BANG: Billion-Scale Approximate Nearest Neighbour Search Using a Single GPU

Karthik Venkatasubba, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada

***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. The growing volume and dimensionality of data necessitates designing *scalable* techniques for ANNS. To this end, the prior art has explored parallelising graph-based ANNS on GPU, leveraging its massive parallelism. The current state-of-the-art GPU-based ANNS algorithms either (i) require both the dataset and the generated graph index to reside entirely in the GPU memory, or (ii) partition the dataset 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 interconnect. We introduce BANG, a first-of-its-kind technique for graph-based ANNS on GPU for billion-scale datasets, that cannot entirely fit in the GPU memory. BANG stands out by harnessing a compressed form of the dataset on a single GPU to perform distance computations while efficiently accessing the graph index kept on the host memory, enabling efficient ANNS on large graphs within the limited GPU memory. BANG incorporates highly optimised GPU kernels and proceeds in phases that run concurrently on the GPU and CPU, taking advantage of their architectural specificities. Furthermore, it enables overlapping communication with computation that results in efficient data transfer between the CPU and GPU. Using a single NVIDIA Ampere A100 GPU, BANG achieves throughputs 50×–400× higher than competing methods for a recall of 0.9 on three popular billion-scale datasets.**

***Index Terms*—Approximate nearest neighbour search (ANNS), graph and tree search, information retrieval, approximate search, vector similarity search, GPU, big data.**

I. Introduction

The k-Nearest-Neighbour-Search (kNN) 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 inefficient. In order to evaluate the exact k-nearest-neighbours of a query point in a d-dimensional dataset having n points, as a consequence of the *curse of dimensionality*, all n points must be examined; this takes $O(nd)$ time [28]. Therefore, it has become commonplace to use Approximate Nearest Neighbour Search (ANNS) for finding the nearest neighbours of a query, to mitigate the curse of dimensionality by sacrificing a small accuracy for speed [68]. ANNS accuracy is typically measured as *recall*, which quantifies the overlap between the retrieved nearest neighbours and the ground truth for a query. ANNS is widely applied in information retrieval [7], [10], [39], recommendation systems [49], [57], search engines [65], and computer vision [70] to search through large datasets of words, documents, images, and multimedia. Advances in deep learning [6] have made ANNS essential for similarity search [68]. ANNS is a key subcomponent of vector databases [47] and is transforming the way vector embeddings are efficiently indexed and queried [27]. Search queries on such massive multidimensional datasets are often processed in batches to meet the high throughput demands, as evidenced by recent efforts [24], [31], [63], [67] leveraging the massive parallelism of GPUs for ANNS. One of the earliest use cases of GPUs in this context was accelerating kNN queries on road networks, such as locating the k nearest cars for ride-sharing [34]. Nowadays, GPUs are increasingly powering vector databases for Retrieval-Augmented Generation (RAG) [45], boosting semantic accuracy and freshness of results in static Large Language Models (LLMs) and supporting high-throughput use cases, such as cloud-based chatbots [5] and unstructured data applications like e-commerce [69].

Billion-scale ANNS is computationally intensive, with pairwise distance calculations well-suited for GPU architectures. Zhang et al. [66] compare CPU- and GPU-based ANNS techniques, and conclude that GPUs offer higher performance and cost-effectiveness for accelerating vector query processing. The Billion-Scale Approximate Nearest Neighbor Search Challenge [53] evaluates ANNS on various hardware platforms (e.g., CPU, GPU, custom hardware) and establishes that GPU-based implementations significantly outperform other approaches. Billion-scale datasets exceed single GPU memory, and hence ANNS on these datasets requires multiple GPUs [24]; this introduces two challenges: (a) a significant escalation in implementation cost (as seen in the Track3 Cost/Power Leaderboards [53]), (b) a mismatch between GPU compute performance and memory resources.

Graph-based ANNS algorithms [21], [29], [36] have been shown to be generally more efficient, in practice, at handling

Received 18 July 2024; revised 8 April 2025; accepted 23 May 2025. Date of publication 19 June 2025; date of current version 13 November 2025. Recommended for acceptance by L. Zhu. *(Corresponding author: Karthik Venkatasubba.)*

Karthik Venkatasubba and Jyothi Vedurada are with the Department of CSE, IIT Hyderabad, Sangareddy 502285, India (e-mail: cs21resch14001@iith.ac.in; jyothiv@cse.iith.ac.in).

Saim Khan was with the Department of CSE, IIT Hyderabad, Sangareddy 502285, India. He is now with Microsoft Bengaluru 560103, India.

Somesh Singh is with Siemens EDA, 38240 Meylan, France.

Harsha Vardhan Simhadri is with Microsoft, Redmond, WA 98052 USA.

Digital Object Identifier 10.1109/TBDATA.2025.3581085

large datasets. However, GPU implementations of these algorithms [24], [46], [67] 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 80 GB device memory, cannot accommodate the entire input data (graph index and data points). Prior solutions, such as sharding, effectively implemented by GGNN [24], incur high memory transfer cost. For example, with the PCIe 4.0 interconnect operating at its peak theoretical transfer rate of 32 GB/s, transferring the DEEP1B dataset (having a data size of 384 GB and a graph index size of 260 GB) from CPU to GPU would take 20 seconds. This would result in a low throughput of less than 500 QPS for 10,000 concurrent queries on the GPU. Alternatively, hashing and compression techniques, as showcased by SONG [67] and FAISS [31], can handle large volumes of data using a single GPU by reducing the data dimensionality or by compressing the vectors. However, these approaches face limitations in achieving high recall on massive datasets. It has been shown that it is feasible to attain high throughputs at high recalls on large datasets using multiple GPUs [24]. However, such approaches have a very high hardware cost which makes them undesirable. Thus, this paper explores an important question: *Can we increase the throughput of ANNS queries without compromising their recall by using a single GPU?*

In this paper, we introduce BANG, a novel GPU-based ANNS method, which is 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 on the GPU while keeping the graph index and the dataset on the CPU, thus enabling efficient ANNS on large datasets. CPU transfers neighbour information to the GPU on demand in every iteration of the search. Furthermore, in contrast to the previous approaches that treat search activity as one monolithic block, BANG divides ANNS activity into distinct phases, in order to maximise resource utilisation and mitigate CPU-GPU data transfer bottlenecks. Specifically, the division into phases leads to three advantages: (1) executing different phases on CPU and GPU in a pipeline (2) optimising each phase's (GPU kernel's) span characteristics separately to maximise parallelism on GPU (3) asynchronously prefetching required data from CPU to GPU for specific phases. Thus, the overall performance of BANG stems from several factors, such as CPU-GPU load balancing, highly optimised GPU kernels (for tasks such as distance calculations, sorting, and updating worklists), prefetching and pipelining.

ANNS methods typically follow the *index* and *search* paradigm whereby data points are first processed to construct an efficient index, and search queries are then processed using this index. In this work, our focus is efficient billion-scale ANNS on GPU utilising an underlying graph index, and therefore we do not construct a graph index but instead utilise an existing one from prior work [29].

Our evaluations show that BANG significantly outperforms the state-of-the-art on four popular real-world billion-size ANNS datasets with varying characteristics across all recall values on a single NVIDIA A100 GPU. Notably, on the SIFT-1B and Deep-1B datasets, we achieve $50.5\times$ higher average throughput over competing methods at a high recall of 0.95.

This paper makes the following main contributions:

- We introduce BANG, a novel GPU-based ANNS method for efficiently searching billion-scale datasets using a single GPU.
- We propose a *phased-execution* strategy that enables pipelined execution of ANNS steps on both CPU and GPU, optimising CPU-GPU load balancing, asynchronous data transfer, and GPU kernel performance for maximising parallelism.
- We present efficient GPU kernels for distance calculations and worklist updates in ANNS and implement prefetching and pipelining to effectively utilise CPU and GPU, reducing traffic over PCIe interconnect and enhancing throughput while retaining recall.
- We conduct an extensive evaluation demonstrating that BANG significantly outperforms state-of-the-art GPU-based ANNS methods in terms of throughput and cost.

II. Background

A. GPU Architecture and Programming Model

Graphic Processor Units (GPUs) are accelerators that offer massive multithreading and high memory bandwidth [26]. CPU (*host*) and GPU (*device*) are connected via an interconnect such as PCI Express that supports CPU $\rightleftarrows$ GPU communications. For our implementation, we use NVIDIA GPU with the CUDA [41] programming model. The main memory space of GPU is referred to as *global memory*. A GPU comprises several CUDA cores, which are organised 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 a SM. Threads within a thread-block communicate and synchronise 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.

B. Graph Index

Several of the best-performing ANNS algorithms are graph-based, achieving both high recall and high throughput, measured as queries per second (QPS). A graph-based ANNS algorithm runs on a *proximity graph*, which is pre-constructed over the dataset points by connecting each point to its nearby points. Our BANG algorithm implements an efficient ANNS method that can run on various proximity graphs, such as kNN [24], HNSW [36], Vamana [29], and others. Manohar et al. [38] conduct a study on various CPU-based ANNS algorithms and show that the *DiskANN* [29] approach, which runs on the Vamana graph index, achieves superior throughput and recall on large datasets. Furthermore, the recent work by Wang et al. [59],

TABLE I
SIZES OF DATA AND GRAPHS FOR VARIOUS DATASETS

Dataset	**Data Size**	**Vamana Graph** [29]	**kNN Graph** [24]
SIFT-1B	128 GB	260 GB	80 GB
Deep-1B	384 GB	260 GB	96 GB
MSSPACEV-1B	100 GB	260 GB	80 GB
Text-to-Image-1B	800 GB	260 GB	80 GB

which proposes enhancements to the Vamana graph index for disk-resident indexes, also highlights the significance of the Vamana graph. Hence, we employ the Vamana graph as the underlying graph index in the implementation of BANG. Vamana graph index is constructed by sharding the large dataset into smaller overlapping clusters. The technique iteratively processes individual clusters and uses *RobustPrune* and *GreedySearch* routines to construct directed subgraphs on the host RAM, which are finally written to an SSD. The final graph is formed by merging the edges in the subgraphs. While prior techniques that construct the graph index using the SNG property [4] aim to reduce distances to the query point along the search path, Vamana further reduces the number of hops to the query point with the RobustPrune property, resulting in a faster search.

III. ANN Search on GPU: Challenges in Handling Billion-Scale Data

We propose BANG, a method for parallel ANNS on a single GPU. There are two main challenges in parallelising ANNS on a single GPU: (1) managing large graphs within the constraints of limited GPU memory, (2) extracting sufficient parallelism for ANNS to fully utilise the hardware resources. In the following subsections, we discuss these challenges in detail.

A. Limited GPU Memory

A primary challenge when dealing with large datasets on GPUs is their massive memory footprint. Table I shows the sizes of various billion-scale ANNS datasets and their respective graph sizes. Notably, even with the recent A100 GPU's maximum global memory capacity of 80 GB, it is evident that the entire input data (base, graph, and query) cannot be accommodated within the GPU memory. To address this challenge, a recent work GGNN [24] effectively implemented sharding as a solution. However, this approach requires frequent swaps of both processed and unprocessed shards between GPU and host RAM, resulting in high memory transfer costs over the PCI interconnect. Further, these swaps may result in an idle time of CPU/GPU, and optimal GPU-CPU coordination is essential during iterative graph traversals of ANNS. 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 from CPU to GPU and will result in a markedly low throughput of less than 500 QPS as explained in Section I.

Although employing a multi-GPU setup can facilitate the effective distribution of shards across all GPUs to store the entire input data (as demonstrated with eight GPUs in GGNN [24]), this approach incurs significant hardware cost. As a result, our focus is on developing an efficient and cost-effective parallel implementation of ANNS using a single GPU.

Alternatively, hashing and quantization 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 [67], which compresses 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 [31], can accelerate query processing in GPU-based ANNS, but it cannot provide high recall.

To mitigate these limitations, we present a novel GPU-based solution wherein we conduct ANNS that does not require the large graph to be present in GPU memory. Instead, the CPU retrieves the required neighbourhood information from the graph index kept on the host RAM and transfers it to the GPU efficiently. Likewise, since the GPU cannot accommodate the entire base dataset, the search process on the GPU employs compressed vectors (instead of the base dataset vectors) for distance calculations. The compressed vectors are generated using the widely utilised Product Quantization [32] technique, employed in several previous works, including FAISS [31] and DiskANN [29]. Thus, BANG alleviates the CPU-GPU data transfer bottleneck by transferring only compressed data to the GPU and eliminating the need to transfer the entire graph. Instead, it fetches neighbour information for a node from the CPU to the GPU on demand.

B. Optimal Hardware Usage

BANG's approach of fetching neighbours from the CPU and calculating distances from them to the query on the GPU using compressed vectors presents two challenges.

The primary challenge is to prevent CPU and GPU idleness by ensuring their concurrent and continuous utilisation despite the dependence on neighbour information between these devices. Furthermore, it is important to consider the volume of data that needs to be transferred between the CPU and the 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 [24] framework. To address these issues, we propose a parallel ANNS implementation BANG that maximises parallelism by efficiently utilising the hardware (CPU, GPU and PCIe bus) with a phased-execution approach to efficiently load balance the ANNS work across CPU and GPU, and to minimise CPU-GPU data transfer bottlenecks. The phased-execution strategy facilitates the execution of different steps of ANNS in a pipeline on both CPU and GPU, which enables: (1) CPU-GPU load balancing to minimise CPU and GPU idle time, (2) using prefetching techniques to transfer data from CPU to GPU asynchronously for certain phases, and (3) optimising the span characteristics of each GPU kernel (corresponding to operations such as distance computation, sorting and updating worklist) separately to maximise parallelism.

The second challenge is to optimise the placement of the data structure according to the computational strengths and memory access capabilities of the CPU and GPU, due to the higher

Algorithm 1: GREEDY SEARCH.

```
Input: A graph index $G$, a query $q$, required no. of
       neighbours $k$, medoid $s$ and a parameter $t$
       $(\geq k)$
Output: A set $\{c_1, c_2, \dots, c_k\}$ of $k$ approx. nearest
        neighbours
1 Initialise visited set $\mathcal{V} \leftarrow \emptyset$ and worklist $\mathcal{L} \leftarrow \{s\}$
2 while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ 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: ANNS Using a Single GPU.

```
Input: $G$, a graph index; $k$, required no. of nbrs; $s$, medoid
       PQDistTable, dist. b/w compressed vectors & queries
       $\mathcal{Q}_\rho$, a batch of $\rho$ queries, and a parameter $t$ $(\geq k)$,
       size of the worklist that controls recall of the search
Output: $\mathcal{K}_\rho := \bigcup_{i=1}^{\rho} \{\mathcal{K}_i\}$, where $\mathcal{K}_i$ is the set of $k$-nearest
        neighbours for $q_i \in \mathcal{Q}_\rho$
1  foreach $q_i \in \mathcal{Q}_\rho$ in parallel do
2      $u_i^*$, $\mathcal{L}_i \leftarrow s$
3      converged $\leftarrow$ $false$
4      while not converged do
5          $N_i \leftarrow$ FetchNeighbors($u_i^*$, $G$) // 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         for $k \leftarrow 1$ to $|N_i'|$ in parallel do // on GPU
12             $\mathcal{D}_i[k] \leftarrow$
                 ParallelComputeDist($N_i'[k], q_i$)
13         $(\mathcal{D}_i', \mathcal{N}_i') \leftarrow$ ParallelMergeSort($\mathcal{D}_i, N_i'$) // on
            GPU
14         $\mathcal{L}_i \leftarrow$ ParallelMerge($\mathcal{L}_i, \mathcal{D}_i', \mathcal{N}_i'$) // on GPU:
            Merge $\mathcal{N}_i'$ with worklist $\mathcal{L}_i$ to keep $t$ NNs at most
15         $u_i^* \leftarrow$ next unvisited nearest node in $\mathcal{L}_i$
16         /* GPU sends node $u_i^*$ to CPU */
17         converged $\leftarrow$ ($\bigwedge_{n \in \mathcal{L}_i}$ Visited($n$) )
18     $\mathcal{K}_i \leftarrow$ $k$ candidates nearest to $q_i$ in $\mathcal{L}_i$
19     /* k-nearest neighbours available on GPU */
```

memory footprint caused by thousands of queries running concurrently on the data in the GPU. For example, when handling parallel queries (Q) on the GPU and managing data structures for tracking visited nodes (V) and their processing, in order to determine whether structures should be placed on the CPU or GPU, several key factors need to be considered, such as 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. BANG handles this by using an efficient implementation of bloom filter [8] on the GPU. We discuss this in detail in Section IV-C.

IV. BANG: BILLION-SCALE ANNS ON A SINGLE GPU

Given a set of available queries, $\mathcal{Q}$, BANG processes the queries in parallel by effectively using CPU (host), GPU (device) and PCIe (data transfer link between CPU and GPU), achieving high throughput on billion-scale data using a single GPU. BANG is a graph-based ANNS technique.

In the case of graph-based ANNS algorithms [24], [29], [36], [67], it is a fundamental approach to use greedy search, depicted in Algorithm 1. A worklist $\mathcal{L}$ is used to hold at most t entries (i.e., nodes in the graph) in the increasing order of the distances of the points from the query point. 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 a proximity graph [24], [29], [36]. Algorithm 2 describes BANG's scheme for batched-query searches on a GPU. Since all queries in a batch are independent, each query search can run in a separate CUDA *thread-block* (Section II-A) independently (see Line 1), resulting in as many thread-blocks as queries. It utilises the GPU's massive parallelism to maximise throughput, as measured by Queries Per Second (QPS). BANG addresses the challenge of the entire graph index not fitting on GPU efficiently by keeping the index on CPU and selectively retrieving from the CPU the neighbours of the visited nodes (i.e., candidate nodes) during query search iteration (Lines 5 and 6). This strategy ensures effective handling

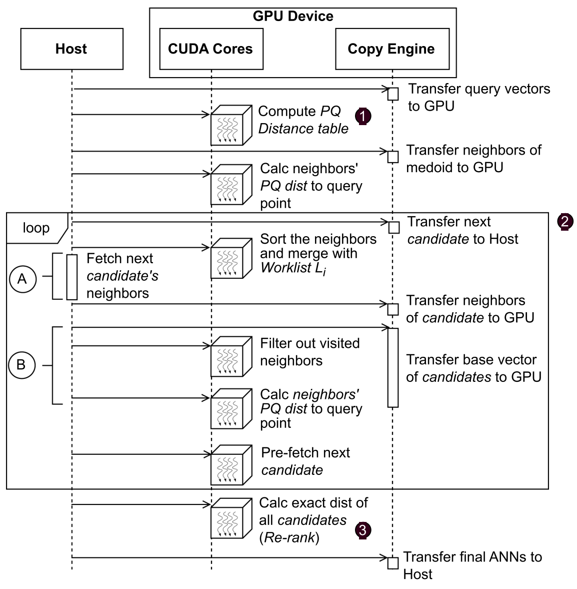
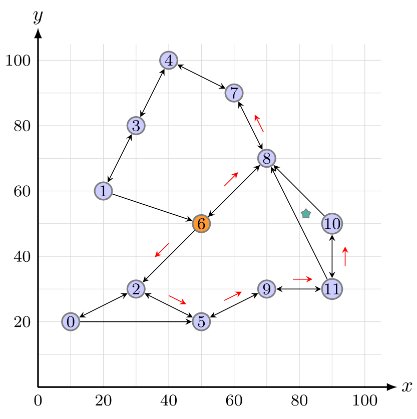

GPU Device
Host
CUDA Cores
Copy Engine
Transfer query vectors to GPU
Compute PQ Distance table
1
Transfer neighbors of medoid to GPU
Calc neighbors' PQ dist to query point
loop
Transfer next candidate to Host
2
Sort the neighbors and merge with Worklist L_i
A
Fetch next candidate's neighbors
Transfer 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)
3
Transfer final ANNs to Host

Fig. 1. Schema of BANG. ▨: CUDA *kernel*; Ⓐ: overlapping execution on the host with GPU kernel; Ⓑ: overlapping GPU kernel with data transfer between host and device.

of billion-scale datasets on a single GPU, enhancing the overall performance of large-scale ANNS.

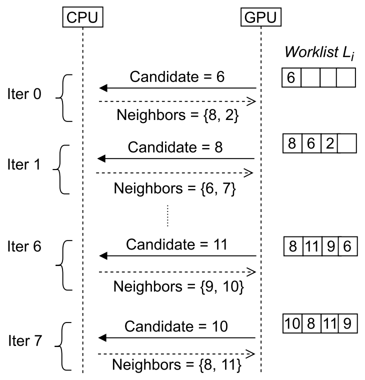
Fig. 1 shows the complete workflow of BANG and the interaction between CPU and GPU. CPU maintains the graph and looks up its adjacency list to retrieve the neighbours of a node sent by the GPU. Similarly, the CPU maintains the complete base dataset



(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)

Fig. 2. Example illustrating BANG workflow for ANNS on Vamana graph. The star indicates the query point $q(82, 53)$.

as it cannot fit on the GPU, and the GPU performs the search using compressed vectors derived from the actual data vectors. Each step shown under the 'CUDA Cores' column in Fig. 1 is executed by an individual CUDA *thread-block* (Section II-A) independently for each query. Fig. 2 shows the workflow on a toy example, having 12 points in the base dataset, and its prebuilt Vamana graph index is shown in Fig. 2(a). We use this running example to explain our technique, BANG.

BANG addresses the challenges discussed in Section III by efficiently utilising CPU, GPU, and PCIe bus hardware through a phased-execution approach. This strategy optimally balances ANNS workload across CPU and GPU while mitigating CPU-GPU data transfer bottlenecks by implementing prefetching techniques for asynchronous data transfer from CPU to GPU during specific phases. Further, by systematically dividing the ANNS activity into distinct steps, each with its own characteristics of work distribution and span, we maximise GPU parallelism, in contrast to other approaches [24], [67] that treat the entire search activity as a single block (i.e., single kernel). That is, we design separate GPU kernels (as different phases) for each of these tasks within the search activity: (1) filtering out previously processed neighbours of the current node u_i^* at Line 7, (2) computing distances of neighbours to the query q_i at Line 11, (3) sorting the neighbours by computed distances at Line 13, and (4) merging the closest neighbours into the worklist at Line 14. Fig. 2(c) shows these steps for each iteration of the query, where the GPU filters and sorts the neighbours by their distance to the query, merges it with L_i, and selects the first candidate node (e.g., 8 in iteration 1) to be visited next. For each individual kernel, we optimise the thread-block size (per query) to maximise GPU occupancy, as a single sub-optimally configured block can significantly reduce performance. Furthermore, this phased-execution strategy allows us to reduce GPU idle time by eagerly predicting the next candidate node u_i^* immediately after neighbour distance calculation rather than waiting for the previous iteration to finish before communicating u_i^* to the CPU.

In addition to greedy search, BANG employs two additional steps which result in three stages that happen one after the other: ❶ the distance table construction, ❷ greedy search, and ❸ re-ranking, as shown in Fig. 1. Since the dataset cannot fit in GPU memory, it is compressed using the popular Product Quantization [32] technique. This technique partitions the d-dimensional dataset vector space into m vector subspaces and performs independent k-means clustering of the dataset points (nodes) within each subspace, resulting in a centroid vector and ID for each cluster. Each dataset point is encoded using the cluster IDs belonging to the point in the respective m vector subspaces. Eventually, the input d-dimensional vector is represented by a shorter m-dimensional vector. Fig. 2(b) shows the cluster ID associated with each node. Since $m = 1$ in the toy example, each 2-dimensional node vector is encoded into a single-byte cluster ID. We calculate the distances between centroid vectors and a given query vector within each subspace in a preprocessing step (❶) and store them in a data structure *PQ Distance Table* (see Fig. 2(b)). During the greedy search stage (❷), distances between dataset points (in compressed form) and query vectors are calculated by summing the precomputed distances of the cluster IDs across all subspaces (i.e., available in *PQ Distance table*). Finally, after the greedy search process converges, BANG performs a re-ranking step (❸) to refine the inaccurate set of k nearest neighbours that might have resulted due to the use of compressed vectors. The re-ranking step calculates the exact distances (Fig. 2(d)) of candidate nodes to the query point using the original data vectors, ensuring precise identification of the final list of k nearest neighbours. Note that the exact distances in Fig. 2(d) and the corresponding PQ distances in Fig. 2(b) happen to be identical for the respective nodes as it is a toy example. However, in practical scenarios they would be different. For the toy 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. BANG performs both reranking and distance table construction on GPU. Note that, for the re-ranking step, only full vectors of selected nodes are sent to GPU, which are fewer (compared to sending the entire dataset) and can thus collectively fit into GPU memory for all queries.

In the following subsections, we describe in detail how we parallelise and optimise the mentioned kernels and each step in Algorithm 2.

A. Construction of $PQDistTable$ in Parallel

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, Algorithm 2 uses approximate distances calculated using compressed vectors at Line 12. 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 the PQ Distance Table (in short, $PQDistTable$). Here, we describe the construction of $PQDistTable$. We describe later (in Section IV-D) how we use these precomputed distances to efficiently compute the *asymmetric distance* [32] 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 [29], [31] that use Product Quantization [32]. We empirically determine $m = 74$ for our setup, guided by the available GPU global memory (Section VI shows the ablation study with varying m values). For example, in Fig. 2(b), after compression, each input vector (2 bytes) gets compressed to a size of 1 byte using $m = 1$ for the product quantization. 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 parallelisation scheme, the span of the algorithm is $O(m \cdot subspace\_size) = O(d)$, where d is the dataset dimension, and $subspace\_size$ is the size of the subspace (i.e., d/m).

B. 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 candidate 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 minimise data transfer overhead. Specifically, from device to host, the transfer includes a list of final approximate nearest neighbours 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 nodes of candidate nodes (Line 6) and base vectors/coordinates 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 streams for task parallelism [55]; BANG makes use of these. During the search, neighbours of a given candidate 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 node's base vector and its neighbourhood list as they are placed next to each other on the host 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 dataset 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 place `cudaStreamSynchronize()` at appropriate locations to honour the data dependencies.

C. Parallel Filtering of Visited Neighbours

Algorithm 2 (Line 5) may encounter the same node multiple times during intermediate iterations of the search. Not filtering visited nodes may degrade throughput due to redundant distance computations (Line 12) and reduce recall by allowing repeated entries in the worklist $\mathcal{L}_i$, potentially displacing more promising nodes and causing premature termination of the search. Our experiments on the SIFT-1B dataset show that this leads to up to a $10\times$ drop in recall compared to when visited nodes are filtered. In each search iteration, Line 5 returns up to $R \times \rho$ neighbours to the GPU, where R is the graph's degree and ρ is the query batch size, necessitating efficient visited checks on this large set to avoid overhead. Tracking visited nodes on the GPU is challenging, as bitmaps incur high memory overhead for large datasets, and dynamic data structures such as priority queues or hash tables are inefficient on GPU as they underutilise its massive parallelism [24], [67]. Therefore, we adopt the well-known Bloom filter for its low memory footprint and suitability for GPU execution.

We use a separate Bloom filter per query and assign one query per thread block to enable fine-grained parallelism (Line 10). Parallel accesses to the Bloom filter do not require global synchronisation, as each query uses a separate filter within a thread block. In case of a race within a thread block, i.e., if thread x sets an entry for a node and thread y tests a different node that hashes to the same entry before x's write completes, y may see the entry as unset. However, this is safe, as it prevents incorrect inclusion of the unvisited node of y. Conversely, in the opposite case, y may see the entry set by x and wrongly mark its own node as visited. While this can be avoided by synchronizing reads before writes using a barrier like `__syncthreads()`, we skip synchronisation due to its overheads, and such cases are rare.

D. Parallel Neighbour 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 IV-A), we can compute the distance of a query point q_i to a node 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.

When computing distances in GPU-based ANNS, summations are commonly conducted using reduction APIs from NVIDIA's CUB [50] library, such as those in [24], [67], 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 subspaces (with each thread processing a segment of size $\frac{m}{g_{size}}$). Each thread sequentially computes the sum of values in a segment using thread-local registers, avoiding synchronisation. Finally, to efficiently add segment-wise sums using g_{size} threads, we explore two approaches: (i) atomics, employing *atomicAdd()* for the final result; (ii) Sub-warp-level reductions, utilising CUB [50] *WarpReduce*. Empirical tuning with $m = 74$, $t_b = 512$, and $g_{size} = 8$ yields optimal performance in our experimental setup, with the second approach, utilising sub-warp level reductions, marginally outperforming the first. Overall, our segmented approach outperforms standard alternatives like CUB *WarpReduce* ($1.2\times$ slower) and *BlockReduce* ($4\times$ 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 neighbours, 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 [16], [22]. However, this approach did not yield significant improvements in locality, and so we do not perform graph reordering.

Work-Span Analysis: The work of the algorithm is $O(R \cdot m \cdot \rho)$, where R is the degree bound of the graph. Owing to the parallelisation scheme, the span of the algorithm is $O(\log\ m)$.

E. 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 $\mathcal{L}_i$. After that, while the CPU fetches 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 optimisation to *eagerly* identify 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 optimisation 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 fetches neighbour IDs. That is, Line 5 executes on CPU concurrently with tasks at Line 13 and 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 an increase in throughput and experimentally found to be 8–12%.

F. 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' in the increasing order of their 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. As 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.

For our use case, we need to sort small lists, typically having up to 64 nodes. We assign one thread-block to a query. Furthermore, 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 IV-G). 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 IV-G). 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 $O(R \cdot log^2(R) \cdot \rho)$. Owing to the parallelisation scheme, the span of the algorithm is $O(log^2(R))$.

G. Parallel Merge

The parallel merge routine that we detail in this section is employed in the parallel merge sort, as explained in the preceding section. Additionally, it is utilised to merge the sorted neighbour list $\mathcal{N}_i'$ with the worklist $\mathcal{L}_i$ at Line 14 in Algorithm 2.

Fig. 3 shows our parallel list merge algorithm on two lists of size four each. Consider two sorted lists L_{in1} and L_{in2} having m and n items respectively. Suppose that an item e in L_{in1} is at position $p_1 < m$, and if inserted in L_{in2} while

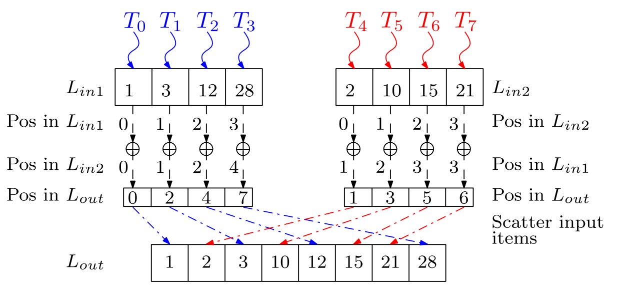


Fig. 3. Parallel List Merge. L_{in1} and L_{in2} are merged to create the merged list L_{out}.

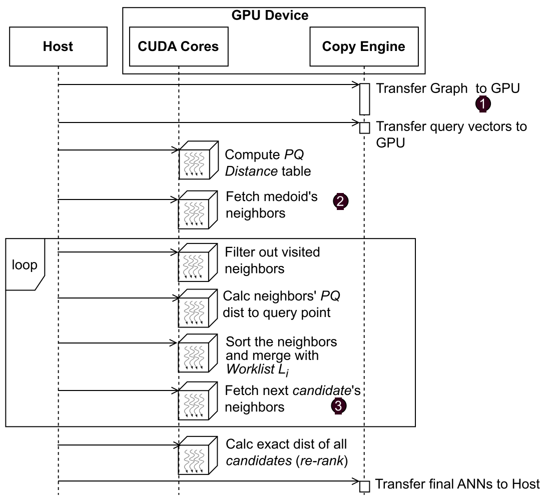

GPU Device
Host
CUDA Cores
Copy Engine
Transfer Graph to GPU ❶
Transfer query vectors to GPU
Compute PQ Distance table
Fetch medoid's neighbors ❷
loop
Filter out visited neighbors
Calc neighbors' PQ dist to query point
Sort the neighbors and merge with Worklist L_i
Fetch next candidate's neighbors ❸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

Fig. 4. Schema of BANG `In-memory` variant.

maintaining sorted order, it will be placed at position $p_2 < n$. Then, 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 Fig. 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 (i.e., $3+4$). Each thread, assigned to an item, concurrently performs a binary search for that item in the other list. We keep the lists in the GPU's shared memory to minimise 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; 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 $O(\ell \cdot log(\ell))$. Owing to the parallelisation scheme, the span of the algorithm is $O(log(\ell))$.

H. Re-Ranking

As discussed, we use approximate distances (computed using PQ distances) throughout the search in Algorithm 2. Hence, we employ a final re-ranking step [61] after the search converges to enhance the overall recall.

We implement the re-ranking step as a separate kernel with optimised thread-block sizes. Re-ranking involves computing the exact (i.e., Euclidean) 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 IV-F. Thanks to the asynchronous data transfers (Section IV-B), the candidates' base dataset vectors 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 $O((d \cdot |\mathcal{C}| + |\mathcal{C}| \cdot log^2(|\mathcal{C}|)) \cdot \rho)$, where $|\mathcal{C}|$ is the maximum number of candidate nodes for a query and d is vector dimension. Owing to the parallelisation scheme, the span of re-ranking is $O(d + log^2(|\mathcal{C}|))$.

V. Implementation Variants

We refer to the implementation described so far in Sections III and IV as BANG. In addition, we provide versions of the implementation specifically optimised for medium-sized datasets (having up to 100-million points) that can fit in the GPU memory along with their respective indexes. We discuss these versions below.

A. In-Memory Version

Since the CPU-GPU interconnect stays constantly busy when we store the graph on the CPU, as in BANG, careful task management on the GPU is needed to avoid idle time due to CPU-GPU interdependency. However, following the same approach is unnecessary for smaller graphs that can fit on a GPU. Keeping the entire graph on the GPU, rather than the CPU, eliminates the memory-intensive process of the CPU fetching neighbours from its memory for GPU-supplied candidates, resulting in enhanced throughput. We refer to this version of the implementation as BANG `In-memory`. The schematic block diagram of this version is shown in Fig. 4. There are two main differences from Fig. 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 accesses to global memory instead of relying on the CPU.

B. Exact-Distance Version

This variant further optimises the BANG `In-memory` version. Instead of using PQ distances within the greedy search loop and subsequently compensating the inaccuracies (resulting due to the use of PQ distances) by the re-ranking step, we directly use exact (Euclidean) distance calculations instead of PQ distances. We refer to this version as BANG `Exact-distance`. The

TABLE II
REAL-WORLD DATASETS IN OUR TEST-SUITE

Dataset	Type	Number of Data Points	d	Vector Size(GB)	Graph Size(GB)	CV Size(GB)
SIFT-1B [33]	uint8	1000,000,000	128	128.0	260.0	74.0
Deep-1B [62]	float	1000,000,000	96	384.0	260.0	74.0
MSSPACEV-1B [14]	int8	1000,000,000	100	100.0	260.0	74.0
Text-to-Image-1B [17]	float	1000,000,000	200	800.0	260.0	74.0
SIFT-100M	uint8	100,000,000	128	12.8	26.0	7.4
Deep-100M	float	100,000,000	96	38.4	26.0	7.4
MSSPACEV-100M	int8	100,000,000	100	10.0	26.0	7.4
Text-to-Image-100M	float	100,000,000	200	80.0	26.0	7.4

CV size stands for compressed vectors size.

main differences from Fig. 4 are: (1) absence of PQ Distance Table computation, (2) exact distance computation (instead of PQ distance summation), (3) absence of re-ranking step.

VI. EXPERIMENTAL SETUP

A. Machine Configuration

Hardware: We conduct our experiments on a machine with 2 x Intel Xeon Gold 6326 CPU (with 32 cores), clock-speed 2.90 GHz, and 660 GiB DDR4 memory. For the Text-to-Image-1B dataset, we used 1133 GiB due to its large size. It houses a 1.41 GHz NVIDIA Ampere A100 GPU with 80 GB global memory and a peak bandwidth of 2039 GB/s. 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 22.04.1 (64-bit). We use *g++* version 11.3 with `-O3`, `-std=c++11` and `-fopenmp` flags to compile the C++ code. We use OpenMP for parallelising the C++ code. The GPU CUDA code is compiled using *nvcc* version 11.8 with the `-O3` flag. The source code of `BANG` is publicly available at *https://github.com/karthik86248/BANG-Billion-Scale-ANN*.

B. Datasets

We present experiments on four popular real-world datasets that originate from a diverse set of applications. We summarise the characteristics of these datasets along with the sizes of the graphs generated using Vamana and compressed vectors in Table II. The *Deep dataset* [62] is a collection of one billion image embeddings compressed to 96 dimensions. The *SIFT dataset* [33] dataset represents the 128-dimensional SIFT (Scale-Invariant Feature Transform) descriptors of one billion images. The *Microsoft SPACEV (MSSPACEV) dataset* [14] 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-scale datasets in our test suite, we pick the first billion points in the dataset. The Text-to-Image (T2I) dataset, derived from the Yandex [17] visual search engine, comprises one billion 200-dimensional image embeddings for indexing. It focuses on a cross-domain setting, where a user provides a textual query, and the search engine retrieves the most relevant images. This scenario introduces the possibility of differing distribution patterns between the dataset and the query vectors, also referred to as an Out-Of-Distribution (OOD) dataset, which makes it challenging to handle. Henceforth, for brevity, we refer to datasets containing one billion data points as *1*B *datasets* and those with 100 million data points as *100* M *datasets*. The 100M versions of the four datasets are generated by extracting the first hundred million points from the respective 1B datasets. We use a query batch size of 10,000 unless otherwise stated, with all queries executed concurrently in a single batch, following prior works [24], [67]. For datasets that provide more than 10,000 queries, we select a subset consisting of 10,000 queries.

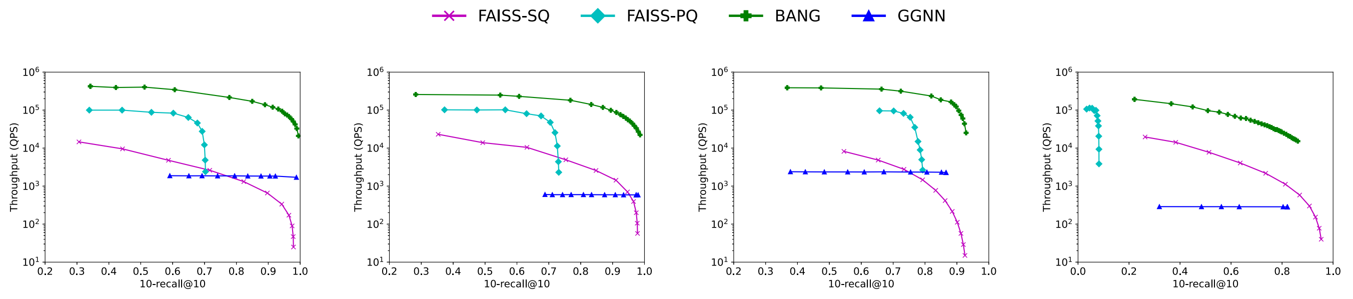
C. Algorithm Parameters: `BANG`

Graph Construction: We run `BANG` search algorithm on Vamana graph, built using `DiskANN` [29]. We specify $R = 64$ (maximum degree bound), $L = 200$ (size of worklist used during build) and σ=1.2 (pruning parameter), following the recommendations of the original source. The rightmost column in Table II shows the size of the resulting graph for each dataset.

Bloom filter: We use a Bloom filter per query, of size ~ 400 kB. Our Bloom filter uses an array of z *bools*, where z is determined using an estimate of the number of processed nodes for a query, a small tolerable false-positive rate, and the number of hash functions used. We use two FNV-1a hash functions [23], which are lightweight, non-cryptographic hash functions often used to implement Bloom filters.

Compression: During graph construction, we configure the parameters for PQ compression such that the generated compressed vectors fit entirely 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 vary (although m remains the same) for different datasets depending upon their dimensionality. For example, for the 128-dimension SIFT-1B dataset, the compression ratio is $74/128 = 0.57$, while for the 384-dimension Deep-1B dataset, the compression ratio is $74/384 = 0.19$. The rightmost column in Table II shows the total size of PQ compressed vectors for each dataset.

Search: The search starts from the *medoid*, a node pre-determined during graph construction. We use the widely used $k{-}recall@k$ [29] recall metric. The goal of ANNS is to efficiently retrieve, for a query point q, a set $\tilde{S}$ of k candidate points from the dataset in order to maximise $k\text{-}recall@k := \frac{|S \cap \tilde{S}|}{k}$, where S is the ground truth of the k nearest neighbours of q in the dataset. We measure the throughput using *Queries Per Second* or *QPS* which is (*number of queries processed / search time*), where search time is the time taken to process all queries. The search parameter t in Algorithm 2 represents the size of the worklist $\mathcal{L}$. The recall increases with t. The lower bound for t is k, and the maximum value of t is empirically set to 152 for our experiments. The Bloom filter (Section IV-C) is created to hold 399,887 entries per query, which we determined empirically. We tune the Bloom filter size to lower values in order to generate lower recall values (below the recall values achieved by using



(a) SIFT-1B (b) Deep-1B (c) MSSPACEV-1B (d) Text-to-Image-1B

Fig. 5. Throughput (y-axis) v/s 10-recall@10 for 1B datasets. y-axis is in log-scale.

$t = k$). To increase statistical significance, we report the average throughput over five independent runs.

D. Baselines

We primarily evaluate BANG against the state-of-the-art open-source ANNS methods GGNN [24] and FAISS [31] that support billion-scale data on a single GPU. While our focus is on billion-scale datasets, we include comparisons on 100M datasets for completeness, including comparison with CAGRA [46], the leading GPU-based ANNS method for 100M points, despite its lack of support for billion-scale data.

GGNN [24]: GGNN provides configurable graph construction parameters *viz.* number of out-going edges (k), number of disjoint points selected for subgraphs (s), number of levels in the hierarchical graph (l), slack factor (τ_b) and number of refinement iterations (r). In the original work, eight GPUs were used for billion-scale datasets, but we configured GGNN to run with a single GPU. We use the recommended build parameters from the original source code repository [25] for most datasets; for others, we follow authors' private email suggestions for optimal performance on our setup. We configure 1B dataset as {k:20, s:32, l:4, τ_b:0.6, r :2, 16 shards} and 100M dataset as {k:24, s:32, l:4, τ_b:0.5, r:2, 2 shards}.

FAISS [31]: FAISS provides users with configuration options for quantizing the base dataset using either Scalar Quantization (SQ) or Product Quantization (PQ); following the FAISS Wiki guidelines [30], we experiment with different IVF index types and PQ/SQ transforms to optimise recall and throughput. For PQ, we configured the index string as: *OPQ32_128*, *IVF262144*, *PQ32*. The first field specifies the vector transformation used for preprocessing, the second denotes the IVF index type, and the third indicates the quantization scheme applied. For SQ, we configured the index string as: *IVF65536, SQ8* as indicated in `Billion-Scale Approximate Nearest Neighbor Search Challenge` [52]. Note that SQ indexes are less space-efficient than PQ but provide better recalls. We performed all experiments on a single GPU.

CAGRA [46]: CAGRA is a state-of-the-art ANNS algorithm in the Nvidia cuVS [43], a high-performance, GPU-accelerated library for efficient vector search. We deployed CAGRA using the RAFT's ANN benchmark collection. We configured CAGRA with the NN-Descent [19] algorithm and set the kNN graph degree to 32. For optimal search results, we vary the internal priority queue size (*itopk* parameter) from 32 to 512.

VII. Results

In this section, we comprehensively compare the performance and solution quality of BANG with various state-of-the-art methods across different datasets detailed in Section VI. We present the results on 1B datasets followed by 100M datasets.

A. Performance on 1B Datasets

Fig. 5 compares BANG against GGNN and FAISS on billion-size datasets. BANG outperforms GGNN and FAISS in throughput for compatible recalls at 10-recall@10 across all four datasets. On the SIFT-1B and Deep-1B datasets, BANG achieves an average (geomean) throughput improvement of 50.5× over the competing methods at a high 10-recall@10 of 0.95. Specifically, for a 10-recall@10 of 0.9, BANG achieves speedups of 55×, 50×, and 400× on the SIFT-1B, Deep-1B, and MSSPACEV-1B datasets, respectively. On the more challenging Text-to-Image-1B dataset, BANG and other methods exhibit lower recall values. Here, too, BANG outperforms the closest baseline by 25× at a 10-recall@10 of 0.8.

To achieve optimal performance in FAISS-PQ, we configure the index values to be large enough to utilise the available GPU memory to the maximum extent possible. However, FAISS-PQ fails to achieve higher recall values (compared to graph-based techniques) because of losses due to compression. The number of distance calculations exponentially increases with recall, as observed in [38], which negatively impacts its throughput. Furthermore, in general, the number of distance computations in FAISS is orders of magnitude higher than in proximity graph-based methods. FAISS-SQ achieves higher recall due to less aggressive compression, as each dimension is encoded with a single byte. In contrast, FAISS-PQ encodes M dimensions using one byte, resulting in higher compression but lower recall. However, FAISS-SQ's larger index size requires data swaps between the CPU and GPU, leading to lower throughput compared to FAISS-PQ.

For GGNN on a single GPU with 1B dataset, data transfer overhead from CPU to GPU inlined with the search impedes the throughput. After utilising the 80 GB of GPU global memory: the SIFT-1B dataset, for example, requires an additional 123 GB

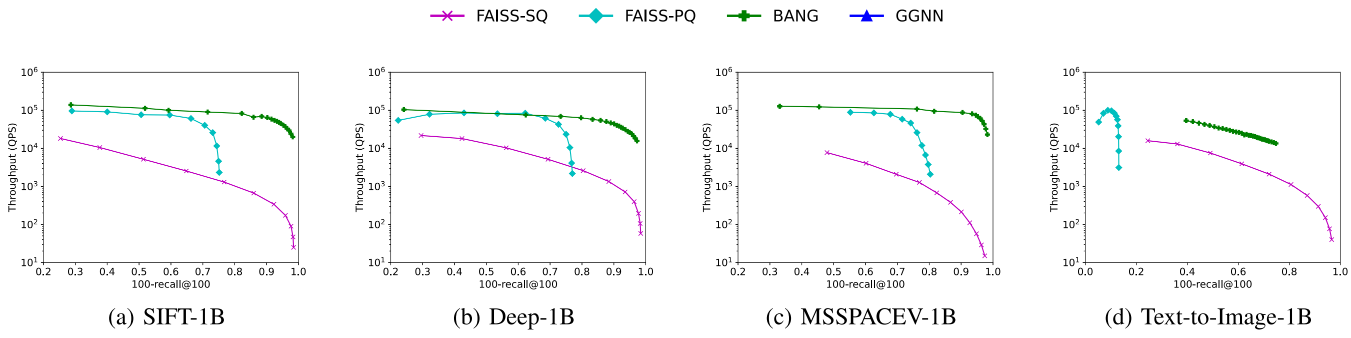


Fig. 6. Throughput (y-axis) v/s 100-recall@100 for 1B datasets. y-axis is in log-scale.

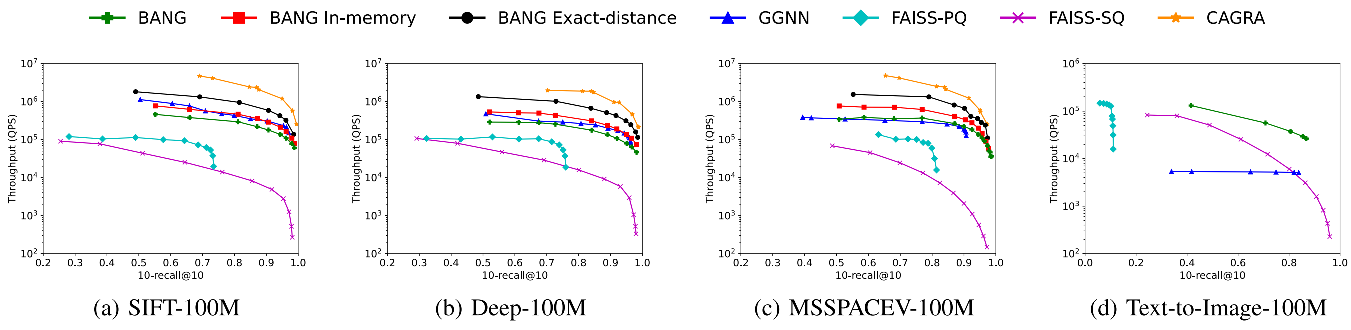


Fig. 7. Throughput (y-axis) v/s 10-recall@10 for 100M datasets. y-axis is in log-scale.

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 75 GB (total 203 GB), respectively. Transferring this additional 123 GB data over the PCIe bus @ 32 GB/s will take more than 3.8 seconds in practice. The original GGNN implementation addressed the memory transfer overhead by using eight GPUs to process the shards in parallel.

BANG search runs on GPU but does not require the graph to reside on the GPU. It eliminates the need to transfer the base dataset to GPU by employing compressed vectors (instead of base vectors) present on the GPU. Furthermore, the reason for the high throughput of BANG is attributed to the massive parallelism achieved on the GPU (Section IV). The re-ranking step, coupled with the integration of Bloom filters to filter processed nodes (prevents duplicate entries in the worklist and allows more deserving nodes to be included in the worklist), significantly contributes to achieving a high recall for BANG.

Fig. 6 presents the results of our experiments with 100-recall@100. While ANNS evaluations in the literature typically focus on 10-recall@10, real-world applications like recommender systems [58] often require fetching a larger number of nearest neighbours using ANNS, which are then filtered for final recommendations. To reflect this, we extend our evaluation to include 100-recall@100. The drop in QPS for BANG is due to the increased worklist length t required to achieve the same recall as 10-recall@10, with the minimum value of t being k, where k is 100 for 100-recall@100. (The effect of varying worklist length on recall and throughput is detailed in Section VII-G). There is no significant throughput deterioration in FAISS, as the same number of clusters and entries within the posting lists are compared regardless of the recall parameter. GGNN, however, exhibits low 100-recall@100 values (< 0.1), so we omit its plots in Fig. 6.

B. *Performance on 100M Datasets*

Fig. 7 shows the comparison of throughput and recall 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 the two variants of BANG (Section V). The BANG In-memory and BANG Exact-distance variants achieve 2× and 3× higher throughput compared to the base variant of BANG since there is no CPU-GPU data transfer overhead in the BANG variants. Interestingly, the BANG Exact-distance variant outperforms GGNN. We might expect the same performance from the BANG Exact-distance variant as GGNN, since it uses only GPU-resident data structures as in GGNN and does not use compressed vectors for distance computations to query points. However, the main difference between the two is the type of underlying proximity graph. The kNN graph structure of GGNN requires more hops or iterations for the search to terminate than BANG Exact-distance. In contrast, in BANG Exact-distance, the Vamana graph contains long-range edges, reducing the hops, enabling faster convergence with fewer distance computations. Note that, compared to BANG and FAISS, for GGNN, the performance improvement on 100M datasets over 1B datasets is significant, since the CPU to GPU data transfer overhead, which is present in the 1B datasets, is not present here.

TABLE III
BREAKDOWN OF THE TOTAL ANNS EXECUTION TIME FOR 10-RECALL@10 = 0.9 AND QUERY BATCH SIZE = 10,000

Activity	**BANG**		**GGNN**		**FAISS-SQ**	
	SIFT-1B	**Deep-1B**	**SIFT-1B**	**Deep-1B**	**SIFT-1B**	**Deep-1B**
	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)
GPU Processing	65.8	77.5	284.8	496.0	13.5	5.7
CPU Processing	15.9	20.6	0.1	0.1	14107.0	6963.9
Data Transfer (CPU $\leftrightarrow$ GPU)	15.4	16.4	5087.6	16303.9	9.6	9.9
Overall Processing	85.3	101.2	5372.5	16800.0	14130.1	6979.5

We compare with CAGRA on all 100M datasets except Text-to-Image, because the dataset (80 GB) and graph index (12.8 GB), which are required for its execution, exceed the 80 GB global memory capacity of the A100 GPU. We observe that CAGRA achieves nearly $1.5\times$–$2\times$ higher throughput than BANG, although it performs $13\times$ more distance computations than BANG. Furthermore, using the NVIDIA Nsight-Compute profiler (NCU) on the Deep-100M dataset, we measure nearly 7 billion floating-point operations for BANG and nearly 12 billion for CAGRA. However, CAGRA achieves a higher memory throughput of 1270 GB/s compared to 577 GB/s for BANG. It also has a small memory access overhead due to uncoalesced accesses of 3% compared to 40% for BANG, as measured by NCU using the metric *derived__memory_l2_theoretical_sectors_global_excessive* [44]. The higher performance of CAGRA is because it uses the highly optimised RAFT [42] primitives for bitonic sort/merge and a space-efficient hash table that fits in shared memory, reducing the access latency. As we target billion-scale datasets, our frequently accessed bloom filter data structure is too large to fit in shared memory; this results in frequent global memory transactions, which impedes performance.

C. Search Latency and Memory Usage

To gain insight into the search latency, we study the time spent by different ANNS methods on (i) CPU execution, (ii) GPU execution, (iii) CPU-GPU data transfer, and (iv) total search time. Table III presents the results for BANG and baseline methods on two 1B datasets at a 10-recall@10 value of 0.9. For baselines, the overall processing time (the last row) is the sum of the first three components ((i)-(iii)). However, for BANG, due to its CPU-GPU load balancing with prefetching and asynchronous transfers, the total time is reduced by approx. 12% on both datasets compared to its unoptimised implementation. Further, the table confirms that GGNN suffers from high transfer overheads due to frequent shard swaps. FAISS-SQ performs only coarse quantization on the GPU and actual search on the CPU, resulting in low GPU processing time. FAISS-PQ variant does not reach the target 10-recall@10 of 0.9 and is thus excluded from the table. For completeness, FAISS-PQ's execution times for 10-recall@10 of 0.7 on SIFT-1B and DEEP-1B are {4154, 45.9, 0.06, 4200} ms and {4302, 24.9, 0.07, 4327} ms, respectively, for (i) CPU, (ii) GPU, (iii) transfer, and (iv) total time. As these values indicate, unlike FAISS-SQ, the entire search in FAISS-PQ is performed on the GPU.

Furthermore, we capture the peak GPU memory usage (in GB) for each method, out of the 80 GB available on the A100 GPU.

TABLE IV
PERCENTAGE BREAKDOWN OF GPU TIME (FROM VALUES IN TABLE III), AND MEMORY USAGE OF BANG KERNELS

Kernel	**SIFT-1B**		**Deep-1B**	
	Time (%)	Mem (MB)	Time (%)	Mem (MB)
Compute PQ Distance table	2	724	2	726
Calc neighbours' PQ dist to query point	46	71300	49	71300
Sort neighbours and merge with Worklist	18	20	17	21
Filter out visited neighbours	20	3819	19	3819
Calc exact dist of all candidates (Re-rank)	4	158	4	448
Pre-fetch next candidate	10	10	9	18

On SIFT-1B, it is 76.0 (BANG), 78.8 (GGNN), 44.6 (FAISS-PQ), and 2.6 (FAISS-SQ); on Deep-1B, it is 76.3, 68.8, 45.9, and 2.5, respectively. The memory usage is dominated by compressed vectors for BANG, posting lists for GGNN, and graph index shards for FAISS-PQ. Thanks to PQ compression, BANG can run on lower-memory GPUs (e.g., 40 GB A100 GPU) while maintaining comparable performance (Section VII-E). FAISS-SQ has the least GPU-memory usage, as only centroids reside on the GPU, with search executed on the CPU. For BANG, we further measure the distribution of GPU time and memory usage for each kernel. Table IV presents the results. The majority of the time and memory is utilised by the key kernels involved in distance calculation and neighbour filtering, which is consistent with the analyses in Sections IV-D and IV-C.

D. Evaluation of Optimisations in BANG

Key optimisations in BANG include multi-kernel approach, prefetching candidate nodes and asynchronous memory transfers between the host and GPU, discussed in Sections IV, IV-E, and IV-B, respectively. We perform experiments on two popular 1B datasets at a baseline 10-recall@10 value of 0.9 to demonstrate the efficacy of the three important optimisations. To measure their impact, we disabled each optimisation individually and recorded the throughput. The performance improvements for SIFT-1B are 10.1%, 8%, and 6.2%, and for Deep-1B are 17.1%, 16.5%, and 13.8%, respectively. These results demonstrate that the proposed optimisations are significant contributors to BANG's superior performance.

E. Effect of Varying Compression Ratio

PQ compression techniques invariably introduce a loss in the recall. To empirically determine the impact of the compression operation on BANG's recall, we ran BANG search on the SIFT-1B dataset and varied m (i.e., the number of subspaces) during each run while keeping the other configurations the same as in Section VI. The results are shown in Fig. 8.

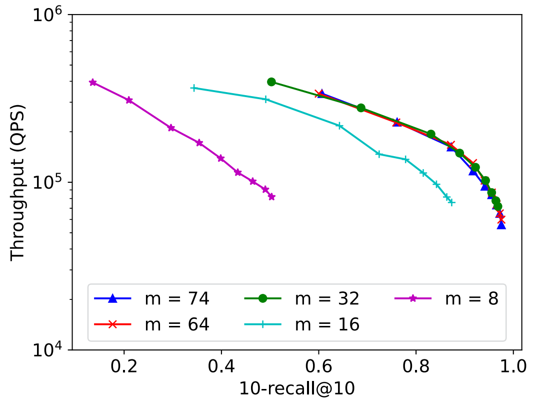


Fig. 8. Throughput (log-scale) vs. 10-recall@10 for BANG on SIFT-1B with varying compression sizes; m = number of vector subspaces present in the compressed vector representation (see Section IV).

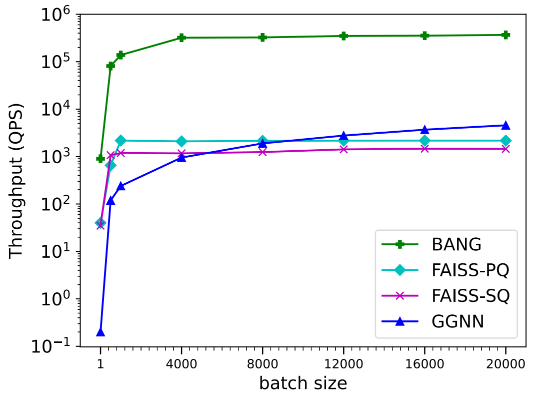


Fig. 9. Throughput (log-scale) vs. query batch size on MSSPACEV-1B dataset. Batch size denotes the number of queries processed concurrently.

While we expect the recall to decrease with high compression (i.e., a lower value of m), the result shows no noticeable change in throughput or recall values even when m is decreased from 74 to 32 (representing compression ratios of 0.57 and 0.25, respectively). We observed that recall does not drop until a compression ratio of 0.25; below this value, recall begins to decrease. Thus, this behaviour allows BANG to operate efficiently on GPUs with limited device memory without compromising recall or throughput. For example, we expect BANG search to deliver comparable recall and throughput on an A100 GPU configured with either 40 GB or 80 GB of device memory. This translates to significant power and cost savings.

One would expect the throughput to increase with the reduction in compression ratio because it reduces the number of distance computations (requiring fewer PQ distance lookups as the number of subspaces reduces) in the distance calculation kernel. Interestingly, we do not notice this pattern in the results. The inaccuracies in the distances calculated using compressed vectors increase as m decreases. Hence, BANG's search path needs more hops and detours to converge. The additional time spent in performing more search iterations outweighs the gains from the computations saved on distance calculations, resulting in no significant throughput improvement.

F. Effect of Varying Query Batch Size

We analyse the impact of input query batch size on throughput by varying it from 1 to 20,000 using the MSSPACEV-1B dataset, which contains more than 10,000 queries. To include all the baseline algorithms for comparison, we configure the parameters to achieve a 10-recall@10 of $\sim$0.8. As shown in Fig. 9, since BANG and the baselines assign one query per thread block, increasing the batch size allows greater GPU concurrency, improving QPS. Throughput increases rapidly for small batch sizes due to more queries running in parallel on all the available GPU cores but stabilises beyond a saturation point when all GPU cores are fully utilised. The saturation points for BANG, FAISS-PQ, and FAISS-SQ occur at batch sizes of approximately 4000, 1000, and 400, respectively. BANG's phased execution strategy enables effective parallelism at higher batch sizes.

TABLE V
VARIATION OF λ WITH RESPECT TO SEARCH PARAMETER t (THE SIZE OF THE WORKLIST), ON SIFT-1B DATASET

t	I	λ	10-recall@10
20	62	210.0	0.75
60	104	73.3	0.91
100	149	49.0	0.95
140	182	30.0	0.97
180	222	23.3	0.98

Smaller λ values lead to higher throughputs.

For GGNN, we observe a relatively higher increase in throughput in relation to the increase in batch size compared to other methods. As discussed in Section VII-C, the data transfer time highly dominates the query processing time. Since the data transfer time remains constant and independent of batch size, increasing the batch size leads to a proportionally greater increase in the numerator (number of queries) compared to the denominator (search time), resulting in a relatively higher improvement in throughput for GGNN.

G. Effect of Varying Worklist Size (t)

The throughput of BANG is inversely proportional to the number of iterations it takes for the search to converge. We conduct an ablation study to investigate BANG's efficiency in terms of the extra iterations it takes to converge compared to the lower bound on the number of iterations. The total number of iterations is lower-bounded by the size of the worklist (i.e., the search parameter t in Algorithm 2) because every entry in the worklist must be visited. An increase in t increases the number of iterations required for the search to converge because the number of search paths increases with more entries in the worklist. We calculate the percentage increase in the number of iterations over the lower bound as $\lambda := \frac{I-t}{t} \times 100$, where I is the actual number of iterations taken.

We vary t from 20 through 180 and measure λ and 10-recall@10 for the SIFT-1B dataset with $m = 64$ chunks. Table V presents the results. Note that recall increases with t. So, higher values of t increase the recall by enabling the exploration of more search paths. As we can observe from Table V, λ decreases with an increase in t. A lower value of λ indicates faster convergence with fewer iterations, confirming the efficacy of the recall improvement strategies for the graph index used.

Furthermore, in each iteration, multiple queries are processed in parallel, each by a separate thread block. The number of queries processed in the first iteration equals the batch size. In our phased-execution strategy, each query takes one step (a phase) at a time, along with all the other queries in the batch. The queries that are converged to termination are excluded from subsequent iterations. The total number of iterations (I) for the greedy search to complete (i.e., for all queries in a batch to converge) is crucial for throughput; fewer iterations result in higher throughput. For a particular t value, we observed that on average, 95% of queries in a batch are completed in $1.1 \times t$ iterations. This demonstrates that most queries take close to the optimal (t) number of iterations to complete. Hence, the parallelisation of BANG does not require sophisticated and heavy-weight thread scheduling mechanisms.

VIII. Related Work

A large body of prior work has tackled the problem of ANNS [60]. In this section, we discuss the popular ANNS techniques targeting multicore CPUs, manycore GPUs and custom accelerators. We also highlight the key differences between BANG and the existing GPU-based ANNS methods.

ANNS on CPU: Traditionally, ANNS has utilised Tree-based methods (like KD-trees [9], [15] and cover trees) and followed a branch-and-bound approach for search navigation. Hashing-based methods are also explored for ANNS. Locality-Sensitive Hash (LSH) [2], [3] depends on developing hash functions that can generate higher collision probabilities for nearby points than those far apart. Both these approaches face scalability challenges due to the growing dimensions and size.

NSW [37] is an early graph-based ANNS method that employs the Delaunay Graph (DG) [20] 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 [36] addresses this by introducing a hierarchical structure to spread neighbours across levels and imposing an upper bound on node degree. However, scaling HNSW to high dimensions and large datasets continues to be challenging. NSG [21] improves graph-based methods' efficiency and scalability, introducing the Monotonic Relative Neighborhood Graph (MRNG) for reducing index size and search path length while also scaling to billion-scale datasets. DiskANN [29] indexes billion-point datasets in hundreds of dimensions through the Vamana graph on commodity hardware (having a 64 GB RAM and an inexpensive solid-state drive), leveraging the properties of NSG and NSW. It introduces a tunable parameter α for a graph degree-diameter trade-off during construction. It proposes compression schemes to reduce memory consumption and computation costs during the search (see Section II-B). SPANN [11] utilises the inverted index methodology. Unlike other inverted index techniques that use compression techniques (that have some invariable losses) to reduce memory footprint, SPANN uses SSD to store a portion of the index and yet outperforms its predecessors by greatly reducing the number of disk accesses. HM-ANN [51] uses memory technologies like Optane PMM to expand the main memory size to fit the graph index and thus avoid compression. It extends popular graph-based techniques like HNSW and NSG to run on the heterogeneous memory architecture with billion-scale datasets. The survey by Manohar et al. [18] extensively studies and compares popular graph-based ANNS algorithms, addressing scalability challenges and proposing parallelisation strategies on CPU.

ANNS 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 [31] utilises quantization techniques for dimensionality reduction. The underlying search structure of FAISS is the inverted-index (IVF) [40], where points are typically assigned to buckets using either single-level clustering or two-level clustering. FAISS achieves high throughputs on the billion scale but with low recall values. PQT [61] extends Product Quantization for better GPU performance but has drawbacks, such as longer encoding lengths than PQ-code and high encoding errors. V-PQT [12] addresses this with Vector and Product Quantization Tree, and Parallel-IVFADC [54] proposes a distributed memory version of FAISS for hybrid CPU-GPU systems scaling up to 256 nodes.

SONG [67], a graph-based ANNS implementation on GPU, offers a notable speedup over state-of-the-art CPU counterparts and significant improvement over FAISS through efficient GPU-centric strategies for ANNS. GANNS [63], built on this foundation, explores enhanced parallelism and occupancy through GPU-friendly data structures and an NSW-based proximity graph construction scheme. In a related effort, TSDG [56] proposes a two-stage graph diversification approach for graph construction and GPU-friendly search procedures catering to various batch query scales. CAGRA [46], a recent graph-based ANNS implementation, leverages modern GPU capabilities (e.g., warp splitting) for substantial performance gains over existing GPU ANNS methods at a million scale. However, SONG, TSDG, GANNS and CAGRA face limitations in scaling to billion-scale datasets due to the necessity of the entire graph index to reside on the GPU.

Graph-based GPU Nearest Neighbor (GGNN) [24] search is a recent implementation that outperforms SONG. It proposes a hierarchical kNN graph construction algorithm. It divides the large input dataset into smaller shards that can fit into GPU memory for performing ANNS on GPU, with the final results merged on the CPU. GGNN can run on a single GPU at a billion scale but with low throughput. To achieve very high throughputs ($> 100,000$) at high recalls (> 0.95), it utilises eight GPUs for billion-scale ANNS.

We compare our work with algorithms that can handle billion-scale datasets on a single GPU. We find that only FAISS and GGNN fall into this category. Further, unlike these GPU-based approaches, BANG does not require multiple GPUs or data sharding techniques to achieve high recall on billion-scale datasets because it harnesses compressed vectors that can easily fit into a single GPU. Furthermore, instead of running the entire ANNS in one large kernel, BANG explores a *phased-execution* strategy, decomposing the ANNS into distinct phases that can be efficiently executed on CPU/GPU. Ultimately, BANG delivers significantly higher recall and throughput compared to these approaches.

ANNS on other Accelerators: Researchers have explored custom hardware, such as FPGAs, for ANNS acceleration, focusing on quantization-based [1], [64] and graph-based [48] approaches. However, these methods suffer from frequent data movement between CPU memory and device memory, leading to high energy consumption and lower throughputs. To address these limitations, vStore [35] 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 [13] focuses on realising ANNS on TPUs. It uses an accurate accelerator performance model considering both memory and instruction bottlenecks.

IX. Conclusion

We presented BANG, a novel GPU-based ANNS method that efficiently handles large billion-size datasets exceeding GPU memory capacity, particularly using a single GPU. It keeps the graph on the CPU and brings together computation on compressed data and optimised GPU parallelisation to achieve high throughput on massive datasets. BANG enables GPU-CPU computation pipelining and overlapping communication with computation. Thus, it is able to optimally utilise the GPU and CPU resources, 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 state-of-the-art GPU-based methods, achieving 50×-400× their throughputs at high recalls. Our evaluations on a single GPU show that BANG delivers high throughput with low hardware and operational costs, making it a highly attractive option for practical deployment.

References

[1] A. M. Abdelhadi, C.-S. Bouganis, and G. A. Constantinides, "Accelerated approximate nearest neighbors search through hierarchical product quantization," in *Proc. 2019 Int. Conf. Field- Program. Technol.*, 2019, pp. 90–98.

[2] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for near neighbor problem in high dimension," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[3] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal LSH for angular distance," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1225–1233.

[4] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proc. 4th Annu. ACM-SIAM Symp. Discrete Algorithms*, USA: Society for Industrial and Applied Mathematics, 1993, pp. 271–280.

[5] Bagisto, "Role of vector database in ecommerce chatbot," 2025. [Online]. Available: https://bagisto.com/en/role-of-vector-database-in-ecommerce-chatbot/

[6] Y. Bengio, Y. Lecun, and G. Hinton, "Deep learning for AI," *Commun. ACM*, vol. 64, no. 7, pp. 58–65, 2021.

[7] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature Biotechnol.*, vol. 33, no. 6, pp. 623–630, 2015.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] L. Cayton, "Fast nearest neighbor retrieval for Bregman divergences," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 112–119.

[10] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug Discov. today*, vol. 23, no. 6, pp. 1241–1250, 2018.

[11] Q. Chen et al., "Highly-efficient billion-scale approximate nearest neighborhood search," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 5199–5212.

[12] W. Chen, X. Ma, J. Zeng, Y. Duan, and G. Zhong, "Hierarchical quantization for billion-scale similarity retrieval on GPUs," *Comput. Elect. Eng.*, vol. 90, 2021, Art. no. 107002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790621000288

[13] F. Chern, B. Hechtman, A. Davis, R. Guo, D. Majnemer, and S. Kumar, "TPU-KNN: K nearest neighbor search at peak flop/s," *Adv. Neural Inf. Process. Syst.*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho and A. Oh, Eds., Curran Associates, Inc., vol. 35, pp. 15489–15501, 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/639d992f819c2b40387d4d5170b8ffd7-Paper-Conference.pdf

[14] S. Contributors, "Spacev1B: A billion-scale vector dataset for text descriptors," 2023, Accessed: Dec., 30, 2023. [Online]. Available: https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B

[15] R. R. Curtin, P. Ram, and A. G. Gray, "Fast exact Max-Kernel search," in *Proc. 2013 SIAM Int. Conf. Data Mining*, 2013, pp. 1–9.

[16] E. H. Cuthill and J. M. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proc. 24th Nat. Conf.*, 1969, pp. 157–172. [Online]. Available: https://api.semanticscholar.org/CorpusID

[17] A. B. Dmitry Baranchuk, "Yandex text-to-image (T2I) dataset," 2024, Accessed: Jan. 04, 2025. [Online]. Available: https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search#13h2

[18] M. Dobson et al., "Scaling graph-based anns algorithms to billion-size datasets: A comparative analysis," 2023, *arXiv:2305.04359*.

[19] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proc. 20th Int. Conf. World Wide Web*, New York, NY, USA: Association for Computing Machinery, 2011, pp. 577–586, doi: 10.1145/1963405.1963487.

[20] S. Fortune, "Voronoi diagrams and delaunay triangulations," in *Computing in Euclidean Geometry*. Singapore: World Scientific, 1995.

[21] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," 2017, *arXiv: 1707.00143*.

[22] J. A. George, "Computer implementation of the finite element method," Ph.D. dissertation, Stanford Univ., USA, 1971. [Online]. Available: https://searchworks.stanford.edu/view/2198775

[23] G. Fowler, L. Curt Noll, and P. Vo, "Fowler-noll-vo hash function," 2025. [Online]. Available: http://isthe.com/chongo/tech/comp/fnv/

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

[25] F. Groh, L. Ruppert, P. Wieschollek, and H. Lensch, "GGNN: Graph-based GPU nearest neighbor search," 2023, Accessed: Dec. 30, 2023. [Online]. Available: https://github.com/cgtuebingen/ggnn

[26] C. C. B. P. Guide, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[27] HYSCALER, "Vector database revolution: Unlocking ai breakthroughs with milvus architecture," 2025. [Online]. Available: https://hyscaler.com/insights/vector-database-revolution-milvus-ai/

[28] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. 30th Annu. ACM Sympo. Theory Comput.*, New York, NY, USA: Association for Computing Machinery, 1998, pp. 604–613, doi: 10.1145/276698.276876.

[29] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, H. Wallach, H. A. Larochelle, F. Beygelzimer, d'E. Alché-BucFox, and R. Garnett, Eds., vol. 32. New York, NY, USA: Curran Associates, Inc., 2019.

[30] H. Jegou, M. Douze, J. Johnson, L. Hosseini, C. Deng, and A. Guzhva, "Faiss wiki," 2023. Accessed: Oct. 2, 2023. [Online]. Available: https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

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

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

[33] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *Proc. 2011 IEEE Int. Conf. Acoust. Speech Signal Process.*, 2011, pp. 861–864.
[34] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu, "A GPU accelerated update efficient index for KNN queries in road networks," in *Proc. 2018 IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 881–892.
[35] S. Liang, Y. Wang, Z. Yuan, C. Liu, H. Li, and X. Li, "Vstore: In-storage graph based vector search accelerator," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 997–1002, doi: 10.1145/3489517.3530560.
[36] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, Apr. 2020.
[37] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Inf. Syst.*, vol. 45, pp. 61–68, 2014.
[38] M. D. Manohar et al., "Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms," in *Proc. 29th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, New York, NY, USA: Association for Computing Machinery, 2024, pp. 270–285, doi: 10.1145/3627535.3638475.
[39] A. C. Mater and M. L. Coote, "Deep learning in chemistry," *J. Chem. Inf. Model.*, vol. 59, no. 6, pp. 2545–2559, 2019.
[40] H. Noh, T. Kim, and J.-P. Heo, "Product quantizer aware inverted index for scalable nearest neighbor search," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 12210–12218.
[41] NVIDIA, "CUDA C programming guide," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[42] NVIDIA, "RAPIDS RAFT: Reusable accelerated functions and tools for vector search and more," 2024. [Online]. Available: https://docs.rapids.ai/api/raft
[43] NVIDIA, "cuVS bench," 2025. [Online]. Available: https://docs.rapids.ai/api/cuvs/nightly/cuvs_bench/
[44] NVIDIA, "Kernel profiling guide," 2025. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html
[45] NVIDIA, "What is retrieval-augmented generation?," 2025. [Online]. Available: https://www.nvidia.com/en-in/glossary/retrieval-augmented-generation
[46] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "CAGRA: Highly parallel graph construction and approximate nearest neighbor search for GPUs," in *Proc. IEEE 40th Int. Conf. Data Eng.*, 2024, pp. 4236–4247.
[47] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," *VLDB J.*, vol. 33, no. 5, pp. 1591–1615, 2024.
[48] H. Peng et al., "Optimizing FPGA-based accelerator design for large-scale molecular similarity search (special session paper)," in *Proc. 2021 IEEE/ACM Int. Conf. Comput. Aided Des.*, 2021, pp. 1–7.
[49] S. Rajput et al., "Recommender systems with generative retrieval," in *Proc. Adv. Neural Inf. Process. Syst.*, 2023, pp. 10299–10315.
[50] Reduction, "CUB library," 2023, Accessed: Dec. 28, 2023. [Online]. Available: https://nvlabs.github.io/cub/
[51] J. Ren, M. Zhang, and D. Li, "HM-ANN: Efficient billion-point nearest neighbor search on heterogeneous memory," in *Advances in Neural Information Processing Systems*, H.M. Larochelle, R. Ranzato, M. Hadsell Balcan, and H. Lin, Eds. Red Hook, New York, USA: Curran Associates, Inc., 2020, pp. 10672–10684.
[52] H. V. Simhadri, "Running the faiss GPU baseline," 2023, Accessed: Jul. 18, 2023. [Online]. Available: https://github.com/harsha-simhadri/big-ann-benchmarks/tree/main/neurips21/track3_baseline_faiss
[53] H. V. Simhadri et al., "Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search," in *Proc. NeurIPS 2021 Competitions Demonstrations Track*, 2022, pp. 177–189.
[54] R. Souza, A. Fernandes, T. S. Teixeira, G. Teodoro, and R. Ferreira, "Online multimedia retrieval on CPU–GPU platforms with adaptive work partition," *J. Parallel Distrib. Comput.*, vol. 148, pp. 31–45, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731520303841
[55] C. Streams, "Streams simplify concurrency," 2023, Accessed: Dec. 28, 2023. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
[56] H. Wang, Y. Wang, and W.-L. Zhao, "Graph-based approximate NN search: A revisit," 2022, *arXiv:2204.00824*.
[57] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 839–848.
[58] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, "An efficient and robust framework for approximate nearest neighbor search with attribute constraint," in *Proc. Adv. Neural Inf. Process. Syst.*, 2024, pp. 15738–15751.
[59] M. Wang et al., "Starling: An I/O-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment," in *Proc. ACM Manag. Data*, vol. 2, no. 1, Mar. 2024, Art. no. 14, doi: 10.1145/3639269.
[60] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," 2021, *arXiv:2101.12631*.
[61] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. P. A. Lensch, "Efficient large-scale approximate nearest neighbor search on the GPU," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2027–2035.
[62] A. B. Yandex and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proc. 2016 IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2055–2063.
[63] Y. Yu, D. Wen, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "GPU-accelerated proximity graph approximate nearest neighbor search and construction," in *Proc. 2022 IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 552–564.
[64] J. Zhang, J. Li, and S. Khoram, "Efficient large-scale approximate nearest neighbor search on opencl FPGA," in *Proc. 2018 IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4924–4932.
[65] J. Zhang et al., "Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2022, pp. 4493–4501.
[66] Z. Zhang, F. Liu, G. Huang, X. Liu, and X. Jin, "Fast vector query processing for large datasets beyond {GPU} memory with reordered pipelining," in *Proc. 21st USENIX Symp. Networked Syst. Des. Implementation*, 2024, pp. 23–40.
[67] W. Zhao, S. Tan, and P. Li, "SONG: Approximate nearest neighbor search on GPU," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1033–1044.
[68] Y. Zhu, R. Ma, B. Zheng, X. Ke, L. Chen, and Y. Gao, "GTS: GPU-based tree index for fast similarity search," in *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024. [Online]. Available: https://doi.org/10.1145/3654945
[69] Zilliz, "Leveraging vector databases for next-level e-commerce personalization," 2025. [Online]. Available: https://zilliz.com/learn/leveraging-vector-databases-for-next-level-ecommerce-personalization
[70] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, "SeRF: Segment graph for range-filtering approximate nearest neighbor search," in *Proc. ACM Manag. Data*, vol. 2, no. 1, Mar. 2024, Art. no. 69, doi: 10.1145/3639324.