BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU

Karthik V
IIT Hyderabad, India
cs21resch14001@iith.ac.in

Saim Khan
IIT Hyderabad, India
saimkhan@alumni.iith.ac.in

Somesh Singh*
LabEx MILYON and LIP (UMR5668), France
somesh.singh@ens-lyon.fr

Harsha Vardhan Simhadri
Microsoft Research, India
harshasi@microsoft.com

Jyothi Vedurada
IIT Hyderabad, India
jyothiv@cse.iith.ac.in

ABSTRACT

Approximate Nearest Neighbour Search (ANNS) is a subroutine in algorithms routinely employed in information retrieval, pattern recognition, data mining, image processing, and beyond. Recent works have established that graph-based ANNS algorithms are practically more efficient than the other methods proposed in the literature, on large datasets. The growing volume and dimensionality of data necessitates designing *scalable* techniques for ANNS. To this end, the prior art has explored parallelizing graph-based ANNS on GPU leveraging its high computational power and energy efficiency. The current state-of-the-art GPU-based ANNS algorithms either (i) require both the index-graph and the data to reside entirely in the GPU memory, or (ii) they partition the data into small independent *shards*, each of which can fit in GPU memory, and perform the search on these shards on the GPU. While the first approach fails to handle large datasets due to the limited memory available on the GPU, the latter delivers poor performance on large datasets due to high data traffic over the low-bandwidth PCIe bus.

In this paper, we introduce BANG, a first-of-its-kind GPU-based ANNS method which works efficiently on billion-scale datasets that cannot entirely fit in the GPU memory. BANG stands out by harnessing compressed data on the GPU to perform distance computations while maintaining the graph on the CPU, enabling efficient ANNS on large graphs within the limited GPU memory. BANG incorporates highly optimized GPU kernels and proceeds in stages that run concurrently on the GPU and CPU, taking advantage of their architectural specificities. Furthermore, it enables overlapping communication with computation while reducing the data transfer between the CPU and GPU over the PCIe bus. We evaluate BANG using a single NVIDIA Ampere A100 GPU on ten popular ANN benchmark datasets. BANG outperforms the state-of-the-art in the majority of the cases. Notably, on the billion-size datasets, we are significantly faster than our competitors, achieving throughputs 40×-200× more than the competing methods for a high recall of 0.9. We will be making all our codes publicly available.

1 INTRODUCTION

The k-Nearest-Neighbor-Search problem is to find the k nearest data points to a given query point in a multidimensional dataset. As the dimensionality increases, *exact* search methods become increasingly less efficient. In order to evaluate the exact k-nearest-neighbors of a query point in a d-dimensional dataset having n datapoints, as a consequence of the *curse of dimensionality*, all the n datapoints must be examined; this takes $O(nd)$ time [25]. Therefore, it is common practice to use Approximate Nearest Neighbor (ANN) search when looking for nearest neighbors in order to mitigate the curse of dimensionality, sacrificing a small accuracy for speed. ANN search is employed as a sub-routine in various fields such as computer vision, document retrieval, and recommendation systems. These applications require searching enormous datasets containing words, images, and documents embedded in a multidimensional space and their search queries are often batch-processed, demanding high throughput. The massively parallel processing capabilities of GPUs can significantly boost the throughput of the ANN search algorithm compared to conventional CPUs.

Graph-based ANNS algorithms [18, 26, 33] have been shown to be generally more efficient in practice at handling large datasets. However, GPU implementations of these algorithms [22, 35, 47] require storing graph data structures in GPU memory, which limits their ability to handle large datasets. Even recent GPUs like the NVIDIA Ampere A100 having 80GB device memory, cannot accommodate the entire input data (graph and data points). Prior solutions, such as sharding, effectively implemented by GGNN [22], result in high memory transfer costs. Additionally, hashing and compression techniques, as showcased by SONG [47] and FAISS [28], can manage large data on a single GPU by reducing data dimensionality or compressing vectors. However, these approaches may have limitations when it comes to high recalls for massive datasets. Furthermore, while multi-GPU configurations, demonstrated by GGNN [22] utilizing eight GPUs, and other approaches [14] that demand multi-CPU setups can provide solutions, they have substantial hardware costs associated. Thus, this paper explores an important question: *Can we increase the throughput of ANN search queries without compromising their recall by using a single GPU?*

In this paper, we introduce BANG, a novel GPU-based ANN search method designed to efficiently handle large datasets, with billions of points, that cannot entirely fit in GPU memory. BANG stands out by employing compressed data for accelerated distance computations

*Contributed to the work while at Microsoft Research India on the GPU while maintaining the graph and the actual data points on the CPU, enabling efficient ANNS on large graphs. CPU transfers neighbour information to the GPU at every iteration of the search. BANG is executed in stages that can run in parallel on CPU and GPU, optimizing resource utilization and mitigating CPU-GPU data transfer bottlenecks. BANG uses *Product Quantization* (PQ) [29] for compressing data and operates on the *Vamana* graph, a technique previously demonstrated success in the DiskANN [26] framework, the state-of-the-art work [14] for billion-scale CPU-based ANNS. BANG incorporates various optimizations, including CPU-GPU load balancing, prefetching, pipelining, bloom filters for search traversal, and highly-optimised GPU kernels for tasks such as distance calculations, sorting, updating worklist etc. Note that our focus is ANN search on the underlying graph-index and hence we do not build a graph but utilize the Vamana graph from DiskANN.

Our evaluations show that BANG significantly outperforms the state-of-the-art in most cases on ten popular ANN benchmark datasets using a single NVIDIA A100 GPU. In particular, on the billion-size datasets, we achieve throughputs 40×-200× more than the competing methods for a high recall of 0.9. On the million-scale datasets, BANG outperforms the state-of-the-art on a majority of the cases. Overall, BANG delivers an arithmetic mean 2× higher throughput than the state-of-the-art for compatible recalls. This paper makes the following main contributions:

- We introduce BANG, a novel GPU-based ANN method that can efficiently search billion-point datasets using only GPU.
- We present highly optimized GPU kernels for calculating distances and updating worklist in ANN search.
- We apply optimizations such as prefetching and pipelining to effectively utilize CPU and GPU, and reduce memory traffic between CPU and GPU over the PCIe interconnect to improve the overall throughput without sacrificing recall.
- We demonstrate that our approach outperforms the state-of-the-art GPU-based ANNS methods.

2 BACKGROUND

2.1 GPU Architecture and Programming Model

Graphic Processor Units (GPUs) are accelerators that offer massive multi-threading and high memory bandwidth [24]. CPU (*host*) and GPU (*device*) are connected via an interconnect such as PCI-Express that supports CPU⇄GPU communications. For our implementation, we use NVIDIA GPUs with the CUDA [4] programming model. The main memory space of GPU is referred to as *global memory*. A GPU comprises several CUDA cores which are organized into *streaming multiprocessors* (SMs). All cores of a SM share on-chip *shared memory*, which is a software-managed L1 data cache. GPU procedures, referred to as *kernels*, run on the SMs in parallel by launching a *grid* of threads. Threads are grouped into *thread blocks*, with each block assigned to an SM. Threads within a thread block communicate and synchronize via shared memory. A thread block comprises *warps*, each containing 32 threads, executing in a single-instruction-multiple-data (SIMD) fashion. Kernel invocation and memory transfers are performed using task queues called *streams*, which can be used for task parallelism.

2.2 A summary of DiskANN and Vamana Graph

To overcome GPU memory limitations and CPU-GPU data transfer overheads for GPU-based ANN search, our work adapts the approach introduced in the DiskANN [26] framework. Further, BANG uses the *Vamana* graph constructed by DiskANN to perform ANN search. Hence, we present some of its key aspects here. DiskANN is a CPU-based graph-based ANNS technique that efficiently constructs the Vamana search graph on a workstation with only 64 GB RAM by sharding the large dataset into smaller overlapping clusters. The technique iteratively processes individual clusters and uses *RobustPrune* and *GreedySearch* to construct directed sub-graphs on RAM, which are then stored on SSD. The final graph is formed by uniting the edges in the sub-graphs. While prior graphs (constructed using the SNG property) reduce distances to the query point along the search path, Vamana emphasizes reducing disk access, achieving low search latency (< 5 ms) with the RobustPrune property. To perform the ANN search, DiskANN retains the Vamana graph on the disk and performs a greedy search on the CPU with *PQ-compressed vectors*. BANG also leverages PQ-compressed vectors to process ANN queries in parallel on GPU. We discuss PQ (Product Quantization) [29] in the following subsection.

2.3 Vector Compression

Product Quantization was first applied to ANNS by Jégou *et al.* [29]. It partitions the d-dimensional dataset into m subspaces and performs independent k-means clustering in each subspace. Each vector is assigned cluster IDs $(0-k)$ on each subspace, forming a compressed m-dimensional vector. These IDs link to unique centroid vectors in the clusters. Distances between centroid vectors and a given query vector within each subspace are computed in a preprocessing step. During ANNS search, distances between dataset points (in compressed form) and query vectors are calculated quickly by summing the precomputed distances of the cluster IDs across all subspaces. This approach accelerates distance calculations by utilizing pre-computed distances of cluster centroids and reducing operations to m dimensions instead of d.

3 ANN SEARCH ON GPU

3.1 Challenges in Handling Billion-scale Data

We propose a parallel implementation of ANN search that leverages PQ vectorization [29] on *Vamana* graph [26] for single-GPU execution, incorporates prefetching and pipelining to efficiently manage CPU and GPU idle time, introduces a novel parallel merge operation for sorting and worklist updates, and integrates bloom filters to handle the visited set of the search traversal for numerous queries on large graphs. There are two main challenges in parallelizing the ANN search on GPUs: (1) managing large graphs within the constraints of limited GPU memory, (2) extracting sufficient parallelism to fully utilize the hardware resources.

Limited GPU Memory.

A primary challenge when dealing with large datasets on GPUs is their massive memory footprint. Table 1 shows the sizes of various billion-scale ANN search datasets and their respective graph sizes. Notably, even with the recent A100 GPU's maximum global memory capacity of 80GB, it is evident that the entire input data

Table 1: Sizes of Data and Graphs for Various Datasets

Dataset	**Data Size**	**Vamana Graph** [26]	**KNN Graph** [22]
SIFT1B	128GB	260GB	80GB
DEEP1B	384GB	260GB	95GB
SPACEV1B	140GB	260GB	112GB

(base, graph, and query) cannot be accommodated within the GPU memory. To address this challenge, recent work GGNN [22] effectively implemented sharding as a solution. However, this approach requires frequent swaps of both processed and unprocessed shards on GPU, resulting in high memory transfer costs. The inherent challenge of processing data in CPU-GPU hybrid platforms requires the transfer of data to GPU memory for computation, followed by the return of the processed data to the main memory for CPU-based operations. Furthermore, sharding the entire data is not viable because even with the PCIe 4.0 interconnect operating at its peak theoretical transfer bandwidth of 32 GB/s, it will take an estimated 20 seconds to transfer the largest dataset (384GB+260GB) from CPU to GPU. This will result in a markedly low throughput (lower bound) of around 500 QPS (Queries per Second) even if 10,000 queries are run concurrently on the GPU.

Although employing a multi-GPU setup can facilitate the effective distribution of shards across all GPUs to accommodate the entire input data (as demonstrated with eight GPUs in GGNN [22]), this approach incurs significant hardware cost. As a result, our focus is on developing an efficient and cost-effective parallel implementation of ANN search on a single GPU.

Alternatively, hashing and compression techniques can handle large data on a single GPU. Hashing effectively reduces data dimensionality to make it fit within GPU memory, as demonstrated by SONG [47], which compressed a 784-byte vector to 64 bytes in the MNIST8M dataset. However, hashing is better suited for smaller datasets and may not achieve high recall on datasets with billions of points. Data compression, as demonstrated by FAISS [28], can accelerate query processing in GPU-based ANN search, with potential trade-offs in recall contingent on the dataset and query parameters.

To remedy these limitations, we present a GPU-based solution wherein we conduct ANN searches with the graph residing on the host, while performing distance computations using compressed data (utilizing Product Quantization [29]) on the GPU. This approach mitigates the CPU-GPU communication bottleneck by transferring only compressed data to the GPU, and it eliminates the need for transferring the entire graph to the GPU by fetching the neighbour information for a node from CPU to GPU on demand.

The approach of performing ANN search with PQ [29] compressed vectors on CPU while maintaining the *Vamana* graph on disk has been introduced in the DiskANN [26] framework. To the best of our knowledge, our work is the first to adapt this approach for GPU-based ANN search to overcome GPU memory limitations and CPU-GPU communication overheads.

Optimal Hardware Usage. The DiskANN approach cannot be applied directly to a CPU-GPU heterogeneous system because GPU-based ANN search faces two scalability challenges that are specific to CPU–GPU systems and do not arise in a CPU-only The first challenge is balancing the workload between the CPU and GPU (which does not apply to disk and CPU in DiskANN). Second, the GPU-based ANN search has a substantially higher memory footprint since it can solve thousands of queries concurrently on GPU and requires information needed for all these queries to be transferred from the CPU at each iteration/hop during the graph traversal.

Given that the graph resides on the CPU, while the distance calculations between graph nodes and query nodes using compressed vectors occur on the GPU, an efficient work distribution strategy must be devised to ensure the simultaneous and continuous engagement of both the CPU and GPU without idleness. This entails distributing tasks to the CPU and GPU based on their respective strengths in terms of memory access latency and computational capabilities to optimize overall performance. Furthermore, it is important to consider the volume of data that needs to be transferred between the CPU and GPU when balancing the work distribution between them to ensure that performance is not constrained by bandwidth limitations caused by prolonged data transfers, as in the GGNN [22] framework.

For instance, for a set of parallel queries (Q) on GPU, when managing the data structures for tracking visited nodes (V) and their subsequent processing, there is the issue of whether structures should be on the CPU or GPU. Designing the workflow strategically requires consideration of several key factors, including the computational complexity associated with the memory size of $Q \times V$, the patterns of memory accesses within V (access times vary with CPU and GPU memory bandwidths), and the time it takes to transfer data $Q \times V$ given the limited bandwidth of the PCIe interconnect between the CPU and GPU.

We propose a parallel ANN search implementation BANG that extracts maximum parallelism by efficiently utilizing the hardware (CPU, GPU and PCIe bus) with a workflow to efficiently load balance the ANN search work across CPU and GPU, performs optimizations such as prefetching and pipelining to manage CPU and GPU idle time, integrates bloom filters to handle the visited set of the search traversal for numerous queries on large graphs, along with having high-optimized kernels for operations such as distance computation, sort and merge on GPU.

3.2 BANG **Overview**

BANG performs approximate nearest neighbour search on a CPU–GPU hybrid system. Figure 1 shows the complete workflow of BANG. Given a set of available queries, Q, BANG processes the queries in parallel by effectively using CPU (host), GPU (device) and PCIe (data transfer link between CPU and GPU). We intend to harness the distinct capabilities (or advantages) of CPU and GPU efficiently and collaboratively use multicore CPU and a single GPU for achieving high throughput even on billion-scale data.

BANG uses the Vamana index graph from DiskANN (see Section 2.2) as the search structure on which to perform query lookups. Since the graph index is so large that it cannot be fitted in GPU memory, it is stored in CPU RAM. As shown in Figure 1, the CPU is used to retrieve the neighbours of a given node by looking up the adjacency list in the graph data structure. Similarly, as the complete actual dataset cannot be fitted on the GPU, the search process on GPU is performed using compressed vectors derived from the actual

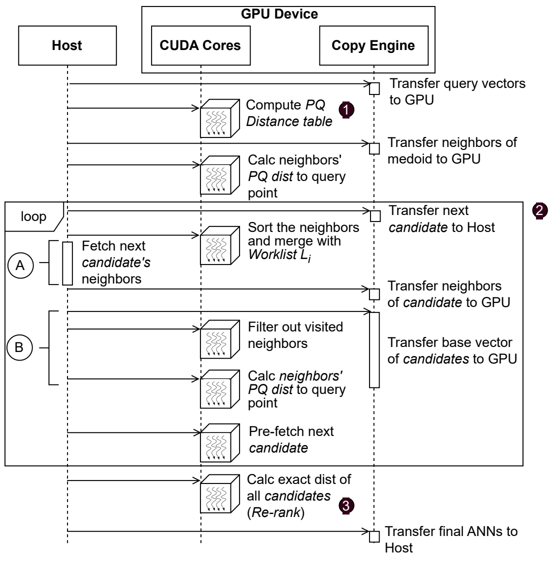

GPU Device
Host
CUDA Cores
Copy Engine
Transfer query vectors to GPU
Compute PQ Distance table
❶
Transfer neighbors of medoid to GPU
Calc neighbors' PQ dist to query point
loop
Transfer next candidate to Host
❷
Sort the neighbors and merge with Worklist L_i
A
Fetch next candidate's neighbors
Transfer neighbors of candidate to GPU
B
Filter out visited neighbors
Transfer base vector of candidates to GPU
Calc neighbors' PQ dist to query point
Pre-fetch next candidate
Calc exact dist of all candidates (Re-rank)
❸
Transfer final ANNs to Host

Figure 1: Schema of BANG.

data vectors. The BANG technique broadly comprises three stages: ❶ Distance Table Construction, ❷ ANN Search, and ❸ Re-ranking.
Distance Table Construction. Consider the toy example shown in Figure 2, having 12 points in the base dataset. The base dataset has two dimensions, and the data type of the coordinate values is an unsigned integer (1 byte, i.e., uint8). The compression technique involves representing a d-dimensional dataset vector with a compressed m-byte code, achieved by partitioning the d dimensions into m vector subspaces, performing subspace clustering, and applying the quantization technique [29] for compression as outlined in Section 2.3. We specify m =1 for the product quantization. So, at the end of the compression step, each node (2 bytes long) will be represented by 1-byte.

Before starting the search, we pre-compute the square of the Euclidean distances of all the cluster centroids to the query points on the GPU in parallel. Note that each subspace has c clusters created using k-means clustering. This data structure is referred to as the PQ Distance table having dimensions $c \times m$ and it remains on GPU until the search terminates. In the example, there are 12 clusters and one subspace, so the dimensionality of the PQ Distance Table is 12×1. The distance between the query point and each cluster centroid is shown tabulated in Figure 2b After generating the PQ distance table, the search begins by retrieving neighbors of a node and determining the PQ clusters they belong to; the approximate distance to the query point is computed by summing pre-computed distances from the PQ Distance Table (compressed vectors).
ANN Search. Since all queries in a batch are independent, they can be handled in an embarrassingly parallel fashion. Each step shown under the CUDA Cores section in Figure 1 runs on an individual CUDA *thread-block* (Section 2.1) independently for each query. Medoid will be used as the starting point for each query search. In the example shown in Figure 2, the medoid happens to be node 6. Once the query is provided, the search starts from the medoid and iteratively progresses toward the query point. The worklist L is empty, so we initialize it with 6. The entries in the worklist are always sorted in ascending order of their distance from the query point. We compute the distance of 6 from the query node using the PQ Distance table (i.e. compressed vectors). With just one element in the worklist, node 6 is immediately selected as the *candidate node* for the next visit without the need for sorting. We transfer node ID 6 from the GPU to the CPU and request the neighbours of node 6. The CPU-side implementation accesses the adjacency list to retrieve neighbours of node 6, providing the GPU with node IDs 8 and 2, concluding iteration 0.

In Iteration 1, GPU filters out previously visited nodes among the neighbours, computes their distances to the query point using the PQ Distance Table, sorts them, merges the sorted list with the worklist, and selects the first candidate node, which is node 8, to be visited next. GPU maintains bloom filter data structure to keep track of visited nodes in order to avoid fetching neighbours of the same points and computing distances. In the figure, we can see that the set of neighbours progressively approaches the query point, and when all nodes in the worklist are visited, the iterations end.
Re-ranking. The candidate node identified and sent to CPU during each iteration is stored in a data structure for use in a final step called *re-ranking*. A re-ranking process can compensate for inaccuracies in the search process caused by distance computation using compressed vectors [26]. Once the search process concludes, their exact distances from the query point are calculated on GPU using the exact base vectors, and the nodes are subsequently sorted based on these exact distances to extract the final k nearest neighbours from the list. For the example, the sorted final list of candidate nodes, ordered by their distances to the query node, includes 10, 8, 11, 9, 6, 7, 5, 2, resulting in the selection of the top 2 nearest neighbours, namely, 10 and 8. Note that, for the re-ranking step, only full vectors of selected nodes are sent to GPU, which are fewer and can thus collectively fit into GPU memory for all queries.

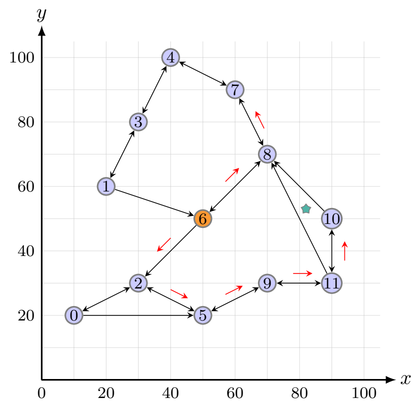
4 BANG: BILLION-SCALE ANN SEARCH ON A SINGLE GPU

4.1 Search Algorithm

In graph-based ANN algorithms, greedy search, illustrated in Algorithm 1, is used to answer search queries. The algorithm begins at a fixed point s and explores the graph G in a best-first order by evaluating the distance between each point in the worklist $\mathcal{L}$ and the query point q, advancing towards q at each iteration and finally reporting k nearest neighbours. BANG performs greedy search on *Vamana* graph from DiskANN (see Section 2.2).

Algorithm 2 describes BANG's scheme for batched query lookups on the Vamana graph. Since all queries in a batch are independent, they can be handled in an embarrassingly parallel fashion (Line 1). Each query runs in a separate CUDA *thread block* (Section 2.1) independently. This will result in as many thread blocks as queries utilizing the massive parallelism GPUs offer to maximize throughput as measured by Queries Per Second (QPS).

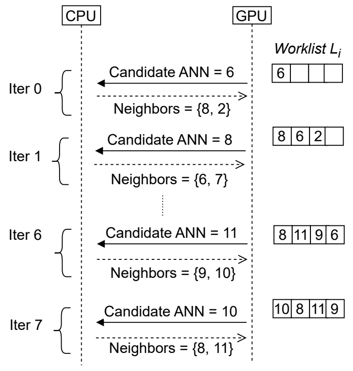
By systematically dividing the ANNS search activity into distinct steps, each with its own characteristics of work distribution and span, we maximize GPU parallelism, in contrast to other approaches [22, 47] that treat the entire search activity as a single



(a)

Node ID	Cluster ID	PQ Distance
0	6	6273
1	9	3893
2	7	3233
3	8	3433
4	1	3973
5	11	2113
6	0	1033
7	2	1853
8	4	433
9	5	673
10	3	73
11	10	593

(b)



(c)

Iteration #	Candidate	Exact Distance
0	6	1033
1	8	433
2	7	1853
3	2	3233
4	5	2133
5	9	673
6	11	593
7	10	73

(d)

Figure 2: Example illustrating BANG workflow for ANN search on Vamana graph. The star indicates the query point $q(82, 53)$.

Algorithm 1: Greedy Search

```
Input: A graph index $G$, a query point $q$, required no. of
       neighbors $k$, medoid $s$ and a parameter $t$ $(\geq k)$
Output: A set $\{c_1, c_2, \ldots, c_k\}$ of $k$ approx. nearest neighbors
1 Initialise visited set $\mathcal{V} \leftarrow \emptyset$ and worklist $\mathcal{L} \leftarrow \{s\}$
2 while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ or $|\mathcal{L}| \neq t$ do
3     $u^* \leftarrow \arg\min_{u \in \mathcal{L} \setminus \mathcal{V}} \| x_u - x_q \|$ // get nearest candidate
4     $\mathcal{V} \leftarrow \mathcal{V} \cup \{u^*\}$
5     $\mathcal{L} \leftarrow \mathcal{L} \cup G.adj(u^*)$
6     Update $\mathcal{L}$ to keep up to $t$ nearest candidates
7 $\mathcal{K} \leftarrow k$ candidates nearest to $q$ in $\mathcal{L}$
8 return $\mathcal{K}$
```

Algorithm 2: BANG: Batched query search on CPU-GPU

```
Input: $G$, a graph index; $k$, required no. of nbrs; $s$, medoid
       $PQDistTable$, dist. b/w compressed vectors & queries
       $Q_\rho$, a batch of $\rho$ queries, and a parameter $t$ $(\geq k)$
Output: $\mathcal{K}_\rho := \bigcup_{i=1}^{\rho} \{\mathcal{K}_i\}$, where $\mathcal{K}_i$ is the set of $k$-nearest
        neighbors for $q_i \in Q_\rho$
1  foreach $q_i \in Q_\rho$ in parallel do
2      $u_i^* \leftarrow s$          // Initialize current node $u_i^*$ of $i^{th}$ query
3      converged $\leftarrow$ $false$
4      while not converged do
5          $N_i \leftarrow$ FetchNeighbors$(u_i^*, G)$      // compute on CPU
6          /* CPU sends nbrs $N_i$ to GPU */
7          foreach $n \in N_i$ in parallel do    // filter nbrs on GPU
8              if not GetBloomFilter$(i, n)$ then
9                  $N_i' \leftarrow N_i' \cup \{n\}$
10                 SetBloomFilter$(i, n)$
11         foreach $n_k \in N_i'$ in parallel do            // on GPU
12             $\mathcal{D}_i[k] \leftarrow$ ParallelComputeDistance$(n_k, q_i)$
13         $(\mathcal{D}_i', N_i') \leftarrow$ ParallelMergeSort$(\mathcal{D}_i, N_i')$  //on GPU
14         ParallelMerge$(\mathcal{L}_i, \mathcal{D}_i', N_i')$        // on GPU: Merge
            worklist $\mathcal{L}_i$ with list $N_i'$ to keep $t$ nearest candidates
15         $u_i^* \leftarrow$ next unvisited nearest node in $\mathcal{L}_i$
16         /* GPU sends node $u_i^*$ to CPU */
17         converged $\leftarrow (\bigwedge_{n \in \mathcal{L}_i}$ Processed$(n)$ $) \wedge |\mathcal{L}_i| = t$
18     $\mathcal{K}_i \leftarrow k$ candidates nearest to $q_i$ in $\mathcal{L}_i$
```

block. For example, we design separate GPU kernels for: (1) constructing the PQ distance table, (2) filtering visited neighbours, (3) computing distances of neighbours to queries, (4) sorting the neighbours by computed distances, (5) merging the closest neighbours into the worklist, and (6) re-ranking the candidate nodes. For each individual kernel, we heuristically optimize the thread block size (per query) to ensure maximum GPU occupancy. A single sub-optimally configured block can significantly reduce performance.

In the following subsections, we detail how we parallelize and optimize the mentioned kernels and each step in Algorithm 2 in order to achieve optimal GPU performance, maximize occupancy, and fully utilize hardware resources.

4.2 Construction of *PQDistTable* in Parallel

Here, we describe the data structure *PQDistTable*, which Algorithm 2 uses as its input for performing ANN search. As discussed previously, since the entire dataset cannot fit on the GPU, the search on the GPU uses compressed vectors derived from the actual data vectors. That is, the algorithm uses approximate distances calculated using compressed vectors at Line 12. This is accomplished with *PQDistTable*, which contains the distance of each of the centroid vectors (obtained after product quantization) to the query vectors (see Section 3.2 and Section 2.3).

For each of the query points in a batch, we compute its squared Euclidean distance to each of the centroids for every subspace produced by the compression. We maintain these distances in a lookup data structure which we call PQ Distance Table (*PQDistTable*). We describe later (in Section 4.5) how we use these precomputed distances to efficiently compute the *asymmetric distance* [29] between a (uncompressed) query point and the compressed data point.

We maintain *PQDistTable* as a contiguous linear array of size $(\rho \cdot m \cdot 256)$, where ρ is the size of the query batch and m is the number of subspaces with each subspace having 256 centroids. The number of centroids is as used in prior works [26, 28] that use *Product Quantization*. We empirically determine $m = 74$ for our setup, guided by the available GPU global memory (Section 6 shows the ablation study with varying m values). As with all other kernels, each thread-block handles one query, resulting in ρ concurrent thread blocks. Furthermore, for a given query $q \in \mathbb{R}^d$, the distance of each of the subvectors of the query $q_s \in \mathbb{R}^{d/m}$ from each of the 256 centroids of a subspace can be computed independently of others in parallel by the threads in a thread block. Note that, for query q, distances of subvectors across m subspaces are computed sequentially within a thread, ensuring an adequate workload per thread and constrained by thread-block size.

Work-Span Analysis. The work of the algorithm is $O((m \cdot subspace\_size) \cdot 256 \cdot \rho)$. Owing to the parallelization scheme, the span of the algorithm is $O(m \cdot subspace\_size) = O(d)$.

4.3 Handling Data Transfer Overheads

As outlined in Algorithm 2, the CPU transfers the neighbours (Line 6) to the search routine executing on the GPU for every node transmitted by the GPU (Line 16). The data transfer between the device (GPU) and the host (CPU) is time-consuming in contrast to the GPU's tremendous processing power (the PCIe 4.0 bus connects the GPU and has a maximum transfer bandwidth of only 32 GB/s). Therefore, BANG transmits only the bare minimum information required in order to minimize data transfer overhead. Specifically, from device to host, the transfer includes a list of final Approximate Nearest Neighbors (ANNs) after the search in Algorithm 2 converges and a candidate node for each query in every iteration (Line 16), while from host to device, it comprises a list of neighbouring vertices (Line 6) and base vectors of the candidates (not shown in the algorithm).

Further, BANG hides data transfer latency with kernel computations using advanced CUDA features. To perform data transfers and kernel operations concurrently, CUDA provides asynchronous `memcpy` APIs and the idea of streams [40]; BANG makes use of these. During the search, neighbours of a given vertex are retrieved using CPU threads for all ρ queries (see Line 5 in Algorithm 2). We leverage the efficient structure of the graph data, allowing sequential memory access of the vertex's base vector and its neighbourhood list as they are placed next to each other on the CPU memory. Hence, immediately after the transfer of the neighbour list to the GPU in each search iteration, we strategically make an asynchronous transfer of the base vectors for future use that are only required during the final re-ranking step on GPU (after the search in Algorithm 2 converges). As a result, the kernel execution engine and the copy engine of the GPU are kept occupied to achieve higher throughput. Thus, in our implementation, we aim to make all the data transfers asynchronous with `cudaMemcpyAsync()` and placing `cudaStreamSynchronize()` at appropriate places to honor the data dependencies.

4.4 Handling Visited Vertices: Bloom Filter

Given the nature of the graph $G_{\mathcal{P}}$ to establish long-range edges [26], Algorithm 2 may encounter the same vertex multiple times during the search, potentially impacting throughput and/or recall. The low throughput issue arises from the possibility of duplicate work during the search, particularly when fetching neighbours (Line 5) and computing their distances to query (Line 12), both of which are computationally intensive. Avoiding this redundancy presents an opportunity for throughput improvement. The low recall issue arises when not filtering out visited neighbours, leading to adding these neighbours to the worklist $\mathcal{L}_i$ multiple times, potentially causing premature termination of the search. Further, maintaining visited vertices is crucial for BANG, as opposed to re-computations [45], as evidenced in our experiments where the recall diminishes to as low as 1/10th when visited vertices are not filtered out. Considering these factors, we keep track of the vertices visited during the traversal to ensure that a node is visited only once.

The most common methods for keeping track of the visited vertices are: i) setting a bit per vertex and ii) using a set or a priority queue or a hash table data structure. While the first approach is fast, the memory footprint depends on the size of the graph and the number of queries in a batch. For a billion-size graph and a query batch size of 10000, it would require $(10^9 \times 10^4)$ bits or $\frac{10^{13}}{10^9 \times 8}$ GB = 125 GB memory. This alone would be too large to fit on the GPU. The second approach has a small memory overhead since it only stores the vertices that are visited. However, a set, priority queue, or hash table, being dynamic data structures, poses challenges for a GPU due to difficulties in maintaining them dynamically and with high irregular dependency. These inherent characteristics result in underutilising GPU computing bandwidth as evidenced by previous works [22, 47] and thus are not conducive to GPU parallelization.

To optimize GPU parallelism within the constraints of limited memory, we employ the well-established Bloom filter [6] data structure for its efficiency in approximate set membership tests and minimal memory footprint, fully leveraging GPU parallelism. We use one bloom filter per query as shown at Line 10 in Algorithm 2. Our bloom filter uses an array of z bools, where z is determined using an estimate of the number of visited vertices for a query, a small tolerable false-positive rate (which is the probability of having false positives), and the number of hash functions used. We use two FNV-1a hash functions [20], which are lightweight non-cryptographic hash functions often used to implement Bloom filters.

4.5 Parallel Neighbor Distance Computation

For each query in a batch, we compute its *asymmetric distance* from the current list of neighbours in each iteration (Line 12 in Algorithm 2). This computation for each query is independent, so we assign one query per thread block. Further, for a query, its distance to each of the neighbours can also be computed independently of the other neighbours. Thanks to the $PQDistTable$ we built previously (Section 4.2), we can compute the distance of a query point q_i to a neighbour n_i by summing the *partial* distances of the centroids across all subspaces in $PQDistTable$, with centroid information obtained from n_i's compressed vector (Sections 2.3 and 3.2).

When computing distances in GPU-based ANN search, summations are commonly conducted using reduction APIs from Nvidia's CUB [38] library, such as those in [22, 47], typically at a thread-block-level or warp-level. However, in our approach, a thread block of size t_b is subdivided into g groups, each with $g_{size} := \frac{t_b}{g}$ threads. These groups collaboratively compute the distance of the query from a neighbour, with g_{size} threads summing up the partial distances of m centroids (with each segment of size $\frac{m}{g_{size}}$). Each thread sequentially computes the sum of values in a segment using thread-local registers, avoiding synchronization. To efficiently add segment-wise sums using g_{size} threads, two approaches are explored: (i) atomics, employing *atomicAdd()* for the final result; (ii) Sub-warp-level reductions, utilizing CUB [38] WarpReduce. Empirical tuning with $m = 74$, $t_b = 512$, and $g_{size} = 8$ yields optimal performance in our experimental setup, with the second approach, utilizing sub-warp level reductions, marginally outperforming the first. Overall, our segmented approach outperforms standard alternatives like CUB WarpReduce (1.2× slower) and BlockReduce (4× slower) for warp-level and thread-block-level reductions.

This kernel constitutes a sizeable chunk of the total run time on billion-size datasets, accounting for approximately 38% on average. The kernel's efficiency is limited by GPU global memory access latency arising from uncoalesced accesses to compressed vectors of various neighbors, a consequence of the irregular structure of the graph. To mitigate this irregularity, we explored graph reordering using the well known Reverse Cuthill Mckee (RCM) algorithm [13, 19]. However, this approach did not yield significant improvements in locality, and so we avoided graph reordering.

Work-Span Analysis. The work of the algorithm is $O(NumNbrs \cdot m \cdot \rho)$. Owing to the parallelization scheme, the span of the algorithm is $O(log\ m)$.

4.6 Prefetching Candidate Nodes

As described in Algorithm 2, candidate node u_i^* of each query q_i will be communicated to the CPU (Line 16) only after the GPU has updated the worklist $\mathcal{L}_i$. While the CPU collates the neighbours (Line 5) and communicates them to the GPU (Line 6), the GPU remains idle, waiting to receive this information. To avoid this delay in starting the current iteration, GPU must be supplied with the neighbour IDs as soon as needed. To address this, instead of sending the candidate nodes at the end of the previous iteration to CPU, we perform an optimization to *eagerly* predict the next candidate nodes immediately after the neighbour distance calculation (i.e., before even sorting the neighbour list by distance and updating worklist $\mathcal{L}_i$).

Our optimization calculates the candidate node u_i^* eagerly by selecting the nearest neighbour in the new neighbour list N_i' and the first unvisited node in worklist $\mathcal{L}_i$ (sorted by distance to query), then choosing the best between the two. The eager selection occurs just before Line 13 in Algorithm 2. Upon dispatching the eagerly selected candidates to the CPU, the GPU continues executing the remaining tasks of the iteration (sorting the new neighbour list and updating the worklist). Concurrently, the CPU collates neighbour IDs. That is, Line 5 executes on CPU concurrently with tasks at Line 13 and Line 14 on GPU. By the completion of the GPU's remaining tasks, the CPU has already communicated the neighbour IDs. This strategy eliminates GPU idle time, resulting in a throughput improvement (experimentally found to be approx. 10%).

4.7 Parallel Merge Sort

To add new vertices closer to q_i than those in $\mathcal{L}_i$ in each iteration of Algorithm 2, we sort the neighbours N_i' based on their respective asymmetric distances to q_i and attempt to merge the eligible ones with $\mathcal{L}_i$. We sort the neighbours using parallel bottom-up merge sort. Conventionally, one worker thread merges two lists (in the conquer phase of the algorithm). Thus, as the algorithm progresses, parallelism decreases since there are fewer lists to merge, and consequently, the amount of work per thread increases as the size of the lists to merge grows. Since most GPU threads remain idle until the sort is complete and the work per thread is high, this scheme is not suitable for GPU processing. We mitigate these issues by merging lists in parallel, using a parallel merge routine described in the following subsection.

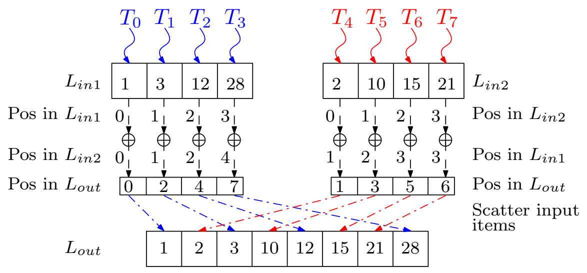


Figure 3: Parallel List Merge. L_{in1} and L_{in2} are merged to create the merged list L_{out}.

For our use case, we need to sort small lists, typically having up to 64 neighbours. We assign one thread block to a query. Further, we assign one thread per neighbour in the list to be sorted by setting the thread block size to the maximum number of neighbours of a node in the graph. We start with sorted lists that have one element each and double their size at each step through the parallel merge routine (Section 4.8). As the lists are small, we can keep them in GPU shared memory throughout the sorting algorithm.

Work-Span Analysis. The work of merging two lists of size t using the parallel merge routine is $O(t \cdot log(t))$ (Section 4.8). The total work $T(n)$ for an array of n numbers follows the recurrence: $T(n) = 2{\cdot}T(n/2){+}n{\cdot}log(n)$, which gives $T(n) = O(n{\cdot}log^2(n))$. Thus, the work of the algorithm is $O(NumNbrs \cdot log^2(NumNbrs) \cdot \rho)$. Owing to the parallelization scheme, the span of the algorithm is $O(log^2(NumNbrs))$.

4.8 Parallel Merge

The merge routine we detail in this section is employed in the merge sort, as explained in the preceding section. Additionally, it is utilized to merge the sorted neighbour list N_i' with the worklist $\mathcal{L}_i$ at Line 14 in Algorithm 2. We leverage an existing parallel list merging algorithm on GPU [21] to suit our specific use case. Figure 3 shows the parallel list merge algorithm on two lists of size four each. Given two sorted lists L_{in1} and L_{in2} with m and n items, where an item e in L_{in1} is at position $p_1 < m$, and if inserted in L_{in2} would be at position $p_2 < n$, the position pos_{Lout} of the element in the merged list L_{out} is determined by $pos_{Lout} \leftarrow p_1 + p_2$. We assign one thread to each element. From the thread ID assigned to the item, we can compute the position of the item in its original list. We use binary search to determine where the item should appear in the other list. For example, in Figure 3, the item 28 in L_{in1} is at index 3, determined by the thread ID. In L_{in2}, its index is 4, found through binary search. Therefore, the index of 28 in the merged list is 7 (3 + 4). Each thread, assigned to an item, concurrently conducts a binary search in the other list. We keep the lists in the GPU's shared memory to minimize search latency. Thus, computing the position of each element in the merged list takes $O(\log(m) + \log(n))$ time. Lastly, elements are scattered to their new positions in the merged list, and in the example, 28 is placed at index 7 in the merged list L_{out}. This step is also performed in parallel, as each thread writes its item to its unique position in the merged list.

Work-Span Analysis. The work of the algorithm to merge two lists each of size l is $O(l \cdot log(l))$. Owing to the parallelization scheme, the span of the algorithm is $O(log(l))$.

4.9 Re-ranking

We use approximate distances (computed using PQ distances) throughout the search in Algorithm 2. We employ a final re-ranking step [43] after the search converges to enhance the overall recall (Section 3.2).

We implement the re-ranking step as a separate kernel with optimized thread block sizes. Re-ranking involves computing the exact L2 distance for each query vector with its respective candidate nodes, followed by sorting these candidates by their distance to the query vector and reporting the *top-k* candidates as nearest neighbours. For a query, its distance from each candidate node is computed in parallel. To sort the candidates according to their exact distance, we use the parallel merge procedure described in Section 4.8. Thanks to the asynchronous data transfers (Section 4.3), the candidate nodes are already available on GPU when this kernel begins. From our experiments, we observed that re-ranking improved the recall by 10-15% for the datasets under consideration.

Work-Span Analysis. The work of the re-ranking step is $O((d \cdot |C| + |C| \cdot log^2(|C|)) \cdot \rho)$, where $|C|$ is the maximum number of candidate nodes for a query and d is vector dimension. Owing to the parallelization scheme, the span of re-ranking is $O(d + log^2(|C|))$.

5 IMPLEMENTATION

We refer to the implementation described so far in Sections 3 and 4 as `BANG Base` (or simply `BANG`). In addition, we provide versions of the implementation specifically optimized for small and medium-sized datasets (having up to hundred million data points) that can fit in the GPU memory. We discuss these versions below.

5.1 In-memory Version

The CPU-GPU link stays constantly busy when we store the graph on the CPU, as in `BANG Base`. The resultant CPU-GPU interdependency also necessitates careful task management on the GPU in order to avoid idle time. Following the same approach is unnecessary for smaller graphs that can fit on a GPU. Persisting the entire graph on the GPU, rather than the CPU, minimizes CPU-GPU communication overhead and inter-dependency, resulting in enhanced throughput. This optimization eliminates the memory-intensive process of the CPU fetching neighbours from main memory for GPU-supplied candidates. It also mitigates the PCIe bus-intensive data transfer between the CPU and GPU during the search process, yielding a notable throughput improvement of up to 50%. We refer to this version of the implementation as `BANG In-memory`. The schematic block diagram of this version is shown in Figure 4. There

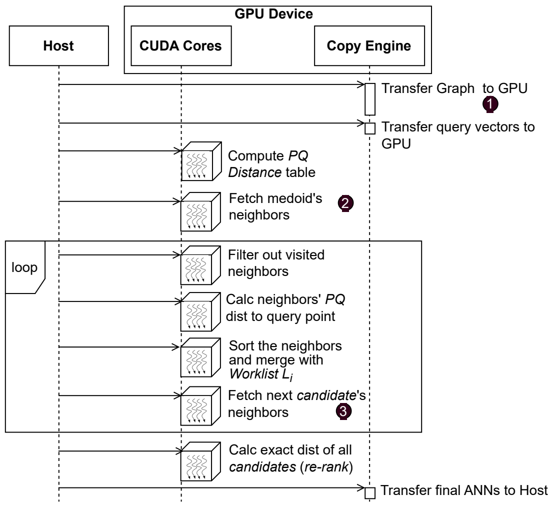

GPU Device
Host
CUDA Cores
Copy Engine
Transfer Graph to GPU
1
Transfer query vectors to GPU
Compute PQ Distance table
Fetch medoid's neighbors
2
loop
Filter out visited neighbors
Calc neighbors' PQ dist to query point
Sort the neighbors and merge with Worklist L_i
Fetch next candidate's neighbors
3
Calc exact dist of all candidates (re-rank)
Transfer final ANNs to Host

Figure 4: Schema of `BANG In-memory` Variant

are two main differences from Figure 1: (1) in step ❶, we transfer the entire graph from CPU to GPU before the search begins (2) steps ❷ and ❸ fetch neighbours on the GPU locally through coalesced access to global memory, instead of relying on the CPU.

5.2 Exact-distance Version

In this version, to further optimize on top of the `BANG In-memory` version, instead of using PQ distances within the search loop and compensating for the inaccuracies by a subsequent re-ranking step, we directly use exact L2 distance calculations. We refer to this version as `BANG Exact-distance`. The main differences from Figure 4 are: (1) absence of PQ distances computation, (2) exact distance computation (instead of PQ distance summation), (3) absence of re-ranking step.

6 EXPERIMENTAL SETUP

6.1 Machine Configuration

Hardware. We conduct our experiments on a machine with an Intel Xeon Gold 6326 CPU with 32 cores (two sockets, 16 cores each), clock-speed 2.90GHz, 48 MiB L3 cache and 660 GiB DDR4 memory. It houses a 1.41 GHz NVIDIA Ampere A100 GPU with 80 GB global memory with a peak bandwidth of 2039 GB/s, and 6912 processing units distributed over 108 streaming multiprocessors (SMs). Each multiprocessor has 164 KB shared memory. The GPU is connected to the host via a PCIe Gen 4.0 bus having a peak transfer bandwidth of 32 GB/s. We use a single GPU for all the experiments.

Software. The machine runs Ubuntu 22.04.01 (64-bit). We use *g++* version 11.3 with the `-O3`, `-std=c++11` and `-fopenmp` flags to compile the C++ codes. We use OpenMP for parallelizing the C++ codes. To compile the GPU codes, we use *nvcc* version 11.8 with the `-O3` flag. We will be making all our codes publicly available.

Table 2: Real-world datasets in our test-suite. Vector Size: Original data size; CV Size: Compressed data size.

Dataset	Type	No. of Data points	d	Vector Size(GB)	Graph Size(GB)	CV Size(GB)
DEEP1B [44]	float	1000,000,000	96	384.00	260.00	74.00
SIFT1B [30]	uint8	1000,000,000	128	128.00	260.00	74.00
SPACEV1B [11]	int8	1000,000,000	100	100.00	260.00	74.00
DEEP100M	float	100,000,000	96	38.00	26.00	9.60
SIFT100M	uint8	100,000,000	128	12.80	26.00	6.40
MNIST8M [7]	uint8	8,090,000	784	6.34	2.10	1.58
GloVe200 [5, 37]	float	1,183,514	200	0.94	0.30	0.10
GIST1M [29, 34]	float	1,000,000	960	3.84	0.26	0.16
SIFT1M [29]	float	1,000,000	128	0.50	0.26	0.06
NYTimes [5, 16]	float	289,761	256	0.29	0.75	0.03

6.2 Datasets

We present experiments on *ten* popular real-life datasets that originate from a diverse set of applications. We summarize the characteristics of the datasets in our test-suite in Table 2. DEEP1B is a collection of one billion image embeddings compressed to 96 dimensions. SIFT1B and SIFT1M represent the 128-dimensional SIFT (Scale-Invariant Feature Transform) descriptors of one billion and one million images, respectively. The SPACEV1B dataset [11] encodes web documents and web queries sourced from Bing using the Microsoft SpaceV Superior Model. This dataset contains more than one billion points, and to be consistent with other billion-size datasets in our test suite, we pick the first billion points in the dataset. DEEP100M and SIFT100M are generated by taking the first hundred million points from the DEEP1B and SIFT1B datasets, respectively. MNIST8M is a 784-dimensional dataset containing the image embeddings of deformed and translated images of handwritten digits. GIST1M contains one million 960-dimensional GIST (Global Image Structure Tensor) descriptors of images. GloVe200 is a collection of 1,183,514 word embeddings with 200 dimensions. NYTimes is a collection of word embeddings with 256 dimensions and 289,761 points. GloVe200 and NYTimes have a skewed distribution of vectors, whereas other datasets have a uniform distribution of vectors. Every dataset except GIST1M and SPACEV1B has 10,000 queries. We run all 10,000 queries in a single batch, as in previous approaches [22, 47]. Since the original query set for GIST1M has 1,000 query points, we repeat it nine times to make it 10,000 points to keep the comparison uniform. Similarly, for SPACEV1B, we pick the first 10,000 points from the query set (of size 29,316).

6.3 Parameter Configuration: BANG

Graph Construction. We run our BANG search algorithm on *Vamana* graph, built using DiskANN [26]. We specify R =64 (maximum vertex degree), L=200 (size of worklist used during build) and σ=1.2 (pruning parameter), following the recommendations of the original source. The sixth column in Table 2 shows the size of the resulting graph for each dataset.

Compression. We specify the parameters for compression (PQ in Section 2.3) such that the generated compressed vectors fit in the GPU global memory. For our setup, we empirically determined the largest value of m to be 74. We use m = 74 across all datasets for all our experiments, unless otherwise stated. Note that as a consequence, the compression ratio could be different for different datasets depending upon their dimensionality. For example, for the 128-dimension SIFT1B dataset, the compression ratio is 74/128 = 0.57, while for the 384-dimension DEEP1B dataset, the compression ratio is 74/384 = 0.19. The last column in Table 2 shows the total size of PQ compressed vectors for each dataset.

Search. The search starts from *medoid*, a node pre-determined during graph construction. We use the standard $k-recall@k$ [26] recall metric, where k is the required number of nearest neighbours. Our query batch size is 10,000. We measure the throughput using the standard definition of *Queries Per Second* or *QPS*. The search parameter t in Algorithm 2, representing the size of worklist $\mathcal{L}$, is varied to generate throughput versus recall plots. The recall increases with an increase in the t value. The lower bound for t is k, and the upper bound is heuristically set to 152 . The bloom filter (Section 4.4) is created to hold 399887 entries. We tune the bloom filter size to lower values in order to generate lower recall values (below the recall values by using $t = k$). To increase statistical significance, we report the throughput averaged over five independent runs.

6.4 Baselines

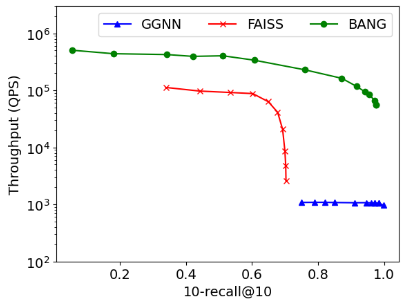
We quantitatively compare BANG with the state-of-the-art open-source techniques that support billion-scale ANN search on GPU(s).

GGNN [22]. It performs ANN query search on the GPU using a hierarchical k-NN graph (constructed on GPU). It divides the large dataset into small shards that can be processed on multiple GPUs in parallel, achieving high throughput and recall. GGNN provides configurable graph construction parameters *viz.*:# out-going edges (k), # disjoint points selected for subgraphs (s), # levels in the hierarchical graph (l), slack factor (τ_b) and # refinement iterations (r). We use the recommended build parameters from the original source code repository [23] for most datasets; for others, we follow authors' private email suggestions for optimal performance on our setup. SPACEV1B: {k:20, s:32, l:4, τ_b:0.6, r:2, 16 shards}. DEEP100M and SIFT100M: {k:24, s:32, l:4, τ_b:0.5, r:2, a single shard}. MNIST8M: {k:96, s:64, the rest are the same as for DEEP100M and SIFT100M}.

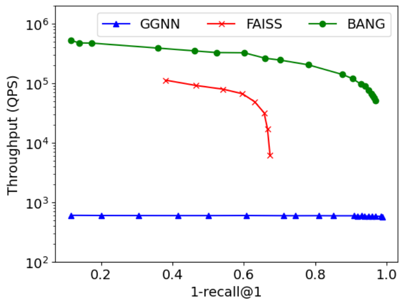
SONG [47]. It is a graph-based ANN search algorithm which requires the graph and dataset entirely in the global memory of the GPU. It handles larger datasets through hashing. SONG uses a *bounded priority queue* whose size can be varied to control the throughput-recall trade-off. In our experiments, we vary the size of the priority queue between 30 and 600 for best results on our setup.

FAISS [28]. It is a billion-scale ANN algorithm available with CPU and GPU implementations. We use the latter. We use the three index construction parameters specified in the FAISS Wiki [27]: *OPQ32_128*, *IVF262144*, and *PQ32*. The first field indicates the *vector transformation* technique (pre-processing) that aids in the subsequent steps of index construction and compression stages. The second field indicates the type of IVF index used. The third field indicates the encoding scheme (quantization) used.

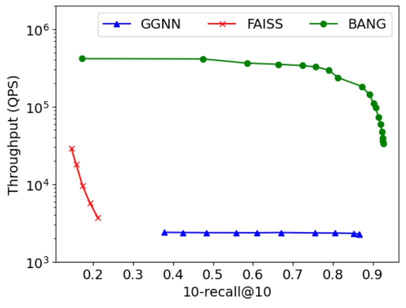
We compare BANG with these baselines on various dataset scales. We compare against FAISS only on 1B datasets, as SONG and GGNN are known to outperform FAISS on million-scale datasets. We do not report the results of SONG for MNIST8M, 100M and 1B datasets since the graph generation was not complete even after 48 hours, showing no visible progress, even after multiple attempts.



(a) SIFT1B



(b) DEEP1B



(c) SPACEV1B

Figure 5: Comparison of BANG with GGNN and FAISS on SIFT1B, DEEP1B and MSSPACEV1B datasets

7 RESULTS

In this section, we provide a comprehensive comparison of the performance and solution quality of BANG with various state-of-the-art methods across different datasets detailed in Section 6.

7.1 Performance on Billion-scale Datasets

Figure 5 shows the comparison of BANG against GGNN and FAISS on billion-size datasets. BANG outperforms GGNN and FAISS in throughput while maintaining the same recall at 10-recall@10 across all three datasets. Overall, BANG achieves outstanding performance with 40×-200× throughput for a high recall of 0.9.

As a result of aggressive compression through quantization, FAISS has a lower recall; lower compression results in an index too large for GPU memory, so it cannot be used. The overhead of data transfer from CPU to GPU impedes the throughput of GGNN on a single GPU (this was addressed by the original GGNN implementation by utilizing eight GPUs). For instance, SIFT1B requires an additional 143 GB of data to be transferred from the CPU to the GPU during the search since the dataset and the graph are each 128 GB and 95 GB, respectively, and the GPU has 80 GB of global memory. Transferring this additional data over the PCIe 4.0 interconnect @ 32GB/s will take close to 5 seconds. By using compressed vectors for search on GPU, BANG avoids the need for transferring this additional data. Furthermore, the reason for the high throughput of BANG can be attributed to the massive parallelism achieved on the GPU (Section 4). The re-ranking step, coupled with the integration of bloom filters to filter visited nodes (allowing more optimal nodes to be included in the worklist), significantly contributes to achieving a high recall for BANG.

7.2 Performance on 100 Million-scale Datasets

Figure 6a and 6b show the performance comparison of BANG with GGNN on 100M datasets. As the data structures (i.e., graph and data points) for 100M datasets fit within GPU memory, we report the performance of all three variants of BANG (Section 5). As can be seen from the plots, on both DEEP100M and SIFT100M dataset, BANG In-memory and BANG Exact-distance variants achieve 2× and 3× higher throughput compared to the base variant of the BANG since there is no CPU-GPU communication overhead. As shown, BANG Exact-distance variant also outperforms GGNN. We might expect the same performance from BANG Exact-distance variant as GGNN since both keep data structures on the GPU and both compute exact distances, but, the underlying graph is different. The k-NN graph structure of GGNN requires more hops or iterations for the search to terminate. In contrast, in BANG Exact-distance, the Vamana graph contains long-range edges, reducing the hops, enabling faster convergence with fewer computations.

7.3 Performance on Million-scale Datasets

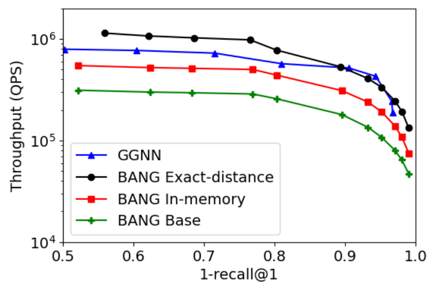
Figure 7 and Figure 8a to 8d show the performance of BANG against GGNN and SONG on million-scale datasets. We report the performance of all three variants of BANG since the points and graph easily fit on GPU. BANG Base does better than SONG but is outperformed by GGNN for all five datasets except MNIST8M (Figure 7). BANG Base incurs latency with CPU-GPU data transfers, giving GGNN an advantage. MNIST8M's high dimensionality increases search latency for GGNN; furthermore, Vamana's long-range edges in BANG reduce the number of hops, providing clear benefits.

The BANG In-memory and BANG Exact-distance variants outperform GGNN for NYTimes, GIST1M and MNIST8M datasets. Achieving high recall close to one is challenging for all variants in the case of NYTimes and GloVe200 datasets due to their skewed and clustered distribution, as also observed in [47]. SIFT1M and GloVe200 are the only datasets where BANG Exact-distance slightly lags behind GGNN.

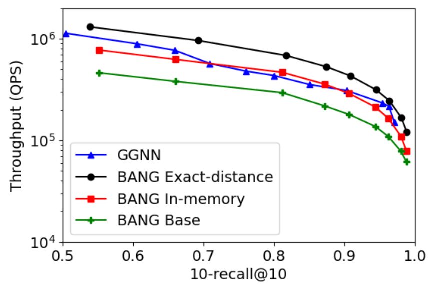
In general, BANG Exact-distance provides more throughput than BANG In-memory since the former requires relatively fewer iterations to reach the query point than the latter, which uses compressed vectors that invariably introduce inaccuracies in distance computations (also needs a re-ranking step). However, for GIST1M, the increased overhead of computing exact distances over a large number of dimensions (960) results in a slight under-performance for BANG Exact-distance compared to using compressed vectors.

7.4 Effect of Varying Compression Ratio

As discussed in Section 2.3, PQ compression techniques introduce recall losses. To assess the impact of the compression ratio on BANG's overall recall, we conducted a study by varying the compression factor m in the PQ compression scheme and performed ANNS on the SIFT1B dataset (other search parameters remained the same as in Section 6). The results are presented in Figure 9. Although recall is expected to decrease with lower compression ratios, we



(a) DEEP100M



(b) SIFT100M

Figure 6: BANG vs. GGNN on 100-million size datasets

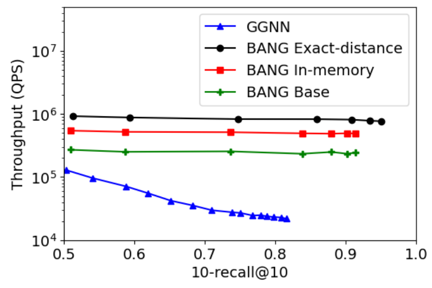
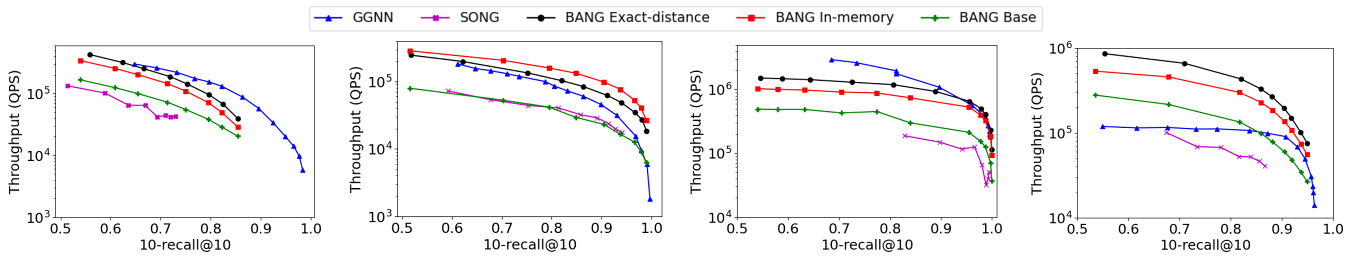


Figure 7: BANG vs. GGNN on MNIST8M



(a) GloVe200 (b) GIST1M (c) SIFT1M (d) NYTimes

Figure 8: Comparison of BANG with GGNN and SONG on the GloVe200, GIST1M, SIFT1M and NYTimes datasets

observed no noticeable change in throughput/recall values even when m is decreased from 74 to 32 (compression ratios of 0.57 and 0.25, respectively). We observed that recall values remain stable until a compression ratio of 0.25, after which they start to decrease. This behaviour allows BANG to operate efficiently on GPUs with limited device memory without compromising recall or throughput. E.g., with SIFT1B, we can achieve the same recall and performance on an A100 GPU with 40 GB as on an A100 with 80 GB, highlighting the potential of BANG in memory-constrained GPU environments.

Throughput should increase when the compression ratio is reduced because it reduces computations in the distance calculation kernel. However, due to more inaccurate distances, BANG search requires additional iterations, taking more hops and detours to reach the final nearest neighbours. This negative effect results in no net gain in throughput with a lower compression ratio.

7.5 Efficiency of Parallel Query Processing

BANG ANN search involves an iterative loop (Algorithm 2), and the total iteration count represents the critical path and total work of a given query. To understand the workload per thread block (recall we parallelize by assigning one thread block per query), we studied the number of iterations per query on the SIFT1B dataset. We choose different values of $\mathcal{L}$ from 20 to 180, number of chunks, m is set as 64, and the rest of the parameters remain the same as in Section 6. The results are captured in Figure 10. Note that the lower bound on total iterations is the size of the worklist $\mathcal{L}$, as every entry in the worklist must be visited. As can be seen, for each run with a specific $\mathcal{L}$ value, 95% of queries were completed in 1.1$\mathcal{L}$ iterations, indicating a promising efficiency in terms of the number of iterations needed. Hence, in our parallelization scheme, we do not propose specific thread scheduling mechanisms, such as work-stealing wherein a thread block which has finished processing its assigned query q_i can partake in processing unprocessed candidate nodes in the worklist $\mathcal{L}_j$ of another query q_j.

8 RELATED WORK

Numerous algorithms [42] are available for ANNS, offering a range of trade-offs w.r.to index construction time, recall, and throughput. **ANN Search Data Structure.** Traditionally, ANNS utilized KD-trees [8, 12]. Tree-based methods, including KD-trees and cover trees, follow a branch-and-bound approach for navigation. Locality-Sensitive Hash (LSH)-based techniques [2, 3] depend on hash functions with higher collision probabilities for nearby points. Both these approaches face scalability challenges with growing dimensions and size.

NSW [32] is an early graph-based ANNS method that employs the Delaunay Graph (DG) [17] for identifying node neighbours, ensuring near-full connectivity but introducing a high-degree search space with increasing dataset dimensionality. The graph includes short-range edges for higher accuracy and long-range edges for improved search efficiency, but this results in an imbalance in the graph's degree due to the creation of "traffic hubs". HNSW [33] addresses this by introducing a hierarchical structure to spread neighbours across levels and imposing an upper bound on node degree. However, HNSW remains non-scalable to high dimensions and large datasets. NSG [18] improves graph-based methods' efficiency

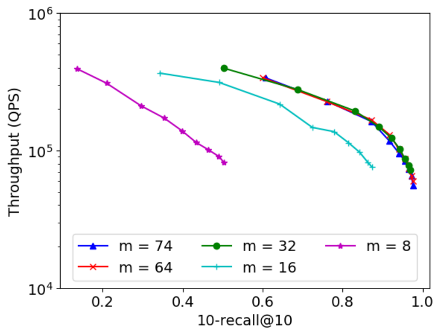


Figure 9: Variation in throughput and recall with m

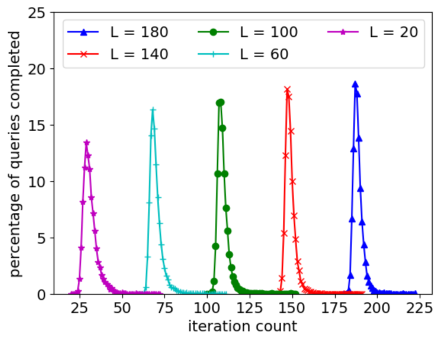


Figure 10: Query completion with iteration count for various $\mathcal{L}$

and scalability, introducing the Monotonic Relative Neighborhood Graph (MRNG), reducing index size and search path length while scaling to billion-scale datasets. DiskANN [26] indexes billion-point datasets in hundreds of dimensions through the Vamana graph on commodity hardware, leveraging NSG and NSW strengths. It introduces a tunable parameter α for a graph degree-diameter trade-off during construction and proposes compression schemes to reduce memory consumption and computation costs during the search (see Section 2.2). The survey [15] extensively studies and compares popular graph-based ANNS algorithms, addressing scaling challenges and proposing parallelization methods. BANG leverages the Vamana graph from DiskANN [26] for our search, with an emphasis on ANN search on GPU, unlike the above-mentioned approaches.

ANN Search on GPU. Due to the complexity and throughput requirement of ANNS, offloading these calculations to massively parallel accelerators like GPU has gained considerable attention. FAISS [28] utilizes Product Quantization for dimensionality reduction, employing an inverted index as the underlying data structure. While FAISS achieves high throughput at a billion scale, it fails to deliver high recalls. [43] extends the Product Quantization concept to a two-level product and vector quantization tree (PQT), exhibiting superior GPU performance over CPU for high-dimensional, large-scale ANNS. However, PQT has drawbacks, such as longer encoding lengths than PQ-code and high encoding errors. Addressing these issues, [9] proposes a novel two-level tree structure called Vector and Product Quantization Tree (V-PQT) that indexes database vectors using two different quantizers, outperforming PQT in recalls. [39] proposes an efficient distributed memory version of FAISS for hybrid CPU-GPU systems scaling up to 256 nodes.

SONG [47], a graph-based ANN implementation on GPU, offers a notable $50 - 180\times$ speedup over state-of-the-art CPU counterparts and significant improvements over FAISS through efficient GPU-centric strategies for ANNS. GANNS [45], built on this foundation, explores enhanced parallelism and GPU occupancy through GPU-friendly data structures and an NSW-based proximity graph construction scheme. However, SONG and GANNS face limitations in scaling to billion-node graphs due to the necessity of the entire graph index on the GPU. In a related effort, [41] proposes a two-stage graph diversification approach for graph construction and GPU-friendly search procedures catering to various batch query scales. Graph-based GPU Nearest Neighbor (GGNN) [22] search is a recent implementation that outperforms SONG. It utilizes eight GPUs for billion-scale ANNS, achieving high recall (close to 1) by dividing the large dataset into smaller shards that fit into GPU memory for parallel graph construction, with the final results merged on the CPU. GGNN achieves very high throughputs (> 100,000) at high recalls (> 0.95). CAGRA [35], a recent graph-based ANNS implementation, leverages modern GPU capabilities (e.g., warp splitting) for substantial performance gains over existing GPU ANNS methods at million-scale. It requires the graph index to fit into GPU memory, and thus, multiple GPUs are needed for billion-scale datasets. Unlike these approaches, BANG is a high-recall, high-throughput ANNS that runs on a single GPU.

ANN Search on Other Accelerators. Researchers have explored custom hardware, such as FPGAs, for ANNS acceleration, focusing on quantization-based [1, 46] and graph-based [36] approaches. However, these methods suffer from frequent data movement between CPU memory and device memory, leading to high energy consumption and lower throughputs. Addressing these limitations, vStore [31] proposes an in-storage computing technique for graph-based ANNS within SSDs, avoiding data movement overhead and achieving low search latency with high accuracy. On the other hand, TPU-KNN [10] focuses on realizing ANNS on TPUs. It uses an accurate accelerator performance model considering both memory and instruction bottlenecks.

9 CONCLUSION

We presented BANG, a novel GPU-based ANNS method that efficiently handles large billion-size datasets exceeding GPU memory capacity, particularly on a single GPU. It brings together computation on compressed data and optimized GPU parallelization to achieve high throughput on large data. BANG enables GPU-CPU computation pipelining and overlapping communication with computation. Thus, it is able to fully take advantage of the GPU and multicore CPU, and reduce the data transfer over the PCIe interconnect between the host and the GPU. As a result, on billion-scale datasets it significantly outperforms the current state-of-the-art GPU-based methods, achieving $40\times$–$200\times$ higher throughput for high recalls; on smaller datasets, it is almost always faster than or comparable to the state-of-the-art methods.

REFERENCES

[1] Ameer M.S. Abdelhadi, Christos-Savvas Bouganis, and George A. Constantinides. 2019. Accelerated Approximate Nearest Neighbors Search Through Hierarchical Product Quantization. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 90–98. https://doi.org/10.1109/ICFPT47387.2019.00019

[2] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for near neighbor problem in high dimension. *Commun. ACM* 51, 1 (2008), 117–122.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. *Advances in neural information processing systems* 28 (2015).

[4] CUDA (Compute Unified Device Architecture). 2023. CUDA Programing Model. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Retrieved: 2023-12-28.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2018. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. arXiv:1807.05614 [cs.IR]

[6] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commununication of the ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[7] Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston. 2007. *Training Invariant SVMs Using Selective Sampling*. 301–320.

[8] Lawrence Cayton. 2008. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*. 112–119.

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

[10] Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. 2022. TPU-KNN: K Nearest Neighbor Search at Peak FLOP/s. arXiv:2206.14286 [cs.PF]

[11] SpaceV Contributors. 2023. SPACEV1B: A billion-Scale vector dataset for text descriptors. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B. Accessed: Dec 30, 2023.

[12] Ryan R Curtin, Parikshit Ram, and Alexander G Gray. 2013. Fast exact max-kernel search. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 1–9.

[13] Elizabeth H. Cuthill and John M. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69*. https://api.semanticscholar.org/CorpusID:18143635

[14] Magdalen Dobson, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. arXiv:2305.04359 [cs.IR]

[15] Magdalen Dobson, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. arXiv:2305.04359 [cs.IR]

[16] D. Dua and C. Graff. [n.d.]. UCI machine learning repository. http://archive.ics.uci.edu/ml

[17] Steven Fortune. 1995. Voronoi diagrams and Delaunay triangulations. *Computing in Euclidean geometry. World Scientific* (1995), 225–265. https://doi.org/10.1142/9789812831699_0007

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[19] J. Alan George. 1971. *Computer implementation of the finite element method*. Ph.D. Dissertation. Stanford University, USA. https://searchworks.stanford.edu/view/2198775

[20] Glenn Fowler, Landon Curt Noll, and Phong Vo. [n.d.]. *Fowler-noll-vo hash function*. http://isthe.com/chongo/tech/comp/fnv/.

[21] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/2304576.2304621

[22] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* (2022).

[23] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P.A. Lensch. 2023. GGNN: Graph-based GPU Nearest Neighbor Search. https://github.com/cgtuebingen/ggnn. Accessed: Dec 30, 2023.

[24] CUDA C Best Practices Guide. 2023. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[25] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[26] 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/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[27] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and Alexandr Guzhva. 2023. FAISS Wiki. https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index. Accessed: Oct 2, 2023.

[28] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[29] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[30] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 861–864. https://doi.org/10.1109/ICASSP.2011.5946540

[31] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. 2022. VStore: In-Storage Graph Based Vector Search Accelerator. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 997–1002. https://doi.org/10.1145/3489517.3530560

[32] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[33] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[34] Aude Oliva and Antonio Torralba. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision* 42 (2001), 145–175.

[35] Hiroyuki Ootomo, Akira Naruse, Corey J. Nolet, Ray Wang, Tamas B. Fehér, and Y. Wang. 2023. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. *ArXiv* abs/2308.15136 (2023). https://api.semanticscholar.org/CorpusID:261276680

[36] Hongwu Peng, Shiyang Chen, Zhepeng Wang, Junhuan Yang, Scott A. Weitze, Tong Geng, Ang Li, Jinbo Bi, Minghu Song, Weiwen Jiang, Hang Liu, and Caiwen Ding. 2021. Optimizing FPGA-Based Accelerator Design for Large-Scale Molecular Similarity Search (Special Session Paper). In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (Munich, Germany). IEEE Press, 1–7. https://doi.org/10.1109/ICCAD51958.2021.9643528

[37] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[38] Reduction. 2023. CUB library. https://nvlabs.github.io/cub/. Retrieved: 2023-12-28.

[39] Rafael Souza, André Fernandes, Thiago S.F.X. Teixeira, George Teodoro, and Renato Ferreira. 2021. Online multimedia retrieval on CPU–GPU platforms with adaptive work partition. *J. Parallel and Distrib. Comput.* 148 (2021), 31–45. https://doi.org/10.1016/j.jpdc.2020.10.001

[40] CUDA Streams. 2023. Streams Simplify Concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. Retrieved: 2023-12-28.

[41] Hui Wang, Yong Wang, and Wan-Lei Zhao. 2022. Graph-based Approximate NN Search: A Revisit. arXiv:2204.00824 [cs.IR]

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

[43] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. 2016. Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[44] Artem Babenko Yandex and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2055–2063. https://doi.org/10.1109/CVPR.2016.226

[45] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 552–564. https://doi.org/10.1109/ICDE53745.2022.00046

[46] Jialiang Zhang, Jing Li, and Soroosh Khoram. 2018. Efficient Large-Scale Approximate Nearest Neighbor Search on OpenCL FPGA. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4924–4932. https://doi.org/10.1109/CVPR.2018.00517

[47] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate nearest neighbor search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1033–1044.