

PathWeaver: A High-Throughput Multi-GPU System for Graph-Based Approximate Nearest Neighbor Search

Sukjin Kim Seongyeon Park Si Ung Noh
Junguk Hong Taehee Kwon Hunseong Lim Jinho Lee

Seoul National University

Abstract

Graph-based Approximate Nearest Neighbor Search (ANNS) is widely adopted in numerous applications, such as recommendation systems, natural language processing, and computer vision. While recent works on GPU-based acceleration have significantly advanced ANNS performance, the ever-growing scale of datasets now demands efficient multi-GPU solutions. However, the design of existing works overlooks multi-GPU scalability, resulting in naïve approaches that treat additional GPUs as a means to extend memory capacity for large datasets. This inefficiency arises from partitioning the dataset and independently searching for data points similar to the queries in each GPU. We therefore propose PathWeaver, a novel multi-GPU framework designed to scale and accelerate ANNS for large datasets. First, we propose pipelining-based path extension, a GPU-aware pipelining mechanism that reduces prior work’s redundant search iterations by leveraging GPU-to-GPU communication. Second, we design ghost staging that leverages a representative dataset to identify optimal query starting points, reducing the search space for challenging queries. Finally, we introduce direction-guided selection, a data selection technique that filters irrelevant points early in the search process, minimizing unnecessary memory accesses and distance computations. Comprehensive evaluations across diverse datasets demonstrate that PathWeaver achieves $3.24\times$ geomean speedup and up to $5.30\times$ speedup on 95% recall rate over state-of-the-art multi-GPU-based ANNS frameworks.

1 Introduction

Various fields, such as computer vision [11, 34, 39], recommendation systems [13], natural language processing [35, 58], and information retrieval [17], utilize datasets consisting of multi-dimensional vectors representing larger data entities (e.g., image, text). These representations are used to efficiently retrieve data relevant to the input queries in various applications. One solution to finding k -closest data points (vectors) is k -Nearest Neighbor Search, where the k -closest data points

are selected from the entire dataset based on the user-defined similarity metric, commonly the L2 distance. However, as the dataset grows, finding the exact solution becomes increasingly difficult due to the curse of dimensionality [15, 28, 38]. To address this challenge, Approximate Nearest Neighbor Search (ANNS) has been widely adopted, especially by products such as vector databases [2, 3, 5, 6, 18, 55]. ANNS performs nearest neighbor search on large datasets with reasonable execution times while maintaining high accuracy.

Among various ANNS solutions [23, 27, 32, 48], graph-based methods [20, 21, 40, 41, 54] have gained significant attention due to their ability to effectively represent neighbor relationships between data points. This representation allows for evaluating fewer data points while achieving higher accuracy compared to alternative methods [56]. Consequently, there has been a growing body of work on optimizing graph-based ANNS performance across diverse hardware platforms. While CPU-based optimizations [9, 12, 16, 22, 42, 43, 61, 64] offer algorithms that maintain accuracy, they are limited by the number of available cores, hindering their scalability when processing large, high-dimensional datasets that demand extensive computation and memory access.

To address such a computational burden, recent works [25, 31, 41, 44, 50, 53, 65] propose GPU-optimized search techniques to leverage the high-performance capabilities of GPUs. A prominent example is CAGRA [44], which accelerates single-GPU search by efficiently computing the L2 distance between queries and data points. It further leverages hash tables to eliminate redundant computations while optimizing for the GPU’s thread and memory hierarchy. Another example is GGNN [25], which partitions large datasets into smaller, independent graphs that fit within a single GPU. This enables data-parallel searches across multiple GPUs.

While these approaches demonstrate significant performance improvements over CPU-based solutions, we identify key bottlenecks that hinder their ability to fully exploit the potential of GPUs:

- **Existing multi-GPU solutions suffer from low scale efficiency.** Prior works [25, 31] divide the dataset across GPUs to support ANNS on large datasets with GPUs, with each GPU independently processing queries (i.e., graph sharding). While this approach enables handling large datasets on GPUs, it requires each query to be processed multiple times across different GPUs, resulting in a low scale efficiency.
- **Majority of random initial nodes turn out to be unnecessary later.** To quickly search for near-optimal data points, existing approaches [41, 44, 50] rely on starting searches from numerous random initial points. However, due to the underlying beam search method, most of the neighbors explored from those random initial points are quickly discarded within a few iterations, leaving only descendants of the few best points. This incurs too much computational and memory overhead for the search, which is unnecessary for the final result.
- **Each iteration mandates too much overhead.** Visiting a vertex in a proximity graph yields distance computation with all its neighbors. Given that the typical degree of proximity graphs is around a few tens, this is a huge burden to the search. However, many of these neighboring data points are not selected as part of the top- k results, leading to unnecessary memory access and computation that hinders overall efficiency.

Based on these observations, we introduce **PathWeaver**, a multi-GPU framework designed to support graph-based ANNS on large datasets with scalable performance with minimal accuracy loss. PathWeaver is carefully designed to address the limitations above while exploiting the parallelism of GPU resources.

To address the scalability limitations of sharding-based multi-GPU search, we leverage our finding that each shard’s seemingly independent search operations can be optimized by sharing intermediate search results with the following shard. We propose **pipelining-based path extension**, a mechanism that passes the search results of each shard to the next GPU in a pipelined manner. This enables subsequent GPUs to start their search from data points closer to the query within their shard, effectively reducing the number of search iterations only with negligible inter-GPU communication cost. One limitation of pipelining-based path extension lies in the first stage, whose search still has to be done from scratch. Our second scheme **ghost staging** addresses this issue by selecting more optimal starting points for each shard by prioritizing data points closer to the query, thereby improving the efficiency of the first stage. Finally, we propose **direction-guided selection**, which skips distance calculations for neighbors whose direction from the parent node significantly deviates from the direction toward the query. This approach further minimizes unnecessary distance computations, reducing both computational and memory overhead.

We implement the proposed techniques in **PathWeaver** and conduct extensive experiments on large datasets. PathWeaver improves the performance by $3.24\times$ geomean speedup compared to the state-of-the-art GPU-based ANNS baselines at 95% recall rate, demonstrating its scalability and effectiveness.

2 Background

2.1 Approximate Nearest Neighbor Search

Given a dataset $\mathcal{D} = \{x_0, x_1, \dots, x_{n-1} \mid x_i \in \mathbb{R}^d\}$ with n data points represented as d -dimensional vectors, and a query $q \in \mathbb{R}^d$, k -Nearest Neighbor Search (k -NNS) identifies k data points $N_k(q)$ that satisfies the following:

$$N_k(q) = \underset{S \subset \mathcal{D}, |S|=k, x_i \in S}{\operatorname{argmin}} \sum dist(q, x_i), \quad (1)$$

where $dist(q, x_i)$ is the similarity measure between the query and data point, typically the L2 distance. As the size of \mathcal{D} grows, performing exact k -NNS becomes computationally expensive due to the substantial memory access and computation overhead. To address this, many applications adopt approximate nearest neighbor search (ANNS), which employs efficient algorithms to approximate k -NNS results with significant reductions in execution time [37].

Among the various ANNS algorithms, graph-based ANNS [20, 21, 40, 41, 54] has been widely used due to its ability to effectively capture the relationships (distances) between data points [56]. Graph-based methods construct a proximity graph \mathcal{G} where nodes represent data points and edges encode distances between them, allowing for compact representations of relationships and facilitating efficient graph traversal to approximate k -NNS results. Assuming that \mathcal{G} is pre-constructed, the proposed PathWeaver targets accelerating the graph search process itself, addressing key inefficiencies and scaling challenges associated with large-scale datasets.

2.2 Approximate Nearest Neighbor Search Method on GPUs

Graph-based Approximate Nearest Neighbor Search (ANNS) algorithms have gained significant traction due to their ability to efficiently navigate complex datasets by leveraging graph structures. We briefly explain the overall search algorithm [44] on a constructed graph \mathcal{G} , as illustrated in Fig. 1a, to provide context for our optimizations. Note that many graph-based search algorithms [25, 53, 65] follow a similar workflow.

The graph-based search operates on two key data structures as shown in Fig. 1a:

- A *priority queue* $p = \{p_0, p_1, \dots, p_{l-1}\}$ of size l , which stores the top- l ($k \leq l$) intermediate nodes sorted by their distances to the query q .

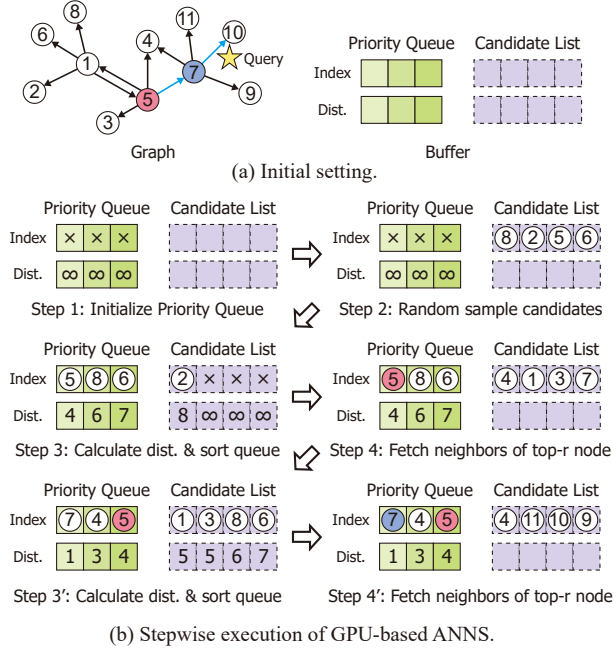


Figure 1: Overview of approximate nearest neighbor search on GPUs.

- A *candidate list* $c = \{c_0, c_1, \dots, c_{m-1}\}$ of size m , which acts as a buffer to hold the neighbors of the nodes being processed.

Given these structures, the algorithm iteratively refines the search process as depicted in Fig. 1b:

- Step 1** The algorithm begins by initializing the priority queue p with dummy indices with ∞ distances.
- Step 2** The candidate list c is populated with m randomly selected nodes from the graph.
- Step 3** The distance $\text{dist}(q, c_i)$ is computed for each candidate node $c_i \in c$ using:

$$\text{dist}(q, c_i) = \|q - c_i\|_2, \quad (2)$$

where elements of the priority queue p and the candidate list c are sorted together based on the distances, ensuring that the closest nodes appear at the top of p .

- Step 4** The neighbors of the top- r ($r \leq l$) nodes in the priority queue are fetched from \mathcal{G} and the candidate list c is populated with these nodes.

Steps 3 and 4 are repeated until the priority queue receives no new entries or the pre-defined maximum number of iterations is reached. Then, the top- k nodes in p are returned as the approximate nearest neighbors.

This algorithm is based on carefully built proximity graphs, which encourage reachability (all vertices are reachable starting from any vertex) and convexity (avoids falling into local

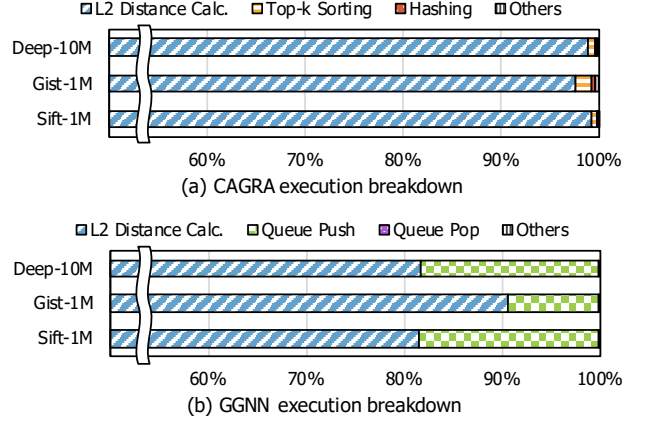


Figure 2: Execution time breakdown analysis of baseline ANNS.

minima) [25, 44]. Based on such characteristics, the search algorithm aims for faster convergence towards the global minima by maintaining the priority queue that stores data points with small L2 distances and quickly dropping long-distance points.

Because the vector dimension typically ranges from a few tens to several hundreds, L2 distance calculation dominates the search time due to the overhead of loading and processing high-dimensional vectors. According to our measurements depicted in Fig. 2a, when using the current state-of-the-art method CAGRA [44], L2 distance calculation accounts for over 95% of the total search time, regardless of the dataset. To further analyze this trend across different methods, we also performed a breakdown of GGNN [25] and depicted in Fig. 2b found that over 80% of the time was still spent on distance calculations. Therefore, the key to high-throughput ANNS is to reduce the number of L2 distance calculations. For a search that converged in i iterations on a j -proximity graph (i.e., each vertex has j neighbors), the rough count of L2 distance calculation is $i \times j \times r$ (including duplicates). To achieve high throughput, PathWeaver mainly aims to reduce the number of search iterations i and number of examined neighbors out of j , while not sacrificing the search accuracy.

3 PathWeaver Design

3.1 Pipelining-based Path Extension

3.1.1 Scalability of Performance on Multiple GPUs

To diagnose the existing multi-GPU search method based on sharding [25], we compare the total number of search iterations between single- and multi-GPU setups using a multi-GPU extension of CAGRA and GGNN, as shown in Fig. 3a. The results indicate that the sharding-based approach scales inefficiently across multiple GPUs in both methods. For the

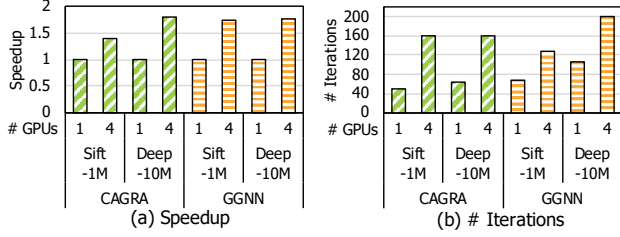


Figure 3: Performance scalability of prior work on Sift-1M and Deep-10M datasets with 4 GPUs.

Sift-1M dataset, using 4 GPUs yields only a $1.39\times$ speedup in CAGRA, resulting in a scaling efficiency of approximately 35%. GGNN shows similar trends, achieving about a $1.7\times$ speedup on both datasets, which corresponds to a scaling efficiency of around 43%. This inefficiency arises from the fact that the multi-GPU baseline performs a separate local search in each GPU’s shard for every query. While this simple query-parallel design provides an easy parallelized solution for large datasets, the number of iterations needed does not reduce linearly with the size of a shard. As a result, the number of total iterations to complete a search increases with more number of shards, as depicted in Fig. 3b. In the case of the Sift-1M dataset in CAGRA method, the total iteration over all the shard scales with the number of shards results in $4\times$ total iteration. In GGNN, increase of total iterations is slightly better at around $2\times$, but not enough to compensate for the overall scalability loss. This highlights the need for an alternative solution to achieve efficient performance.

3.1.2 Pipelining-based Path Extension Design

To amend the issue identified above, we propose *pipelining-based path extension*, a multi-GPU solution that enhances scalability by leveraging intermediate search results from other GPUs in a pipelined manner. As illustrated in Fig. 4a, the key idea is to create inter-shard edges from each node to the closest node in the adjacent shard.

The baseline sharding [25] (Fig. 4b) begins by randomly partitioning the input dataset into independent shards, with the number of shards matching the number of GPUs (4 shards in this example). After allocating these shards to each GPU, independent graphs are built for each shard/GPU. Note that in this baseline approach, there is no edge between the nodes in different graphs. Once the graphs are built, the baseline performs independent ANNS for all queries by starting from randomly selected nodes within each graph. Upon completing the search for all queries, each GPU stores the top- k data points for each query specific to its graph. For instance, after the searches are finished, $N \times k$ candidate data points are outputted across N GPUs. These results are then offloaded to the CPU for reduction to get the final top- k data points for each query. While this method provides an easily paralleliz-

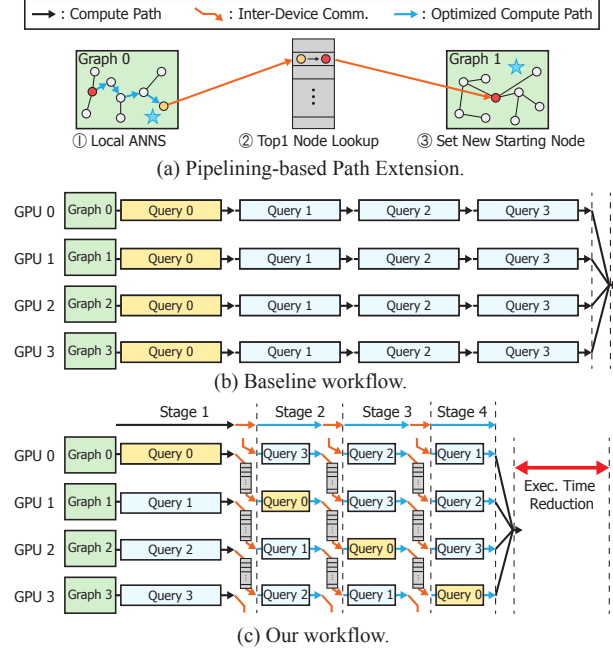


Figure 4: Illustration of the Pipelining-based path extension design.

able solution with almost no communication, it requires more search iterations as revealed in Section 3.1.1 and thus scales inefficiently.

In the proposed pipelining-based path extension design illustrated in Fig. 4c, the dataset is randomly partitioned in the same way as the baseline sharding approach to build independent graphs G_i for the shard on GPU i . However, after the graph construction, pipelining-based path extension takes a step further by creating uni-directional node-to-node connections between the shards of adjacent GPUs as shown in Fig. 4a. We define G_i to be adjacent to shards graph $G_{(i+1)\%N}$ for N GPUs, which form a ring topology. From each vertex in a shard, the nearest neighbor vertex is found in the adjacent shard, and an inter-shard edge is added between them. For example, the 0th GPU’s graph connects all of its nodes to the closest node in the 1st GPU’s graph, and nodes in the 3rd GPU to the 0th GPU, forming a ring-like uni-directional connection among graphs/shards. These inter-shard edges can be expressed as a mapping I :

$$I(u) = \underset{w \in G_{(i+1)\%N}}{\operatorname{argmin}} \operatorname{dist}(u, w) \quad \forall u \in G_i. \quad (3)$$

These connections are constructed once during the graph build phase and can be reused across searches. As detailed in Section 5.7, the additional overhead for creating these inter-shard edges is minimal.

To perform a search with the interconnected graph, each GPU is assigned a $|Q|/N$ chunk from the set of queries Q and starts the search as in the baseline sharding. After each query

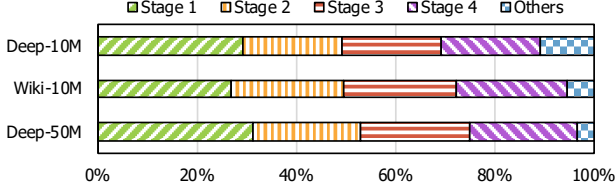


Figure 5: Execution time breakdown analysis with 4 GPUs after applying the pipelining-based path extension design according to the pipeline stages.

has converged in all GPUs, a part of the final local result Z is taken. From each of its vertices $z \in Z$, it continues searching from $I(z)$, which naturally takes place in the adjacent shard as if the search path has been extended. Because $I(z)$ has been designed to be close to the query, the search converges significantly faster than the baseline in fewer iterations. For example, in Fig. 4c, when searching with query batch 0 (yellow box), the search process begins on GPU 0 with graph 0. After the search completes on GPU 0, the result is passed to GPU 1, which continues the search on graph 1. As shown in the figure, the search time in stage 2 (blue arrow) is reduced compared to that of stage 1 (black arrow). This process is executed simultaneously on all GPUs and is repeated in a pipelined manner until every GPU has processed the queries originating from all other GPUs. Then the results are reduced in the CPU, similarly to the baseline.

As discussed in Section 6.4, because the only data transfer involved is the local result between the adjacent GPUs, the communication overhead is almost negligible compared to the substantial reduction in search time.

3.2 Ghost Staging

3.2.1 Execution Time Breakdown after Pipelining-based Path Extension

Fig. 5 illustrates the execution time breakdown of searching with pipelining-based path extension. As expected, the first search stage of pipelining-based path extension is the primary bottleneck. For example, the first search stage in the Deep-50M dataset takes up to 31%, while the other stages consume only up to 22%.

This bottleneck arises as the first search stage does not benefit from pipelining-based path extension and starts from random, potentially distant data points. As a result, more iterations are required to converge on the appropriate top- k candidates. As all GPUs are performing the initial stage of their own query, reducing this overhead could bring additional speedup.

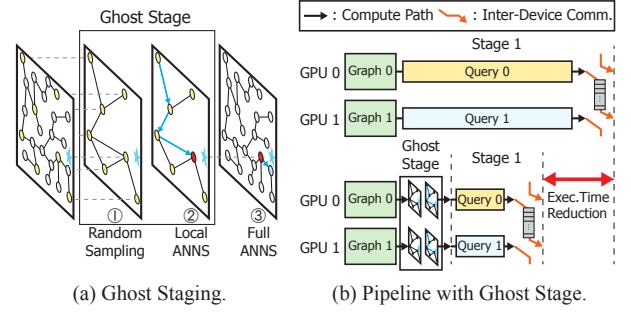


Figure 6: Illustration of the ghost staging design.

3.2.2 Ghost Staging Design

Motivated by pipelining-based path extension’s ability to identify better starting points for subsequent GPU search stages, we propose *ghost staging*, a kind of hierarchical search [40] solution for improving the initial search stage on each GPU. Ghost staging adds a small auxiliary shard before the first stage, such that a high-quality data point close to the query can be located only in a few iterations. This approach significantly reduces the number of iterations needed during the initial search stage.

Fig. 6 illustrates how ghost staging reduces the number of search iterations during the first stage while maintaining accuracy. To create an auxiliary shard, ① ghost staging randomly samples a fixed number of random data points within a shard, referred to as ghost nodes. These ghost nodes are connected based on distances, creating a lightweight network. In addition, the inter-shard edges are added between the ghost nodes and the original nodes. Due to the relatively small number of ghost nodes, the preparation for ghost staging only adds negligible overhead to the graph build process (see Section 5.7). During ghost staging, ② a fixed number of starting points are chosen among the ghost nodes and the search algorithm from Section 2.2 is applied to locate ghost nodes near the query within a few iterations. ③ Once the relevant ghost nodes are identified, the search transitions to the original graph similar to pipelining-based path extension.

The high performance of ghost staging stems from ghost nodes acting as central hubs and their interconnections as high-ways within the expansive original graph. Moreover, ghost staging maintains high accuracy by locally searching the original graph for the remaining iterations. This is highlighted by comparing the search process of the baseline method and ghost staging in Fig. 6b. To reach the same data point close to the query, the baseline has to traverse many vertices due to the original graph’s size. On the other hand, during ghost staging, taking a single step forward among ghost nodes has a similar effect of passing through multiple original graph data points. Even if ghost staging overshoots and bypasses the query, it can backtrack by the remaining search on the original graph.

Table 1: Unused Distance Calculation

Dataset	#Total Visits	#Discarded Visits	Ratio
Sift-1M [8, 32]	2.32E+7	2.00E+7	86.2%
Gist-1M [8, 32]	3.17E+6	2.81E+6	88.9%
Deep-10M [10]	2.68E+7	2.28E+7	85.0%

3.3 Direction-Guided Selection

3.3.1 Analysis on Unused Distance Calculation

Despite reducing the search iterations with pipelining-based path extension and ghost staging, the search procedure remains heavily dominated by the numerous L2 distance calculations. This partially stems from having to add all neighbors of each top- r node in the priority queue (Section 2.2), where the number of neighbors is usually in the order of a few tens [44]. While such a large number of neighbors is necessary to ensure reachability and convexity, it often introduces unnecessary computational overhead. During the search, we found that the majority of the data points added to the priority queue never reached the top- k region of the priority queue and were eventually discarded (unused) without being considered again because they were too far from the query compared to the other candidates.

In Table 1, we quantitatively measured the portion of the considered nodes, where **#Total Visits** denotes the total number of nodes in the proximity graph that are accessed during the search, and **#Discarded Visits** represents the nodes that are visited but never remain in the candidate buffer until the end of the search process to be included in the final top- k results. As depicted in the table, we find that the ratio of discarded node visits, which lead to unnecessary distance calculations, can exceed 80%. This indicates an interesting opportunity for additional optimization, where we could remove a substantial portion of the unused calculations if we can identify them in advance.

3.3.2 Direction-Guided Selection Design

We propose *direction-guided selection*, an optimization technique designed to accelerate the search process within each iteration. Direction-guided selection allows the search algorithm to bypass data points that are significantly unaligned with the query’s direction relative to the current target node, as shown in the top graph in Fig. 7. By leveraging a set of lightweight operations including direction table reads and local sorting, direction-guided selection significantly reduces expensive vector reads and distance calculations.

The direction-guided selection process operates as follows: (a) During the off-line preprocessing phase, a compressed direction table is generated, storing the sign bits of the differences between the source node and its neighbors in the proximity graph. Each sign bit vector encodes the rough di-

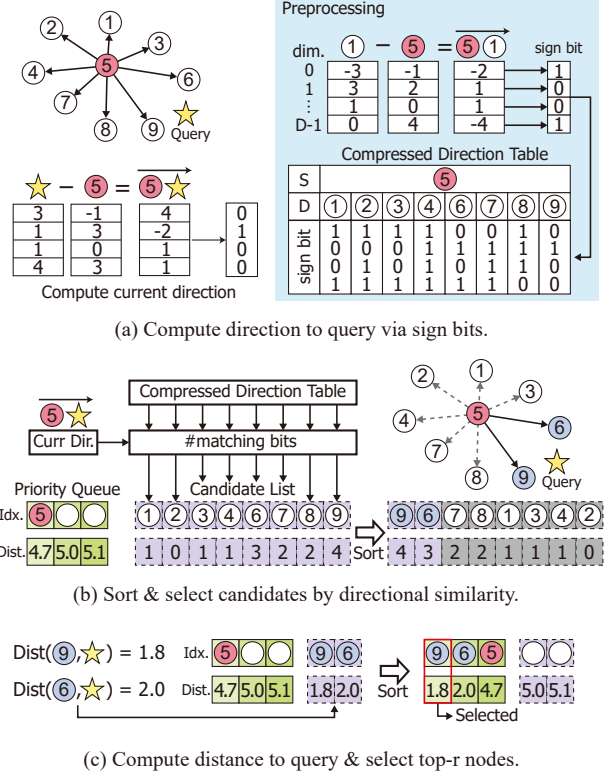


Figure 7: Illustration of the direction-guided selection design.

rection of an edge [46] and serves as a fast proxy to guide traversal during run-time search on the GPU. (b) On each iteration of the search, the sign bits of the direction between the query and the visiting node (node 5 in the example) are calculated. Then the number of matching bits is counted for each neighbor using the compressed direction table. The neighbors are sorted by their number of matching bits in descending order, and the top- n candidates are selected. In the example, with $n=2$, nodes 9 and 6 are selected. (c) The distance is calculated between the query and the selected neighbors (node 9 and 6 in the example), and the priority queue elements are sorted by the order of their distances (e.g., nodes 9, 6, 5). Finally, top- r nodes from the priority queue are selected (e.g., node 9 with $r=1$) and they become the next visiting nodes. To maintain accuracy, the last few iterations are regarded as cool-down phase, and the search is conducted without the selection. We use 30% of the max iteration as the default value.

Direction-guided selection effectively reduces the number of distance calculations while maintaining high accuracy. This is achieved by prioritizing candidates most aligned with the query’s direction, significantly narrowing the search space. There is a small chance that direction-guided selection can drop meaningful candidates, because of the compression error in the direction table. However, due to the reachability and

convexity of the proximity graphs, this would typically result in an increased number of iterations until convergence instead of accuracy. We empirically found that the accuracy drop is almost negligible while the gain from the reduction in candidates dwarfs the increase in the number of iterations.

4 Implementation Details

PathWeaver’s search kernel has been implemented based on the CAGRA search kernel, adopting its query-per-thread-block approach along with GPU-friendly bitonic sorting and hash tables. To maximize the benefits of fast register communication within a warp, we set the threadblock size to be equal to that of a warp (i.e., 32 threads).

To minimize the communication overhead in pipelining-based path extension, the total set of queries is sent to all devices, which has a negligible size compared to the dataset. Each device processes independent graphs generated from its respective shard and maintains a unique inter-shard edge table for its linked shard. At runtime, only a partial chunk of the query batch is processed, and the result is forwarded to the next GPU device.

While the number of results sent per query is a tunable parameter, we empirically choose to send only one to minimize communication overhead. Additionally, for pipelining-based path extension, a shard-to-shard lookup table needs to be pre-computed. Every node within a single shard acts as a query to perform a search in the adjacent shard. The top-1 result from the search is stored in the lookup table for pipelining-based path extension.

Ghost staging can be implemented similarly to pipelining-based path extension, except that instead of connecting neighboring shards, it connects the extracted smaller auxiliary shard with the original shard.

In direction-guided selection, during processing of the sign bit table, all 1-bit signs are packed into one uint32. To efficiently compress the vector between the current node and the query, each thread subtracts the vector’s elements and converts them into 1-bit values. During the runtime search process, the `__shfl_xor_sync()` intrinsic is used within a warp to enable efficient processing by performing fast intra-warp shuffle operations. Next, the precomputed sign bit table is looked up to retrieve the approximate sign bits of neighboring vectors. The similarity with the query vector is then computed by performing bit-wise XOR operations on uint32 units, followed by a `__popcnt()` intrinsic call to count the total number of different bits. Finally, a min-sort is performed to find the node with the highest similarity. As mentioned above, direction-guided selection requires a precomputed compressed sign bit table generated on the CPU using multi-threading, where each thread handles the edges of a single parent node. For each neighbor, element-wise comparisons are performed, and the results are compressed into a uint32 using bit-wise shift and OR operations.

Table 2: Datasets used in evaluation

Target	Dataset	Dim. (d)	Size (n)	Type
Single GPU	Sift-1M [8, 32]	128	1M	float
	Gist-1M [8, 32]	960	1M	float
	Deep-10M [10]	96	10M	float
Multi-GPU	Wiki-10M [7]	768	10M	float
	Deep-10M [10]	96	10M	float
	Deep-50M [10]	96	50M	float

5 Evaluation

5.1 Experiment Setup

Environment. We have implemented and evaluated PathWeaver on a server with four NVIDIA RTX A6000 GPUs and an AMD EPYC 9124 16-Core CPU, where two GPUs are connected via NVLink Bridge [4], and each GPU pair is linked through a host PCIe switch. All environments run on Ubuntu 22.04 with CUDA version 12.1 and PyTorch 2.4.1.

Baselines. The following CPU and GPU baselines were chosen for evaluation.

- **CAGRA** [44] is the state-of-the-art GPU framework for graph-based ANNS. CAGRA proposes a heuristically optimized proximity graph for parallel search operations and incorporates techniques such as warp splitting and forgettable hashing to enhance search performance. We used the official implementation of the authors, where we extended it for multi-GPU settings (denoted as ‘CAGRA w/ Sharding’).
- **HNSW** [40] is a popular graph-based ANNS implemented for CPUs. HNSW introduces a hierarchical proximity graph, with each layer structured as an NSW [41] graph representing a subset of points. The search operation begins at the top layer and traverses through the hierarchy, achieving improved performance and accuracy. Because HNSW only supports execution on CPUs, we evaluated HNSW only in the single-GPU environment for fair comparison. We utilized 64 threads on CPUs for evaluation.
- **GGNN** [25] is another GPU implementation for graph-based ANNS. GGNN builds on the HNSW-inspired proximity graph, specifically optimizing it to leverage massive parallelism during the graph construction phase. Additionally, GGNN enhances search performance by efficiently utilizing shared memory and enabling parallel operations on data structures. To the best of our knowledge, GGNN is the only graph-based ANNS framework that supports multiple GPUs out of the box, where it uses the sharding method.

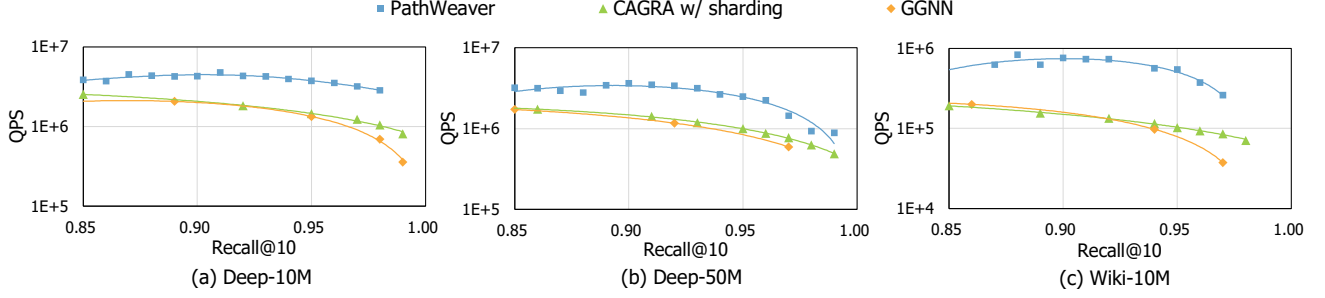


Figure 8: Performance comparison on multi-GPU environment.

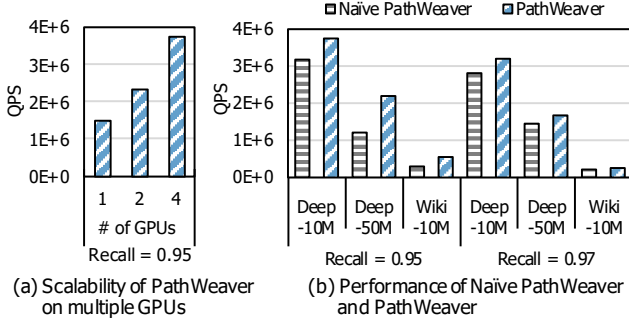


Figure 9: Performance comparison of Recall@10 between the naïve and pipeline method on multiple GPUs.

Datasets. We evaluate our method on a total of six datasets in which the vector dimension ranged from 96 to 960 as reported in Table 2. We analyzed single-GPU performance using datasets of varying sizes (1M, 10M) and dimensions (96, 128, 960) to demonstrate our method’s effectiveness. For multi-GPU experiments, we selected larger datasets, scaling up to 50M in size and 768 in dimension. We took the first 10 million and the first 50 million part to create the Deep-10M and Deep-50M datasets from the Deep-1B dataset [1]. For graph building of PathWeaver, we used CAGRA’s graph build algorithm, which offers the fastest build speed on GPUs. To ensure fairness, the out-degree of graphs was fixed to 64 for all datasets for PathWeaver’s search and the CAGRA baseline evaluation.

Query Batch Size. We evaluated the performance and accuracy of PathWeaver and baselines using a query batch size of 10,000 for single-GPU tests. The exception was Gist-1M, where we used a batch size of 1,000. For multi-GPU evaluations, we employed a batch size of 60,000 to achieve high throughput and fully utilize multiple devices.

Performance Metrics. We tested two key metrics for the evaluation, following the typical performance metrics in the previous ANNS frameworks. The two key metrics and the details are as follows:

1. **Recall:** This metric quantifies the accuracy of the ANNS

results by comparing them to the ground truth k -NNS results. For a query q , recall is defined as:

$$Recall@k = \frac{|\mathcal{N}_k^{KNNS}(q) \cap \mathcal{N}_k^{ANNS}(q)|}{|\mathcal{N}_k^{KNNS}(q)|}, \quad (4)$$

where $\mathcal{N}_k^{KNNS}(q)$ represents the k -NNS result, and $\mathcal{N}_k^{ANNS}(q)$ is the result obtained from the ANNS framework. Similar to prior work [25, 44, 53, 65], we target 95% recall@10 for most of the analyses unless otherwise noted.

2. **Queries Per Second (QPS):** This metric captures the throughput of the framework by measuring the number of queries processed per second.

The primary objective of ANNS frameworks is to balance these two metrics—achieving high recall to ensure accuracy while maximizing QPS to deliver fast query processing.

5.2 Evaluation on Multiple GPUs

5.2.1 Performance Comparison on Multiple GPUs

Fig. 8 presents the performance comparison of PathWeaver with other multi-GPU baselines in a QPS-recall plot. Because CAGRA [44] does not support a multi-GPU environment out-of-box, we extended it with the sharding method as discussed by the authors. As shown in the results, PathWeaver outperforms all baselines. At a high recall of 95%, PathWeaver achieves $3.24\times$ geomean speedup over CAGRA, the best-performing baseline. Moreover, PathWeaver achieves at most $5.30\times$ speedup at the same recall rate in the Wiki-10M dataset. A similar advantage is maintained at a moderate recall 88%–92%, where PathWeaver achieves $3.36\times$ speedup over CAGRA.

Among datasets, Deep-10M and Deep-50M both show similar QPS of around 3×10^6 for PathWeaver and 1.1×10^6 for both CAGRA and GGNN. However, for Wiki-10M, the throughput is an order of magnitude lower in all frameworks because Wiki-10M is composed of very wide vector indices of 768. This aligns with our finding that the ANNS throughput is less impacted by the size of the graphs. Even though

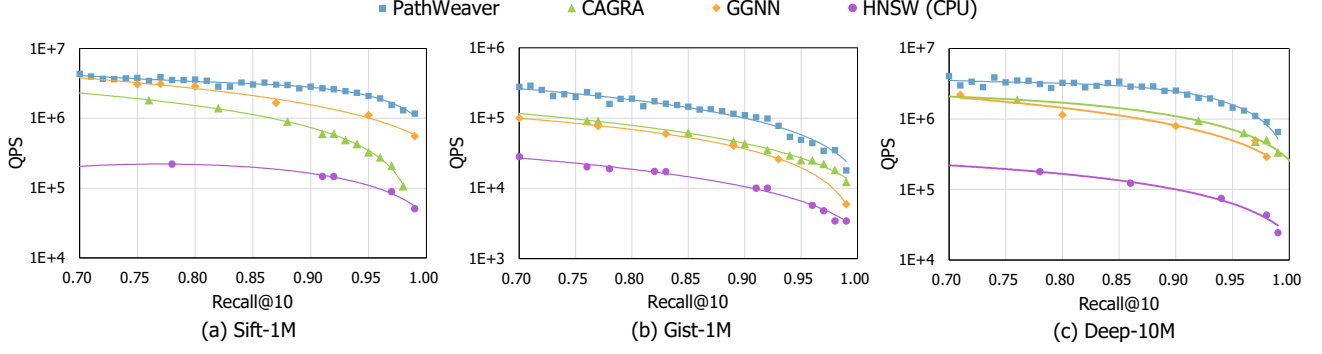


Figure 10: Performance comparison on a single GPU.

Deep-50M has five times more vertices than that of Deep-10M, the number of iterations until convergence is similar, which supports the motivation for pipelining-based path extension. However, having a wider vector per vertex in Wiki-10M dataset directly affects the performance. In this case, PathWeaver achieves more speedup than the other two datasets mainly due to direction-guided selection.

To demonstrate the efficiency and scalability of PathWeaver, we evaluated its performance while increasing the number of GPUs from 1 to 4, at a recall of 95%. The results are shown in Fig. 9a. Compared to that of using a single GPU, using four GPUs achieves $2.47\times$ more speedup when comparing the fastest cases of each GPU setting, which represents 62% scale efficiency. Compared to that of the baselines shown in Fig. 3 for the same dataset Deep-10M, the scale efficiency is improved by 17%.

Fig. 9b further evaluates the impact of pipelining-based path extension on multiple GPUs. Compared to PathWeaver using the sharding method of baseline (denoted as ‘Naïve PathWeaver’), the advantage of pipelining-based path extension is maintained across various datasets and target recall values. This indicates that PathWeaver scales stably on various settings.

5.3 Performance Comparison on a Single GPU

While PathWeaver provides a significant speedup with multiple GPUs, its benefit remains in single-GPU settings. Except for pipelining-based path extension, PathWeaver can still benefit from ghost staging and direction-guided selection on a single-GPU setting. We plotted the performance of PathWeaver and other baselines in Fig. 10. In addition to the GPU-based methods, we additionally study HNSW [40], a baseline method implemented on a CPU.

Overall, PathWeaver achieves a speedup of $3.43\times$ over CAGRA. This speedup is primarily attributed to reduced distance computations due to direction-guided selection and fewer iterations resulting from the application of ghost staging. Similar to the observations from multi-GPU experiments, Gist-1M

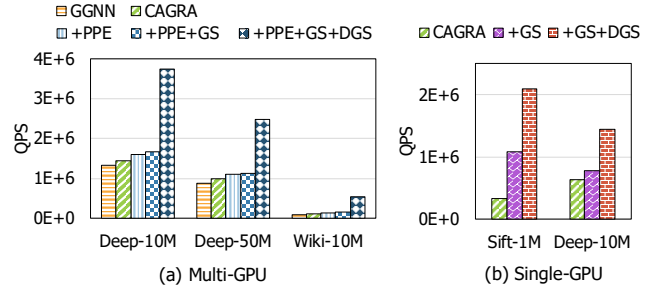


Figure 11: Ablation study of PathWeaver on (a) multi-GPU and (b) single-GPU settings. ‘+PPE’, ‘+GS’, and ‘+DGS’ represent PathWeaver with pipelining-based path extension, ghost staging, and direction-guided selection, respectively.

results in the slowest throughput due to its larger vector dimension of 960. In all cases under evaluation, PathWeaver exhibited better QPS-recall trade-off compared to the baselines.

5.4 Ablation Study

To demonstrate the impact of each scheme in PathWeaver, we conducted an ablation study using the Deep-10M, Deep-50M, and Sift-1M datasets on four GPUs, as well as Deep-10M and Sift-1M on a single GPU. In the results shown in Fig. 11, PPE, GS, and DGS represent pipelining-based path extension, ghost staging, and direction-guided selection, respectively.

For the ablation study on multiple GPUs, we used GGNN and CAGRA as baselines, and gradually applied PPE, GS, and DGS to analyze the effect of each, as shown in Fig. 11a. On top of the baselines, each component of PathWeaver adds consistent speedup across datasets with similar trends. In addition, Fig. 11b shows the ablation study on the single-GPU setting. Although pipelining-based path extension cannot be applied in this case, ghost staging provides higher speedup. This is because in the single-GPU setting, the entire search can be regarded as the first stage, which benefits from the ghost stage.

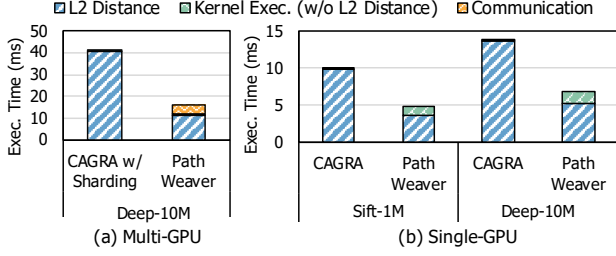


Figure 12: Search execution time breakdown on (a) multi-GPU and (b) single-GPU setting.

5.5 Execution Time Breakdown

To investigate how the execution time is spent in PathWeaver, we further broke down the execution time into three components: L2 distance calculation, rest of the kernel execution time, and inter-GPU communication time. The rest of the kernel includes random number generation, neighbor fetching, distance sorting, and hash table management for avoiding duplicate node visits. They are evaluated at 95% recall, with Deep-10M dataset for multi-GPU setting and an additional Sift-1M dataset for the single-GPU setting. To obtain the breakdown, we used `clock64()` function to separate the portion of L2 distance calculation within a kernel. For breakdowns of multi-GPU execution time, we measured the execution time with a certain portion of the code disabled, similar to the CPI stack [19] method.

The multi-GPU results are shown in Fig. 12a. We compare PathWeaver with extended CAGRA implementation for multi-GPUs (CAGRA w/ Sharding). As shown in the result, the L2 distance calculation is the dominating factor in both the CAGRA w/ Sharding and the fully optimized PathWeaver. In CAGRA w/ Sharding, no execution time is attributed to inter-GPU communication, since all searches are performed independently without any data exchange across GPUs. In PathWeaver, the communication time is incurred because of inter-stage data transfer of pipelining-based path extension. However, this results in fewer iterations required until convergence in subsequent stages, leading to a smaller L2 distance calculation. The rest of the kernel’s portion slightly increases with optimizations in PathWeaver, due to the additional direction flag lookup process introduced by direction-guided selection. However, its portion is almost negligible in both CAGRA w/ Sharding and PathWeaver, making its impact minimal.

Fig. 12b depicts the breakdown on a single GPU. Because there is no communication, most of the time is spent on the L2 distance calculation. Thus, the speedups come from ghost staging and direction-guided selection. Instead of communication, the miscellaneous kernel execution time slightly increases in PathWeaver, which accounts for the ghost stages and the reduction of the dominant L2 distance calculation.

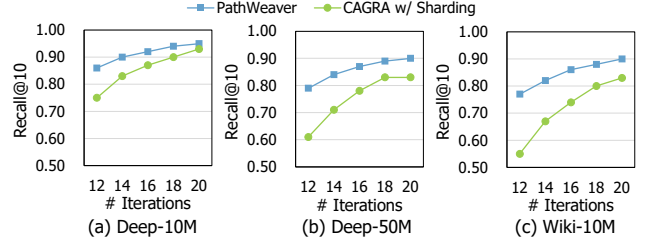


Figure 13: Accuracy comparison on different number of iterations.

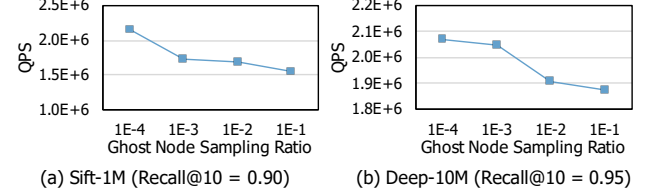


Figure 14: Sensitivity study on the relationship between the ghost node sampling ratio and QPS.

5.6 Detailed Analysis

In this subsection, we further investigate the effect of PathWeaver in terms of pipelining-based path extension, ghost staging, and direction-guided selection.

5.6.1 Pipelining-based Path Extension

To demonstrate the effect of pipelining-based path extension, we analyzed how the recall rate evolves by increasing the max number of search iterations in Fig. 13 on three datasets (Deep-10M, Deep-50M, and Wiki-10M). PathWeaver achieves high recall values with significantly fewer iterations. This is due to pipelining-based path extension, which enables each GPU to initiate the search process from a data point closer to the query using the search results from other GPUs. In contrast, the baseline requires substantially more iterations to reach comparable recall rates. For example, in the Deep-10M dataset, the baseline reaches a recall rate 0.90 with 18 iterations, while PathWeaver achieves this with only 14 iterations.

5.6.2 Ghost Staging

To evaluate the impact of the sampling ratio of ghost nodes on search performance, we conducted search operations on the Sift-1M and Deep-10M datasets using a single GPU, while varying the sampling ratio for building the ghost shard. As shown in Fig. 14, higher QPS was observed at lower sampling ratios of ghost nodes. For example, the QPS is $1.39\times$ higher for sampling ratio 0.0001 compared to 0.1 in the Sift-1M dataset. We attribute this to the connections among the smaller set of ghost nodes, which facilitate larger iteration jumps

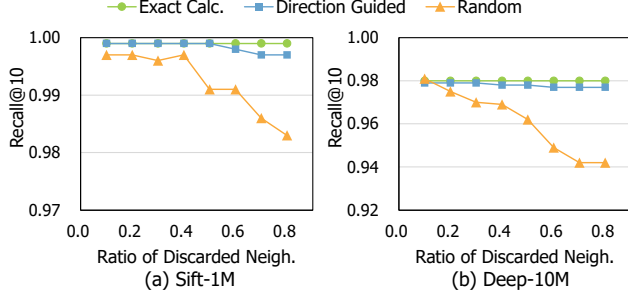


Figure 15: Comparison of neighbor selection strategies by varying the ratio of discarded neighbors.

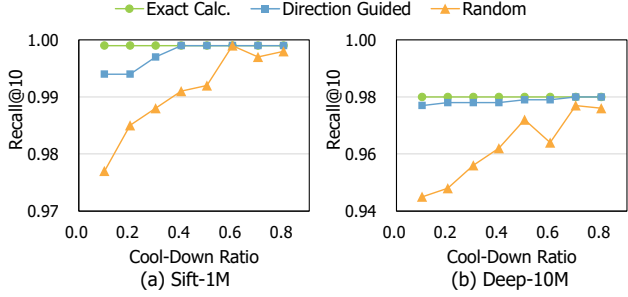


Figure 16: Comparison of neighbor selection strategies by varying the cool-down ratio of search iterations against the total iteration. A small cool-down ratio indicates that direction-guided selection or random selection is applied during the majority of the iterations.

within the original graph. The result indicates that a small-sized ghost shard is often sufficient, which provides another explanation for why ghost staging can bring speedup.

5.6.3 Direction-Guided Selection

To demonstrate the effectiveness of using direction bits as the neighbor discarding metric in direction-guided selection, we compared the performance of direction-guided selection against exact calculation (no discarding) and random neighbor discarding (randomly selecting neighbors).

Initially, the recall rate was evaluated for various discarded neighbor ratios while using a cool-down ratio of 0.5, as shown in Fig. 15. Compared to exact calculations, which achieve ideal recall, neighbor discarding in direction-guided selection results in a slight recall drop of at most 0.003, whereas random neighbor discarding causes a significant recall degradation of at most 0.038 for the Deep-10M dataset. Notably, PathWeaver maintains robust recall even with a discarding ratio of 0.7, in contrast to the significant recall decline observed with the random method.

Next, we analyze the recall rate by adjusting cool-down ratios with a fixed neighbor discarding ratio at 0.5, as shown

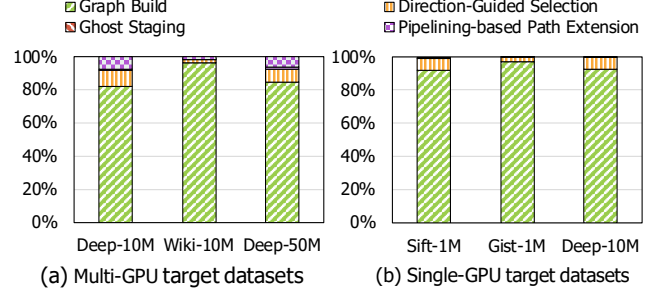


Figure 17: Graph build time overhead analysis.

in Fig. 16. Similar to the prior analysis, PathWeaver shows robust recall even when the cool-down ratio decreases, while the random method fails to maintain high recall.

For example, at a cool-down ratio of 0.3, neighbor discarding in direction-guided selection demonstrates only a minor recall drop of 0.002, compared to a significant degradation of 0.032 with the random discarding approach on the Deep-10M dataset. Both analyses demonstrate the effectiveness of using the direction of vectors when filtering out neighbors.

5.7 Graph Build Time Analysis

The preprocessing steps of PathWeaver, such as generating inter-shard connection edges for pipelining-based path extension, ghost nodes connections for ghost staging, and direction bit vectors for direction-guided selection, necessitate an overhead analysis to evaluate their impact. Therefore, we perform a breakdown analysis of the proximity graph build time, as shown in Fig. 17. Note that PathWeaver uses the graph build algorithm of CAGRA [44], represented as ‘graph build’.

The results indicate that the overall overhead is less than 10% for datasets targeting a single GPU. The overhead of constructing ghost node connections was lower than generating direction bit vectors. This is because the latter requires creating direction bit vectors for all edges in the graph, while the former requires connecting a relatively smaller number of data points. For multi-GPU target datasets, the overhead was less than 4% for Wiki-10M, while it reached 15% for Deep-50M. This is because building the graph also involves many L2 distance calculations, whose overhead is proportional to the vector dimension, similar to the search kernel. Additionally, the overhead of constructing ghost shards remains negligible, with the overheads of generating inter-shard connections and direction bit vectors being comparable. The minimal overhead of creating inter-shard connections arises from leveraging the existing connections in the proximity graph. These connections provide information about which node in the next shard is closest to the target node in the current shard. Overall, for both cases of single- and multi-GPU settings, the additional graph build overhead from PathWeaver’s designs is small.

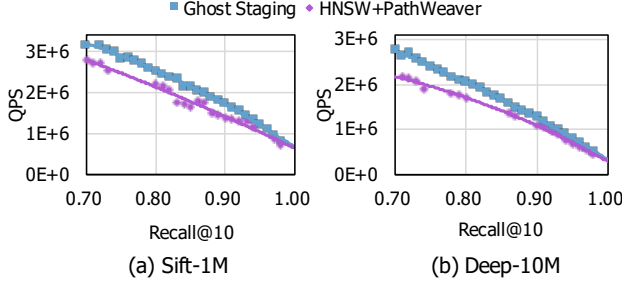


Figure 18: Comparing ghost staging and GPU-based HNSW.

6 Discussion

6.1 Comparing Ghost Staging and Existing Hierarchical Graph Approaches

Although ghost staging is a kind of hierarchical method, it differs from existing methods such as HNSW [40] and GGNN [25] in its objective and implementation. HNSW sequentially inserts each node into level n with exponentially decreasing probability as n increases, constructing a proximity graph at each layer. On the other hand, GGNN partitions the nodes, builds a graph within each partition, and finally merges them by selecting nodes for the next layer.

In contrast, ghost staging starts from an already-built proximity graph, and the extra layer is constructed by sampling vertices and connecting them. While existing hierarchical solutions aim to improve overall reachability and convexity by introducing hierarchy during graph construction, our approach uses the additional ghost stage solely to efficiently identify an entry node on the base proximity graph.

To investigate the difference, we conduct searching on the HNSW graph with our GPU-based search kernel and compare the results with those of ghost staging (direction-guided selection and pipelining-based path extension are disabled for fair comparison). As depicted in Fig. 18, ghost staging consistently achieves faster search speed compared to the HNSW graph even when it's searched with GPUs.

6.2 Managing Dynamic Updates

While PathWeaver currently targets static graphs, it is worthwhile to discuss dynamic updates to the proximity graphs [60]. Because PathWeaver is based on shards, any update would only involve modifying the affected shard. When rebuilding is needed, the cost mostly comes from the affected local graph, because the auxiliary data introduced by PathWeaver only accounts for a small portion as reported in Fig. 17.

For a small number of insertions, nodes can be added to one of the existing shards, followed by a rebuilding for the associated graph and the auxiliary data. If the number of insertions is small enough, the growth of the shard will not

cause load imbalance. In such a scenario, the inter-shard edges can be incrementally updated because the small change in the local graph does not affect the similarity between existing vertices. For a large number of insertions, an additional shard can be created without modifying existing ones. For a small number of deletions, a deleted node may still act as a bridge between its neighbors. To preserve connectivity, a deletion flag can be used to logically remove the node. However, when a substantial portion of a shard is deleted, rebuilding the shard and its associated structures becomes beneficial.

6.3 Comparing Direction-Guided Selection with Static Graph Pruning Strategy

In ANNS, several approaches perform graph pruning [44, 50] in a static manner to reduce memory usage and improve query efficiency. This strategy reduces graph density by eliminating less important edges during index construction. While static pruning offers a consistent structure across all queries, it may not be optimal for query-specific behaviors.

On the contrary, direction-guided selection can be considered a dynamic pruning strategy. Instead of permanently discarding edges, it dynamically chooses a subset of neighbors during the search process, based on the direction to the query. One potential issue of its fixed top- n selection is that it might discard important candidates. Even though we did not observe a significant drop in recall in our experiments, one could instead prune based on the similarity criteria, as done in [66] for neural ranking. Such a method could preserve good candidates, at the potential cost of warp imbalances due to non-uniform pruning.

6.4 Overhead Analysis of Pipelining-based Path Extension

In this subsection, we quantify the communication overhead of pipelining-based path extension. First, the volume of inter-GPU communication is only $Q \times b_{idx}$, where Q is the number of queries and b_{idx} is the number of bytes per transmitted index. In contrast, the amount of GPU memory accesses scales with $I \times J \times Q \times v \times b_{elem}$, where I is the number of search iterations, J is the out-degree of each node, v is the vector dimension, and b_{elem} is the number of bytes per vector element.

Although inter-GPU channels (e.g., NVLink) are typically over $10\times$ slower than GPU's memory bandwidth, the dominant cost still lies in the GPU's memory term, as the product $J \times v$ often exceeds 10^4 . As a result, the reduction in I achieved by pipelining-based path extension has a much greater impact on total latency than the relatively small inter-GPU communication cost.

7 Related Works

7.1 Graph-based ANNS Solutions

CPU-based solutions. A range of CPU-based solutions has been proposed to enhance the construction and traversal of proximity graphs for approximate nearest neighbor (ANN) search. Key studies, such as [20, 21, 40, 41, 54], focus on designing efficient graph structures that are leveraged by the beam search algorithm to achieve scalable and effective ANN retrieval. To improve search efficiency, several methods incorporate further optimizations [9, 12, 16, 22, 42, 43, 61, 64]. For example, subgraph sampling and edge pruning techniques are utilized in [64] to reduce traversal overhead, while [16] explores reordering graph structures to enhance cache efficiency. Additionally, [12] introduces a novel distance function approximation that accelerates the search process, demonstrating significant performance gains in graph-based ANNS.

GPU-based solutions. SONG [65] represents the first graph-based ANN search solution for GPUs, leveraging GPU parallelism to address the bottleneck of computationally expensive distance calculations while optimizing data structures through various approaches. GANNS [62] and GGNN [25] build upon this by focusing on GPU-friendly implementations that efficiently utilize shared memory for maintaining data structures and parallelizing their operations. CAGRA [44] enhances graph construction and search processes on GPUs, employing techniques such as warp splitting and forgettable hash to achieve high throughput. On the other hand, PathWeaver moves beyond the sole reliance on proximity graphs during the search process. By integrating information from adjacent shards and considering query directionality, it achieves enhancements in both speed and accuracy.

Other platforms. Graph-based ANNS solutions have also been explored on alternative platforms to enhance performance [30, 36, 51, 57, 59, 63, 67]. Pyramid [67] introduces a hierarchical near-memory-computing (NMC) architecture specifically designed for efficient graph-based ANNS. Computational storage devices (CSD) are leveraged in works such as [36, 51], where software-hardware co-design accelerates search algorithms by integrating computation closer to data storage. DF-GAS [63] proposes a distributed FPGA-as-a-Service architecture, enabling parallel search operations across both full graphs and subgraphs. CXL-ANNS [30] employs memory disaggregation from the host via CXL to optimize graph-based ANNS performance.

7.2 Scaling ANNS to Large Datasets

For ANNS on large datasets, various approaches focus on compressing data vectors. Quantization-based methods [23, 26, 32, 33] achieve this by clustering nodes within a graph and replacing them with centroid-based representations. Similarly, SONG [65] utilizes 1-bit random projection to enable large

datasets to fit within GPU memory.

Another prominent approach to handling large datasets involves hierarchical or hybrid architectures. Methods such as [24, 29, 45, 47, 49, 50] utilize storage devices to store large datasets, whereas BANG [53] combines CPU and GPU resources by maintaining the large graph index in CPU memory and leveraging GPU acceleration. FusionANNS [52] employs a combination of CPU-GPU collaboration and SSDs to optimize performance. These approaches enable cost-efficient processing of large datasets, though the achievable speedup remains inherently limited by the constraints of single-machine architectures.

Sharding datasets to fit into device memory and utilizing multiple devices has been widely investigated. Methods such as [14, 25, 31, 36] employ a multi-device strategy by partitioning datasets across devices, processing queries independently on each, and aggregating the results on the host. While these approaches enable distributed processing, achieving substantial and consistent speed-up as the number of devices increases often remains challenging. In contrast, our pipelining-based path extension achieves a proportional speedup with the number of machines used, effectively addressing the scalability limitations observed in previous methods.

8 Conclusion

We propose PathWeaver, a multi-GPU framework for graph-based ANNS that provides scalable and efficient performance on large datasets. PathWeaver addresses the main bottlenecks of graph-based ANNS through three key innovations: GPU-to-GPU communication to reduce unnecessary search iterations, a representative dataset search to refine starting data points, and data point filtering to minimize unnecessary memory access and distance calculation. Our evaluation results demonstrate that PathWeaver outperforms both CPU- and GPU-based solutions, achieving higher performance while maintaining accuracy.

Availability

We have open-sourced the code of PathWeaver and made it publicly available. The GitHub repository for PathWeaver can be found at <https://github.com/AIS-SNU/PathWeaver>.

Acknowledgement

This work was partially supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2022R1C1C1011307), and Institute of Information & communications Technology Planning & Evaluation (IITP) (RS-2024-00395134, RS-2024-00347394, RS-2023-00256081, RS-2021I1211343). Jinho Lee is the corresponding author.

References

- [1] Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. <https://big-ann-benchmarks.com/neurips21.html>.
- [2] Chroma. <https://trychroma.com/>.
- [3] MongoDB. <https://www.mongodb.com/>.
- [4] NVIDIA NVLink Bridge. <https://www.nvidia.com/en-us/products/workstations/nvlink-bridges/>.
- [5] Pinecone. <https://www.pinecone.io/>.
- [6] Weaviate. <https://weaviate.io/>.
- [7] Wiki-all Dataset. https://docs.rapids.ai/api/cuvs/stable/cuvs_bench/wiki_all_dataset/.
- [8] L. Amsaleg and H. Jégou. Datasets for approximate nearest neighbor search. <http://corpus-texmex.irisa.fr/>.
- [9] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Accelerated Nearest Neighbor Search with Quick ADC. In *International Conference on Multimedia Retrieval (ICMR)*, 2017.
- [10] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [11] Nandan Banerjee, Ryan C. Connolly, Dimitri Lisin, Jimmy Briggs, and Mario E. Munich. View management for lifelong visual maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.
- [12] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Finger: Fast inference for graph-based approximate nearest neighbor search. In *ACM Web Conference (WWW)*, 2023.
- [13] 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 *ACM International Conference on Information & Knowledge Management (CIKM)*, 2022.
- [14] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems*, 2019.
- [15] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Symposium on Computational Geometry (SCG)*, 1994.
- [16] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. In *International Conference on Neural Information Processing Systems (NIPS)*, 2022.
- [17] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 1990.
- [18] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The Faiss library. *arXiv:2401.08281*, 2024.
- [19] P. G. Emma. Understanding Some Simple Processor-performance Limits. *IBM Journal of Research and Development*, 1997.
- [20] Cong Fu and Deng Cai. Efanna : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv:1609.07228*, 2016.
- [21] 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 (VLDB)*, 2019.
- [22] Jianyang Gao and Cheng Long. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proceedings of the ACM on Management of Data (PACMOD)*, 2023.
- [23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013.
- [24] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *ACM Web Conference (WWW)*, 2023.
- [25] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data*, 2023.

- [26] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning (ICML)*, 2020.
- [27] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment (VLDB)*, 2015.
- [28] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing (STOC)*, 1998.
- [29] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *arXiv:2211.12850*, 2022.
- [30] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *USENIX Annual Technical Conference (ATC)*, 2023.
- [31] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 2021.
- [32] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.
- [33] Yannis Kalantidis and Yannis Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.
- [34] Tanaka Kanji, Chokushi Yuuto, and Ando Masatoshi. Mining visual phrases for long-term visual SLAM. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.
- [35] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*, 2020.
- [36] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Transactions on Computers*, 2023.
- [37] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *International Conference on Management of Data (SIGMOD)*, 2020.
- [38] 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 & Data Engineering*, 2020.
- [39] Ting Liu, Charles Rosenberg, and Henry A. Rowley. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *IEEE Workshop on Applications of Computer Vision (WACV)*, 2007.
- [40] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.
- [41] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 2014.
- [42] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024.
- [43] Sameer A. Nene and Shree K. Nayar. A Simple Algorithm for Nearest Neighbor Search in High Dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.
- [44] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *IEEE International Conference on Data Engineering (ICDE)*, 2024.
- [45] Yu Pan, Jianxin Sun, and Hongfeng Yu. LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing. In *IEEE International Conference on Big Data (BigData)*, 2023.
- [46] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *IEEE/CVF conference on computer vision and pattern recognition workshops (CVPRW)*, 2020.

- [47] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2020.
- [48] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.
- [49] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv:2105.09613*, 2021.
- [50] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.
- [51] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable billion-point approximate nearest neighbor search using SmartSSDs. In *USENIX Annual Technical Conference (ATC)*, 2024.
- [52] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuechang Zhang, Junhua Zhu, and Yu Zhang. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. *arXiv:2409.16576*, 2024.
- [53] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv:2401.11324*, 2024.
- [54] Javier Vargas Muñoz, Marcos A. Gonçalves, Zanoni Dias, and Ricardo da S. Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition*, 2019.
- [55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A Purpose-Built Vector Data Management System. In *International Conference on Management of Data (SIGMOD)*, 2021.
- [56] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment (VLDB)*, 2021.
- [57] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai “Helen” Li, and Yiran Chen. NDSEARCH: Accelerating Graph-Traversal-Based Approximate Nearest Neighbor Search through Near Data Processing. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2024.
- [58] Frank F. Xu, Uri Alon, and Graham Neubig. Why do nearest neighbor language models work? In *International Conference on Machine Learning (ICML)*, 2023.
- [59] Weihong Xu, Junwei Chen, Po-Kai Hsu, Jaeyoung Kang, Minxuan Zhou, Sumukh Ping, Shimeng Yu, and Tajana Rosing. Proxima: Near-storage Acceleration for Graph-based Approximate Nearest Neighbor Search in 3D NAND. *arXiv:2312.04257*, 2023.
- [60] Zhaozhuo Xu, Weijie Zhao, Shulong Tan, Zhixin Zhou, and Ping Li. Proximity graph maintenance for fast online nearest neighbor search. *arXiv:2206.10839*, 2022.
- [61] Shangdi Yu, Joshua Engels, Yihao Huang, and Julian Shun. PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search. *arXiv:2312.03940*, 2023.
- [62] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *IEEE International Conference on Data Engineering (ICDE)*, 2022.
- [63] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.
- [64] Minjia Zhang, Wenhan Wang, and Yuxiong He. Grasp: Optimizing graph-based nearest neighbor search with subgraph sampling and pruning. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2022.
- [65] Weijie Zhao, Shulong Tan, and Ping Li. SONG: Approximate Nearest Neighbor Search on GPU. In *IEEE International Conference on Data Engineering (ICDE)*, 2020.
- [66] Weijie Zhao, Shulong Tan, and Ping Li. Guitar: Gradient pruning toward fast neural ranking. In *International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2024.

- [67] Zhenhua Zhu, Jun Liu, Guohao Dai, Shulin Zeng, Bing Li, Huazhong Yang, and Yu Wang. Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search. In *ACM/IEEE Design Automation Conference (DAC)*, 2023.