CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs

1st Hiroyuki Ootomo
NVIDIA
Tokyo, Japan
ORCID: 0000-0002-9522-3789

2nd Akira Naruse
NVIDIA
Tokyo, Japan
ORCID: 0000-0002-3140-0854

3rd Corey Nolet
NVIDIA
Maryland, USA
ORCID: 0000-0002-2117-7636

4th Ray Wang
NVIDIA
Shanghai, China
ORCID: 0000-0001-8982-0571

5th Tamas Feher
NVIDIA
Munich, Germany
ORCID: 0000-0003-2095-4349

6th Yong Wang
NVIDIA
Shanghai, China
ORCID: 0009-0005-0906-8778

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00323

***Abstract*—Approximate Nearest Neighbor Search (ANNS) plays a critical role in various disciplines spanning data mining and artificial intelligence, from information retrieval and computer vision to natural language processing and recommender systems. Data volumes have soared in recent years and the computational cost of an exhaustive exact nearest neighbor search is often prohibitive, necessitating the adoption of approximate techniques. The balanced performance and recall of graph-based approaches have more recently garnered significant attention in ANNS algorithms, however, only a few studies have explored harnessing the power of GPUs and multi-core processors despite the widespread use of massively parallel and general-purpose computing. To bridge this gap, we introduce a novel parallel computing hardware-based proximity graph and search algorithm. By leveraging the high-performance capabilities of modern hardware, our approach achieves remarkable efficiency gains. In particular, our method surpasses existing CPU and GPU-based methods in constructing the proximity graph, demonstrating higher throughput in both large- and small-batch searches while maintaining compatible accuracy. In graph construction time, our method, CAGRA, is 2.2–$27\times$ faster than HNSW, which is one of the CPU SOTA implementations. In large-batch query throughput in the 90% to 95% recall range, our method is 33–$77\times$ faster than HNSW, and is 3.8–$8.8\times$ faster than the SOTA implementations for GPU. For a single query, our method is 3.4–$53\times$ faster than HNSW at 95% recall.**

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

I. Introduction

The importance of Approximate Nearest Neighbor Search (ANNS) has grown significantly with the increasing volume of data we encounter. ANNS is particularly relevant in solving the k Nearest Neighbor Search (k-NNS) problem, where we seek to find the k vectors closest to a given query vector, typically using a distance like the L2 norm, from a dataset of vectors. The simplest exact solution for k-NNS involves exhaustively calculating the distance between the query vector and all vectors in the dataset, then outputting the k vectors of the smallest distances (top-k) as results. However, this approach becomes infeasible for large datasets due to the sheer number of similarity computations required, making scalability an issue. In many practical applications, exact results are not always necessary and ANNS can strike a balance between throughput and accuracy, reducing the computational burden and enabling the scaling to large datasets. The impact and use-cases for ANNS are widespread and include several disciplines spanning data mining and artificial intelligence, such as language models in natural language processing [14], [30], computer vision [2], [13], information retrieval [17], recommender systems, and advertising [3], [15]. ANNS also forms the core of many important classes of data science and machine learning algorithms such as clustering [20], classification [21], manifold learning and dimensionality reduction [22]. Various different categories of algorithms for ANNS have been proposed and are well-studied, including hashing-based [4], tree-based [25], quantization-based [12], and graph-based methods.

The graph-based method for ANNS relies on a *proximity graph*, or a graph that represents the similarity relationships among data points within a dataset. Graph-based methods involve two primary steps: constructing a proximity graph from a dataset and traversing it to find the k closest nodes to the input query. The question of determining the optimal proximity graph structure is not easily answered theoretically. For instance, the graph quality as a k-nearest neighbor graph does not necessarily guarantee higher accuracy [29]. Therefore, researchers have focused on optimizing the graph's efficiency and structure heuristically and empirically. One notable approach is the NSW (Navigable Small World) graph proposed by Malkov et al. [19], which leverages the Small World phenomenon [27] to enhance the search performance of the proximity graph. Building upon this idea, HNSW (Hierarchical Navigable Small World) [18] graphs address issues present in NSW, where a few nodes have a large degree, hindering the high-performance search. HNSW addresses this problem by introducing a hierarchical graph structure and setting an upper bound on the maximum degree. Another approach is NSG (Navigating Spreading-out Graph), proposed by Fu et al. [8], which approximates a Monotonic Relative Neighborhood Graph (MRNG) structure to help guarantee low

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00323

search complexity. These are just a few examples of graph structures used in ANNS, and numerous other graphs have been well-studied in the field [29].

When it comes to implementing graph-based ANNS methods, few studies have introduced high-performance implementations optimized for data-center servers capable of harnessing the massive parallelism offered by GPUs. The memory bandwidth to load the dataset vectors can be a bottleneck of the search throughput, not only in graph-based, but also in other ANNS implementations. Since a GPU, typically has high memory bandwidth, it is potentially suitable for ANNS. One notable implementation is SONG [33], which stands as the first graph search implementation on GPUs, using various optimization techniques. These techniques include employing the open addressing hash table and performing multi-query searches within a warp. With these optimizations, SONG has higher throughput than IVFPQ on GPU included in FAISS [11] and HNSW on CPU. Similarly, GANNS is a GPU-friendly graph search and construction method tailored for NSW, HNSW, and k nearest neighbor graphs on GPUs [32]. This approach further advances the efficiency of graph-based ANNS on GPU architectures by modifying data structures for GPUs and reducing their operation time. Additionally, Groh et al. present GGNN, a fast graph construction and search method designed explicitly for GPUs [9]. Their work also improves the overall performance of ANNS on GPUs by improving data structures for GPUs and utilizing fast shared memory. Some studies have pointed out and managed the issue that graph construction can be time-consuming by using the advantage that a proximity graph can be reused once it is constructed. Unfortunately, there still remains a critical challenge in efficiently designing proximity graphs that are well-suited for GPU architectures in both construction and search. Despite the progress made in using GPUs for graph-based ANNS, most existing studies focus on adapting or optimizing existing graphs for GPU utilization rather than specifically designing proximity graphs from the ground up to fully leverage the GPU's capabilities in both graph construction and subsequent search operations.

This paper proposes 1) a proximity graph suitable for parallel computing in graph construction and query search and 2) a fast ANNS graph construction and search implementation, **CAGRA** (Cuda Anns GRAph-based), optimized for NVIDIA GPU. Our graph is search implementation-centric and designed to increase the efficiency of a massively parallel computing device, like the GPU, rather than theoretical graph quality.

The summary of our contributions is as follows:

- We propose a proximity graph for ANNS and its construction method suitable for massively parallel computing. This method reduces the memory footprint and usage, which can be the performance bottleneck in the graph optimization process, by avoiding exact similarity computation.
- We provide a search implementation optimized for GPU, which is designed to gain high throughput in both single and large-batch queries. We harness software warp splitting and forgettable hash table management to utilize the GPU resource efficiently.
- We demonstrate that CAGRA achieves a higher performance in graph construction and query than the state-of-the-art graph-based ANNS implementations for CPU and GPU. In graph construction time, CAGRA is 2.2–27× faster than HNSW. In large-batch query throughput in the 90% to 95% recall range, CAGRA is 33–77× faster than HNSW and is 3.8–8.8× faster than the SOTA implementations for GPU. For a single query, CAGRA is 3.4–53× faster than HNSW at 95% recall.

II. Background

A. Approximate Nearest Neighbor search

In the k-NNS problem, we obtain k vectors $x_{i_1}, x_{i_2}, \ldots, x_{i_k} \in \mathbb{R}^n$ that satisfy the following condition from a dataset $\mathcal{D} \in \mathbb{R}^{n \times N}$:

$$i_1, i_2, \ldots, i_k = k\text{-argmin}_i \, \mathrm{Distance}\left(x_i, q\right), \tag{1}$$

where $q \in \mathbb{R}^n$ is a given query vector, $k\text{-argmin}_i$ is the top-k arguments in ascending order, and $\mathrm{Distance}(\cdot)$ ($\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$) is a distance measure. The distance is typically the L2 norm or cosine similarity. Although k-ANNS obtains k similar vectors to a given query vector, the results are not always exact. We evaluate the accuracy of the results for a query as recall:

$$\mathrm{recall} = |U_{\mathrm{ANNS}} \cap U_{\mathrm{NNS}}| / |U_{\mathrm{NNS}}|, \tag{2}$$

where U_{ANNS} is the set of resulting vectors obtained by ANNS and U_{NNS} is by NNS. We denote the recall of k-ANNS as "recall@k". There is typically a trade-off between the recall and throughput (QPS; Query Per Second).

B. CUDA

The GPU, or Graphics Processing Unit, has been broadly used for general-purpose computing in recent years, whereas it was initially developed strictly for graphics processing. NVIDIA proposes and has been developing CUDA, which allows us to leverage the high-performance computing capability of the GPU for general-purpose computing. While the GPU has higher parallelism and memory bandwidth than the CPU, we have to be able to abstract parallelism from an algorithm and map it to the architecture of the GPU to leverage its high performance. Therefore, not all applications can gain higher performance by just using the GPU. We briefly introduce the architecture of NVIDIA GPU in the following sections, and further information on the architecture can be found in [23].

1) Thread hierarchy: In the NVIDIA GPU thread hierarchy, 32 threads in a group called "warp" execute the same instruction simultaneously. On the other hand, different instructions are not executed in a warp in parallel, leading to a performance degradation called "warp divergence". A group of up to 32 warps composes a CTA (Cooperative Thread Array), or thread block, and a CTA is executed on a single GPU streaming multiprocessor (SM). The SM is like a core in a multi-core CPU, and since there are many SMs on recent GPUs, we can launch and operate multiple CTAs at a time.

2) Memory hierarchy: In the memory hierarchy of the GPU, the device memory has the largest size, and all threads can access the same memory space. Shared memory, on the other hand, is a local memory used within each CTA and all the threads in the CTA share the memory space. While the size of shared memory is much smaller than the device memory, it has lower latency and higher bandwidth. Registers are local data storage for each thread and have lower latency than the shared memory.

C. Related work

Various algorithms for graph-based ANNS were introduced in Sec. I so we focus this section specifically on highly parallel and GPU-accelerated graph-based ANNS implementations whilst outlining how our contributions compare to them.

1) SONG: SONG is the first graph-based ANN implementation for GPU proposed by Zhao *et al.* [33]. Unlike CAGRA, this method does not contribute a faster graph construction technique and relies on other methods like NSW [19], NSG [8], and DPG [16]. SONG proposes a dataset and several optimizations for the GPU, in which they use open address hash table, bounded priority queue, and dynamic allocation reduction. They have achieved 10–$180\times$ speed up on the GPU compared to single-threaded HNSW on CPU.

2) GGNN: GGNN is a GPU-friendly implementation of graph-based ANNS proposed by Groh *et al.* [9] that, like CAGRA, provides both a high-throughput search implementation and a fast graph construction technique that can utilize high parallelism. GGNN was demonstrated to outperform SONG in large-batch searches.

3) GANNS: GANNS is also a GPU-accelerated graph construction and search method proposed by Yu *et al.* [32]. They propose a GPU-based NSW graph construction method and show that both the proximity graph construction and search performance are better than SONG.

Unfortunately, between GGNN and GANNS, it is unclear which is the state-of-the-art graph-based ANNS GPU implementation since there is no study on comparison between them. In addition, all of the above GPU implementations are focused on applications with a large number of queries. To the best of our knowledge, no GPU implementation outperforms the CPU implementation for applications with small-batch queries. In this paper, we will show that CAGRA outperforms both the CPU and GPU in graph construction and search.

III. CAGRA GRAPH

In this section, we explain the design and features of the CAGRA graph. While some graphs are designed to follow or approximate graphs with some theoretical properties, including monotonicity, the CAGRA graph is a search implementation-centric and heuristically optimized graph. The CAGRA graph has the following features:

- **Fixed out-degree** (d). By fixing the out-degree, we can utilize the massive parallelism of GPU effectively. Most graph-based ANNS algorithms build and utilize non-fixed out-degree graphs. In single-thread execution on a CPU, a non-fixed degree for each node has an advantage in that we can reduce the less important distance computation by keeping only essential edge connections. However, in the case of GPU, too small a degree doesn't fully saturate the computing resources allocated to each CTA, leading to lower hardware utilization. Rather, it is better to expand the search space using all the available compute resources, as it won't increase the overall compute time. Another advantage is that fixing the degree allows more uniform computation, thus creating less load imbalance during the parallel graph traversal phase, which would lead to low hardware utilization. We set the degree depending on the dataset and required recall and throughput.
- **Directional**. Since the out-degree is fixed, the graph is naturally directional.
- **No hierarchy**. HNSW, for instance, employs a hierarchical graph to determine the initial nodes on the bottom layer. However, in the case of GPU, we can obtain compatible initial nodes by randomly picking some nodes and comparing their distances to the query, thus employing the high parallelism and memory bandwidth of GPU. The detail of the search algorithm is in Sec. IV.

Two main steps are involved in the construction of the CAGRA graph: 1) building a k-NN graph and 2) optimizing it to enhance search recall and throughput. We chose k-NN graph as the base graph because the fixed out-degree graph is well-suited for efficient GPU operations, and we can rapidly build it using nearest neighbors descent (NN-descent) [29] even on the GPU [28]. The following section outlines the heuristic graph optimization approach and its parallel computation algorithm.

A. Graph optimization

The primary objective of CAGRA graph optimization is to enhance reachability among a large number of nodes. Reachability is characterized by two properties: 1) whether we can traverse from any arbitrary node to another arbitrary node and 2) the number of nodes we can traverse from one node within a specified number of path traversals.

To assess property 1), we measure the number of strongly connected components (CC) in the graph. The number of CC is determined as follows:

There is no guarantee that the base kNN graph is not disconnected [20], and the weak CC represents the number of subgraphs in the graph. Additionally, in the case of a directional edge graph, there may be scenarios where traversal from one node to another is not possible, even if the graph is not entirely disconnected. A graph is considered strongly connected when an arbitrary node in the graph can reach any other node. The number of strong CC is the count of node groups in the graph, where each group forms a strongly connected subgraph. A smaller number of strong CC are preferred because a larger number of CC can lead to more unreachable nodes starting from a search start node.

To assess property 2), we utilize the average 2-hop node count ($N_{2\text{hop}}$) for all nodes in the graph as the metric. The 2-hop node count of a given node is defined as the number

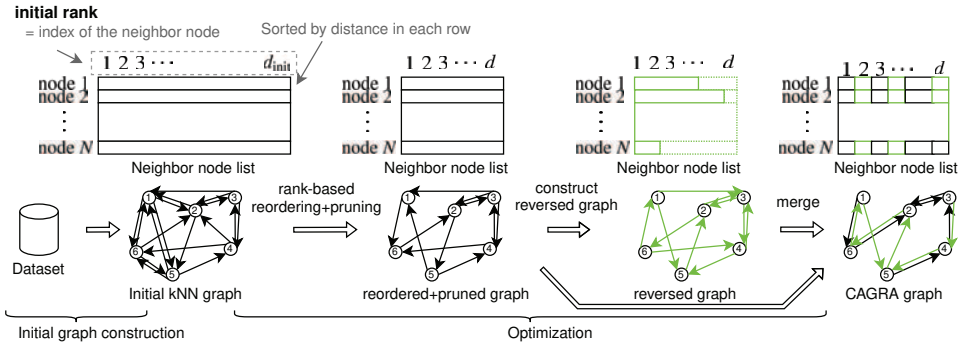

Fig. 1. Construction flow of the CAGRA graph.

of nodes that can be reached in two steps from the node. Its maximum value is determined as $N_{2\text{hop}}^{\max} = d + d^2$ where d is the degree of the graph. A higher average 2-hop node count indicates that more nodes can be explored during specific search iteration steps.

In the CAGRA graph optimization, two key techniques are employed on the initial kNN graph: **reordering** and **reverse edge addition**. Reordering is a technique that reorders each edge of the initial kNN graph in an order that increases the diversity of the graph, rather than in the order of its length, and has the primary effect of increasing 2-hop node counts. Reverse edge addition is a technique often used in other graph-based ANN implementations and improves node reachability while reducing strong CC values.

B. Graph construction and optimization algorithm

The CAGRA graph is a directed graph where the degree, d, of all nodes is the same. The construction of the graph consists of two stages: 1) construction of an initial graph and 2) optimization, as shown in Fig. 1.

1) Initial graph construction: We construct a k-NN graph as an initial graph where the degree of each node is $k = d_{\text{init}}$. We use NN-Descent [5] to construct the graph and will typically set d_{init} to be $2d$ or $3d$, where d is the degree of the final CAGRA graph. As a final step in this process, we sort the connected node list of each node in ascending order based on distance from the source node. This sorting operation can be efficiently executed in parallel using GPUs since no dependency exists in the computation for each individual node list. We assume that the initial k-NN graph has sufficient connectivity among nodes. And the goal of the CAGRA graph optimization is to reduce the degree of the graph to reduce the size while preserving the reachability.

2) Graph optimization: The optimization process involves two steps: 1) reordering edges, and 2) adding reverse edges. It takes the initial graph as input and produces the CAGRA graph as output. This process offers notable advantages: i) it no longer requires the dataset or distance computation, and ii) it allows for many computations to be executed in parallel without complex dependencies.

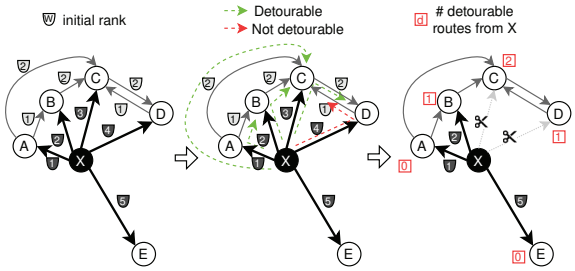

Fig. 2. CAGRA edge reordering and pruning flow. We assume pruning edges from the node X. **Left**: The initial rank of the edges from X and other related edges. **Middle**: Possible two-hop routes, classified as detourable and not detourable by Eq. 3. We use the rank instead of the distance. **Right**: The number of detourable routes of each node connected to X. The edges are discarded from the end of the list ordered by the number of detourable routes. In this example, the nodes A, B, and E are preserved as the neighbors of node X, although the node E is the farthest in the initial neighbors of node X in the distance.

In the reordering edges step, we determine the significance of an edge, **rank**, to prune the edges at the end of the entire optimization. Existing pruning algorithms prune an edge from one node to another if it can be bypassed using another route (detourable route) that satisfies certain criteria. For instance, in NGT [10], it defines the detourable route from X to Y as a pair of two edges as follows:

$$(e_{X\to Z}, e_{Z\to Y}) \text{ s.t. } \max(w_{X\to Z}, w_{Z\to Y}) < w_{X\to Y}, \quad (3)$$

where $e_{\cdot\to\cdot}$ and $w_{\cdot\to\cdot}$ denote a directed edge between two nodes and the distance, respectively, and Z is a node with a direct connection from X and a direct connection to Y. Based on this pruning, we consider two reordering techniques, **distance-based** and **rank-based** reordering, and adopt rank-based in the CAGRA graph optimization. While the complexity of both the reordering operations is $\mathcal{O}(Nd^3)$, the distance-based strategy requires distance calculations between one node and its neighbors.

In distance-based reordering, we first count the number of detourable routes for each edge and reorder the node list by the counts in ascending. A smaller number of detourable routes for an edge indicates that this edge is more important to keep the 2-hop node counts. Then, we set the position of an edge

in the node list as a rank of the edge, which is an indicator of the importance of the edge. The computation of distance-based reordering has high parallelism since we can count the detourable routes for each edge in parallel. However, we need to compute the distances on the fly during the operation or make a distance table before the operation, making this method impractical for a large dataset. In the former case, we need $N \times d_{\text{init}} \times (d_{\text{init}} - 1)$ distance computations, and in the latter case, we need a distance table with $N \times d_{\text{init}}$ entries on memory, where N is the size of the dataset.

For rank-based reordering, we set the position of the edge in the neighbor node list, which is sorted by distance at the end of the initial graph construction, as the initial rank, similar to distance-based reordering. Then, we reorder the edges in the same way as distance-based reordering, but we use the initial rank instead of the distance, as shown in Fig. 2. In other words, we approximate the distance by the initial rank. This approximation allows us not to compute the impractical amount of distance computations and not to store the large size of the distance table in memory. We set the order index of a node as the rank, the same as distance-based. We adopt rank-based reordering in the CAGRA graph optimization and only keep top d neighbors for each node (pruning).

After reordering, we create a reversed graph where all edges have opposite directions of the reordered and pruned graph. Since the number of incoming edges per node, or in-degree, is not fixed in the reordered graph, the out-degree of each node in the reversed graph is also not fixed. However, we set the upper bound of the degree to d because of an attribute of the next operation. And we make the reversed graph so that the reversed edges are sorted by the rank in the pruned graph in ascending order. It means "*Someone who considers you are more important is also more important to you*". The process of adding reverse edges has a complexity of $\mathcal{O}(Nd)$.

Finally, we merge the pruned graph and reversed graph. In this process, we basically take $d/2$ children for each parent node from each graph and interleave them. When the number of children for a parent node in the reversed edge graph is fewer than $d/2$, we compensate them by taking from the pruned graph.

To find an optimal d, we build graphs with different numbers, such as 32, 64, and 96, and measure their search performance. There is no deterministic way to find the parameter in a single shot since it depends on the dataset and user requirements. This is not unlike the hyper-parameters ofother methods, for example, the maximum out-degree of the HNSW (libhnsw) graph. Increasing the out-degree improves the recall while the search throughput degrades.

C. Evaluation of the CAGRA graph and optimization

This section reveals the following question:

Q-A1 How much do the CC and 2-hop node counts improve with the CAGRA graph optimization?

Q-A2 How fast is rank-based reordering compared to distance-based?

TABLE I
DATASETS USED IN THE EVALUATIONS

	Dim (n)	Size (N)	Data type	CAGRA graph degree (d)
SIFT-1M [1]	128	1M	float	32
GIST-1M [1]	960	1M	float	48
GloVe-200 [24]	200	1183514	float	80
NYTimes [1]	256	290K	float	64
DEEP-1M [2]	96	1M	float	32
DEEP-10M [2]	96	10M	float	32
DEEP-100M [31]	96	100M	float	32

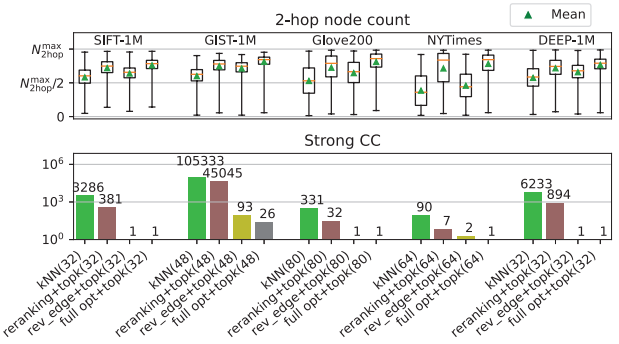

Fig. 3. The 2-hop node counts and strong CC comparison among a k-NN graph, partially and fully optimized graphs by CAGRA from an initial k-NN graph. The number in each bracket in the label is the degree of the graph (d), and we set the degree of the initial graph as $d_{\text{init}} = 3d$.

Q-A3 Does rank-based reordering have compatible recall with distance-based?

We use the datasets in Table I. All experiments are conducted on a DGX A100 server equipped with AMD EPYC 7742 CPU (64 cores) and NVIDIA A100 (80GB) GPU, high-end processors released within a similar timeframe. We put both the dataset and graph on the device memory of the GPU. In the CAGRA graph construction, we build an initial k-NN graph on GPU and optimize it on CPU.

1) **Q-A1:** *Connected components and 2-hop node count:* In the CAGRA graph construction, two optimizations are performed on the initial k-NN graph: reordering and reverse edge addition. Then, *how much effect does each optimization have?* To evaluate this, we have compared the properties of a standard k-NN graph, a partially optimized CAGRA graph (using only one optimization), and a fully optimized CAGRA graph (using both optimizations).

The results of the 2-hop node count and the strong CC experiments are shown in Fig. 3. In the case of the 2-hop node count, we observe that both optimizations increase the average 2-hop count, and the effect of the reordering is more significant compared to the reverse edge addition. The results also show that reverse edge addition significantly affects the strong CC more than reordering.

[1]Datasets for ANNS: http://corpus-texmex.irisa.fr/
[2]First 1M and 10M vectors of DEEP-100M dataset [31].

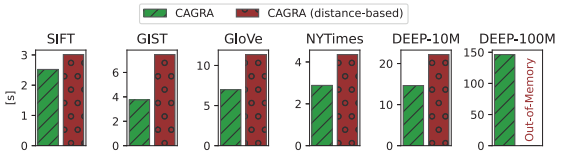

Fig. 4. CAGRA graph optimization time comparison with rank- and distance-based reordering.

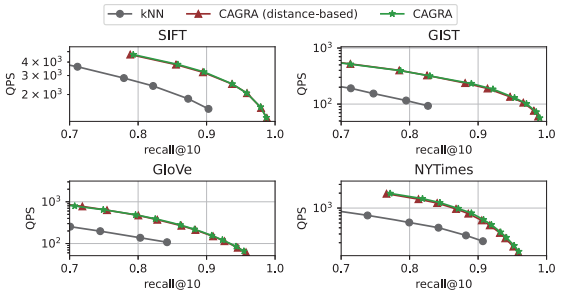

Fig. 5. CAGRA search performance comparison between the graphs optimized by rank- and distance-based reordering. CAGRA performs rank-based reordering while CAGRA (distance-based) performs distance-based.

2) **Q-A2:** *The reordering method's advantage on compute time:* In the reordering optimization, we avoid the distance computation altogether, reducing the computational overhead, and leading to faster optimization. So then, *how does that improve the total optimization time?* We have measured the optimization time, as shown in Fig. 4. The rank-based CAGRA optimization is faster than the distance-based for all datasets by as much as $1.9\times$. Furthermore, while we were still able to perform the rank-based optimization, we experienced an out-of-memory error that prevented us from performing the distance-based optimization on DEEP-100M. These results show that rank-based optimization is faster than distance-based and supports larger datasets.

3) **Q-A3:** *Seach performance comparison to distance-based optimization:* The recall that a graph can potentially achieve and the number of iterations to obtain a specific recall will vary by the graph construction methods, including the reordering priority criteria in the CAGRA graph optimization. In CAGRA, we reduce the graph optimization time, avoiding distance computation and instead using rank as the priority criteria. So then, *does the CAGRA graph have the compatible search performance compared to distance-based optimization?* To answer this question, we have tested both rank-based and distance-based reordering during CAGRA graph optimization and measured the throughput and recall of a query search process using the graph, as shown in Fig. 5. This confirms the recall-throughput balance is almost the same while the rank-based graph construction time is shorter, as demonstrated in Q-A2.

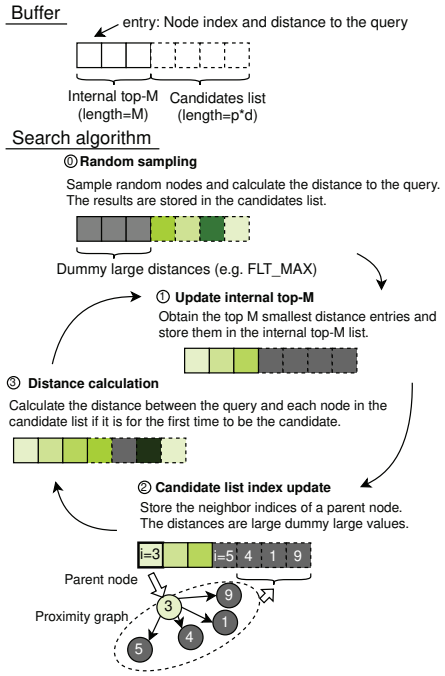

Fig. 6. *Top*: The buffer layout used in the CAGRA search. *Bottom*: The algorithm of the CAGRA search.

IV. CAGRA Search

In this section, we explain CAGRA's search algorithm and how we map it onto the GPU.

A. Algorithm

The CAGRA search algorithm uses a sequential memory buffer consisting of an **internal top-M list** (typically known as a priority queue in other algorithms) and its **candidate list**, as shown at the top of Fig. 6. The length of the internal top-M list is $M(\geq k)$, and the candidate list is $p \times d$, where p is the number of source nodes of the graph traversed in each iteration, and d is the degree of the CAGRA graph. Each buffer element is a key/value pair containing a node index and the corresponding distance between the node and the query.

As shown at the bottom of Fig. 6, The search calculation is as follows:

⓪ **Random sampling** (initialization step): We choose $p \times d$ uniformly random node indices using a pseudo-random number generator and compute the distance between each node and the query. The results are stored in the candidate list. We set the internal top-M list with dummy entries where the distance values are large enough to be the last in the next ① sorting process. For instance, if the distance is stored in `float` data type, `FLT_MAX`.

① **Internal top-M list update**: We pick up top-M nodes with the smallest distance in the entire buffer and store the results in the internal top-M list.

② **Candidate list index update** (graph traversal step): We pick up all neighbor indices of the top-p nodes in the

internal top-M list where they have not been parents. The result node indices are stored in the candidate list. This step does not calculate the distance between each node in the candidate list and the query.

③ **Distance calculation**: We calculate the distance between each node in the candidate list and the query only if this is the node's first time being in the candidate list for the query. This conditional branch prunes unnecessary computations since distances don't need to be recomputed if they were already computed in a previous iteration. For instance, if a node has already been in the list and the distance is

- small enough to stay in the top-M list, then it should already be in the list.
- large enough not to be in the top-M list, then it should not be added again.

We process ① $\sim$ ③ iteratively until the index numbers in the top-M list converge, meaning they remain unchanged from the previous iteration. Finally, we output the top-k entries of the internal top-M list as the result of ANNS.

B. Elemental technologies and designs

This section explains the elemental technologies and designs we use in the CAGRA search implementation on GPU.

1) Warp splitting: As described in Sec. II-B1, a warp consists of 32 threads that execute the same instruction simultaneously, representing the smallest parallel thread group in the hardware. In the CAGRA search implementation, we introduce a software-level division of the warp into even smaller thread groups, referred to as **teams**. Each team consists of a specific number of threads, which we term the **team size**.

This division allows us to enhance GPU utilization for the following reasons: Consider the latency of device memory load, where a 128-bit load instruction is the most efficient. We typically map one distance computation to one warp and utilize warp shuffle instructions to compute it across the threads collaboratively. However, when the dataset dimension is 96, and the data type is `float` (4-byte), the total bit length of a dataset vector is 3072 bits, which is smaller than the bit length loaded when all 32 threads in a warp issue the 128-bit load instruction, 4096 bits. Consequently, this will leave some threads in the warp which do not issue the load instruction, resulting in inefficient GPU usage. Now consider mapping one distance computation to a team with a team size of 8, and one team can load 1024 bits in one instruction. This allows the entire vector to be loaded by repeating the loading three times in all threads of the team. Additionally, the other teams within the same warp can calculate the distances between the query and the other nodes in the candidate list, thereby maximizing GPU utilization and efficiency. Although we split the warp into teams in software, we don't encounter warp divergence since all of the teams in each warp still execute the same instructions.

2) Top-M calculation: In ①, we obtain top-M distance entries from the buffer. Since we can assume that the individual internal top-M list parts have already been sorted, we can reduce the computation compared to the full top-M computation to the buffer. More specifically, we first sort the candidate buffer and merge it with the internal top-M buffer through the merge process of the bitonic sort [11]. We use the single warp-level bitonic sort when the candidate buffer size is less or equal to 512, while we use a radix-based sort using within a single CTA when it is larger than 512. This design is based on the observation that when the candidate buffer is small enough, we can quickly sort the candidate list right in the registers of a single warp without the shared memory footprint, while we need to use the shared memory when the list length is large, resulting in a performance degradation compared to the radix-based sort.

3) Hash table for visited node list management: In ③, we calculate the distance between the query and each node in the candidate list only the first time the node appears in the list. This requires a mechanism for recording whether a node has been in the list before and, in a similar manner to the SONG algorithm [33], we use an open addressing hash table to manage the visited node list in the CAGRA search.

The number of potential entries in the hash table is calculated as $I_{\text{max}} \times p \times d$, where I_{max} represents the maximum number of search iterations. We set the hash table size to at least twice this value to reduce the likelihood of hash collisions.

If we place the hash table in limited memory, such as shared memory, and it exceeds the memory's capacity, we use a smaller hash table with periodic *table resetting*, meaning the hash table evicts previously visited nodes at certain intervals. After resetting the table, we only register the nodes present in the internal top-M list to the hash table at that moment. Although this process may increase the number of distance computations, catastrophic recall degradation will not occur, as mentioned in [33]. We refer to this type of hash table management, which is meant for limited memory and uses table resetting, as **forgettable hash table management**. We set the number of entries of the hash table as $2^8 \sim 2^{13}$ and the reset interval as typically $1 \sim 4$ depending on the graph and search parameters M, d, and p. This hash table management reduces the shared memory usage per query, typically ≤ 4 kB, resulting in higher parallel efficiency in a large-batch query on almost all generation of NVIDIA GPUs.

4) 1-bit parented node management: In step ②, we select a specific number of nodes that have not previously been parents and assign their neighbor indices to the candidate list. To keep track of whether a node has acted as a parent, we utilize the Most Significant Bit (MSB) of the index variable in the buffer as a flag for recording this information. An alternative approach could involve using another hash table, but we choose not to adopt it due to a latency disadvantage. We need to search the entry in the hash table if we use the alternative method, while we can check whether a node has acted as a parent just by reading the MSB of the node index in the list. This method comes with a disadvantage, however, as it imposes a limitation on the size of the dataset and restricts it to half of the maximum value representable by the index data

TABLE II
THE SUMMARY OF THE DIFFERENCE BETWEEN THE SINGLE- AND MULTI-CTA MODES.

	single-CTA	multi-CTA
Use case	large-batch	small-batch or higher recall is required
per 1 query	single CTA	multiple CTA
Hash table location	Shared memory	Device memory
Hash table management	Forgettable	Standard

type. For instance, when the data type used is `uint32_t`, the supported maximum size of the dataset is only $2^{31}-1$, compared to the maximum value of $2^{32}-1$ if we didn't utilize the MSB for this flag.

C. Implementation

This section explains the features and optimizations of the CAGRA search implementation on the GPU. The CAGRA search contains separate implementations for **single-CTA** and **multi-CTA**. While the basic search strategy and operations are the same in both implementations, mapping the hardware to the queries differs. The summary of these implementations is shown in Table II.

1) Single-CTA implementation: As the name implies, the single-CTA implementation is designed to efficiently process queries by mapping each query to one CTA. The target batch size for this implementation ranges from middle to large values, such as 100 and above. Leveraging the parallel processing capabilities of GPUs, multiple of these single-CTAs can be executed simultaneously, enabling efficient handling of multiple queries and effective GPU resource utilization. However, we note that relatively small batch sizes can leave the GPU resources underutilized, leading to suboptimal performance.

To implement the single-CTA approach, we have developed a kernel function that handles the entire search process ⓪ ∼ ③, placing the hash table in shared memory rather than device memory. As part of our optimization exploration, we have also considered an alternative implementation involving separate kernel functions, with each function handling a specific step of the search process (⓪ ∼ ③). However, extensive testing revealed that the overhead of launching multiple kernels outweighs any potential performance gains. As a result, we have concluded that adopting the multi-kernel approach is not advantageous, and we instead prefer our single-kernel implementation.

In the throughput analysis of the implementation, we have observed that the memory bandwidth of the device memory limits the performance of the kernel function when the query batch size and the dimension of the dataset are large. We propose an approach involving low-precision data types for dataset vectors to address the memory bandwidth limitations and enhance overall throughput. By reducing the memory footprint, this optimization technique aims to expedite data transfer between memory and processing units, thereby increasing throughput. However, it is crucial to carefully assess the trade-off between throughput enhancement and the potential impact on recall.

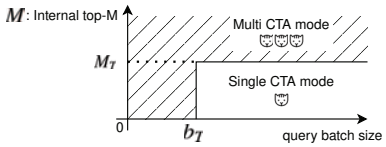

Fig. 7. The rule to choose a suitable CAGRA search implementation.

2) Multi-CTA implementation: The multi-CTA implementation is designed to map one query to multiple CTAs, with a target batch size typically ranging from small values, such as $1 \sim 100$. In contrast to the single-CTA implementation, which launches only as many CTAs as the query batch size, the multi-CTA approach maximizes GPU resource utilization by employing multiple CTAs to process a single query. As a result, this implementation achieves higher GPU utilization and enhances query processing efficiency, even when dealing with small batch sizes.

In this implementation, the hash table is stored in device memory as it needs to be shared with multiple CTAs. Each CTA traverses graph nodes, managing its own internal top-M list and candidate list by setting the number of parent nodes as 1 while sharing the hash table. Therefore, while we search up to $p \times d$ nodes in each iteration in single-CTA, we search up to *the number of CTA we launch* $\times d$ nodes in each iteration. Since we typically set $p = 1$ to maximize the throughput of single-CTA, the number of nodes visited in each iteration in multi-CTA is larger than in single-CTA, leading to higher recall if the number of iterations is the same.

We explored an alternative approach involving graph-sharding, which is commonly used for multi-node ANNS computations [11], with the goal of maximizing GPU resource utilization for small batch sizes. However, we decided against it for practical reasons that pose challenges to execution optimizations, such as the reliance on specific graph structures to create the subgraphs, as well as the shuffling and splitting of the indices to create sub-datasets of the target dataset.

Subsequently, we independently built graphs for each sub-dataset, similar to the graph construction method used in GGNN [9]. During the search phase, assigning each sub-graph to a single CTA did result in high GPU resource utilization when the number of sub-datasets was sufficiently large, however, despite its potential advantages, this method presented several issues. For example, determining the optimal number of splits depends on factors such as the query batch size and the hardware configuration, specifically the number of SMs on the GPU. Creating a series of sub-graphs for each batch size and GPU configuration is not feasible in practice, and it makes this approach impractical. Nevertheless, we recognize that the sharding technique could be well-suited for extending graph-based ANNS to a multi-GPU environment, where each GPU is assigned to process one sub-graph independently.

3) Implementation choice: As mentioned above, we have two implementations targeting small and large batch sizes and

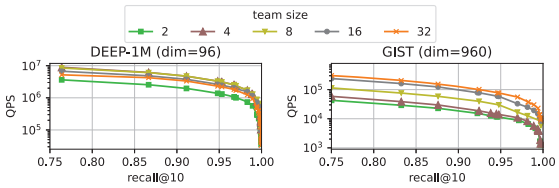

Fig. 8. Search performance comparison among different team sizes.

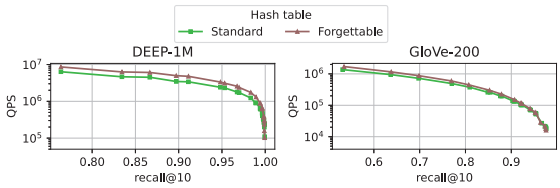

Fig. 9. Search performance comparison between two hash table management methods: standard and forgettable.

we select the implementation based on both the batch size and the internal top-M size, as shown in Fig. 7. We use the multi-CTA implementation when the query batch size is smaller than a threshold b_T or when the internal top-M size is larger than a threshold M_T, since the computing cost of (1) is large in the single-CTA implementation for these cases, increasing the computing time. When the multi-CTA implementation is not used, we fall back to the single-CTA implementation. While the proper thresholds depend on the hardware, we recommend $M_T = 512$ and $b_T =$ "the number of SMs on the GPU" empirically.

D. Evaluation of the CAGRA search implementation

This section reveals the following question:

Q-B1 How much effect does the warp splitting have on the throughput?

Q-B2 How much effect does the forgettable hash have on the throughput?

Q-B3 Which cases single-CTA is faster than multi-CTA?

1) **Q-B1:** *The effect of team size in throughput:* We split the warp into multiple teams in software to efficiently utilize the GPU resources, as mentioned in Sec. IV-B1. Then, *how much effect does this warp splitting have?* We compare the performance among different team sizes for the DEEP-1M and GIST datasets in Fig. 8. In the evaluation result for DEEP-1M, a relatively small dimension dataset, we can gain the highest performance when the team size is 4 or 8 while maintaining recall. When the team size is too small, such as 2, the number of registers per thread becomes too large, leading to performance degradation. On the other hand, in the search performance for GIST, a relatively large dimension dataset, we achieved the highest performance when the team size was 32. In this case, we can utilize the GPU resources efficiently even if we do not split the warp. Instead, decreasing the team size causes significant performance degradation due to increased register usage.

2) **Q-B2:** *The effect of forgettable hash table management in throughput:* When we place the hash table in shared memory, we use the *forgettable hash table management*, a small-size hash table that is reset periodically, instead of the standard hash table in device memory, as mentioned in Sec. IV-B3. Then, *how much faster is the forgettable hash on shared memory than the standard one on the device memory*, and *how much recall is reduced by the periodic reset?* We compare the search performance between the forgettable hash in shared memory and the standard hash in device memory when using them in single-CTA in Fig. 9. In this experiment, we reset the hash table for every iteration in the forgettable hash. In both datasets, DEEP-1M and GloVe, we have confirmed the forgettable hash achieves compatible or higher search throughput compared to the standard hash. The throughput gain observed in GloVe is slightly smaller than in DEEP-1M. This discrepancy can be attributed to the fact that in GloVe, the overhead of hash table operations becomes relatively smaller when dealing with larger dimension dataset vectors, as the primary computational load shifted towards distance calculations.

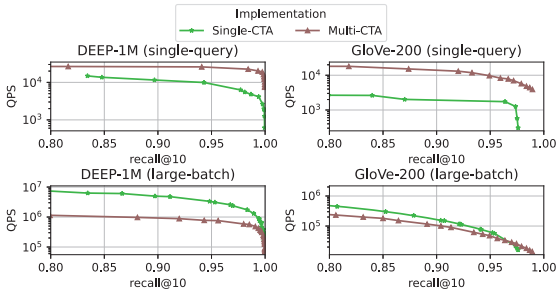

Fig. 10. Search performance comparison between single-CTA and multi-CTA implementations for single-query (top) and large-batch query (bottom). The batch size in large-batch query search is 10K.

3) **Q-B3:** *Search performance comparison between single- and multi-CTA implementations:* We have measured the search performance of the single- and multi-CTA implementations for the DEEP-1M and GloVe datasets, as shown in Fig. 10. In the context of a single query, the multi-CTA approach outperforms the single-CTA approach for both the DEEP-1M and GloVe datasets. However, in a large-batch query, we observe divergent outcomes. For the DEEP-1M dataset, the single-CTA method demonstrates superior search performance. On the other hand, in the case of GloVe, if a higher recall is required, the multi-CTA method achieves better results. This discrepancy can be attributed to the nature of the GloVe dataset, which is considered to be "harder" than DEEP-1M [16]. Achieving higher recall on the GloVe dataset necessitates searching through more nodes, in other words, increasing internal top-M, which is a requirement that the multi-CTA approach fulfills effectively.

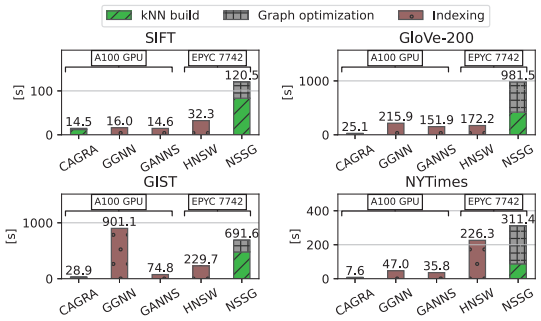

Fig. 11. Graph construction time comparison among CAGRA and other graph-based ANNS implementations.

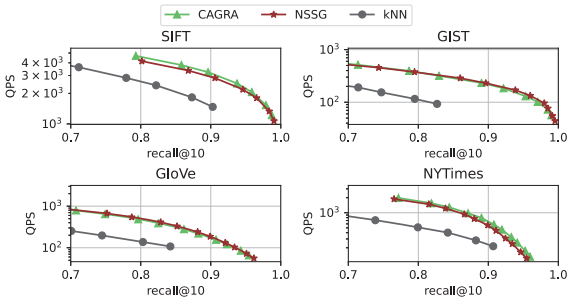

Fig. 12. The search performance comparison between graphs created by CAGRA and NSSG using NSSG single-threaded search implementation.

V. Experiments

We compare CAGRA with the following graph-based ANNS implementations:

1) **GGNN** [9]: One of the current state-of-the-art GPU implementation candidates.
2) **GANNS** [32]: One of the current state-of-the-art GPU implementation candidates.
3) **HNSW** [18]: Well-known state-of-the-art implementation and proximity graph for CPU.
4) **NSSG** [7]: An implementation with search and graph construction processes similar to CAGRA. NSSG also starts the search process with random sampling.

While the out-degree of the CAGRA graph is fixed, it is not fixed for the other four implementations. Therefore, we align the average out-degree for each dataset to make the comparison between them as fair as possible.

In the experiments, we answer the following questions:

Q-C1 How fast is the CAGRA graph construction?
Q-C2 How comparable is the search quality of the CAGRA graph?
Q-C3 How much better is the CAGRA search performance in batch processing?
Q-C4 How much better is the CAGRA search performance in online processing?
Q-C5 Does CAGRA support large datasets?

A. Q-C1: *Graph construction time*

The result of the graph construction time is shown in Fig. 11. We measured the entire graph construction time, including memory allocation, dataset file load, and data movement. NSSG first builds a k-NN graph explicitly and then optimizes it similarly to CAGRA, whereas GGNN, GANNS, and HNSW do not. Therefore, in the case of CAGRA and NSSG, we show the breakdown of the initial k-NN graph build and its optimization time, while the others are only the entire construction time. CAGRA is compatible with or faster than the other CPU and GPU implementations. In comparing implementations for GPU, CAGRA is 1.1–31× faster than GGNN, and 1.0–6.1× than GANNS. And it is 2.2–27× faster than HNSW.

B. Q-C2: *Graph search quality*

To evaluate the search quality of the graph, we compare the search performance of the CAGRA graph to the NSSG graph using the NSSG search implementation in both graphs. To do so, we load the CAGRA graph into NSSG and use NSSG search to find nearest neighbors. Using the same search implementation with different graphs allows us to directly compare the quality of the graphs. Fig. 12 demonstrates that while a kNN graph results in low search accuracy, the CAGRA and NSG graphs show comparable performance. Although many graphs have high performance, we use NSSG since its search implementation is similar to CAGRA, including that the search process starts from the random sampling, and it has better or compatible search performance compared to most of them [29]. If we were to use another search implementation unsuitable for CAGRA, for instance, HNSW or NSG, the search performance would be disadvantageous to CAGRA. In this evaluation, we first build an NSSG graph and calculate the average out-degree of the graph. Then, we build a CAGRA graph for the dataset setting the out-degree as the largest value less than or equal to the average out-degree in multiples of 16. The comparison of search performance is shown in Fig. 12, and the results indicate that the search performance of the CAGRA graph is almost at the same level as that of the NSSG graph across all four datasets. In summary, the evaluation demonstrates that the CAGRA graph achieves a search performance comparable to the NSSG graph, which is one of the highest-performance graphs for ANNS.

C. Q-C3: *Recall and throughput in large-batch query search*

In the batch processing use of ANNS, large-batch search performance is crucial. This use case is suitable for GPU since it is easy to extract parallelism, and this makes the best use of the single-CTA implementation in CAGRA. Then, *How fast is CAGRA compared to other ANNS GPU implementations and the state-of-the-art CPU implementation in large-batch search?* We have compared the recall and throughput among CAGRA and the other methods, as shown in Fig. 13. Since the search implementation of NSSG is not multi-threaded, and using it would not be a fair comparison, we measured the performance of NSSG using the search implementation for

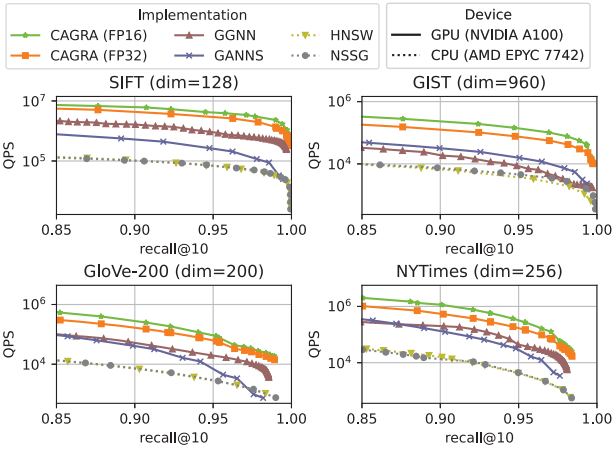

Fig. 13. Large-batch search performance comparison among CAGRA and other graph-based ANNS implementations (batch size=10K). CAGRA (FP32) indicates that the dataset is stored in FP32, while CAGRA (FP16) is converted to FP16.

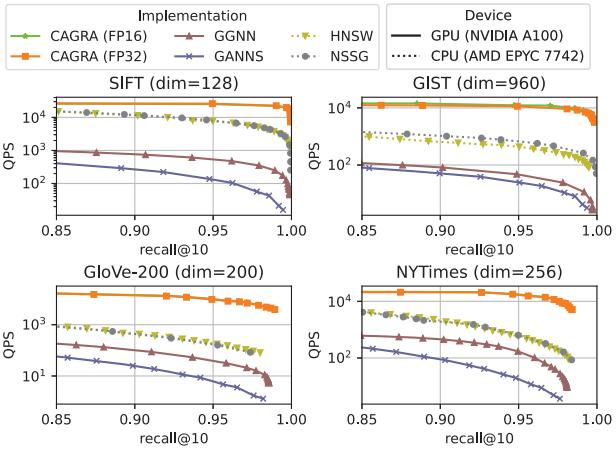

Fig. 14. Single-query search performance comparison among CAGRA and other graph-based ANNS implementations. CAGRA (FP32) indicates that the dataset is stored in single-precision FP32, while CAGRA (FP16) is converted to half-precision FP16.

the bottom layer of the HNSW graph. In the performance measurement of HNSW and NSSG, we have tried multiple OpenMP thread counts, up to 64, and plotted the fastest of them. The results show that the performance of CAGRA is higher than the other ANNS methods on both CPU and GPU. In the 90% to 95% recall range, our method is 33–77× faster than HNSW and is 3.8–8.8× faster than the other GPU implementations. Since the memory bandwidth of the device limits the throughput of CAGRA, we can gain higher throughput using half-precision (FP16) in dataset vector data type. We demonstrate that half-precision does not degrade the quality of results while still benefitting from higher throughput.

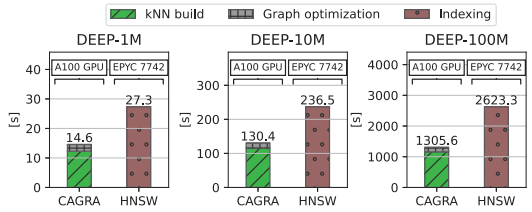

Fig. 15. Graph construction time comparison between CAGRA and HNSW for DEEP-1M, 10M, and 100M datasets.

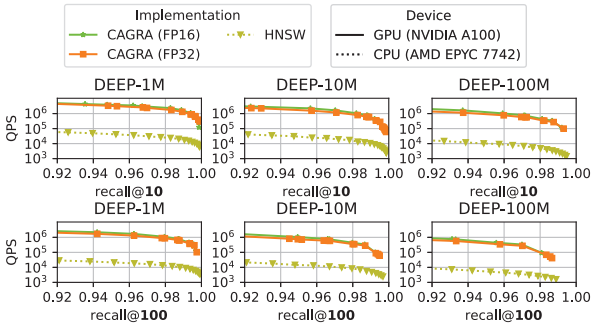

Fig. 16. Search performance comparison between CAGRA and HNSW for DEEP-1M, 10M, and 100M datasets in recall@10 (top) and recall@100 (bottom). The batch size is 10K.

D. Q-C4: *Recall and throughput in single-query search*

In the online processing use of ANNS, single-query performance is crucial. In this use case, an implementation for multi-batch processing on GPU is typically unsuitable since it can not efficiently utilize the GPU resources. In CAGRA, we propose the multi-CTA implementation to address this inefficiency. Then, *How fast is CAGRA compared to the other fast ANNS implementations for CPU in single-query?* We have compared the recall and throughput of CAGRA to the other methods, as shown in Fig. 14. Our results show that CAGRA has a 3.4–53× higher search performance than HNSW at 95% recall. Since the GGNN and GANNS methods are optimized for large-batch queries, their single-query throughputs are much slower than even HNSW and NSSG on CPU. While the performance of CAGRA (FP16) and CAGRA (FP32) in SIFT, GloVe, and NYTimes are very similar, CAGRA (FP16) is slightly better in GIST. This discrepancy is from the larger dimensionality of GIST compared to the other datasets, which require more memory bandwidth to load the dataset.

E. Q-C5: *Large size dataset*

In recent years, the query operations for larger and larger datasets are attracting attention [26]. So, *does CAGRA support large datasets?*, and *how do larger datasets affect CAGRA's search performance?* We measured the graph construction and search performance of CAGRA in the DEEP-1M, 10M, and 100M datasets, as shown in Fig. 15 and Fig. 16. The graph construction time increases proportionally with the dataset size. During the search performance comparisons, we observe

that as the dataset size grows, CAGRA's recall declines slightly but follows a similar trend to HNSW and the degradation in both recall and throughput is not significant. Based on our findings, we believe that CAGRA remains capable of handling larger datasets while maintaining this trend unless the dataset exceeds the device memory capacity. In these cases, a multi-GPU sharding technique [6] discussed in Sec. IV-C2 and data compression schemes, such as product quantization, are some of the ways to address the memory capacity problem, though further performance investigation is required.

VI. Conclusion

In this paper, we proposed a fast graph-based ANNS method called CAGRA, which is designed to perform well on NVIDIA GPUs by harnessing their increased computing capacity and superior memory bandwidth. CAGRA performs a heuristic optimization to the initial k-NN graph to improve the reachability from each node to other nodes with a highly parallel computation-friendly algorithm. CAGRA has better search performance than other state-of-the-art graph-based ANNS implementations on both CPU and GPU for both large-batch and single queries. CAGRA is available in the open-source NVIDIA RAPIDS RAFT library, which can be found on GitHub (https://github.com/rapidsai/raft).

References

[1] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, January 2020.

[2] Nandan Banerjee, Ryan C. Connolly, Dimitri Lisin, Jimmy Briggs, and Mario E. Munich. View management for lifelong visual maps. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, November 2019.

[3] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, and Bo Zheng. Approximate Nearest Neighbor Search under Neural Similarity Metric for Large-Scale Recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, October 2022.

[4] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, June 2004.

[5] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, March 2011.

[6] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. LANNS: a web-scale approximate nearest neighbor lookup system, December 2021.

[7] Cong Fu, Changxu Wang, and Deng Cai. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 4139–4150, August 2022.

[8] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, pages 461–474, 2019.

[9] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik Lensch. GGNN: Graph-based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data*, 2022.

[10] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data, October 2018. arXiv:1810.07355 [cs].

[11] Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, July 2021.

[12] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, January 2011.

[13] Tanaka Kanji, Chokushi Yuuto, and Ando Masatoshi. Mining visual phrases for long-term visual SLAM. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2014.

[14] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through Memorization: Nearest Neighbor Language Models. 2020.

[15] Ping Li, Weijie Zhao, Chao Wang, Qi Xia, Alice Wu, and Lijun Peng. Practice with Graph-based ANN Algorithms on Sparse Data: Chi-square Two-tower model, HNSW, Sign Cauchy Projections, June 2023.

[16] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering*, August 2020.

[17] Ting Liu, Charles Rosenberg, and Henry A. Rowley. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *2007 IEEE Workshop on Applications of Computer Vision (WACV '07)*, February 2007.

[18] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv:1603.09320 [cs]*, August 2018. arXiv: 1603.09320.

[19] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, September 2014.

[20] Corey J. Nolet, Divye Gala, Alex Fender, Mahesh Doijade, Joe Eaton, Edward Raff, John Zedlewski, Brad Rees, and Tim Oates. cuSLINK: Single-linkage Agglomerative Clustering on the GPU, June 2023.

[21] Corey J. Nolet, Divye Gala, Edward Raff, Joe Eaton, Brad Rees, John Zedlewski, and Tim Oates. GPU Semiring Primitives for Sparse Neighborhood Methods. In *Proceedings of Machine Learning and Systems*, pages 95–109, 2022.

[22] Corey J. Nolet, Victor Lafargue, Edward Raff, Thejaswi Nanditale, Tim Oates, John Zedlewski, and Joshua Patterson. Bringing UMAP Closer to the Speed of Light with GPU Acceleration. *Proceedings of the AAAI Conference on Artificial Intelligence*, May 2021.

[23] NVIDIA. CUDA C++ Programming Guide. 2023.

[24] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, October 2014.

[25] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, June 2008.

[26] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search, May 2022. Number: arXiv:2205.03763 [cs].

[27] Jeffrey Travers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, December 1969.

[28] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. Fast k-NN Graph Construction by GPU based NN-Descent. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, October 2021.

[29] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *arXiv:2101.12631 [cs]*, May 2021. arXiv: 2101.12631.

[30] Frank F. Xu, Uri Alon, and Graham Neubig. Why do Nearest Neighbor Language Models Work?, January 2023. arXiv:2301.02828 [cs].

[31] Artem Babenko Yandex and Victor Lempitsky. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[32] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, May 2022.

[33] Weijie Zhao, Shulong Tan, and Ping Li. SONG: Approximate Nearest Neighbor Search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, April 2020.