Directional Beam Search: I/O-Efficient ANNS on GPU

Zikun Wang
Northeastern University
Boston, United States
wang.zikun@northeastern.edu

Hunter McCoy
Northeastern University
Boston, United States
mccoy.hu@northeastern.edu

Prashant Pandey
Northeastern University
Boston, United States
p.pandey@northeastern.edu

ABSTRACT

Approximate nearest neighbor search (ANNS) is a critical problem in modern machine learning and information retrieval, and GPU-based solutions offer strong performance through massive parallelism and high memory bandwidth. State of the art ANNS techniques build a traversable graph index over the vectors and perform a greedy beam search at query time to retrieve the nearest neighbors. However, the existing beam search approach is wasteful and heavily bound by memory access throughput. We propose *directional beam search*, which stores a compact locality-sensitive hashing (LSH) summary on each graph edge and uses it to estimate neighbor distances during the search, reducing the I/O per explored node to a single probe. We implement it in Jasper and compare against conventional beam search, the state-of-the-art GPU library CAGRA, and the scale-out beam search in BANG. Scaled out to host memory, it reaches up to 28× higher throughput than conventional beam search and, unlike BANG, requires no additional GPU memory for quantized vectors, yielding higher recall on all datasets and up to 8× higher throughput than BANG. On device memory, directional beam search achieves up to 7.7× higher throughput than CAGRA at recall 10@10 ≥ 90%.

VLDB Workshop Reference Format:
Zikun Wang, Hunter
McCoy, and Prashant Pandey. Directional Beam Search: I/O-Efficient ANNS on GPU. VLDB 2026 Workshop: The 2nd Workshop on Vector Databases.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/saltsystemslab/jasperpy.

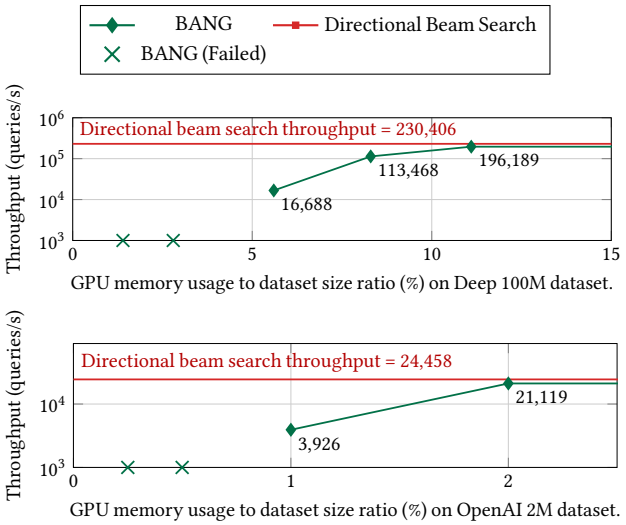


Figure 1: Throughput at recall 10@10 ≥ 90%. BANG requires additional GPU memory to store quantized vectors, and achieving high throughput requires storing larger quantized vectors on the GPU. In contrast, the host-side variant of directional beam search requires no additional GPU memory.

1 INTRODUCTION

Approximate nearest neighbor search (ANNS) is a foundational primitive across modern machine learning systems. It powers retrieval-augmented generation (RAG), where relevant documents are fetched from large corpora to ground large language model outputs; it is increasingly used to manage long-context inference by retrieving the most relevant entries from an otherwise prohibitively large key-value (KV) cache [21]; and it remains a workhorse of large-scale recommender systems [3]. As the underlying datasets grow to hundreds of millions or billions of vectors, the efficiency and scalability of ANNS directly determine the cost and latency of these applications.

Among the many families of ANNS algorithms, graph-based methods have consistently been shown to offer the best performance–accuracy trade-off [14, 20]. These methods organize the dataset into a navigable proximity graph and answer a query by performing a greedy *beam search*: starting from an entry point, the search repeatedly explores the neighborhood of the most promising candidate, computing the distance from the query to each neighbor in order to decide where to move next.

The fundamental bottleneck of graph-based ANNS is memory access rather than computation. Exploring a single node requires fetching its neighbor list and then the full vector of every neighbor in that list, resulting in many random memory probes per step of the search. This problem is greatly amplified in GPU-accelerated ANNS. To scale beyond the limited capacity of GPU memory, the graph topology and the raw vectors are often kept in host memory, so each of these random probes becomes a fine-grained transfer over the comparatively slow PCIe bus, leaving the GPU's massive compute and bandwidth largely idle.

Scaling GPU ANNS to billion-vector datasets has been approached from two directions. The first direction compresses the dataset to fit within GPU memory, relying on inverted indices and vector quantization to trade accuracy for capacity [2, 4, 5, 12]. A second set of approaches instead scale out beyond GPU memory, keeping the full-precision vectors in host memory or on SSD and relying on CPU/GPU cooperative filtering and re-ranking to answer queries [22]. The most notable scale-out system, BANG [23], keeps product-quantized (PQ)

Proceedings of the VLDB Endowment. ISSN 2150-8097.

versions of all vectors resident in GPU memory and uses them to approximate distances during the search, fetching the full vectors from host memory only for a final re-ranking step. This avoids fetching full vectors on every probe and delivers strong throughput.

However, this design is fragile: it requires the PQ representation of *every* vector in the dataset to fit in GPU memory at all times. In many deployments, the GPU memory available to ANNS is only a small fraction of the dataset size. This is especially acute for GPU-native RAG during LLM inference, where the model weights and the KV cache consume the bulk of GPU memory and leave the retrieval index only a thin slice. When the ratio of available GPU memory to dataset size is small and as higher accuracy demands larger quantized codes, requiring even the quantized vectors to be GPU-resident becomes a hard scalability wall, making efficient scale-out GPU ANNS essential. For example, on the Deep dataset, BANG must spend roughly 2 GB of GPU memory on PQ codes just to reach 90% recall (10@10), and even then delivers 8× lower throughput than the host-side variant of directional beam search; only when given substantially more GPU memory does it pull ahead (Figure 1).

In this paper we propose *directional beam search*, a query algorithm that eliminates the redundant probes incurred when exploring a node. Instead of fetching every neighbor's vector to compute exact distances, directional beam search incurs *exactly one* probe per explored node. The key idea is to store, alongside each edge, a compact summary of the edge's direction obtained through locality-sensitive hashing (LSH). At query time we use these summaries to bin the neighbors of a node and estimate which one is closest to the query vector, and we follow only that estimated-closest neighbor deeper into the search. Because the per-edge summaries are small and stored inline with the graph, a single probe retrieves everything needed to decide the next step, without ever fetching the neighbors' full vectors.

We implement directional beam search in Jasper, a GPU-accelerated ANNS engine [15], and evaluate it in both on-device and host-memory using the same search kernel in each. On device memory, it outperforms all baselines, achieving up to 7.7× higher throughput than the state-of-the-art GPU library CAGRA [18] at recall 10@10 ≥ 90%.

When scaled out to host memory, directional beam search reaches up to 28× higher throughput than conventional beam search, whose performance collapses under PCIe-bound random access. It is competitive with BANG while removing BANG's core limitation: it keeps nothing extra in GPU memory and is not bottlenecked by PQ quality. Directional beam search performs 1.8× faster than BANG on BigANN 10M 100@100 at recall ≥ 90%. On high-dimensional data this gap is even bigger, on OpenAI (100@100), BANG tops out at 84.6% recall whereas directional beam search reaches 99.0% while matching or exceeding its throughput.

2 PRELIMINARIES

We study the k-nearest-neighbor problem: given a dataset of n vectors in $\mathbb{R}^d$ and a query vector q, return the k dataset vectors closest to q under the Euclidean distance. Exact search is prohibitively expensive at scale, so approximate nearest neighbor search (ANNS) instead returns k candidate neighbors and is measured by *recall* $k@k$, the fraction of the true k nearest neighbors contained in the k returned results. Directional beam search builds on two pieces of background, which we review here. Section 2.1 introduces locality-sensitive hashing, which we use to summarize the direction of each graph edge and estimate distances, and Section 2.2 describes the GPU memory hierarchy, which governs the cost of the memory accesses that dominate graph-based ANNS. We defer the graph index and the beam search procedure that our method modifies to Section 3.

2.1 Locality Sensitive Hashing

Locality sensitive hashing (LSH) is a technique for finding similar items in a dataset without comparing all items to each other [9, 10]. Whereas a traditional hash function seeks to minimize the number of collisions (two keys receiving the same output value), a locality sensitive hash function attempts to cause collisions between similar input keys while avoiding collisions for dissimilar ones.

Formally, a hash family $\mathcal{H}$ is (r,cr,p_1,p_2)-sensitive if there exists a pair of distance cutoffs r and cr where $c > 1$ and probabilities $p_1 > p_2$ such that, for a pair of keys x and y with distance between them $d(x,y)$ and a hash function h drawn at random from $\mathcal{H}$, $\Pr[h(x) = h(y)] \geq p_1$ if $d(x,y) \leq r$ and $\Pr[h(x) = h(y)] \leq p_2$ if $d(x,y) \geq cr$.

In the context of graph-based ANNS indices, LSH is good at encoding general information about a vector's direction. Previous works like PiPNN [19] utilize hyperplane LSH to speed up the graph construction process. In our work, we build our estimation based on an LSH technique called cross-polytope hashing [1]. We choose cross-polytope hashing due to its simplicity and performance. Details of the hashing can be found in Algorithm 1. Overall, cross-polytope hashing randomly rotates the vector and assigns the hash as the vector's dimension with the largest magnitude. Moreover, it can assign multiple hashes to the vector by probing the vector's top-k dimensions with the largest magnitudes. In Section 4.4, we explore how cross-polytope hashing helps us with distance estimation.

2.2 GPU Memory Hierarchy

On GPUs, the memory hierarchy is divided into three tiers: L1 cache/shared memory, L2 cache, and device memory. Each streaming multiprocessor (the processing unit) contains its own L1 cache. Part of this L1 cache can be dedicated as shared memory, which is directly addressable by the application. If an L1 cache miss occurs, the data will be fetched from the L2 cache, which is shared among all streaming multiprocessors on a GPU (and from device memory if an L2 cache miss occurs). Both L1 and L2 have a cache line size of 128 bytes and are fetched in 32-byte units.

CUDA allows applications to directly map pinned host memory from CPU to GPU, using functions like `cudaMallocHost`. When a kernel accesses the pinned host memory, the request goes through the PCIe connection and is stored directly in the L1 cache. Since the GPU does not store or cache any copy of the pinned host memory, every L1 cache miss on pinned host memory must go through PCIe. Even though this allows us to scale datasets larger than the GPU memory onto CPU memory, it is orders of magnitude slower due to the high latency and low bandwidth of the PCIe connection between CPU and GPU.

3 JASPER

Our implementation is built on top of the Jasper [15] ANNS index. Jasper is a state-of-the-art ANNS index for in-memory ANNS queries

on NVIDIA GPUs. Jasper is based off of the Vamana index, using the BeamSearch and RobustPrune algorithms from DiskANN [11] and the batch parallel construction algorithm from ParlayANN [14]. It supports batch and incremental insertions, queries, and deletions.

3.1 Vamana and Graph ANNS Indices

Jasper builds a Vamana index, which is a form of *graph index*. A graph-based ANNS represents vectors as vertices in a directed graph. Typically, these indices construct a *navigable proximity graph*, in which a query can be answered by greedily walking from an entry point toward the query along graph edges using beam search (see Section 3.2) [11, 13].

ANNS graphs are directed graphs where each vertex has a bounded outgoing edge degree R. The edge set E is selected to ensure that the graph remains sparse while still preserving navigability. Most modern indices achieve this through a *neighbor-pruning* rule that, when selecting the out-edges of a vertex, discards a candidate neighbor whenever an already-selected neighbor lies in a similar direction but closer, yielding a small, diverse, and well-distributed edge set [7, 11].

Other examples include HNSW [13], which layers navigable small-world graphs into a hierarchy to provide logarithmic-expected-length search paths; NSG [7], which sparsifies an approximate k-nearest-neighbor graph while preserving monotonic paths to the query; and Vamana, the graph constructed by DiskANN [11] to support billion-scale search from SSD on a single machine. ParlayANN [14] accelerates the Vamana index construction using a batch-parallel construction algorithm.

On GPUs, CAGRA [18] builds and searches proximity graphs using NN-Descent [6] followed by a custom pruning round, and Jasper [15] implements the Vamana batch-construction algorithm directly on GPUs.

3.2 Beam Search

In beam search, the Vamana index is iteratively traversed to find the approximate nearest neighbors of a given query vector [7, 11, 13]. The search always starts from a selected starting point, typically the *medoid*, the dataset point closest to the centroid of the graph.

During beam search, two lists are maintained: the **visited list**, and the **frontier**. The visited list maintains a set of nodes expanded during traversal, and the frontier is the set of best nodes encountered so far. The frontier is sized at a set with known as the **beam width** as it denotes the maximum fan-out of the greedy search. The frontier contains the ID of a node, whether or not that node has been visited yet, and the distance between that node and the target query. The search begins with the visited list empty and the medoid marked unvisited as the only node in the frontier.

In each iteration, we pop the closest unvisited vertex and read in the IDs of its outgoing neighbors. These neighbors are referred to as the **candidates** of this iteration, with the number of candidates in a given round being bounded by the outgoing degree R. For each candidate, we load the full vector and calculate its distance to the query. Any candidate that is already in the frontier is pruned: duplicates are not allowed.

Once the distances are gathered, we sort the candidates by distance and then merge them into the frontier. If the resulting list is larger than the max frontier size, it is clipped. This is the end of the iteration: the current vector is marked as visited and added to the visited list, and we then select a new node to visit.

This procedure repeats until all vertices in the frontier have been visited. After that, the frontier and visited set are returned, with the frontier representing the set of nearest neighbors for the query.

To run a Vamana index efficiently on GPUs, Jasper integrates the following optimizations into the Vamana procedures:

- *Shared memory for query state:* per-query state (frontier, candidate buffer, distances) lives in shared memory (directly addressable L1 cache), and Jasper aggressively optimizes to reduce shared memory usage and increase occupancy, allowing it to schedule more queries simultaneously.
- *Warp specialization to minimize divergence:* the irregular phases of beam search are confined to a single warp to minimize control-flow divergence.
- *Specialized loads to reduce register pressure:* distance computations in Jasper are tiled into 16-byte chunks to bound the per-thread accumulator state.
- *Specialized quantization for GPU memory access patterns:* Jasper implements the RaBitQ [8] quantization algorithm. This allows it to process quantized vector distances directly using sequential global-memory loads, avoiding the random codebook lookups and read amplification incurred by Product Quantization.

Queries are executed in parallel on the GPU, with a thread block assigned to each query. Candidate distance calculations are performed in parallel inside of query as well to boost throughput.

4 DIRECTIONAL BEAM SEARCH

Directional beam search optimizes the existing beam search approach by reducing the number of I/Os during queries. In this section, we cover the design of Directional beam search, the storage format, and the design of its distance estimator using cross-polytope hashing.

4.1 Design

In the original beam search algorithm and the parallel beam search implmenented in Jasper [15], the search eagerly explore all the candidate neighbors' distances to the query vector in parallel, bringing in each neighbor through an independent load. To avoid the cost of performing R independent vector lookups, we delay the calculation of exact distances by estimating them using the stored neighborhood routing information. This effectively reduces the number of I/O accesses for each beam search exploration from $1+R$ to 1, optimizing the search algorithm when the vectors and the index are stored in host memory.

We introduce a new list in directional beam search called the **candidate list**, which we use to explore the graph via the estimated distance. During directional beam search, we maintain three lists of elements in total.

- The **candidate list** maintains a set of vertices that are waiting to be explored, and the vertices are ordered by the LSH estimated distances (the LSH estimation technique is explained in Section 4.4).
- The **frontier list** maintains the set of vertices that got explored and added to the final result. It is ordered by the exact distances rather than the LSH estimations. When the search is completed, we return the top-k vertices in the frontier as the set of approximate nearest neighbors.

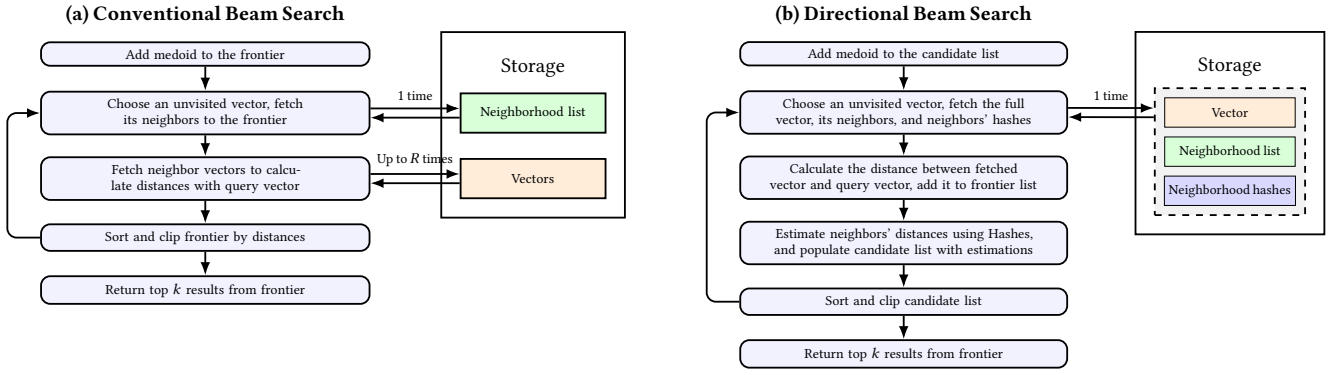


Figure 2: A side-by-side comparison between conventional beam search and directional beam search's I/O access pattern. R **is the maximum number of neighbors a vector can have.**

- The **visited list** maintains the set of all nodes that have been explored and added to the frontier list. We use this list to avoid adding duplicate elements to the candidate list.

We show the exact steps of directional beam search in Figure 2(b). Initially, we insert a starting vertex in the candidate list. In each iteration, we pop the element with the shortest estimated distance to the query vector from the candidate list. We then fetch that vertex's full vector representation, the list of outgoing neighbors, and the neighbor LSH hashes.

Using these hashes, we estimate the distance for each new neighbor in the list that we haven't visited, and then add them to the candidate list in sorted order using the estimated distance. Since we also fetched the full vector for the explored vertex, we can calculate the true distance between the explored vertex and the query vector, and we add this node to the frontier list. The same node will also be marked as visited in the visited list, so that we don't traverse the same node more than once.

The final step of each iteration is to sort the candidate list by the estimated distances and clip it to the target beam width. As in conventional beam search, directional beam search repeats the above steps until either a set limit is reached or there are no more candidates available to explore. In our implementation, we set the limit to be double the size of the beam width. Once the traversal is finished,

4.2 Index layout

To facilitate our search, we store additional information per edge. For each edge, we store k cross-polytope hashes and a 2-byte `bf16` distance, where k is an adjustable parameter for the accuracy vs. size trade-off. Within each hash, we reserve the first bit for the `sign` of the hashed dimension and use the remaining bits for the dimension index. Therefore, depending on the dataset's vector dimension, we can fit each hash in 1 byte if the dimension ≤ 128; otherwise, we use 2 bytes per hash.

4.3 Building hashes

After constructing the graph index and before queries begin, we need to compute the hashes for each neighborhood list and store them alongside the graph index. We refer to this phase as the building phase. In Section 5.2, we measure the overhead incurred by each step of the building phase.

Algorithm 1 Cross-polytope hash for each edges.

Require: For vector u's neighbors $v_i \in V$, given k as the number of hashes.

1: $\texttt{hash}_{(u,v)} = []$
2: $\texttt{mag}_{(u,v)} = ||v-u||$
3: $\hat{r} = \frac{v-u}{||v-u||}$
4: **for** $i \in \{0,\ldots,k-1\}$ **do**
5: $\quad j_i \leftarrow$ index of the i'th largest $|\hat{r}|$
6: $\quad \texttt{hash}_{(u,v)}[i] \leftarrow (j_i, \text{sign}(\hat{r}[j_i]))$
7: **end for**
8: **return** $\texttt{hash}_{(u,v)}, \texttt{mag}_{(u,v)}$

Pre-rotate the dataset. First, given a dataset of n vectors with d dimensions each, we generate a random orthogonal matrix R with $d \times d$ dimensions, and apply R to every vector to perform a random rotation on all of the vectors. Naively, this rotation costs $O(n \cdot d^2)$ operations. Using Hadamard Pseudo-Rotation Transform (proposed in [1]), we can do this rotation in $O(n \cdot d \log d)$ time. We only have to pay the rotation cost once during the initial building phase.

Store the edge magnitude. For each neighbor v in a neighbor list for u, we store the distance $||u-v||$ in 16 bits using the `bf16` format.

Hash and store the k hashes per edge. For each neighbor v in the neighbor list of u, we store k cross-polytope hashes to represent the edge. We show this calculation in Algorithm 1.

Calibrate constants for each of the k ranks. In the last step we only stored the indices of the k largest dimensions for the normalized relative vector $\hat{r}$, yet this is not enough information to estimate the dot product in Section 4.4. We can estimate the magnitude of each of the k ranks through sampling, as shown in Algorithm 2. We first generate a number of randomly oriented unit vectors in d dimensions, and compute the expected value for each of the top-k dimensions across all of the sampled vectors.

Algorithm 2 Calibrate constants for each of the k ranks.

Require: For a dataset with d dimensions, given k as the number of hashes, samples n vectors.

```
S = Sampled n unit vectors
c_per_rank = []
for v ∈ S do
    sorted ← sort(|v|, descending)
    for i ∈ {0,...,k−1} do
        c_per_rank[i] += sorted[i]
    end for
end for
norm_denorm = 0
for i ∈ {0,...,k−1} do
    c_per_rank[i] = c_per_rank[i]/n
    norm_denorm += c_per_rank[i]^2
end for
return c_per_rank, norm_denorm
```

4.4 Distance estimation

During each query iteration, we explore one vector and its corresponding neighborhood. When we load in avector while exploring, we get the explored vector u and all of its neighbors' indices. We also fetch the approximate information for this neighbor list from the build phase: for each neighbor v of node u, we store the magnitude of $v-u$ and k hashes (each representing a dimension index). Along with the newly loaded information, we have the exact query vector q stored in shared memory.

Our goal is to use this information to accurately estimate the Euclidean distance between each neighbor vector v and the query vector q. We can express this distance using the following expansion.

$$\|q-v\|^2 = \|(q-u)-(v-u)\|^2 \tag{1}$$

$$= \|q-u\|^2 + \|v-u\|^2 - 2\cdot\langle q-u, v-u\rangle \tag{2}$$

Since we know the exact vector q and u, we can calculate the first term $\|q-u\|^2$. We also already stored the magnitude between v and u, so we know $\|v-u\|^2$. Now our job is to estimate the term $\langle q-u, v-u\rangle$. Recall from our building phase that the hashes are for the normalized vector $\hat{r} = \frac{v-u}{||v-u||}$ in Algorithm 1. We can expand the dot product by splitting out the magnitude as:

$$\langle q-u, v-u\rangle = \|v-u\| \cdot \langle q-u, \hat{r}\rangle \tag{3}$$

We derive an accurate estimate of $\langle q-u, \hat{r}\rangle$ using our stored estimated information about $\hat{r}$. For each rank i, the top-k cross-polytope hashes give us the coordinate index j_i (`coord`) of $\hat{r}$'s i'th largest dimension and its sign $s_i \in \{-1,+1\}$ (`sign`). From Algorithm 2, we also have a calibrated constant c_i (`c_per_rank`) estimating how large the i'th rank should be for a randomly oriented unit vector, together with $Z = \sum_{i=0}^{k-1} c_i^2$ (`norm_denorm`). Using this information, we can use the following estimator:

$$\langle q-u, \hat{r}\rangle \approx \frac{1}{Z}\sum_{i=0}^{k-1} c_i s_i (q-u)[j_i] \tag{4}$$

The division by $Z = \sum_{i=0}^{k-1} c_i^2$ compensates for the entropy of $\hat{r}$ that is discarded when we keep only its top-k coordinates, and it makes the estimator unbiased in the regime that matters most for nearest-neighbor search: when q is close to v, the vector $q-u$ is nearly aligned with $\hat{r}$. Concretely, suppose $q-u = \alpha\hat{r}$. Then each captured term contributes $c_i s_i (q-u)[j_i] \approx \alpha c_i^2$, so the numerator is approximately $\alpha\sum_{i=0}^{k-1} c_i^2 = \alpha Z$, and dividing by Z recovers the true value $\langle q-u, \hat{r}\rangle = \alpha$. Without this normalization the truncation to top-k coordinates would systematically shrink the estimate.

4.5 Implementation details

The graph index is constructed using Jasper on the GPU, and the hashes and neighborhood distances are populated using another CUDA kernel. We implement directional beam search in Jasper using CUDA C++. The major differences between beam search in Jasper and directional beam search are:

- We maintain the candidate list alongside the original frontier in shared memory.
- We incorporate parallel distance estimation in the beam search.

It is worth noting that the same directional beam search implementation is used for both the device memory and host memory configurations; the only difference is the allocation method for the vectors and graph, which is switched from device memory (`cudaMalloc`) to pinned host memory (`cudaMallocHost`). This unified implementation demonstrates the versatility of our approach across different levels of the memory hierarchy.

5 EVALUATION

In this section, we evaluate the performance of *directional beam search* implemented in Jasper against the default beam search in Jasper (*conventional beam search*). We evaluate the performance on both device memory and host memory. We also compare the against BANG (both in device and host) and CAGRA (only in device). Finally, we perform a micro-benchmarks to evaluate the accuracy and cache hit rate of the *directional beam search*.

5.1 Setup

System setup. All experiments are conducted on a machine with an Intel(R) Xeon(R) 6710E CPU running at 2.40GHz with 1TB of memory. The GPU is an NVIDIA RTX PRO 6000 Blackwell Server Edition with 96GB of GDDR7 memory and peak bandwidth of 1597GB/s. The GPU is connected through PCI Express 5.0 x16 with peak read throughput of 64GB/s. We compiled our code using CUDA version 13.0. The source code is publicly available on GitHub: https://github.com/saltsystemslab/jasperpy/.

Dataset. To evaluate our performance, we choose the four datasets from the BigANN benchmark repository [20]. We employ 16 hashes for the LSH configuration in directional beam search. We report the information regarding all the datasets and corresponding overhead of the hash storage in Table 1.

Baselines. We compare *directional beam search* with the following baselines:

- Jasper [15]: Jasper is a state-of-the-art ANNS index for in-memory ANNS queries on NVIDIA GPUs. Jasper is based off of the Vamana index, using the BeamSearch and RobustPrune algorithms from DiskANN [11] and the batch parallel construction algorithm from ParlayANN [14].

Dataset	#Vec	Dim	MaxDeg	#Hash	Overhead	Total Size
BigANN	10M	128	64	16	11.5 GB	16.6 GB
Deep	10M	96	64	16	11.5 GB	16.0 GB
Gist	1M	960	64	16	1.0 GB	3.2 GB
OpenAI	2.3M	1536	64	16	2.4 GB	10.0 GB

Table 1: Space required to store hashes and total index for each of the datasets tested. The hash overhead is the space required to store the 16 hashes of each of 64 neighbors for every vertex in the dataset, and the total space is the entirety of memory needed to store the index, including vector, graph, and hash information.

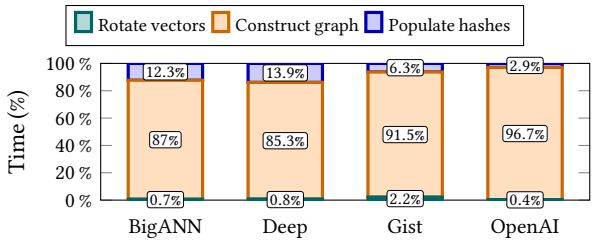


Figure 3: Construction time breakdown by percentage. For all datasets, graph construction takes up over eighty five percent of the total time, with the majority of the remaining time going towards populating the per-neighbor hashes. Vector rotation takes no more than three percent of construction time.

- CAGRA [18]: CAGRA is the state-of-the-art GPU ANNS algorithm, implemented in Nvidia's library CUVS [16]. It constructs the graph index using NN-descent and offers an optimized beam search kernel for queries. It only supports an in-GPU-memory index and does not scale out to CPU memory.
- BANG [23]: BANG supports billion-scale GPU ANNS by storing the product-quantized (PQ) vectors in GPU memory, while keeping the original vectors and graph in CPU host memory. In each beam search iteration, the GPU fetches the neighborhood for the explored vector and calculates its distance using the PQ vectors. When the beam search is complete, BANG re-ranks the results using the exact vectors. Jasper also offers a version of the search where the graph is stored in GPU memory.

5.2 Construction Overhead

In Figure 3, we show the overhead of computing hashes for directional beam search in Jasper compared to the base graph index construction time. The cost of rotating the data vectors at the start is negligible due to the fast matrix multiplication operation on GPU. The main construction time is still dominated by graph construction, cost between 85% to 96% of the total time.

5.3 Query Performance

We evaluate the query performance of directional beam search against other baselines. There are two set of experiments. The device memory experiments store vectors and graph index in GPU device memory. The host memory experiments scales the query to CPU memory. For the host version of directional beam search and conventional beam search implemented on Jasper, where we store the vectors and the graph index in host pinned memory. BANG stores the PQ vectors in device memory while keeping the full vectors and graph index in CPU memory.

Host memory. In Figure 4, we show the results for query throughput when vectors and the graph index are scaled out to CPU memory. On datasets with small vectors (BigANN and Deep), directional beam search outperforms BANG across all settings except BigANN 10@10, where the throughput and recall of the two methods match. On BigANN 10M 100@100 with recall $\geq 90\%$, directional beam search achieves 1.8× higher throughput than BANG. On Deep 10M 100@100 with recall $\geq$ 90%, directional beam search achieves 1.7× higher throughput than BANG.

As the vector size grows larger with Gist and OpenAI, BANG struggles to maintain recall at high beam widths. For Gist 100@100 and OpenAI 100@100, BANG's highest recall values are 93.25% and 84.58%, respectively. In contrast, directional beam search reaches 99.94% and 99.04% recall while maintaining better throughput. For these datasets, BANG's performance is heavily bounded by the quality of the PQ vectors it stores. Among these settings, BANG achieves better throughput than directional beam search only on Gist 1M 10@10, where it is 1.5× faster.

Across all the datasets, the conventional beam search baseline implemented on Jasper struggles to maintain throughput due to the PCIe bandwidth limit. For large datasets like OpenAI, its throughput drops below 1,000 queries per second at higher recall.

Device memory. In Figure 5, we show the results for query throughput when vectors and the graph index are stored in GPU memory. Directional beam search achieves the highest throughput at high recall across all datasets. For small datasets like BigANN and Deep, directional beam search shares the same performance as conventional beam search until recall $\geq$ 90%. When recall $\geq$ 90%, conventional beam search's throughput degrades rapidly. Both CAGRA and BANG consistently achieve worse throughput than directional beam search on these two datasets. For example, on the Deep 10M 10@10 dataset, directional beam search reaches 2.5× the throughput of CAGRA when recall is $\geq 99\%$.

Directional beam search performs exceptionally well on datasets with large vectors like Gist and OpenAI, which have 960 and 1536 dimensions, respectively. Despite the large vectors, directional beam search achieves high recall while maintaining high throughput. On the Gist 1M 100@100 dataset, directional beam search achieves 99.93% recall at 30,323 queries per second. In comparison, the strongest baseline, CAGRA, achieves only 98.97% recall at 14,636 queries per second using its highest beam width. On the OpenAI 2M 100@100 dataset, directional beam search achieves 99.04% recall at 48,121 queries per second, while the strongest baseline, CAGRA, achieves only 97.99% recall at 9,044 queries per second.

This performance improvement stems from the smaller memory accesses required by directional beam search at each iteration. Take the OpenAI dataset as an example. In the conventional beam search implementation (used by all three baselines), each iteration accesses up to $64 \times 1536 \times$ `sizeof(fp16)` $= 196.6$kB, ignoring the size of the neighborhood list. In contrast, directional beam search accesses only 5.2kB, a 37× reduction.

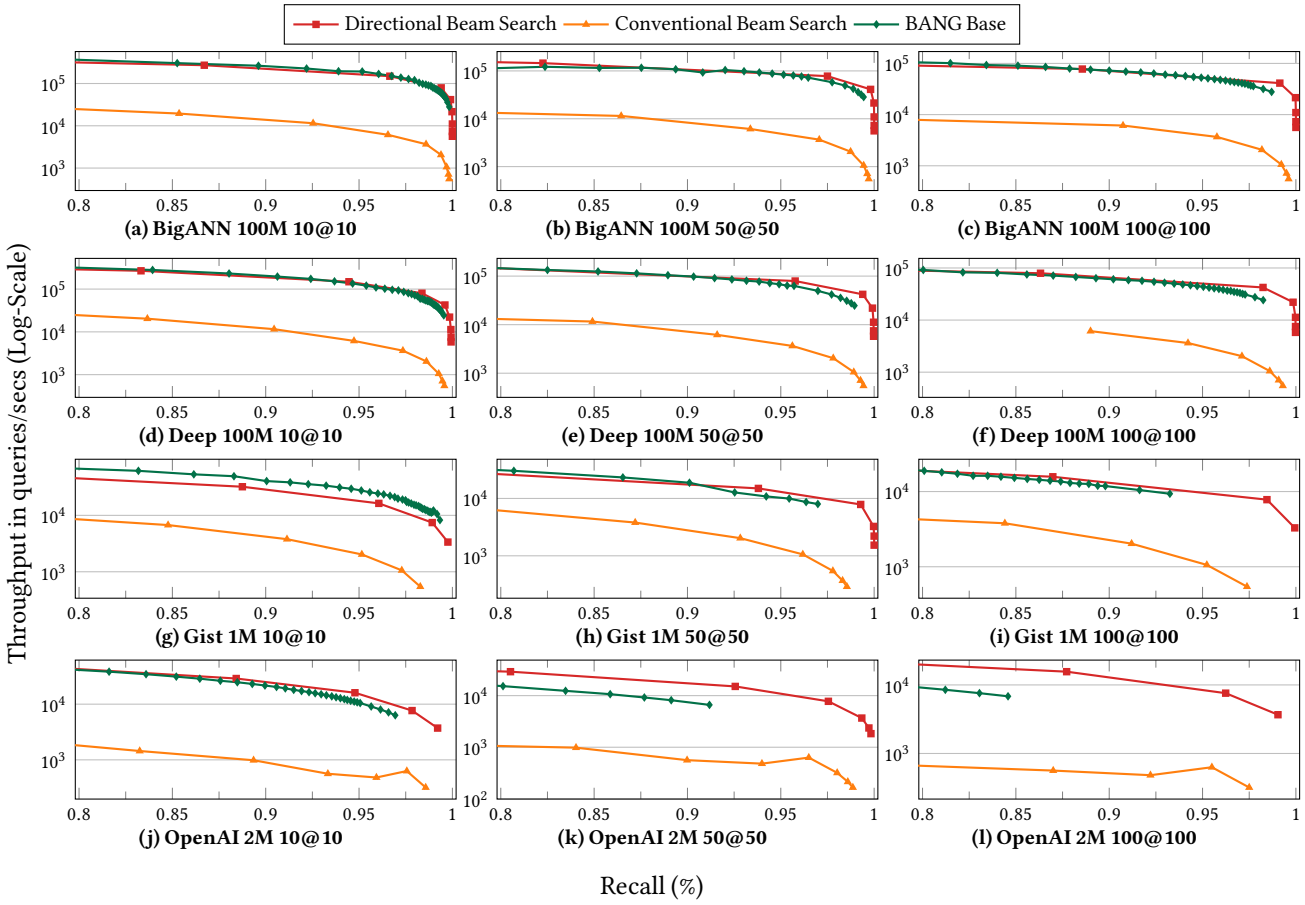


Figure 4: Query recall/throughput on CPU host memory. Y-axis is on log-scale; higher and more to the right is better.

5.4 Microbenchmarks

Estimator error distortion test. In Figure 6, we empirically measure the error rate of our distance estimator on the BigANN and OpenAI datasets. We first build the graph using Jasper, then sample 10,000 neighborhoods. For each neighbor in the sampled neighborhoods, we record the exact distance and the estimated distance produced by our estimator (number of hashes = 16) to a random query vector from the dataset. The estimates for both datasets are centered around the $y = x$ line; however, the OpenAI dataset is slightly more spread out because its high dimensionality means that 16 hashes cannot faithfully represent the actual vector.

Cache misses between conventional and directional beam search. In Figure 7, we compare the number of cache misses between directional beam search and conventional beam search as we increase the beam width. We collect the statistics using NVIDIA Nsight Compute [17]. At the highest beam width (1024), directional beam search incurs 9.7× fewer L1 cache misses. Reducing memory accesses is crucial for ANNS queries, and this lower cache miss rate directly explains the better throughput of directional beam search in Figure 4.

GPU memory constraint when scaling out to CPU memory. The biggest difference between directional beam search and BANG lies in how each handles GPU memory when scaling out to CPU host memory. Directional beam search stores everything in host memory and frees up any extra space for other potential uses (e.g., cache). When BANG scales out to host memory, it still needs to store PQ vectors in device memory, and the size of these PQ vectors directly affects its throughput. We demonstrate this in Figure 1, where we vary the memory space allocated to BANG's PQ vectors and plot its throughput on the Deep 100M and OpenAI 2M datasets, respectively. We plot directional beam search's throughput as a red horizontal line, since it does not require any additional GPU memory. It is worth noting that BANG relies on the PQ method from DiskANN [11], which hard-caps the size of each PQ vector at 96 bytes. Providing more memory than this does not increase the PQ vector size beyond the cap.

On the Deep 100M dataset, BANG fails to reach ≥ 90% recall 10@10 until we provide it with more than 5% of the original vector size, which is about 2 GB. With 2 GB of PQ vectors, BANG's throughput is 8× lower than that of directional beam search. As we increase the available memory to 4 GB, BANG is able to achieve higher throughput than directional beam search.

On the OpenAI 2M dataset, due to the hard cap of 96 bytes per PQ vector imposed by the DiskANN code, BANG struggles to match the throughput of directional beam search. Even with the largest

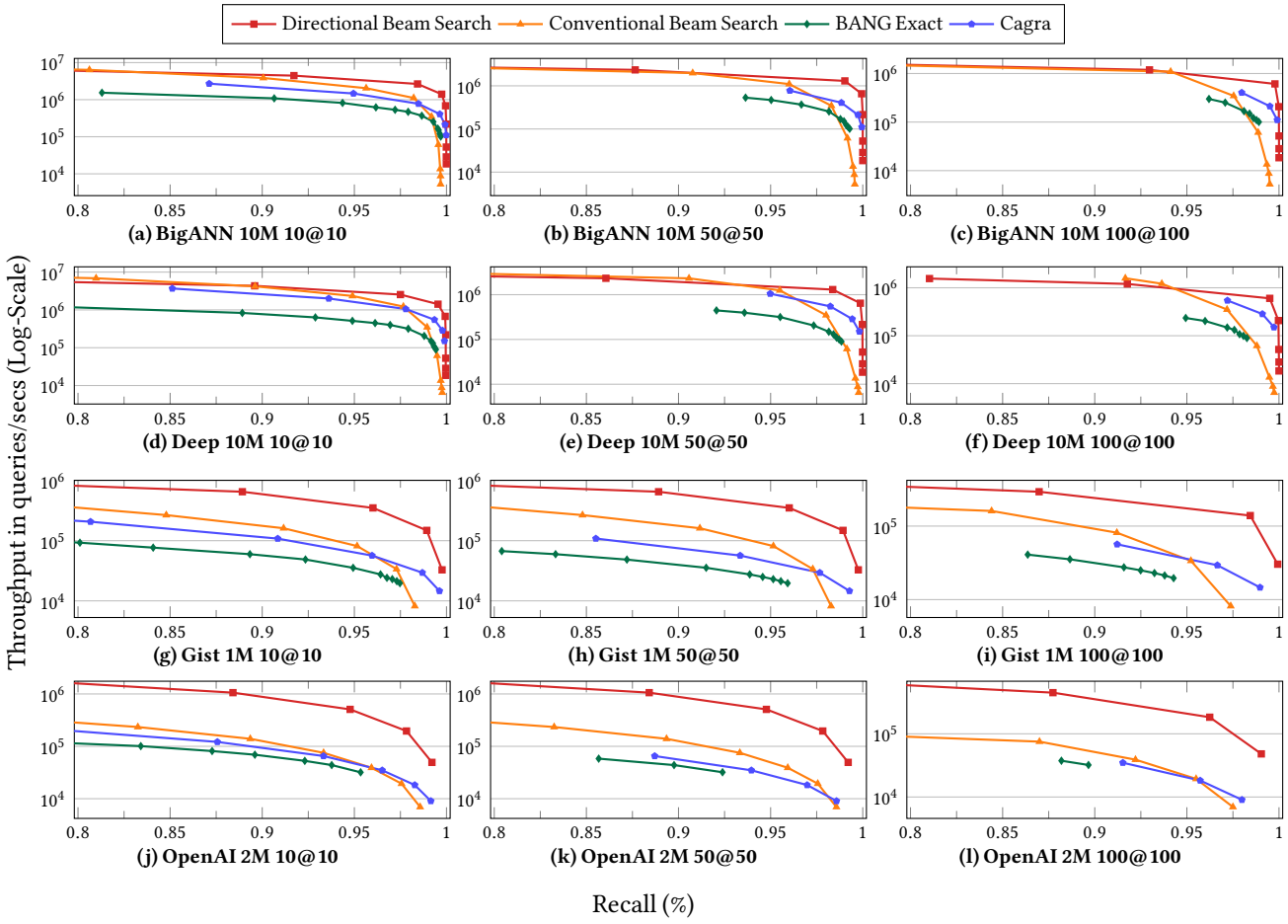
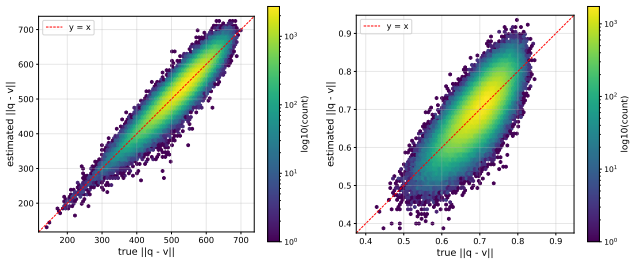


Figure 5: Query recall/throughput on GPU device memory. Y-axis is on log-scale; higher and more to the right is better.



(a) BigANN 10M **(b) OpenAI 2M**

Figure 6: Estimation error distortion test. We sampled 10,000 neighborhood from each dataset and estimated the distance of the neighbors against a random query vector. X-axis is the exact distance. Y-axis is the estimated distance.

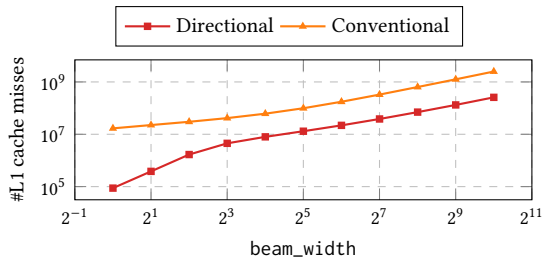


Figure 7: Number of L1 cache misses between directional beam search and conventional beam search. Lower is better.

allowed PQ vector size of 96 bytes, directional beam search still achieves 1.15× higher throughput at the same recall.

6 CONCLUSION

We presented *directional beam search*, a GPU-based ANNS query algorithm that cuts the I/O of beam search to a single probe per node by storing a compact locality-sensitive hash summary of each edge's direction. It delivers high throughput and recall on both device and host memory, achieving up to 28× higher throughput than conventional beam search.

ACKNOWLEDGMENTS

This research was funded in part by NSF grant OAC 2517201, 2513656.

REFERENCES

[1] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1225–1233.

[2] Alan Araujo, Willian Barreiros, Jun Kong, Renato Ferreira, and George Teodoro. 2025. IMI-GPU: Inverted multi-index for billion-scale approximate nearest neighbor search with GPUs. *J. Parallel and Distrib. Comput.* 200 (2025), 105066. https://doi.org/10.1016/j.jpdc.2025.105066

[3] Yoram Bachrach, Yehuda Finkelstein, Ran Gilad-Bachrach, Liran Katzir, Noam Koenigstein, Nir Nice, and Ulrich Paquet. 2014. Speeding up the Xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM Conference on Recommender Systems* (Foster City, Silicon Valley, California, USA) *(RecSys '14)*. Association for Computing Machinery, New York, NY, USA, 257–264. https://doi.org/10.1145/2645710.2645741

[4] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. 2019. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems* 99 (Oct. 2019), 295–307. https://doi.org/10.1016/j.future.2019.04.033

[5] Wei Chen, Xiao Ma, Jiangfeng Zeng, Yaoqing Duan, and Grace Zhong. 2021. Hierarchical quantization for billion-scale similarity retrieval on GPUs. *Computers & Electrical Engineering* 90 (March 2021), 107002. https://doi.org/10.1016/j.compeleceng.2021.107002

[6] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures. In *Proceedings of the 20th International Conference on World Wide Web (WWW) (WWW '11)*. Association for Computing Machinery, 577–586. https://doi.org/10.1145/1963405.1963487

[7] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474. https://doi.org/10.14778/3303753.3303754

[8] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3, Article 167 (2024), 27 pages. https://doi.org/10.1145/3654970

[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 518–529.

[10] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*. Association for Computing Machinery, 604–613. https://doi.org/10.1145/276698.276876

[11] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[12] Jeff Johnson, Matthijs Douze, and Herve Jegou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/tbdata.2019.2921572

[13] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[14] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 270–285. https://doi.org/10.1145/3627535.3638475

[15] Hunter McCoy, Zikun Wang, and Prashant Pandey. 2026. GPU-Accelerated ANNS: Quantized for Speed, Built for Change. arXiv:2601.07048 [cs.DB] https://arxiv.org/abs/2601.07048

[16] NVIDIA. 2026. cuvs. https://github.com/rapidsai/cuvs. https://github.com/rapidsai/cuvs

[17] NVIDIA Corporation. 2026. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. Accessed: 2026-06-11.

[18] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. arXiv:2308.15136 [cs.DS] https://arxiv.org/abs/2308.15136

[19] Tobias Rubel, Richard Wen, Laxman Dhulipala, Lars Gottesbüren, Rajesh Jayaram, and Jakub Łącki. 2026. PiPNN: Ultra-Scalable Graph-Based Nearest Neighbor Indexing. arXiv:2602.21247 [cs.DB] https://arxiv.org/abs/2602.21247

[20] 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. 2022. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. arXiv:2205.03763 [cs.LG] https://arxiv.org/abs/2205.03763

[21] Ryan Synk, Monte Hoover, John Kirchenbauer, Neel Jain, Alex Stein, Manli Shu, Josue Melendez Sanchez, Ramani Duraiswami, and Tom Goldstein. 2025. Exploiting Sparsity for Long Context Inference: Million Token Contexts on Commodity GPUs. arXiv:2502.06766 [cs.CL] https://arxiv.org/abs/2502.06766

[22] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2024. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. https://doi.org/10.48550/ARXIV.2409.16576

[23] Karthik Venkatasubba, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2025. BANG: Billion-Scale Approximate Nearest Neighbour Search Using a Single GPU. *IEEE Transactions on Big Data* 11, 6 (Dec. 2025), 3142–3157. https://doi.org/10.1109/tbdata.2025.3581085