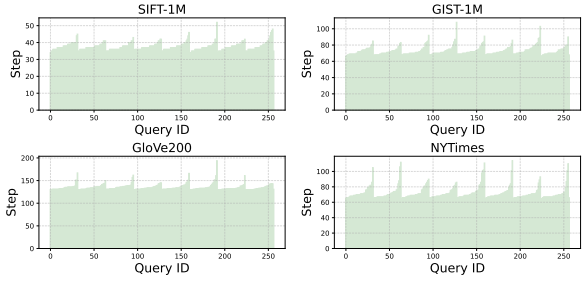

Fig. 2: The distribution of query steps within a batch

Even in small batch, there is a noticeable unevenness in the number of steps between queries. Fig. 2 shows the distribution of steps within a batch (batch size is 32). Without loss of generality, we selected 8 batches for each dataset to show the results. In GIST1M, The distribution of query steps ranges from about 70 to 100. The slowest query takes up to 32.4% more steps than the fastest query. The completion time of the batch is predominantly determined by the tail latency, severely impacting the latency of the queries that are not in the tail. We refer to this phenomenon as the ***query bubble***. During the processing of the slowest query, some cores (responsible for processing faster queries) remain idle in single kernel processing.

The above analysis motivates us to reduce the impact of the ***query bubble*** by utilizing idle GPU cores as much as possible and allow completed queries to exit earlier.

B. Limitations of GPU Parallel Algorithm

In Graph-based ANNS, queries can be processed in parallel on GPU. Intra-CTA refers to the parallel processing of threads within a CTA, where threads compute partial dimensions and merge sub-distances. To achieve higher thread parallelism beyond the capacity of a single CTA, multiple CTAs should be employed, which is referred to as Multi-CTA. Currently, CAGRA [25] implements Multi-CTA, where each CTA maintains its own candidate list and enters random entry point, executing the intra-CTA search mode. The CTAs share a visited table and merge their immediate TopK after the search is complete.

However, we find several bottlenecks. The search within intra-CTA incurs significant overhead from maintaining the candidate list, as sorting new neighbors and candidate points is computationally expensive. As shown in Fig. 3, the sorting overhead can reach 19.9% to 33.9% across different datasets. Even though our intra-CTA implementation draws from the more efficient maintenance of parallel data structures in GANNS [23], this overhead remains significant and directly affects latency.

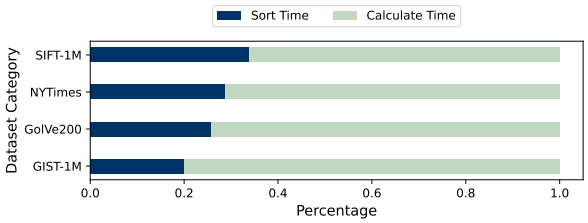

Fig. 3: The percentage of time on calculation and sorting

The Multi-CTA approach [25] requires merging the TopK results from each CTA into final TopK after the search completes. Since merging across CTAs requires access to global memory, it cannot utilize the more efficient shared memory. Moreover, merging multiple ordered TopK results has limited parallelism. Even with a divide-and-conquer approach, nearly half of the cores remain idle, leading to inefficient resource utilization.

Optimizing the sorting overhead in intra-CTA and addressing the TopK merge issue in Multi-CTA are significant challenges.

IV. Design

In this section, we introduce ALGAS, a low-latency GPU-accelerated ANN search system. The overview of ALGAS is illustrated in Fig. 6.

For the query bubble issue, we propose a dynamic batching mechanism that splits the batch into independent slots. By implementing persistent kernel function, each slot can operate independently on the GPU without batch synchronization, which reduces the query bubble between batches, thereby preventing the GPU cores from being idle. We introduce this method in Section IV-A. Additionally, we develope a parallel search algorithm optimized for the GPU, which reduces the frequency of maintenance sorting on the candidate list during the search process and offloads the TopK merging to the CPU. This allows the GPU to focus more on efficient and parallel distance computation. We show the design details in Section IV-B. To determine the optimal parallel parameters, we incorporate specific tuning based on hardware and software parameters to maximize GPU utilization as shown in Section IV-C.

A. Dynamic Batching

To reduce the query bubble in batch processing, an intuitive solution is to merge multiple batches, eliminating batch boundaries and reducing the wasted space.

We propose a dynamic batching designed to reduce bubble between batches. Our objective is to efficiently organize the submitted query requests, thereby minimizing inefficiencies in batch processing. As shown in Fig. 4, compressing multiple batch bubble not only reduces the overall completion time for multiple batches but also allows shorter queries to return results faster.

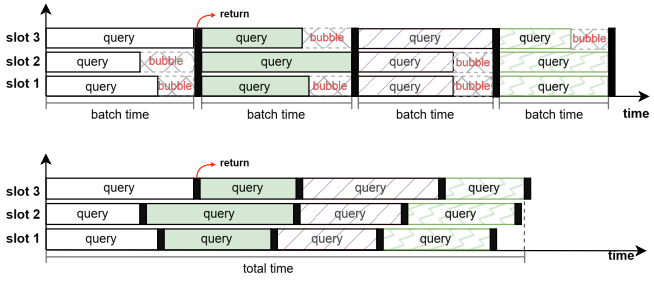

Fig. 4: Static Batching vs Dynamic Batching

In our design, we transform the batching process into independent slots that do not affect each other. Each slot is fully responsible for the execution lifecycle of a single query, including dispatching the query from the host to the GPU and returning the result to the host after computation. Once a slot completes executing a query, it can immediately dispatch the next prepared query without waiting for all queries in the same batch to finish.

To enhance the synchronization of slot-level information between the host and the GPU, we introduce the state for each slot and categorize them into 5 states. Below, we briefly outline each state and the transitions between them.

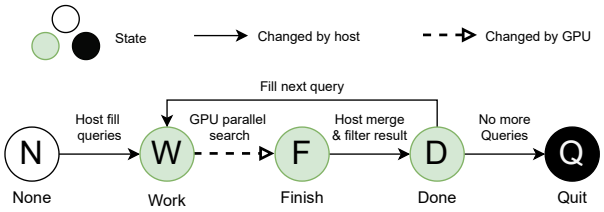

Fig. 5: State Transition Process of Query

- **None** After a slot is initialized, its state is in *None*, indicating that the slot can accept a new query request.
- **Work** The host sends a query to a slot and fill to the GPU, subsequently updating the states of the corresponding CTAs to *Work*. Once the CTAs associated with the slot detect that the state has been set, it will receive the query and start performing the search.
- **Finish** After completing the search, CTA is responsible for pushing the query results to the designated location. Subsequently, the CTA changes the state to *Finish*.
- **Done** When the host perceives that all CTAs are in the *Finish*, it retrieves the corresponding query results. After completing the current query, the slot has two options: acquire the next query and change all states of its CTAs to *Work* or exit directly.
- **Quit** This state indicates that the slot has exited and will no longer accept new queries.

Persistent Kernel Function: The GPU search algorithms often implement single-kernel function for performance. However, the single-kernel design presents an implementation challenge for dynamic batching, which requires operations across multiple batches.

Dynamic batching requires checking the state of each slot. A simple approach is to partition the kernel. The kernel performs a fixed number of steps and then exits, allowing the host to check completed queries and add new ones. But this approach incurs significant memory access overhead. Each kernel checking requires the repeated loading of data into shared memory. Deciding the timing for checking is quite challenging because overly frequent checks can increase overhead, while infrequent checks may not eliminate enough bubble.

To address this, we implement a persistent kernel function that allows for state checking without splitting the kernel. Specifically, we change the kernel's checking method to a polling mode, enabling the kernel to poll and modify states on the GPU side rather than being entirely controlled by the host. As shown in Fig. 6, each CTA assign to a kernel slot maintains its own state, enabling it to operate independently and eliminating the need for kernel exit, thereby reducing overhead.

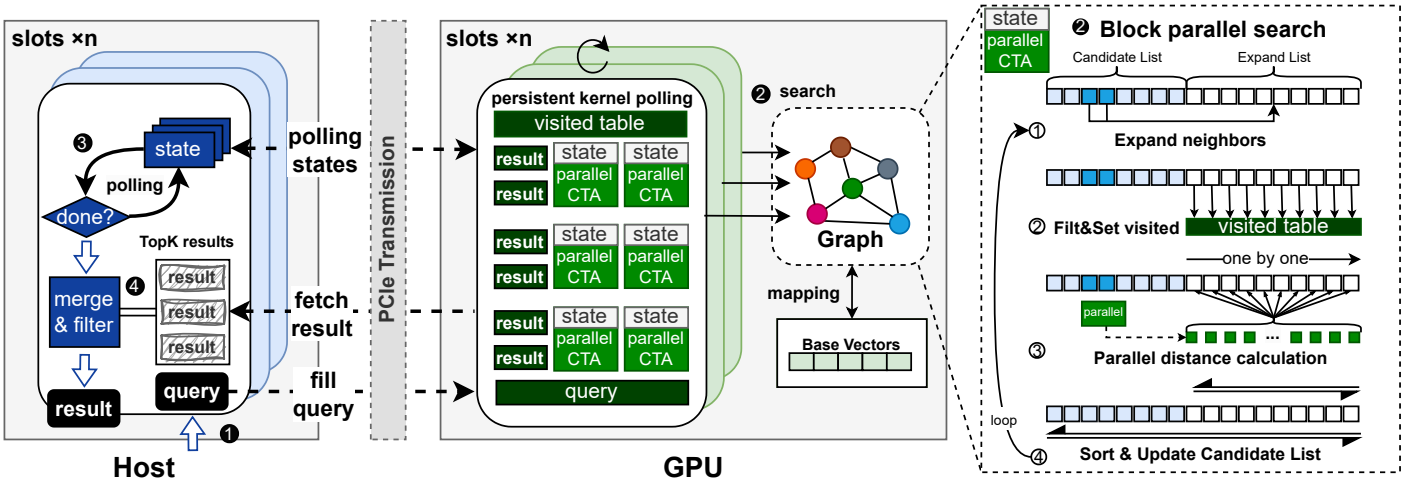

Fig. 6: The Overview of ALGAS

B. Search Algorithm

In this section, we introduce beam extend to optimize the sorting overhead within intra-CTA. Next, we introduce the optimization of the TopK merging for Multi-CTA through GPU-CPU cooperation. Finally, we gradually outline the specific search process in Fig. 6.

Beam Extend in Intra-CTA: To increase the percentage of distance computations on the GPU, it is essential to reduce sorting operations. We analyze the role of candidate list maintenance operation in the query search process. Searching on the graph is a type of greedy search, where in each iteration, the query needs to select the closest unvisited candidate point and then compute the distance to its neighbors. After adding new points to the candidate list, maintenance is required to ensure that the next iteration can continue to select closest candidate point.

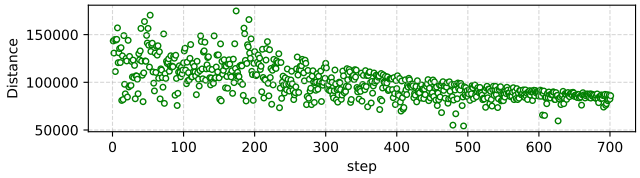

Fig. 7: Distribution of Distance with Respect to Steps

This process prompts us to consider the necessity of maintaining the candidate list in each iteration. In Fig. 7, We collect distance data for the query during the search process. We find the distances between the query and the search points decrease sharply in the early stages, while they gradually converge in the later stages. This indicates that the early stage primarily focuses on locating the region where the TopK resides, resulting in significant distance variations. In contrast, the later stage conducts a diffusion search within that small region, leading to smaller distance changes.

Frequent sorting during the diffusion phase seems unnecessary because this phase requires accessing most points within a small region, and most of these points will eventually be visited. Therefore, there is no need to ensure a strictly greedy selection. Especially at high recall, a larger candidate list necessitates a broader access region.

We propose the beam extend method to optimize sorting overhead, considering it in two phases. In time of phase division, we can determine this time based on the offset of the currently selected candidate point within the list. In the early phase, referred to as the localization phase, we maintain the candidate list in each iteration using a standard greedy search method, allowing us to quickly identify the small region containing the TopK. In the later phase, referred to as the diffusing phase, we execute the beam extend method. Specifically, during the diffusing phase, we skip some sorting operations for several iterations, performing sorting after a few iterations. From the perspective of selecting candidate points, we choose several points at once for neighbor expansion. The new points generated from several iterations are added to the candidate list through a single maintenance operation. Although this approach may result in a less greedy search, it does not significantly impact recall, as most of the points in the small region will ultimately be visited. This reduces the number of sorting operations during the search process, directly lowering latency.

GPU-CPU Cooperation in Multi-CTA: The multi-CTA method requires an additional TopK merge operation to merge sub-sequences from multiple CTAs. Performing the merge operation within the Multi-CTA search kernel is inefficient, as cross-CTA data synchronization incurs slower global memory access and reduces GPU utilization. Even a divide-and-conquer merging approach can result in nearly half of threads remaining idle. More importantly, the independent merge kernel conflicts with the design of persistent kernels in dynamic batching, as the persistent kernel must be interrupted to launch the merge kernel.

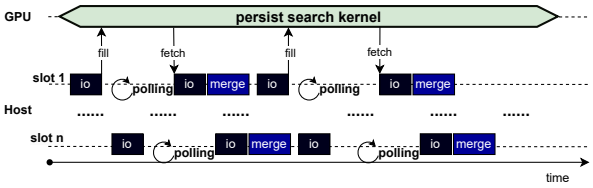

Fig. 8: Timeline of GPU-CPU Cooperation

We address this challenge by approaching the problem from a higher level. The TopK merge operation is a critical step in the execution path and represents the final operation performed on the GPU. As a result, offloading this operation to the CPU does not incur additional GPU-related computations, such as resuming GPU processing.

In Fig. 8, we decouple the TopK merge operation from the GPU. In this way, the GPU focuses on low-communication parallel search, while the CPU leverages its low memory access latency to perform merging tasks. At this point, the GPU does not require any additional synchronization across CTAs, except for sharing the visited table. This method integrates well with dynamic batching, which decouples batch into independent slots. The CPU can perform the TopK merge operation smoothly for each slot, without waiting for batch synchronization, thereby reducing the overhead associated with TopK merge operations. In the data organization for slots and CTAs, we place the CTA results of the same query in a continuous address space. The CPU only needs a single sequential I/O operation to retrieve all results for the queries across CTAs. We provide a detailed discussion of this process next.

Process of Query Searching: We outline the search algorithm in specific steps, with each step corresponding to a particular number in Fig. 6. Assuming we need to search for the TopK data points, with a candidate list of max length L, an expand list length, and T CTAs used for the search.

❶ Send Query: The host sends a query to the slot, setting CTAs states of slot to *Work*. The CTAs in the slot poll the state and initiate the search once the *Work* state is detected. Each CTA initializes a part of the visited table, implemented as a bitmap.

❷ Search in CTA: Each CTA follows the same process: ① The CTA selects the closet unvisited point as candidate point from its candidate list. Then, the CTA obtains the neighbors of this candidate point and adds them to the expand list. If all points in the candidate list have been visited, the state changes to *Finish* and the search exits. ② The CTA checks the points in the expand list using the bitmap to determine whether their distances have been calculated. It filters out the visited points and marks the unvisited points in the bitmap. ③ The CTA calculates the distances between the query and the points in the expand list. It distributes the dimension according to the number of threads in the CTA, with each thread computing a portion of the dimension distance, and then aggregates the distances by shuffle operation. ④ The CTA performs a parallel-friendly bitonic sort. First, it sorts the expand list based on distances. Then, it merges the candidate list and expand list through this sorting, keeping the top L (where $L \geq$ TopK) points in the candidate list.

❸ Host State Monitoring: The host polls the states of all corresponding search CTAs. Once all states of the slot are detected as *Finish*, the host fetches all results. It loads the next query into the slot, sets its states, and the slot proceeds with the next search round.

❹ Result Merge&Filter: Host uses a priority queue to merge TopK, filters out unnecessary data, submits the results.

C. Adaptive Tuning Scheme

To maximize throughput and minimize latency, it is essential to fully utilize the cores and storage resources on the GPU. Since the dynamic batching kernel function has been modified into a persistent kernel, there are differences in resource allocation compared to normal CUDA kernels. Specifically, we need to ensure that all slots can activate simultaneously. This requires that the allocation of logical resources for each slot remains within these physical limitations to ensure simultaneous activation.

To facilitate management and shuffle operations, we set the number of threads per block to match the warp size. We develop an optimal parallel scheme by adjusting $N_{parallel}$ based on the hardware parameters. In this context, $N_{parallel}$ indicates how many CTAs are assigned for each query. To support the simultaneous execution of all blocks, the following condition must be satisfied:

$$N_{parallel} \cdot slot \leq N_{SM} \cdot N_{\ max\_block\_per\_SM}$$

N_{SM} is the number of multiprocessors, and $N_{max\_block\_per\_SM}$ is the number of blocks that each SM can simultaneously support. With a fixed number of slots, we need to dynamically adjust $N_{parallel}$. This allows us to determine the suitable $N_{block\_per\_SM}$:

$$N_{block\_per\_SM} = \text{align}(N_{parallel} \cdot slot / N_{SM})$$

The candidate list and expand list are frequently accessed data structures during the search process. To enhance efficiency, we store them in shared memory, referred to as $M_{avail\_per\_block}$. It is clear that $M_{avail\_per\_block}$ will vary based on the sizes of the candidate list and expand list.

In the CUDA configuration, we allocate a portion of shared memory for each block, while the remaining unallocated shared memory acts as a runtime cache. For datasets with higher dimensions, the reserved shared memory per block may be insufficient, resulting in performance degradation. Therefore, we allow for the reservation of additional shared memory as cache during parameter adjustment, denoted as $M_{reserved\_per\_block}$, with its specific size adjustable based on the data dimension.

The maximum shared memory allocated to blocks within a multiprocessor is limited by M_{per_SM}, which represents the "Shared Memory per Multiprocessor." To support the maximum number of $N_{parallel}$ and persistent kernel, we need

to maximize the number of blocks that a multiprocessor can run simultaneously. The total shared memory used by blocks within the same multiprocessor cannot exceed the maximum supported by the multiprocessor; otherwise, they cannot run simultaneously. Therefore, the maximum shared memory available must satisfy:

$$M_{\text{avail\_per\_block}} \leq M_{\text{per\_SM}}/N_{\text{block\_per\_SM}} - M_{\text{reserved\_per\_block}}$$

$M_{\text{avail\_per\_block}}$ is the available share memory per block, and $M_{\text{per\_SM}}$ is the capacity of shared memory per multiprocessor.

Regarding the timing for activating beam search, we first define a search threshold $\text{offset}_{\text{beam}}$. When the searcher expands to include a point located at $\text{offset}_{\text{beam}}$ within the candidate list, we consider this to be the later stage of the search, at which point we start beam search.

V. Optimization

In this section, we discuss key optimizations in ALGAS. First, we address the optimization of state transmission concerning PCIe I/O bottleneck. Then, we describe the use of parallel processing on the host to handle queries.

A. State Optimization

We find that increasing the number of slots leads to a higher volume of state transmitted via PCIe. This increase results in an I/O bottleneck that affects the transmission of query vectors and results for all slots. Especially in small-dimensional datasets, it frequently involves cross-PCIe data exchanges. To address this, we employ copies of the state to optimize unnecessary PCIe I/O.

State transfer between the host and GPU primarily involves synchronizing information, with the host polling GPU states and generating a large number of small PCIe I/O transactions. However, in most cases, the polling frequency of the host exceeds the rate of state changes. As a result, much of the polling is ineffective, as the state often hasn't changed. While using blocking mode can reduce PCIe I/O, its performance is generally not as good as polling.

To address this, we use local copy of slot states on the host and GPU. By polling the local slot state, we can avoid cross-PCIe communication. When the GPU updates the local slot state, it only needs to perform a single PCIe transfer to update the corresponding remote slot state. In Fig. 9, we prepare state copies for each slot on both the host and GPU, located in their respective memory, and map address of copies to each other using GDRcopy [27]. When the host or GPU needs to change a state, it updates the local state and makes a single PCIe transfer to update the remote state copy. During polling, we do not generate any PCIe transfers. Although both the GPU and host can modify states, no consistency conflicts arise because only one side has modification right at any given time. The GPU has modification right only after a query is received, while the host retains modification right for all other states.

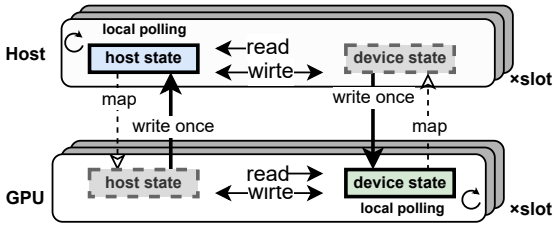

Fig. 9: Mechanism of State Optimization

B. Parallel Processing on Host

The host needs to send queries, retrieve results, and perform merging and filtering operations. Due to the use of dynamic batching, the return timing of each query is uncertain, which requires the host to poll these queries and retrieve the results via IO streams. When the batch size is larger, such as above 32, a single host thread may struggle to respond to so many queries. This issue is particularly pronounced in datasets with smaller dimensions or quantities, where computations return rapidly, necessitating more frequent processing by the host.

To address this issue, we adopt a parallel processing model, assigning each thread a subset of slots. We also allocate private IO streams for each host thread to prevent IO blocking. In the implementation, each host thread initializes its designated slots and relevant metadata. They employ a concurrent query manager module to handle query distribution. The slots managed by a single host thread share a common IO stream for handling queries. Each host thread monitors the states of CTAs within its slots, and once all CTAs in a slot have finished, it retrieves results sequentially through the stream. Finally, the host performs merge and filter operations before handling a new query.

VI. EXPERIMENTS

This section presents the experiments conducted to evaluate the search performance efficiency of the proposed ALGAS.

We conduct our experiments on a server running Ubuntu 20.04.1 with Linux kernel version 5.15.0. This server comprises two NUMA nodes, each containing a CPU featuring 10 physical cores (Intel Xeon Silver 4210 @ 2.20GHz) and 128GB of DRAM. Additionally, the server includes an Nvidia Quadro A6000 GPU [28], utilizing the 535.104.05 GPU driver. For more details regarding the GPU configuration, refer to Table II.

TABLE II: RTX A6000 Device Properties

Property	Value
Shared memory per block	48 KiB
Shared memory per multiprocessor	100 KiB
Reserved shared memory per block	1 KiB
deviceProp.sharedMemPerBlockOptin	99 KiB
Number of SMs	84
Max blocks of SM	16
Max threads per block	1024
Warp size	32

We select four datasets for the experiments. Table III presents the data sizes and dimensions. To verify ALGAS can support general GPU graph, we use NSW-GANNS graph [23] and CAGRA graph [25] on the experiments. We compare ALGAS with three search methods:

- **CAGRA** [25]: This is one of the state-of-the-art works, with advantages in single-query latency.
- **GANNS** [23]: A representative work on GPU. For comparison purposes, we make a modification to GANNS, allowing for a smaller batch size to be dispatched to the GPU instead of sending the entire query set.
- **IVF** [21]: IVF implemented by FAISS-GPU, a widely used open-source library for approximate Nearest Neighbor Search.

TABLE III: Properties of Dataset

	Vertices	Dimension	Metric
SIFT1M [29]	1M	128	Euclidean
GIST1M [29]	1M	960	Euclidean
GLoVe200 [30]	1.18M	200	CosineSimilarity
NYTimes [31]	0.29M	256	CosineSimilarity

A. Performance Comparison

We compare the performance of different graph-based methods under small batch size, using a batch size of 16 as a representative case. Similar performance trends are observed for other small batch sizes. The TopK parameter is fixed at 16, and the candidate list size is the primary factor controlling recall. Fig. 12 illustrates the variation in average latency for different TopK, with red numbers indicating recall rates. Other parameters are optimized using an adaptive tuning scheme. Both CAGRA and GANNS employ the same batch size for query execution.

As shown in Fig. 10 and Fig. 11, ALGAS demonstrates superior latency and throughput. The figure labels follow the format: the first part represents the graph type, and the second part represents the search method (e.g., CAGRA-ALGAS denotes searching the CAGRA graph using the ALGAS method). GANNS, due to the absence of a multi-CTA implementation, fails to fully utilize GPU resources in small-batch settings, resulting in lower throughput. Compared with CAGRA, ALGAS reduces latency by 21.9%-35.4%, and improves throughput by 27.8%-55.2%.

B. Latency on Dynamic Batching

To evaluate the performance of dynamic batching, We collect and sort latency data for all queries to highlight the differences between the two batching modes, as shown in Fig. 13. It is evident that dynamic batching generally results in lower latency compared to static batch process. In dynamic batching, queries with faster response times can return results earlier.

C. Different Batch Size

Batch size impacts throughput and limits the maximum number of CTAs and shared memory available to each query, which in turn can increase the latency of individual queries.

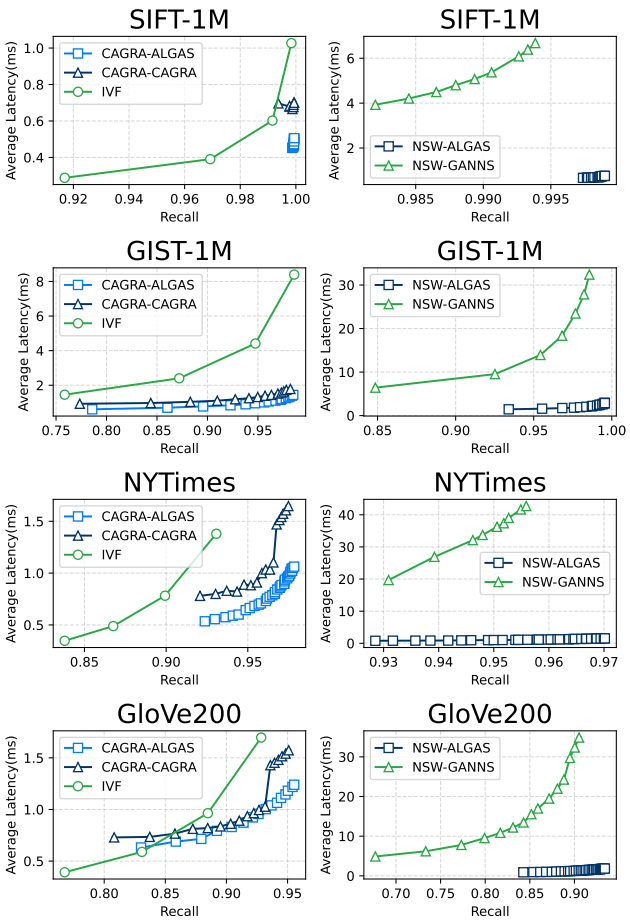

Fig. 10: Latency in Different Graphs and Methods

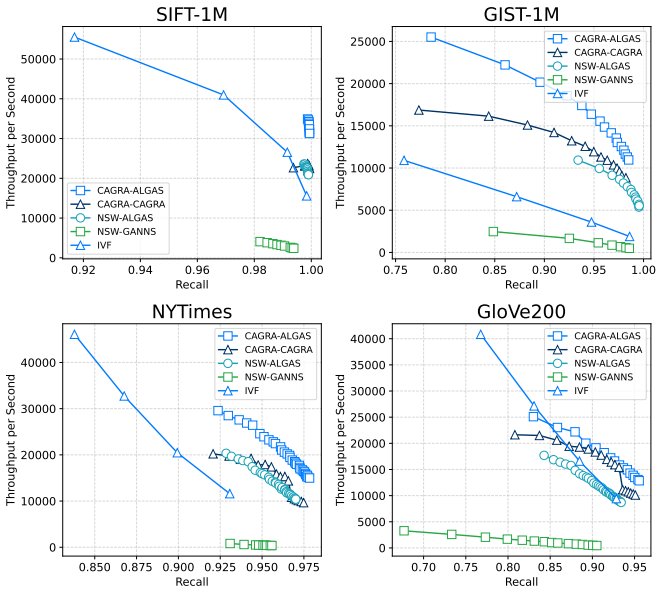

Fig. 11: Throughtput in Different Graphs and Methods

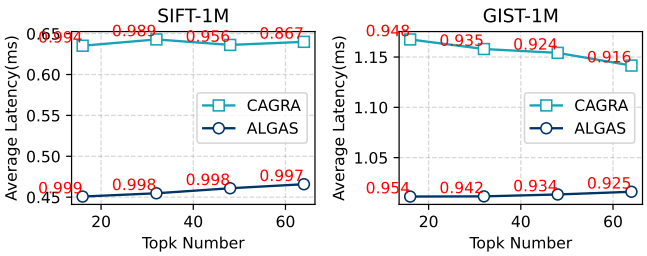

Fig. 12: Compare with Different Topk

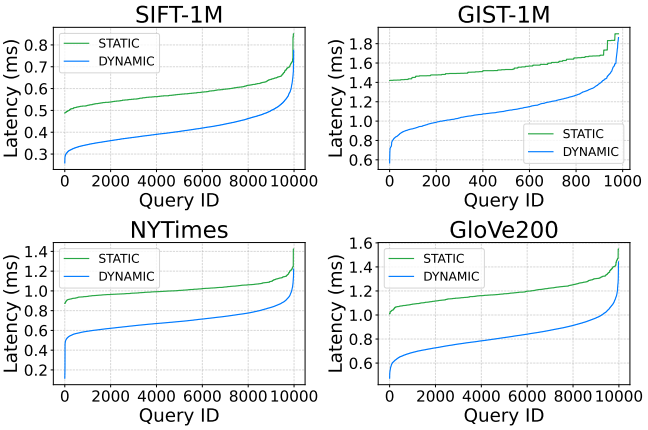

Fig. 13: Sorted Query Latency among Different Batch Mode

Batch size also affects data transfer over PCIe, with larger batch requiring more frequent data exchanges between the host and the GPU, such as dispatching queries and collecting results. In Fig. 14 and Fig. 15, we evaluate the performance of ALGAS under different batch sizes with fixed recall. ALGAS achieves an increase in throughput of up to 18.8% to 145.9% compared to CAGRA, along with latency reductions of approximately 17.7% to 61.8%.

D. Performance of Beam Extend

In Fig. 16, we utilize 8 CTAs in parallel to evaluate the performance of beam extend. "Greedy Extend" indicates that beam search is not employed. Utilizing beam extend to reduce the number of sorting operations in the later stages of the search significantly enhances throughput under high recall while also reducing the average latency. This effect arises from a decrease in the number of sorting operations needed in the later stages of the search. Fig. 17 demonstrates the percentage of time spent on sorting during the search process before and after the implementation of beam extend, indicating a reduction of approximately 14.2% to 25% in search time.

E. Performance of Parallel Processing on Host

In Fig. 18, we evaluate the performance of host parallel processing. SIFT-1M benefits from enhancements in the host thread. Due to its lower dimension, SIFT-1M experiences more frequent I/O operations, leading to more pronounced gains. Additionally, the implementation of local polling based

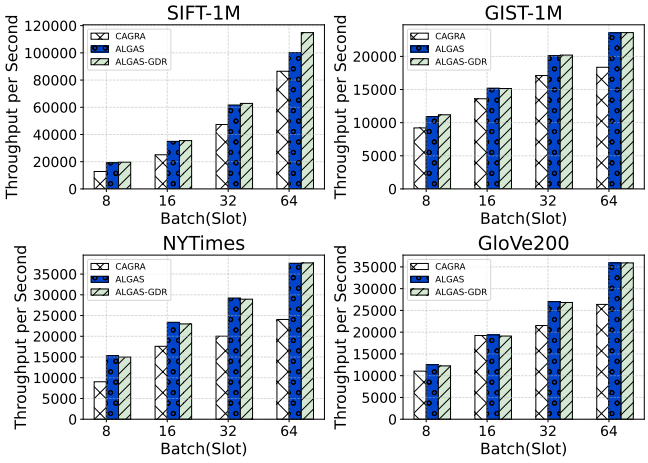

Fig. 14: Batch-Throughput among Different Methods

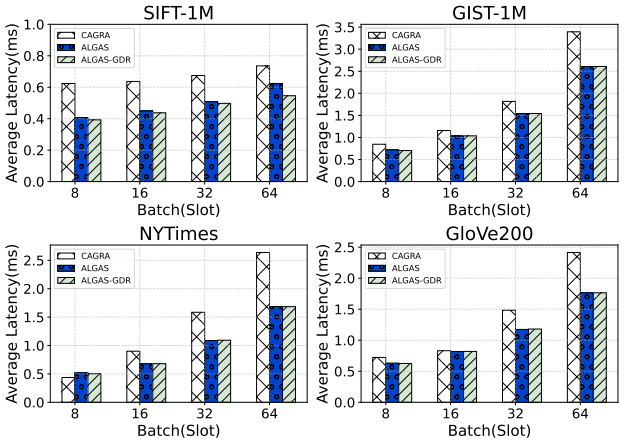

Fig. 15: Batch-Latency among Different Methods

on GDRCopy [27] utilizes less PCIe bandwidth, resulting in improved scalability.

VII. Related Works

ANNS can be broadly categorized into four types: **Tree-based Methods:** These methods organize data through multi-dimensional tree structures, making them suitable for low-dimensional data (e.g. [9]–[11]). **Locality-sensitive Hashing based Methods:** These methods use random projections to map high-dimensional data into multiple buckets, with a higher probability of similar data points being mapped to the same bucket (e.g. [7], [8]). **Quantization-based Methods:** These techniques compress high-dimensional vectors into lower-dimensional codewords, thereby reducing both data dimension and computational complexity (e.g. [12]–[14]). **Graph-based Methods:** These methods construct a graph structure among nodes, performing well in high-dimensional spaces and providing high search efficiency and recall rates (e.g. [15]–[20], [32]).

In GPU-accelerated graph ANNS, SONG [22] first introduces the GPU to speed up search processes based on graphs.

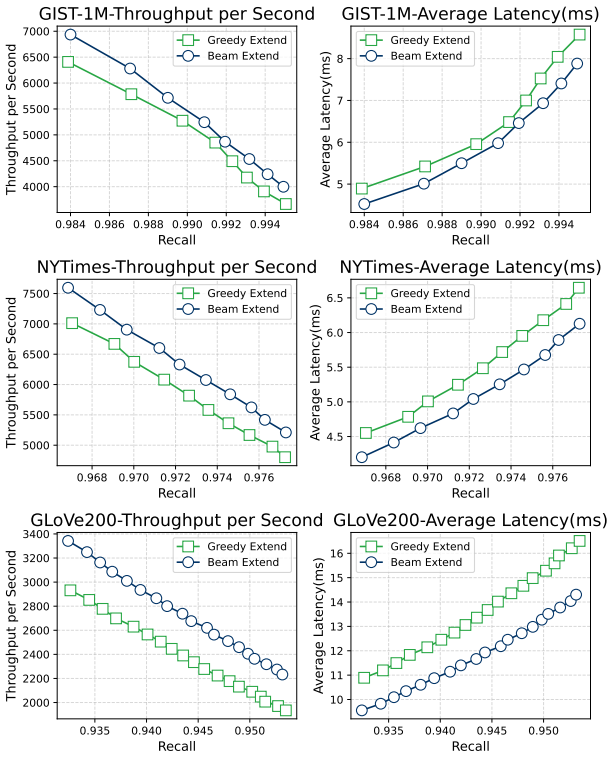

Fig. 16: Beam Extend in Different Datasets

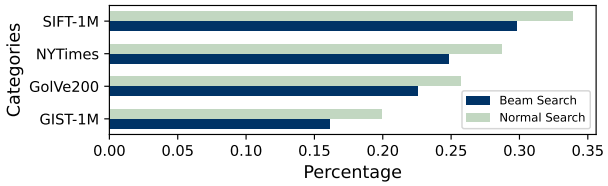

Fig. 17: Sorting Percentage After Beam Extend Optimization

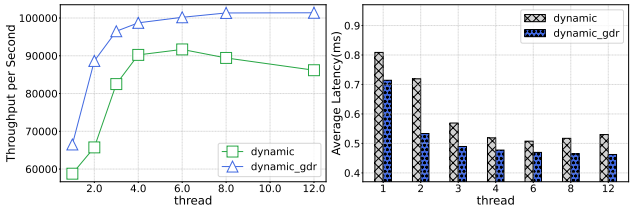

Fig. 18: Performance of Host Parallel Processing

Compared to the CPU method, SONG significantly improves throughput. However, it ignores the issues of latency and small batch. GANNS [23] eliminates storage costs associated with accessing metadata by introducing some redundant computations, and optimizes the operational overhead of graph ANNS data structures on GPU. Additionally, GANNS proposed a fast, GPU-friendly algorithm for constructing HNSW/NSW graphs [17], [18], taking full advantage of GPU parallelism for batch processing and merging, which significantly reduce construction time. GGNN [24] presents a GPU-based method for hierarchical graph construction and search, where sub-graphs are constructed in parallel by partitioning the dataset. The presence of sub-graphs enables concurrent searches across these sub-graphs. CAGRA [25] proposes a fixed out-degree graph optimized for GPU, implementing single-CTA and multi-CTA methods in the search algorithm to accommodate large batch and single query.

In disk-based implementations, notable examples include DiskANN [16] and SPANN [33]. For GPU-based implementations, there are quantization-based methods, including Faiss [21], Robustiq [34] Billion-Scale Similarity Search [35] and RUMMY [36]. RTNN [37] proposes to formulate neighbor search as a ray tracing problem. Bang [38] implements DiskANN on GPU. Furthermore, there are ANNS implementations based on other devices, including DF-GAS [39], which is based on FPGA, SmartANNs [40], which is based on SmartSSD and CXL-ANN [41], which utilizes CXL technology. VDTuner obtains appropriate index parameters through optimization methods [42].

VIII. Conclusion

In this paper, we propose a general graph-based Approximate Nearest Neighbor Search framework on GPU, named ALGAS. ALGAS addresses the issue of query bubble under small batch. By employing dynamic batching, it eliminates the bubble between batches and utilizes a carefully designed search algorithm to enhance performance for small batch. Our experiments demonstrate that ALGAS performs comparably to state-of-the-art industrial implementations across various small batch sizes. Compared to other works, ALGAS can reduce latency up to 21.9%-35.4%, throughput can increase up to 27.8%-55.2%.

IX. Acknowledgements

This research is supported in part by NSF of China (grant numbers, 62272252, 62272253) and the Fundamental Research Funds for Central Universities.

References

[1] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2614–2627.

[2] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," *The VLDB Journal*, pp. 1–25, 2024.

[3] R. Chen, B. Liu, H. Zhu, Y. Wang, Q. Li, B. Ma, Q. Hua, J. Jiang, Y. Xu, H. Deng *et al.*, "Approximate nearest neighbor search under neural similarity metric for large-scale recommendation," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 3013–3022.

[4] P. Li, W. Zhao, C. Wang, Q. Xia, A. Wu, and L. Peng, "Practice with graph-based ann algorithms on sparse data: Chi-square two-tower model, hnsw, sign cauchy projections," *arXiv preprint arXiv:2306.07607*, 2023.

[5] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[6] F. F. Xu, U. Alon, and G. Neubig, "Why do nearest neighbor language models work?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 38 325–38 341.

[7] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," *Advances in neural information processing systems*, vol. 28, 2015.

[8] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 950–961.

[9] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[10] M. E. Houle and M. Nett, "Rank-based similarity search: Reducing the dimensional dependence," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 1, pp. 136–150, 2014.

[11] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X.-S. Hua, "Trinary-projection trees for approximate nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 2, pp. 388–403, 2013.

[12] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 3887–3896.

[13] 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.

[14] Y. Chen, T. Guan, and C. Wang, "Approximate nearest neighbor search by residual vector quantization," *Sensors*, vol. 10, no. 12, pp. 11 259–11 273, 2010.

[15] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.

[16] 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," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[17] 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.

[18] 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.

[19] C. Fu and D. Cai, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv preprint arXiv:1609.07228*, 2016.

[20] 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, 2021.

[21] 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.

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

[23] 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 (ICDE)*. IEEE, 2022, pp. 552–564.

[24] 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.

[25] 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 (ICDE)*. IEEE, 2024, pp. 4236–4247.

[26] NVIDIA. (2007) Cuda toolkit. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[27] NVIDIA Corporation, "Magnum IO GDRCopy," https://developer.nvidia.com/gdrcopy, Sep. 2023, [Online; accessed 2023-10-04].

[28] NVIDIA, "Rtx 6000 ada generation graphics card," https://www.nvidia.com/en-us/design-visualization/rtx-6000/, 2024, accessed: October 4, 2024.

[29] "SIFT and GIST," 2010. [Online]. Available: http://corpus-texmex.irisa.fr/

[30] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

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

[32] P. Chen, W.-C. Chang, J.-Y. Jiang, H.-F. Yu, I. Dhillon, and C.-J. Hsieh, "Finger: Fast inference for graph-based approximate nearest neighbor search," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 3225–3235.

[33] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *Advances in Neural Information Processing Systems*, vol. 34, pp. 5199–5212, 2021.

[34] W. Chen, J. Chen, F. Zou, Y.-F. Li, P. Lu, and W. Zhao, "Robustiq: A robust ann search method for billion-scale similarity search on gpus," in *Proceedings of the 2019 on international conference on multimedia retrieval*, 2019, pp. 132–140.

[35] 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, 2021.

[36] Z. Zhang, F. Liu, G. Huang, X. Liu, and X. Jin, "Fast vector query processing for large datasets beyond {GPU} memory with reordered pipelining," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 23–40.

[37] 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.

[38] S. Khan, S. Singh, H. V. Simhadri, J. Vedurada *et al.*, "Bang: Billion-scale approximate nearest neighbor search using a single gpu," *arXiv preprint arXiv:2401.11324*, 2024.

[39] S. Zeng, Z. Zhu, J. Liu, H. Zhang, G. Dai, Z. Zhou, S. Li, X. Ning, Y. Xie, H. Yang *et al.*, "Df-gas: a distributed fpga-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 283–296.

[40] B. Tian, H. Liu, Z. Duan, X. Liao, H. Jin, and Y. Zhang, "Scalable billion-point approximate nearest neighbor search using {SmartSSDs}," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 1135–1150.

[41] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "{CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 585–600.

[42] T. Yang, W. Hu, W. Peng, Y. Li, J. Li, G. Wang, and X. Liu, "Vdtuner: Automated performance tuning for vector data management systems," *arXiv preprint arXiv:2404.10413*, 2024.