An Experimental Study of GPU-Based Graph ANN Search Algorithms

Yinzuo Jiang and Shimin Chen*
SKLP and Center for Advanced Computer Systems, ICT, CAS
University of Chinese Academy of Sciences
{jiangyinzuo22s, chensm}@ict.ac.cn

***Abstract*—Approximate nearest neighbor search (ANNS) is the key operation in vector databases. Graph-based ANNS algorithms are among the best-performing and most used methods on CPUs. However, the graph search procedures are challenging to optimize on GPUs. Existing GPU-based algorithms propose a variety of designs. In this paper, we conduct an experimental study of five representative GPU-based graph ANNS algorithms (i.e., SONG, CUHNSW, GANNS, GGNN, CAGRA) in order to compare their overall performance and understand the impact of the different design aspects.**

***Index Terms*—ANN search, GPU, graph index, evaluation**

I. Introduction

Vector databases are widely used in image search, recommendation systems, and recently in retrieval augmented generation (RAG) for large language models (LLMs). The key operation in vector databases is approximate nearest neighbor search (ANNS). Given a query vector q, ANNS retrieves the (approximate) k nearest neighbors (kNN) of q in the vector database $D = \{d_1, d_2, \ldots, d_n\}$. The database D typically contains millions to even billions of vectors. Each vector consists of hundreds to thousands of floating point features. Therefore, a brute-force method to obtain the accurate kNN results can be prohibitively expensive. Since approximate answers often suffice for many applications, industry and academia mainly focus on ANNS as a feasible solution to the kNN problem.

Existing ANNS methods build a variety of structures to reduce the kNN search cost, including hashing-based [1]–[3], tree-based [4]–[6], quantization-based [7]–[9], and graph-based structures [10]–[13]. Graph-based ANNS algorithms have been shown to be among the best-performing ANNS methods on CPUs. They construct a proximity graph on the vectors in D. Each vertex v represents a vector d_v. A directed edge (u, v) either connects two vectors that are close or provides properties of the small world for accelerating the graph search [10]–[13]. Given a query q, ANNS starts from a set of entry vertices and performs a variant of A* heuristic search on the graph to obtain the kNN results.

Recent studies exploit GPUs for accelerating graph-based ANNS algorithms [14]–[18]. Compared to CPUs, GPUs support higher levels of parallelism and enjoy higher memory bandwidth. For example, an NVIDIA A100 GPU consists of 108 Streaming Multiprocessors (SMs). Each SM contains 64 CUDA cores. The bandwidth of the GPU's device memory (a.k.a. global memory) is up to 1555 GB/s. Existing studies show that distance computation can be easily parallelized with GPUs. However, it is more challenging to exploit GPUs' Single Instruction Multiple Threads (SIMT) programming model to optimize the graph search procedures. Therefore, existing algorithms propose a variety of designs.

In this paper, we conduct a performance study of five representative GPU-based graph ANNS algorithms (i.e., SONG [14], CUHNSW [15], GANNS [16], GGNN [17], CAGRA [18]) using four representative vector data sets. Our goal is to compare their overall performance and understand the performance impact of their different designs. To our knowledge, this is the first systematic evaluation study for the graph ANNS algorithms on GPUs. Our code is publicly available at https://github.com/schencoding/gpu-graph-anns.

The rest of the paper is organized as follows. Section II reviews the general framework of the graph-based ANNS algorithms, and compares the five different algorithms in this study. Then, Section III describes the experimental methodology. After that, Section IV presents our evaluation study. Finally, Section V concludes the paper.

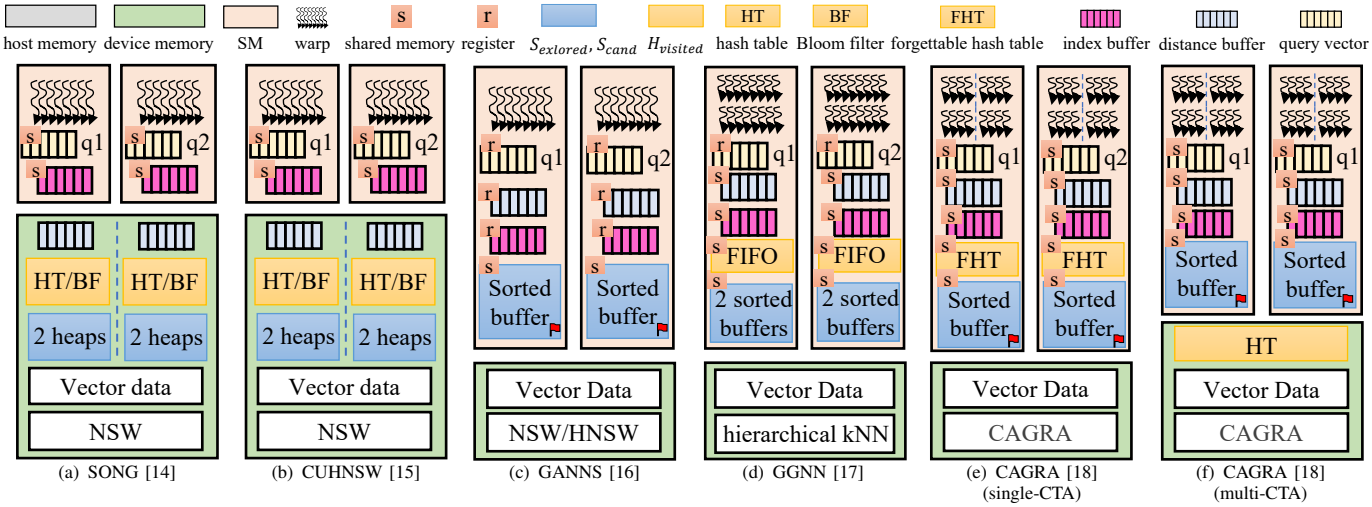
II. GPU-Based Graph ANN Search Algorithms

In this section, we first describe the general framework of the graph-based ANNS algorithms, then compare the designs of five state-of-the-art GPU-based graph ANNS algorithms.

Graph-Based ANNS. Algorithm 1 lists the pseudo code of the parallel graph-based ANNS algorithm. Given the vector data set D and a batch of query vectors Q, the algorithm searches the proximity graph G to find the approximate kNN for every query vector q. Popular distance metrics include the Euclidean distance, angular distance, inner product, etc.

The algorithm employs the following main data structures. First, S_{cand} contains visited vertices, while $S_{explored}$ contains vertices that have been explored. A vertex v is visited if distance(v, q) has been computed. A vertex u is explored if all its neighbors have been visited. S_{cand} and $S_{explored}$ store both the vertex IDs and the corresponding computed distances. The *popTopK* procedure returns a given number of entries with the smallest distances. The *addAndEvict* procedure inserts new entries. Since GPU-based algorithms often allocate fixed space for the two structures, *addAndEvict* removes entries with the largest distances if the allocated space is full. Second, $H_{visited}$ records the information of visited vertices for the purpose of

*Shimin Chen is the corresponding author.



(a) SONG [14] (b) CUHNSW [15] (c) GANNS [16] (d) GGNN [17] (e) CAGRA [18] (single-CTA) (f) CAGRA [18] (multi-CTA)

Fig. 1. GPU-based graph ANNS algorithms.

Algorithm 1: Parallel graph-based ANNS.

Input: A proximity graph G, vector dataset D, the batch of query vectors Q, the number of nearest neighbors to find k

Output: top-k approximate nearest neighbors for every query $(R_1, R_2, ..., R_{|Q|})$

1 **parallel for** q in Q **do**
2 $S_{cand} \leftarrow \emptyset$; $S_{explored} \leftarrow \emptyset$;
3 $H_{visited} \leftarrow P_{entry}$;
4 $Dist_{entry} \leftarrow$ parallelComputeDistance(P_{entry}, q, D);
5 S_{cand}.addAndEvict$(P_{entry}, Dist_{entry})$;
6 **repeat** sd times **do**
7 **if** satisfyStopCriterion$(S_{cand}, S_{explored})$ **then**
8 **break**
9 $(P_{cur}, Dist_{cur}) \leftarrow S_{cand}$.popTopK$(sw)$;
10 $S_{explored}$.addAndEvict$(P_{cur}, Disk_{cur})$;
11 $P_{nb} \leftarrow \emptyset$;
12 **parallel for** v in P_{cur} **do**
13 **parallel for** u in G.neighbors(v) **do**
14 **if** $u \notin H_{visited}$ **then**
15 $H_{visited}$.insert(u);
16 $P_{nb} \leftarrow P_{nb} \cup \{u\}$;
17 $Dist_{nb} \leftarrow$ parallelComputeDistance(P_{nb}, q, D);
18 S_{cand}.addAndEvict$(P_{nb}, Disk_{nb})$;
19 $(R_q, Dist_q) \leftarrow S_{explored}$.popTopK$(k)$;
20 **return** $(R_1, R_2, ..., R_{|Q|})$;

avoiding redundant distance computation. Third, the vertices in P_{entry} are the starting points for the graph search. Finally, $Dist$ buffers are temporary buffers for distance computation.

The algorithm performs each query in the query batch Q in parallel (Line 1). Note that all data structures in the algorithm except G, D, Q, and P_{entry} are private in the processing of a query. For each query q, S_{cand} and $H_{visited}$ are initialized with the entry vertices in P_{entry}, and $S_{explored}$ starts with an empty set (Line 2–5). Then, the algorithm goes into the main loop to perform at most sd iterations of exploration (Line 6). The loop can stop early if certain stop criterion is satisfied (Line 7–8). Each iteration pops sw vertices with the smallest distances from S_{cand} (Line 9), move them to $S_{explored}$ (Line 10), and explores the vertices in parallel (Line 12–18). The algorithm avoids visiting the same vertex multiple times by checking the vertex about to visit against $H_{visited}$ (Line 14). The newly visited vertices are added to $H_{visited}$ and S_{cand}. Finally, the algorithm returns the top-k vertices with the smallest distances from $S_{explored}$ (Line 19– 20).

The capacity of S_{cand} and $S_{explored}$, the search depth sd, and the search width sw are all hyper-parameters. Tuning these parameters achieve different query throughput vs. recall trade-offs. Moreover, the stop criterion can also be tuned to achieve better recalls. For example, one way is to allocate k entries for S_{cand} and $S_{explored}$ and terminate if the minimal distance in S_{cand} is greater than the maximal distance in $S_{explored}$. To improve recalls, existing algorithms increase the capacity of the structures to be much larger than k [14]–[16], [18] or introduce slacks to the minimal and maximal distance comparison [17].

GPU-Based Graph ANNS Algorithms. In this paper, we study five state-of-the-art GPU-based graph ANNS algorithms, i.e., SONG [14], CUHNSW [15], GANNS [16], GGNN [17], and CAGRA [18], as depicted in Figure 1. All the algorithms follow the general framework in Algorithm 1. They perform the ANNS on a single GPU. The vector data set D and the graph G can fit into the GPU device memory. Table I shows the main different features of the five algorithms.

First, there are mainly two types of proximity graphs: variants of kNN graphs [11] and variants of NSW [12]. In a kNN graph, the edges connect a vertex to its k nearest neighbors. Efficient approximate methods are often employed to construct the kNN graphs quickly [11]. NSW [12] contains both short-range and long-range links to achieve the properties of the small world. HNSW [13] is a hierarchical version of NSW in the same spirit of the skip list. The GPU algorithms except CUHNSW ensure that the number of out-neighbors is fixed. In this way, the out-neighbors of vertices in G can be placed in an array and easily located using vector IDs.

TABLE I
DIFFERENCES OF GPU-BASED GRAPH ANNS ALGORITHMS.

Algorithm	Graph structure	Query parallelism	sw	S_{cand} and $S_{explored}$	$H_{visited}$
SONG [14]	NSW, fixed-degree	a thread block per query	sw=1	heaps in device memory, single-threaded	hash table or Bloom filter in device memory, single-threaded
CUHNSW [15]	HNSW, degrees are not fixed	a thread block per query	sw=1	heaps in device memory, single-threaded	hash table in device memory, single-threaded
GANNS [16]	NSW / HNSW, fixed-degree	a thread block per query	sw=1	bitonic sort / sorting-based merge on shared memory, multi-threaded	no check for visited, avoid the use of $H_{visited}$
GGNN [17]	hierarchical kNN, fixed-degree	a thread block per query	sw=1	parallel insert to a sorted ring-buffer in shared memory, multi-threaded	ring-buffer in shared memory, multi-threaded
CAGRA [18]	CAGRA (optimized kNN), fixed-degree	single-CTA: a thread block per query; multi-CTA: multiple thread blocks per query	$sw \geq 1$	warp-level bitonic sort / radix-based sort, multi-threaded	forgettable hash table in shared memory, multi-threaded

Second, all algorithms except CAGRA computes ANNS for a query using a thread block. All the threads in a thread block are run on a single SM on GPUs. In comparison, CAGRA supports two modes. The single-CTA mode processes a query with a single thread block, while the multi-CTA mode can exploit multiple thread blocks (and thus multiple SMs) for a query. Specifically, multi-CTA CAGRA explores multiple vertices using multiple thread blocks in parallel. The distance computation is typically parallelized with multiple threads in a thread block. To compute distance(v, q), every thread in the thread block computes a partial result on a disjoint subset of v's dimensions. Then, warp-level primitives are often used to aggregate the partial results into the full distance.

Third, the number of vertices to explore (sw) in each iteration of Algorithm 1 is different. all algorithms except CAGRA explores a single vertex at a time. In comparison, CAGRA can explore multiple vertices in parallel. sw can be tuned to achieve different query throughput and recalls.

Finally, the designs of S_{cand}, $S_{explored}$, and $H_{visited}$ are different. SONG and CUHNSW perform single-threaded accesses, while GANNS, GGNN, and CAGRA exploit all threads in the thread block(s) to process the structures.

For S_{cand} and $S_{explored}$, the algorithms either use heaps in single-threaded accesses or perform sorting or inserting in multi-threaded execution. SONG and GGNN implement S_{cand} and $S_{explored}$ as two disjoint data structures. The other algorithms store S_{cand} and $S_{explored}$ in the same structure, and use a flag to indicate the explored vertices (shown as the red flag in Figure 1).

For $H_{visited}$, the most common implementation is based on a hash table. Due to space and efficiency considerations, the algorithms often simplify the design. SONG considers removing entries from the hash table or employing a Bloom filter for the visited structure. CUHNSW allows a conflicting entry to replace an existing entry in the hash table. GANNS performs distance computation without checking the visited information. (It only checks if a vertex is already explored when it is inserted into $S_{explored}$.) CAGRA aims to fit the hash table into the shared memory in the single-CTA mode. When it is too full, the forgettable hash table is reset and reinitialized with entries in S_{cand} and $S_{explored}$. Note that removing entries from $H_{visited}$ may result in redundant distance computation, but does not impact the recall. In contrast, GGNN employs a ring buffer as $H_{visited}$ and scans all the items in the buffer in parallel for a $H_{visited}$ check.

III. EXPERIMENTAL METHODOLOGY

Machine Configuration. The experimental machine is equipped with two Intel(R) Xeon(R) E5-2640 v4 CPUs (2.40GHz, 10 cores / 20 threads, 25 MB L3 cache per CPU), 128 GB DDR4 memory, and a NVIDIA A100-PCIE-40GB GPU (108 SMs, 64 FP32 CUDA Cores per SM, 192 KB of combined shared memory and L1 data cache per SM, 40 MB L2 cache, 40 GB HBM2 global memory with 1555 GB/s memory bandwidth) [19]. The machine runs Ubuntu 22.04.5 LTS with Linux kernel 5.15.0-126-generic. All programs are compiled with GCC 11.4.0 and NVCC 11.8 with optimization flag -O2. The CUDA runtime version is 11.8.

Solutions to Compare. We compare five state-of-the-art GPU-based graph ANNS algorithms and four baseline algorithms[1]:

- *SONG* [14]: According to the SONG paper [14], we choose the best $H_{visited}$ design, i.e., a hash table with selected insertion and visited deletion optimizations.
- *CUHNSW* [15]: This is an open-source GPU-based implementation of HNSW. Its graph index format is compatible with the CPU-based hnswlib. As a result, the out-degree of vertices in its graph is not necessarily fixed.
- *GANNS* [16]: GANNS supports both NSW and HNSW graphs. According to the GANNS paper [16], we choose the better-performing NSW graph.
- *GGNN* [17]: We evaluate GGNN release_0.5.
- *CAGRA* [18]: We evaluate the CAGRA implementation in cuVS [20]. CAGRA automatically selects the single-CTA or the multi-CTA mode based on the query batch size and the capacity of $S_{explored}$.
- Three GPU-based baselines: We evaluate GPU-based *IVF Flat*, *IVF PQ* (with refine), and *Brute-force* algorithms

[1]Please note that CAGRA [18], the most recent GPU-based graph ANNS work, compares GGNN, GANNS, CAGRA, and CPU-based HNSW in the experiments. In comparison, we conduct a more extensive study with five more algorithms, i.e., SONG, CUHNSW, and three GPU-based baseline algorithms.

TABLE II
VECTOR DATA SETS USED IN THE EXPERIMENTS.

Name	Dim	# of Vectors	# of Queries	Distance
DEEP [24]	96	1,000,000	10,000	Euclidean
SIFT [25]	128	1,000,000	10,000	Euclidean
GloVe [26]	200	1,183,514	10,000	Angular
GIST [25]	960	1,000,000	1,000	Euclidean

in cuVS [20]. Besides graph-based algorithms, IVF algorithms, which divide vectors into K-means clusters, are also widely used. The IVF ANNS first finds $nprobe$ nearest cluster centroids, then computes the distances for all vectors in the corresponding clusters. IVF PQ employs product quantization [7] to accelerate distance computation. Note that the distance computation between a batch of query vectors and a set of vectors in the data set can be formulated as matrix multiplication operations [21] and accelerated by highly optimized GEMM libraries that leverage the GPU's tensor cores.

- One CPU baseline: We evaluate hnswlib v0.7.0 [22] as a representative CPU-based *HNSW* [13] algorithm. We run hnswlib with either 1 thread or 40 threads. Each thread processes a query in the query batch in parallel.

All the algorithms are integrated into the cuVS bench, an ANNS benchmarking framework in cuVS [20]. We use cuVS 24.12. We add instrumentation code to the algorithm implementations for performance analysis purposes. Moreover, for SONG, CUHNSW, and GANNS, we expose their internal hyper-parameters so that we can tune them to obtain throughput-recall curves. Furthermore, we find that SONG, CUHNSW, and GANNS do not cache the immutable data vectors or graph indices in the GPU, incurring repeated overhead for copying data to the GPU. Such operations are unnecessary and can be rectified. Therefore, we remove the redundant copy operations in our experiments for the sake of fairness of comparison.

Hyper-Parameters to Tune. We tune the following knobs as mentioned in Algorithm 1 to achieve different throughput-recall trade-offs: 1) search depth sd, 2) search width sw, 3) out-degree of the proximity graph, 4) capacity of $|S_{cand}|$ and $|S_{explored}|$, 5) slack factor τ in the stop criterion.

Data Sets. We use four representative vector data sets from the ANN-Benchmarks [23]. The details of the data sets are shown in Table II. We ensure that each data set fits into the GPU. (For DEEP, which contains 1 billion vectors, we use a subset of 1 million vectors so that the data set can fit into the GPU.) Moreover, CUHNSW [15] and CAGRA [18] do not support the angular (cosine) distance. We normalize vectors in GloVe and compute the Euclidean distance on the normalized vectors. All the data types in our experiments are 32-bit floats.

Questions to Answer. We aim to answer the following questions in the evaluation:

Q1: Which is the best GPU-based graph ANNS algorithm?
Q2: How do GPU-based Graph ANNS algorithms compare to the baseline algorithms?
Q3: What are the performance bottlenecks of the GPU-based graph ANNS algorithms?
Q4: What is the impact of different design features?

IV. PERFORMANCE EVALUATION

Overall Performance. Figure 2 compares all ANNS algorithms for retrieving top-10 nearest neighbors on the four data sets. We maximize the query batch size and thus the inter-query parallelism to obtain the best performance. The batch size is 10000 except GIST, whose test set contains 1000 queries. The x-axis varies the recall from 0.8 to 1.0, while the y-axis reports the query throughput in QPS (queries per second) in the logarithmic scale. For each algorithm, we adjust the hyper-parameters to obtain a set of (recall, QPS) points, then connect the Pareto-optimal points to form the curve. Therefore, the closer to the top-right corner the better.

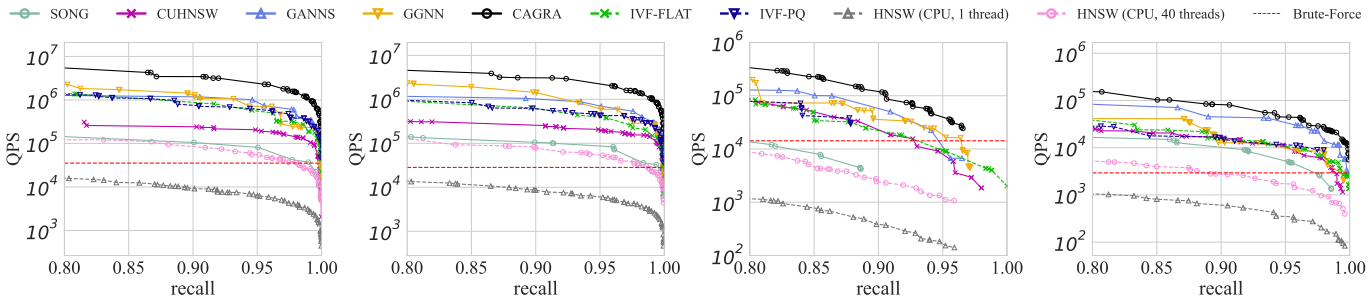
Among the four data sets, DEEP and SIFT are relatively simple. The recalls of a large number of points in Figures 2(a) and 2(b) are close to 1. GloVe requires a longer search time to achieve a high recall rate. GIST sees the highest vector dimensions (i.e., 960). As a result, the QPS on GloVe and GIST is relatively lower than that on DEEP and SIFT.

OB 1. CAGRA is the best GPU-based graph ANNS algorithm (Q1). Among GPU-based graph ANNS algorithms, CAGRA shows the best performance, followed by GANNS, GGNN, CUHNSW, and SONG. GANNS out-performs GGNN for GIST and is comparable to GGNN for the other data sets. They are clearly better than CUHNSW and SONG. SONG is the worst performing GPU-based graph ANNS algorithm.

OB 2. GPU-based IVF algorithms out-perform CUHNSW and SONG. Brute-force is competitive when the recall is close to 1 (Q2). The QPS of IVF algorithms is better than CUHNSW and SONG, and comparable to GANNS and GGNN (especially when the recall is above 0.95). Brute-force returns the accurate results, and hence its recall is 1. In the figures, we draw a dotted red line for brute-force. When the recall is close to 1, brute-force out-performs all other algorithms for GloVe, and is quite competitive for the other data sets. Both IVF and brute-force algorithms can use highly optimized matrix multiplication libraries for batch vector distance computations, thereby better utilizing the GPU's computation capability.

OB 3. CUHNSW out-performs CPU-based HNSW (Q2). Comparing the 40-thread and 1-thread HNSW curves, we see that multi-threading greatly improves the performance of CPU-based HNSW. Moreover, comparing CUHNSW and the 40-thread CPU-based HNSW, we see that the GPU implementation can significantly improve the performance of HNSW.

Time Breakdown. Figure 3 shows the time breakdown for the GPU-based graph ANNS algorithms on SIFT and GIST for the configurations obtaining recalls greater than and closest to 0.95. We measure the time for computing distances, accessing various data structures (i.e., S_{cand}+$S_{explored}$, $H_{visited}$, or other structures), and allocating memory in GPU. CUHNSW invokes a CUDA kernel to search each HNSW layer copied from the CPU. Therefore, CUHNSW has two more components (i.e., CPU and upper-layer search).



(a) DEEP, batch size = 10000 (b) SIFT, batch size = 10000 (c) GloVe, batch size = 10000 (d) GIST, batch size = 1000

Fig. 2. Overall performance for retrieving top-10 nearest neighbors.

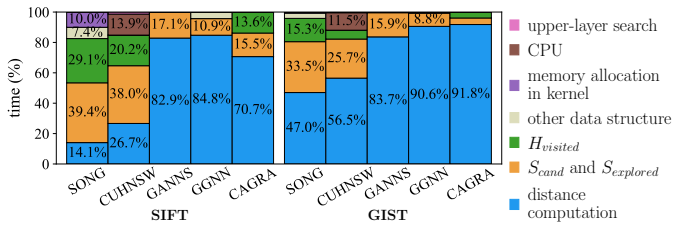


Fig. 3. Time breakdown of GPU-based ANNS algorithms.

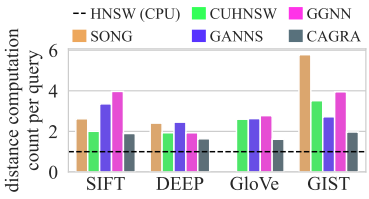


Fig. 4. Number of computed distances normalized to that of CPU-based HNSW.

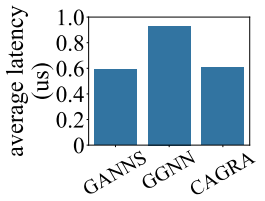


Fig. 5. Cost of accessing search data structures (SIFT).

OB 4. Single-threaded accesses to the graph search data structures can be a performance bottleneck (Q3,Q4). SONG and CUHNSW spend over 50% of the time in accessing S_{cand}+$S_{explored}$ and $H_{visited}$ on the SIFT data set. They perform single-threaded accesses to the graph search data structures. In comparison, GANNS, GGNN, and CAGRA all parallelize such accesses, and hence the components for accessing these data structures become much smaller.

OB 5. GANNS, GGNN, and CAGRA spend most of their time in distance computation (Q3). If we focus on the higher-performing algorithms (i.e., GANNS, GGNN, and CAGRA), distance computation takes over 70% of the total time. Moreover, as the number of feature dimensions increases (from 128 dimensions in SIFT to 960 dimensions in GIST), the algorithms spend higher fraction of their time in distance computation. Consequently, to achieve higher performance, it is important to reduce the number of computed distances.

Number of Computed Distances. Figure 4 reports the number of computed distances per query for the configurations obtaining recalls greater than and closest to 0.95. We normalize the number of distances of each point to that of CPU-based HNSW with the same recall. (Linear interpolation is used if the results of the CPU-based HNSW do not contain the exact recall.) Note that the number of computed distances is determined by the graph structure (e.g., the out-degree, the neighbor relationship), the search width, the stop criterion, and even the implementation of the search data structures (which may record only a subset of the visited information and hence only partially avoid redundant distance computations).

OB 6. CAGRA achieves the smallest number of computed distances per query among GPU-based graph ANNS algorithms. Combined with the time breakdown results, this observation explains why CAGRA out-performs the other GPU-based graph ANNS algorithms.

OB 7. The number of computed distances of CUHNSW is much worse than that of CPU-based HNSW. While CUHNSW's graph data structure is compatible with hnswlib, CUHNSW sees much higher number of computed distances per query. This indicates that either its constructed graph has lower quality, or the search data structures incur higher number of redundant distance computation (e.g., because $H_{visited}$ allows conflicting entries to replace existing entries).

Access Cost of Search Data Structures. Figure 5 compares the cost of accessing S_{cand}, $S_{explored}$, and $H_{visited}$ for the three higher-performing GPU-based graph ANNS algorithms. We divide the total time of accessing the structures by the number of computed distances, and report the average latency.

OB 8. CAGRA and GANNS have better implementations of the search structures than GGNN (Q4). As described in Table I, for $H_{visited}$, GGNN employs a ring-buffer and performs parallel checking for all the items in the ring-buffer, which incurs $O(|H_{visited}|)$ cost. In comparison, CAGRA employs a hash table with $O(1)$ cost and GANNS does not check the visited information. For S_{cand} and $S_{explored}$, GGNN inserts each point into a sorted buffer by moving the existing items and inserting the new item. In comparison, both GANNS and CAGRA sort all points visited in an iteration and merge them with existing items, thereby reducing the item moving cost.

Scalability Varying GPU Resources. Figure 6(a) compares the performance of GPU-based graph ANNS algorithms varying GPU resources on the SIFT data set. For each algorithm, we choose the setting that achieves the maximum QPS with recall $\geq$ 0.95 in Figure 2. We use NVIDIA Multi-Instance GPU (MIG) [19] to configure GPU instances with 1/7, 2/7, 3/7, 4/7, and 7/7 of the A100's resources. The y-axis reports QPS normalized to that with 1/7 resources.

OB 9. As the amount of GPU resources increase, GGNN, GANNS, and CAGRA show good scalability. Comparing the

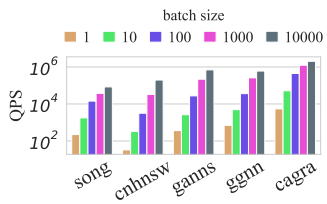


Fig. 7. Varying query batch size (SIFT).

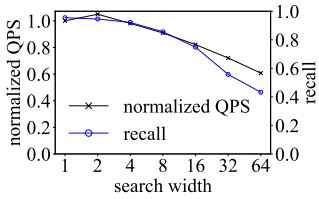


Fig. 8. Impact of search width in CAGRA (SIFT).

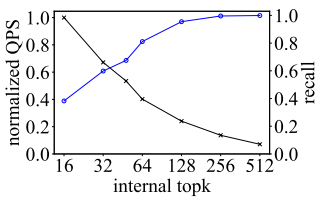


Fig. 9. Impact of capacity of S_{cand} and $S_{explored}$ in CAGRA (SIFT).

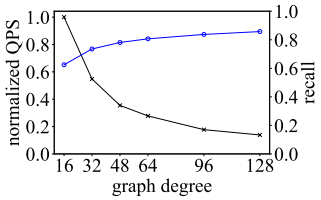
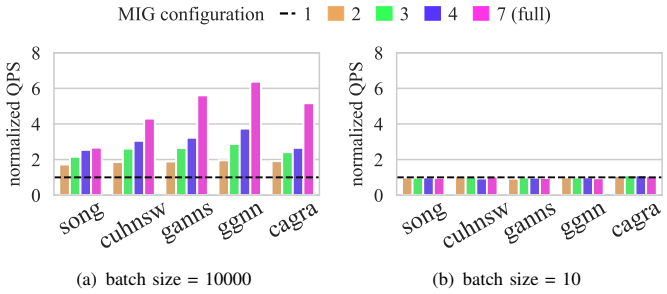


Fig. 10. Impact of out-degree in CAGRA (SIFT).



(a) batch size = 10000 (b) batch size = 10

Fig. 6. Varying GPU resources using MIG (SIFT).

cases with 1/7 and 7/7 resources, we see that GGNN, GANNS, CAGRA, CUHNSW, and SONG obtain 6.4x, 5.6x, 5.2x, 4.3x, 2.7x speedups, respectively. The three higher-performing algorithms can effectively utilize the increasing hardware resources with large query batches.

Impact of Query Batch Size. Figure 7 compares the performance of GPU-based graph ANNS algorithms varying the query batch size on the SIFT data set. We vary the size of each query batch from 1 to 10000. For example, when the query batch size is 10, we perform 1000 batches of queries. We report the maximum QPS for configurations whose recalls are greater than 0.95. Moreover, Figure 6(b) varies the GPU resources when the batch size is 10.

OB 10. GPU-based graph ANNS algorithms exhibit drastic performance drops at small query batch sizes. The multi-CTA mode helps reduce the performance loss of CAGRA (Q4). Comparing 10 and 10000 batch sizes in Figure 7, the higher-performing algorithms, GANNS, GGNN, CAGRA suffer performance drops of 266x, 121x, and 40x, respectively. Comparing 1 and 10000 batch sizes, GANNS, GGNN, CAGRA slows down by 1921x, 863x, and 373x, respectively. As described in Table I, most graph ANNS algorithms run a query per thread block. A100 supports up to 32 thread blocks per SM, and 3456 thread blocks for 108 SMs in total. When the query batch size is 1–1000, there are fewer numbers of thread blocks than the maximum capacity. As a result, it is more difficult to hide the latency to access the global memory (e.g., for reading the vector data for distance computation). In comparison, CAGRA automatically switches to the multi-CTA mode for small query batch sizes, which employs multiple thread blocks per query, alleviating this problem.

OB 11. The multi-CTA mode of CAGRA cannot fully utilize the increasing GPU resources (Q4). As shown in Figure 6(b), we see that all algorithms, including CAGRA, show poor scalability when the query batch size is 10. Even the multi-CTA design falls short of fully exploiting the GPU resources.

Impact of Hyper-Parameters. We focus on the best-performing algorithm, CAGRA, to understand the performance impact of hyper parameters. Figures 8, 9, and 10 show the QPS and the recall of CAGRA while varying the search width, the internal top-k size (which is the capacity of S_{cand} and $S_{explored}$), and out-degree of the proximity graph. Each point in the figure is the average across all the configurations with the same hyper-parameter. The QPS is further normalized to the one at the lowest x-value.

OB 12. QPS and recall tend to grow in reverse directions for most hyper parameters except search width. High search widths cause poor QPS and recalls (Q4). In Figures 9 and 10, increasing the hyper-parameter leads to higher number of visited vertices. Therefore, the QPS decreases while the recall increases. This trend exists for most other hyper-parameters (whose figures are omitted for space reasons.) In contrast, in Figures 8, we see that high search widths lead to both poor QPS and poor recalls. The best performance appear at around 1 and 2. That is, it is not beneficial to explore more than 2 vertices in parallel in each iteration.

V. Conclusion

In this paper, we have evaluated five representative GPU-based graph ANNS algorithms (i.e., SONG, CUHNSW, GANNS, GGNN, CAGRA) using four vector data sets. We have obtained a number of observations from the experimental results to answer the four raised questions. Our observations can be summarized into the following guidelines for designing GPU-based graph ANNS algorithms:

- It is important to perform parallel accesses to the graph search data structures. Otherwise, this can become a performance bottleneck.
- After optimizing the search structure access cost, the main optimization goal is to reduce the number of computed distances per query.
- Hyper-parameters should be carefully selected. The search width should be set to at most 2.
- It is significant but challenging to better utilize the GPU resources for small query batches.

Acknowledgment

This work is partially supported by National Key R&D Program of China (2023YFB4503600) and Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

REFERENCES

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998* (J. S. Vitter, ed.), pp. 604–613, ACM, 1998.

[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, eds.), pp. 518–529, Morgan Kaufmann, 1999.

[3] A. Shrivastava and P. Li, "Fast near neighbor search in high-dimensional binary data," in *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I* (P. A. Flach, T. D. Bie, and N. Cristianini, eds.), vol. 7523 of *Lecture Notes in Computer Science*, pp. 474–489, Springer, 2012.

[4] D. A. White and R. C. Jain, "Similarity indexing with the ss-tree," in *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA* (S. Y. W. Su, ed.), pp. 516–523, IEEE Computer Society, 1996.

[5] C. Silpa-Anan and R. I. Hartley, "Optimised kd-trees for fast image descriptor matching," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*, IEEE Computer Society, 2008.

[6] P. Ram and K. Sinha, "Revisiting kd-tree for nearest neighbor search," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019* (A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, eds.), pp. 1378–1388, ACM, 2019.

[7] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, 2011.

[8] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *2013 IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA, June 23-28, 2013*, pp. 2946–2953, IEEE Computer Society, 2013.

[9] J. Paparrizos, I. Edian, C. Liu, A. J. Elmore, and M. J. Franklin, "Fast adaptive similarity search through variance-aware quantization," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pp. 2969–2983, IEEE, 2022.

[10] R. Paredes and E. Chávez, "Using the *k*-nearest neighbor graph for proximity searching in metric spaces," in *String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005, Proceedings* (M. P. Consens and G. Navarro, eds.), vol. 3772 of *Lecture Notes in Computer Science*, pp. 127–138, Springer, 2005.

[11] W. Dong, M. Charikar, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011* (S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, eds.), pp. 577–586, ACM, 2011.

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

[13] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, 2020.

[14] W. Zhao, S. Tan, and P. Li, "SONG: approximate nearest neighbor search on GPU," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pp. 1033–1044, IEEE, 2020.

[15] "Cuhnsw," 2021. [Online]. Available: https://github.com/js1010/cuhnsw.

[16] Y. Yu, D. Wen, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Gpu-accelerated proximity graph approximate nearest neighbor search and construction," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pp. 552–564, IEEE, 2022.

[17] F. Groh, L. Ruppert, P. Wieschollek, and H. P. A. Lensch, "GGNN: graph-based GPU nearest neighbor search," *IEEE Trans. Big Data*, vol. 9, no. 1, pp. 267–279, 2023.

[18] 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 *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pp. 4236–4247, IEEE, 2024.

[19] "Nvidia a100 tensor core gpu architecture," pp. 43–43, 2025. [Online]. Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[20] "cuvs: Vector search and clustering on the gpu," 2025. [Online]. https://github.com/rapidsai/cuvs.

[21] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Trans. Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[22] "hnswlib." https://github.com/nmslib/hnswlib.

[23] M. Aumüller, E. Bernhardsson, and A. J. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Inf. Syst.*, vol. 87, 2020.

[24] A. Babenko and V. S. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 2055–2063, IEEE Computer Society, 2016.

[25] "Datasets for approximate nearest neighbor search," 2025. [Online]. Available: http://corpus-texmex.irisa.fr/.

[26] 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, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (A. Moschitti, B. Pang, and W. Daelemans, eds.), pp. 1532–1543, ACL, 2014.