GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction

Yuanhang Yu[♮], Dong Wen[†♮], Ying Zhang[♮], Lu Qin[♮], Wenjie Zhang[†] and Xuemin Lin[†]

[♮]*AAII, University of Technology Sydney, Australia*

[†]*The University of New South Wales, Australia*

[♮]yuanhang.yu@student.uts.edu.au; {ying.zhang, lu.qin}@uts.edu.au;

[†]dong.wen@unsw.edu.au; {zhangw, lxue}@cse.unsw.edu.au;

***Abstract*—The approximate nearest neighbor (ANN) search in high-dimensional space offers a wide spectrum of applications across many domains such as database, machine learning, multimedia and computer vision. A variety of ANN search algorithms have been proposed in the literature. In recent years, proximity graph-based approaches have attracted considerable attention from both industry and academic settings due to the superior search performance in terms of speed and accuracy. A recent work utilizes a graphics processing unit (GPU) to accelerate the ANN search on proximity graphs. Though significantly reducing the distance computation time by taking advantage of the massive parallelism of GPUs, the algorithm suffers from the high expenses of data structure operations. In this paper, we propose a novel GPU-accelerated algorithm that designs a novel GPU-friendly search framework on proximity graphs to fully exploit the massively parallel processing power of GPUs at key steps of the search. Also, we propose GPU-accelerated proximity graph construction algorithms which can build high-quality representative proximity graphs with efficient parallel implementations. Extensive experiments on benchmark high-dimensional datasets demonstrate the outstanding performance of our proposed algorithms in both ANN search and proximity graph construction.**

I. Introduction

Nearest neighbor (NN) search has been studied as a fundamental problem in many areas such as databases, computer vision and recommendation systems. Given a distance metric δ and a query point q, it is to find a point in a point set P that is closest to q. The exact search can be costly due to *the curse of dimensionality* [1]. Therefore, many researchers have diverted their attention to efficient approximate algorithms for the trade-off between accuracy and efficiency. A lot of approximate nearest neighbor (ANN) search algorithms are proposed. According to a recent ANN search benchmark paper [2], the proximity graph-based methods have outstanding search performance and can achieve high recall by only retrieving a small number of points.

Proximity graph-based methods build proximity graphs on a set of data points where each point is a graph vertex and some other (close) points will be chosen as neighbors. Figure 1 shows an example of the proximity graph. We can traverse the proximity graph from an entry vertex and quickly identify the promising nearest neighbors of the query point by greedy heuristics. Due to the high dimensionality of data points, the massive distance computation is the dominant cost while searching on the proximity graph. On the other hand, it is well-known that GPUs are promising platforms for solving simple data-intensive computing tasks (e.g., bulk distance computation) due to having such an enormous number of cores and high memory bandwidth. Considering the GPU-CPU interaction is not expensive in the context of ANN search, GPU-based solutions have been developed by industry such as Proxima from Alibaba Group [3] and Faiss from Facebook [4] for a variety of applications such as information retrieval, recommendation and advertisement. Therefore, it is rather natural to accelerate the proximity graph ANN search with GPU-based solutions. SONG [5] is the state-of-the-art algorithm that follows this line of research. By decomposing the search algorithm into 3 stages in each search iteration, SONG can comfortably parallelize the performance-crucial distance computation on a GPU in the distance computation stage, which significantly speeds up the search, compared to CPU solutions. Nevertheless, we notice that the data structure operations become the bottleneck of the search algorithm due to the execution dependency involved, which is unfriendly to GPUs. As shown in time breakdown of SONG (Figure 7) in our experiment, around $50-90\%$ of time when searching on navigable small world (NSW) graphs comes from the data structure operations such as lookup and maintenance of data structures.

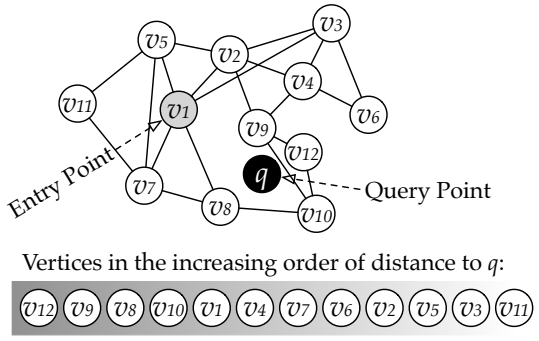


Fig. 1: An example of proximity graph G_1

The key reason behind the low data structure operation efficiency is that SONG still follows the search paradigm of CPU-based solutions. Specifically, the relevant structure operations (i.e., search and update of the priority queue and the hash table) can be efficiently implemented on the CPU. However, it is difficult for a GPU to dynamically maintain data structures with high irregular dependency. Although a set of ANN-specific optimization techniques have been proposed to eliminate dynamic GPU memory allocations and trade computations for less GPU memory consumption, SONG relies on a single thread in each thread block for data structure operations in its implementation for a good overall performance. This

inherently underutilizes the GPU computing bandwidth.

In this paper, we design a GPU-friendly search algorithm on proximity graphs such that all key steps can efficiently and fully exploit the massive parallelism of GPU. The key idea is to use two lazy strategies *lazy update* and *lazy check* such that the maintenance of the data structures can be easily processed in parallel. Consequently, we can further accelerate the proximity graph-based ANN search compared to the state-of-the-art alternative. Moreover, as reported in [2], index construction time of proximity graphs is usually more expensive compared to other categories of ANN search algorithms. This motivates us to develop GPU-based algorithms to accelerate construction of three representative proximity graphs. As with the search algorithm, the key challenge is how to fully exploit the massive parallelism of GPUs. We find that: (1) it is cost-prohibitive to directly adopt the CPU-based sequential processing-based NSW graph construction algorithm; and (2) the brute-force parallel implementation leads to NSW graphs of poor quality. Here, we propose a divide-and-conquer approach to achieve both goals at the same time: (1) fully exploit the massive parallelism of GPUs; and (2) build high-quality NSW graphs. hierarchical navigable small world (HNSW) graphs and k nearest neighbor graphs.

Contribution. The principal contributions of this paper are summarized as follows.

• We develop a GPU-accelerated ANN search algorithm on proximity graphs. Compared to the state-of-the-art, we design a GPU-friendly search paradigm so that we can efficiently and fully exploit the massive parallelism of GPUs at key steps of the search.

• We propose the first GPU-based NSW graph construction algorithm, which can make good use of the immense parallelism of GPUs and still maintain the graph's high quality. We also show that our techniques can be easily extended for construction of other proximity graphs.

• Our comprehensive experiments on representative high-dimensional datasets demonstrate the superior performance of our proposed algorithms. In terms of ANN search, we achieve up to 5x speedup with the same accuracy, compared to the state-of-the-art GPU-based algorithm. Regarding NSW graph construction, our approach can achieve 40-50x speedup on most datasets with the same graph quality, compared to the single-thread CPU-based NSW graph construction algorithm.

II. Preliminary

We formalize the problem of ANN search on proximity graphs (Section II-A) and reviewing the search algorithm on proximity graphs along with representative proximity graphs (Section II-B). We then introduce GPU architecture (Section II-C) and present the state-of-the-art GPU-based proximity graph search algorithm (Section II-D).

A. Problem Definition

Definition 1: (**K Nearest Neighbor Search**). Given a set of points P ($|P| > k$) and a distance function δ in a space S, k nearest neighbor search for a query point $q \in S$ is to return a set of points $N \subseteq P$ ($|N| = k$) such that $\forall u \in N, \forall v \in P \setminus N$,

$$\delta(u, q) \leq \delta(v, q). \quad (1)$$

Due to *the curse of dimensionality* [1], considerable research efforts turn to searching approximate k nearest neighbors (ANN) when the dimensionality of the space S is high. In this way, we might find a good trade-off between result quality and search efficiency. Let $X = \{x_i | 1 \leq i \leq k\}$ denote the result by an approximate algorithm, a common way to measure precision for the query point q is defined as $\frac{|X \cap N(q)|}{k}$ where $N(q)$ includes k nearest neighbors of the query q.

A variety of proximity graphs have been proposed in the literature to facilitate approximate nearest neighbor search. Below is a general definition of the proximity graph followed by two important properties.

Definition 2: (**Proximity Graph**). Given a set of points P in a multidimensional space S and a distance function δ, a proximity graph $G = (V, E)$ of P consists of a set of n vertices $V = P$ where each vertex in V is uniquely associated with a point in P, and a set of m edges E each of which connects two vertices in V.

Without loss of generality, we assume a proximity graph G is a directed graph. Whenever there is no ambiguity, we use a point and its corresponding graph vertex exchangeably in this paper. Generally, there are two key properties for proximity graphs used for ANN search:

(1) For each vertex v, all or the majority of its outgoing neighbors are close to v in terms of the given distance function δ. A few outgoing neighbors not close to v might also be included to speed up the search in some proximity graph models (e.g., long edges in NSW graph [6]).

(2) All vertices have the same (similar) number of outgoing neighbors. We may use an upper bound d_{max} and a lower bound d_{min} to control the number of outgoing neighbors in proximity graphs. The bounded number of neighbors is friendly for GPU-based graph processing. Note that we only keep the outgoing neighbors in adjacency lists.

Problem statement. In this paper, we aim to develop efficient *GPU-accelerated* algorithms for approximate nearest neighbor (ANN) search on a proximity graph as well as constructing proximity graphs.

B. Proximity Graph Search and Construction

Search on proximity graph. Though a variety of proximity graph models have been proposed in the literature, most of them use the *beam search* strategy which is an A^*-like search algorithm with a limited budget. The key idea of the beam search on proximity graphs is the greedy heuristic that guides the search, i.e., (1) choose the unvisited neighbor vertex which is closest to the query q in the graph search. (2) apply the backtracking to avoid a locally optimal solution under a particular budget, i.e., search more nearest neighbors than required for exploring neighbors of local optimum.

Algorithm. Given a proximity graph $G = (V, E)$, a query point q and a number k of returned nearest neighbors, Algorithm 1 searches nearest neighbors in iterations. (1) A max-heap N, a min-heap C and a hash table H are initialized (Lines 1-3). (2) A start vertex v_s is added to C and H to start iterations up (Lines 4-5). (3) As iterations begin, for each iteration, (a) pick the vertex v_c closest to q in C (Line 7). (b) check whether v_c is closer than the current k-th nearest neighbor v_f in N (Line 9). If the number of vertices in N is smaller than k, we regard

$\delta(v_f, q) = INF$. (c) If the condition does not hold, the search terminates (Line 10). Otherwise, v_c is added to N (Line 14), and its unvisited outgoing neighbors are inserted into C for subsequent exploration and marked as visited (Lines 15-17).

Example 1: Given the proximity graph G_1, the query point q in Figure 1 and $k = 4$, Algorithm 1 returns k-nearest neighbors of q. In the beginning, N, C and H are empty. Suppose v_1 is chosen as the entry point, it is added to C and H. Then, the first iteration starts up. In iteration 1, as the closest point to q in C, v_1 is popped up from C. Afterwards, v_1 is directly pushed into N because the number of points in N is lower than k. Last, all neighbors of v_1 are sequentially added to C and H because they are neither visited. In iteration 2, v_8 is popped up from C and pushed into N. Its unvisited neighbor v_{10} is added into C and H. In subsequent iterations, the search algorithm traverses the path $v_{10} \rightarrow v_{12} \rightarrow v_9$, thereby visiting all neighbors of points on this path. After iteration 5, the closest point to q in C is v_4, which is worse than v_{10} (the furthest point to q in N). Hence, traversal terminates, and v_{12}, v_9, v_8 and v_{10} in N are returned.

Algorithm 1: Search on Proximity Graph

Input: A proximity graph $G = (V, E)$, a query point q and the number of returned nearest neighbors k
Output: A set N of k nearest neighbors of q in V

```
1  C := ∅;          /* candidate set */;
2  N := ∅;      /* top k result so far */;
3  H := ∅;       /* visited points set */;
4  pick a start vertex v_s;
5  C := C ∪ {v_s}; H := H ∪ {v_s};
   // search on the proximity graph
6  while |C| > 0 do
       // pop the point v_c closest to q
7      v_c := C.Min(); C.Pop();
       // k-th closest neighbor so far
8      v_f := N.Max();
9      if δ(v_c, q) > δ(v_f, q) and N.size() = k then
10         break;
11     if N.size() > k then
12         N.Pop();
13     N := N ∪ {v_c};
14     for each outgoing neighbor u of v_c in G do
15         if u ∉ H then
16             C := C ∪ {u}; H := H ∪ {u};
17 return N;
```

Construct proximity graph. A variety of proximity graph models have been proposed in the literature. Here, we focus on two representative models: Navigable Small World (NSW) graph and K Nearest Neighbors (KNN) graph.

Navigable Small World Graph. In an NSW graph, the outgoing edges of each vertex include short-range links and long-range links. Short-range links are regarded as an approximation of the Delaunay Graph [7], and long-range links maintain the properties of the small world [8]. The construction process is a serial insertion of all points in the dataset. For each new point, (1) d_{min} nearest neighbors are selected from points in the current graph (all points are selected if the number of points in the current graph is less than d_{min}). (2) selected nearest neighbors are *bidirectionally* linked with the new point. The insertion continues until all points have been inserted into this graph. G_1 in Figure 1 is constructed as an NSW graph where d_{min} is set to 2. For example, among three neighbors of v_7, v_1 and v_5 are picked because they are closest to v_7 when v_7 is inserted; v_8 is picked because v_7 is one of the closest neighbors of v_8 when v_8 is inserted.

As its important variant, the hierarchical navigable small world graph (HNSW) is a hierarchical graph where each layer is an NSW graph for a subset of points.

K Nearest Neighbors Graph. In a KNN graph, each point is connected to its k (Here, $k = d_{min} = d_{max}$) nearest neighbors. NN-Descent [9] is often used to construct the KNN graph. It first randomly picks neighbors for each point and iteratively improves the quality of neighbors of each point by exploring the neighbors of its neighbors. The process terminates when the precision improvement of the KNN graph is small enough.

C. GPU Architecture

Thread hierarchy. We introduce threads on the GPU from two perspectives: hardware and programming model, in charge of execution and organization respectively.

Hardware. The GPU consists of several streaming multiprocessors (SM), including some streaming processors and other resources such as dispatch units.

Programming model. A program executed by a GPU is called a *kernel*, the execution of a kernel is supported by thousands of threads. These threads are organized into several *thread blocks*. A thread block contains hundreds of threads. After a kernel is launched, thread blocks will be assigned to SMs, and cores inside each SM execute these threads in a single instruction multiple threads (SIMT) fashion, at the granularity of a single *warp* which is the smallest execution unit from the view of hardware and contains 32 consecutive threads.

Memory hierarchy. There are a variety of memories such as global memory, shared memory, caches and register in GPU. They differ in capacity, latency and visibility.

D. State-of-the-art GPU Solution

SONG [5] is the state-of-the-art GPU solution for the proximity graph ANN search. According to [5], distance computation consumes over 95% of execution time in the CPU-based proximity graph ANN search (Algorithm 1). To improve the performance, SONG focuses on exploiting the enormous parallelism of the GPU to accelerate the dominant segment. This is supported by decomposing Algorithm 1 into a three-stage workflow that includes: (1) *candidates locating*; (2) *bulk distance computation*; and (3) *data structures updating*. Below are some details.

Data structures & memory usage. SONG follows the same search framework as the graph-based CPU solutions (i.e., Algorithm 1), and data structures in a block mainly include priority queues N and C, an open addressing hash table H. The differences are: (1) C is implemented in the form of a min-max heap with size k, which can save memory consumption without sacrificing performance. (2) H always includes points

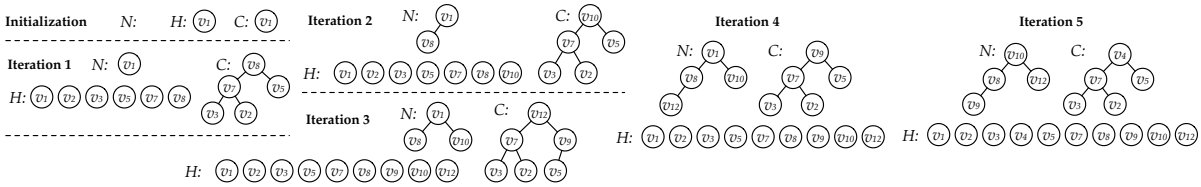


Fig. 2: An example of beam search (Algorithm 1)

in $N \cup C$. This makes H with fixed size $2k$. (3) Two auxiliary arrays *cand* and *dist* with fixed size d_{max} are used to enable collaboration between threads in the block.

To manage them, N, C and H are stored at the local memory and maintained by a single thread called host thread, while *cand* and *dist* reside in the shared memory. Besides, the query point q is also stored in the shared memory.

Three-stage search. We illustrate the search process. Notice that a point will be inserted (resp. removed) in (resp. from) H if it is inserted (resp. removed) in (resp. from) N or C.

Candidates locating. Here, N is updated based on C, and points that are potential to be loaded into C are searched. Specifically, (1) The host thread extracts the closest vertex v_c in C. (2) The furthest vertex v_f in N is obtained and compared with v_c. (3) If v_c is better than v_f, v_c and its unvisited neighbors are recorded. Otherwise, the search process terminates.

The record is completed by the host thread. It first removes v_c from C and adds v_c into N. Then, the unvisited neighbors of v_c are sequentially loaded into *cand*. The process of checking whether neighbors of v_c are visited is completed by the host thread's accessing of H point by point. When v_c is inserted and N is full, v_f is deleted.

Bulk distance computation. For each point $cand_i$ in *cand*, threads access the sub-vector of $cand_i$ from the global memory. Computation is independently carried out in parallel on these sub-vectors among threads. Finally, warp-level primitive *__shfl_xor_sync* is used to aggregate these partial results of each thread, and the sum is put into *dist*.

Data structures updating. Then, C is updated. the host thread sequentially inserts points from *cand* into C based on distances stored in *dist*. During the insertion, the size of C is never beyond k. That is, if C is full and the new point is better than the worst point in C, the worst point is removed before the new point is added.

III. Proximity Graph Search

First, we introduce the motivation of our GPU-based proximity graph search (Section III-A). We next present the search algorithm (Section III-B) and a theoretical analysis (Section III-C).

A. Motivation

We note that the distance computation can be significantly accelerated by GPU-based solutions because the sub-vectors of an entire feature vector (i.e., a point in high-dimensional space) are independent and can naturally be processed simultaneously. For instance, we can simultaneously use 32 threads in a warp to compute the Euclidean distance between two points (i.e., vectors) in 640-dimensional space where each thread takes care of the 20-dimensional sub-vector, and the partial results can be easily aggregated by warp-level primitives. However, as stressed in [5] and illustrated in Figure 7 in Section V, the bottleneck of the state-of-the-art graph search becomes the data structure operations. This is because SONG still follows the search paradigm of CPU-based solutions and the corresponding data structure operations (e.g., dynamic maintenance of the priority queue and hash table) are expensive on a GPU. Though a set of optimizations have been explored, SONG uses a single thread for data structure operations in each query to avoid irregular dependency among threads in their implementations. This inherently underutilizes the GPU computing bandwidth since multiple threads with the same instruction will be invoked at the same time even when the smallest computing unit, i.e., the warp, is used for each query.

To better maintain the data structures in the GPU solution, we apply the lazy strategy on the updating and checking operations. Though the lazy strategy has been widely adopted in the literature, we would like to point out that the existing works on proximity graph based ANN search, including the state-of-the-art GPU solution, maintain the data structures in an eager manner. To better apply a lazy strategy for the GPU-friendly solution, we need to choose new data structures and re-design the search paradigm, which, together, pose new challenges. In this paper, we re-design the search algorithm so that the maintenance of data structures is friendly to GPU. The key challenge is to implement efficient parallelism during the update of data structures while ensuring that iterations can keep running. We focus on the following two perspectives.

Candidate selection. It is non-trivial to have an efficient GPU implementation for priority queues as stressed in [5]. A set of optimization techniques has been proposed by SONG such as bounded priority queue optimization and selected insertion optimization to accommodate the GPU computation. Nevertheless, SONG has to rely on a single thread for the above process because it extracts the closest point sequentially from a priority queue, which is efficient in terms of following the greedy heuristics and quickly approaching the target but not friendly for the parallel computation in GPU.

Lazy update. To alleviate the above issue, an alternative is to use a concurrent heap [10] that supports parallel operations. Unfortunately, it is inefficient in the context of ANN search. This is because the number of points stored in the heap is few in our problem. As a result, the depth of the heap is low. This means that we have to sequentially update almost all vertices in the heap when we insert or delete points.

Instead of dynamically maintaining the candidate set and current top k result with two priority queues, we use a *lazy update* strategy with two fixed-length arrays N and T with lengths l_n and l_t, respectively. We say that

- A vertex is a **visiting** vertex regarding q if its distance to

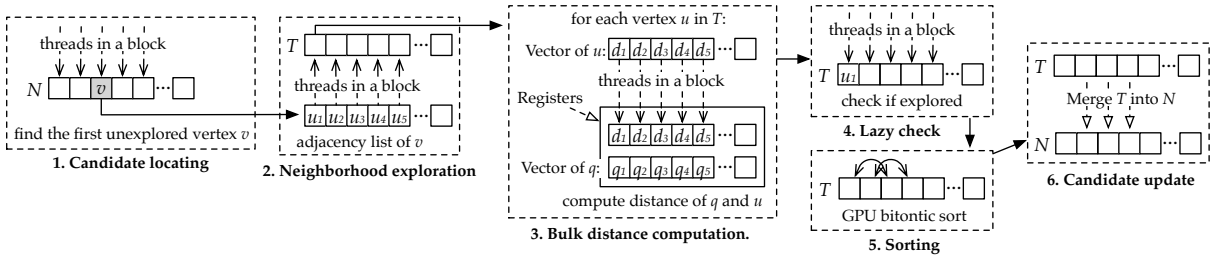


Fig. 3: A diagram of search algorithm GANNS

q will be calculated.

○ A vertex is an **exploring** vertex regarding q if the distances of its neighbors to q will be calculated.

N keeps the top k results and potential exploring vertices at the same time. In each iteration, a vertex in N is picked as the exploring vertex; T stores visiting vertices that are outgoing neighbors of the exploring vertex.

The selection of an exploring vertex can be based on warp-level primitives. N and T can be updated by GPU sorting and merging algorithms, i.e., the maintenance of the candidate set and current top k result can be efficiently paralleled.

Visited vertices mark. During search, a vertex might be accessed multiple times when its neighbors are explored. Thus, it is necessary to mark the visited vertices to reduce the redundant computation. In CPU-based proximity search, a hash table can be used for efficiently checking if a vertex has been visited (Line 16 of Algorithm 1). In this way, re-computation of the visited vertices can be avoided. In [5], two possible alternatives are discussed for this purpose: opening addressing hash table and bloom filter, followed by visited deletion optimization. The open addressing hash table with a single thread is used in the implementation of SONG because the overhead of synchronization does not pay-off. As an alternative, one may wonder if the well-known bitmap hashing can be applied since it can be easily paralleled without any synchronization cost. Unfortunately, this is not efficient on the GPU because of the high latency of the random memory accesses involved in the warp threads and the limited on-chip memory [11].

Lazy check. Based on the above observation and the fact that the distance computation in the context of GPU computation is very efficient, we do not check if a neighbor of v has been visited. Consequently, we can avoid the use of a hash table at the cost of some redundant distance computations. Note that we still assess if a vertex has been explored by checking the array N to avoid the propagation of this redundant distance re-computation.

B. GPU-based Proximity Graph Search

We first show the data structures used in the search algorithm and corresponding memory space allocation. The search algorithm is presented next.

Data Structures and Memory Allocation. Data structures in our search algorithm, as well as corresponding GPU memory allocation are as follows. (1) The proximity graph G, high-dimensional points (i.e., features), and high-dimensional queries, are kept in the GPU global memory. (2) N and T are allocated in the shared memory. (3) The coordinate values (i.e., feature) of each point, including the query q, will be assigned to the registers.

Search Algorithm. Note that we allocate a thread block for one query. Since ANN search of each query points is independent, the *inter-block level parallelism* is immediate by utilizing multiple thread blocks. Next, we illustrate our GPU-based ANN search algorithm in one thread block.

Initially, T is empty, and N includes the start vertex v_s. We use $N[v]$ (resp. $T[v]$) to denote the array element associated with the vertex v in N (resp. T), and $N[v]$.explored (resp. $T[v]$.explored) to indicate if v has not been explored. Then, search starts up. Each iteration of the search is illustrated in Figure 3. It consists of 6 phases as follows.

(1) *Candidate locating.* When each iteration starts, the first vertices v with $N[v]$.explored = false in N will be identified. In particular, threads access the flag $explored$ of vertices in parallel. Warp-level primitives *__ballot_sync* and *__ffs* are used to aggregate these flags to reveal the first unexplored vertex. If these vertices are all explored, subsequent vertices are processed in the same way. The search will be terminated if all vertices in N are already explored.

(2) *Neighborhood exploration.* Let $\{u_i\}$ denote outgoing neighbors of v. They are loaded into T by collaboration of threads in the thread block, and we set $T[u_i]$.explored as false. Then, v is marked as explored.

(3) *Bulk distance computation.* Distances between vertices in T and q are computed one by one. For each vertex in T, its corresponding d-dimensional vector (point) is transferred from the GPU global memory to the registers of n_t threads. Each thread is responsible for the computation of its corresponding sub-vector, and the partial results are aggregated by the warp-level primitive *__shfl_down_sync*.

(4) *Lazy check.* As discussed in Section III-A, there might be redundant distance computation because we do not check whether a vertex has been visited when it is inserted to T. Nevertheless, we will exam if vertices in T have been explored before they are merged into N to prevent the propagation of redundant computation. Given that vertices in N are sorted by their distances w.r.t q, we perform a parallel binary search for vertices in T. $T[v]$.explored is set to true if v is already in N because we do not need to consider v for neighborhood exploration again.

(5) *Sorting.* The well-known GPU algorithm bitonic sort [12] is employed to sort vertices in T based on their distances w.r.t q and the flag $explored$, and ties are broken by vertex ID.

(6) *Candidate update.* Since vertices in T and N have been ordered, we apply the bitonic sorting-based merge [4] algorithm to retrieve l_n closest vertices among $T \cup N$, and keep

results in N for further processing. Note that it is possible that v has been explored in N, then discarded from N. In this scenario, v will not appear in N again since the distance of the l_n-th vertex of N will keep decreasing during the search. In this way, we can avoid exploring the neighbors of a vertex multiple times though its distance to q might be re-computed.

Example 2: Given $k = 4$, the proximity graph G_1 and the query point q in Figure 1, our search algorithm returns k-nearest neighbors of q. Similarly, we assume that the entry point is v_1. In the beginning, it is loaded into N, and its flag *explored* is initialized as false.

In iteration 1, the first point not yet explored in N is the point v_1. Therefore, its neighbors v_2, v_3, v_5, v_7 and v_8 are loaded into T, and the flag *explored* of v_1 in N is set to true. Then, distances between vertices in T and q are computed. During lazy check, no point has a duplication in N. As a result, none of the points are set as explored. After sorting, the order of vertices in T becomes v_8, v_7, v_2, v_5 and v_3. Last, we update N by merging it with T. In iteration 2, v_8 is recognized as the next explored point and its neighbors are loaded, computed, checked, sorted and merged. Subsequently, the path $v_{10} \rightarrow v_{12} \rightarrow v_9$ is traversed. In iteration 5, the only unexplored point in N, v_9 is chosen as the next explored point. However, its neighbors make N unchanged. Meanwhile, points in N are all explored. Hence, the process terminates, returning v_{12}, v_9, v_8 and v_{10}.

As shown, though our search algorithm has the same search path as Algorithm 1 and SONG, neighbors of the exploring point in each iteration are considered in a batch rather than one by one. This increases parallelism and reduces the cost of maintaining data structures. Moreover, compared with Algorithm 1 and SONG, our search algorithm consumes less memory because it aborts the maintenance of visited points.

Remarks. To justify the practicality of solving ANN search on the GPU, we also investigate the impact of data transfer between CPU and GPU on total time cost. Specifically, compared with the time of querying on the GPU, the time of data transfer between CPU and GPU is negligible. This is because (1) data transfer occurs only at the beginning and end of search, and the size of data is minor compared with bandwidth. For instance, the size of results is around 1MB when there are 2000 queries in one batch and k is set to 100. However, the bandwidth of data transfer (PCI Express 3.0×16) is around 10GB/s; (2) CUDA provides a stream mechanism that supports asynchronous processing of kernel computation and data transfer. That is to say, data transfer can be overlapped with querying on the GPU even when several batches of points need to be processed on the GPU.

C. Analysis

Memory Usage. Memory usage is of importance in GPU-based algorithms. The proposed search algorithm makes much effort to (1) avoid additional buffer such as auxiliary arrays; (2) reduce the usage of shared memory of one thread block to enhance potential parallelism, i.e. the sizes of N and T are profitable. For instance, l_n usually takes 32, 64 or 128 while l_t, is set to d_{max} which is 32 by default; and (3) utilize registers. As reported in [13], the register is the largest SRAM structure on the chip, usually around 256KB per SM. Instead of ignoring this important memory structure, we load the vector of q and points into registers for the distance computation.

Time Complexity. Given the number n_t of threads in a thread block, the complexity of phases (1) and (2) is $O(\frac{l_n}{n_t})$ and $O(\frac{l_t}{n_t})$, respectively; for phase (3), the complexity is $O(l_t \times \frac{n_d}{n_t})$ where n_d is the dimensionality of the points; for phase (4), the complexity is $O(\log l_n \times \frac{l_t}{n_t})$; for phase (5), the complexity is $O(\log^2 l_t \times \frac{l_t}{n_t})$; and for phase (6), the complexity is $O(\log l_n \times \frac{l_n}{n_t})$. Putting them together, the overall cost is $O(\log l_n \times \frac{l_t+l_n}{n_t} + l_t \times \frac{n_d}{n_t} + \log^2 l_t \times \frac{l_t}{n_t})$ time for each iteration of the search. As a reference, the time complexity of three phases of SONG is $O(l_t)$, $O(l_t \times \frac{n_d}{n_t})$ and $O(l_t \times \log l_n)$, respectively. Theoretically, the speedup of both bulk distance computation and data structure operation is linear to the number of threads n_t within a thread block.

IV. Proximity Graph Construction

We first present the motivation of our GPU-based NSW graph construction algorithm (Section IV-A). We next describe the algorithm (Section IV-B) and give an analysis (Section IV-C). Then, we briefly show the techniques for the construction of two other popular proximity graphs: HNSW and KNN graphs(Section IV-D).

A. Motivation

Intuitively, the improvement of the search performance by SONG and our search algorithm can immediately accelerate NSW graph construction. However, it is nontrivial due to the sequential insertions during construction.

Straightforward methods. Generally, we have two straightforward GPU implementations for NSW graph construction as follows.

Sequential method. We can strictly follow the *sequential processing* of the points $\{v_1, v_2, \ldots\}$. Specifically, for a new point v_i, we conduct ANN search on the current NSW graph G, put its k nearest neighbors obtained from ANN search $\{u_1, \ldots, u_k\}$ as the adjacency list of v_i, and update the adjacency list of u_i with v_i as well. Then v_{i+1} will be processed on the new graph G', including edges created by the insertion of v_i.

Parallel method. We can simply design a parallel method by *parallel processing* of the points $\{v_i, v_{i+1}, \ldots\}$. In particular, ANN search on the current NSW graph G is performed in parallel for each point in the batch, and results are aggregated to update edges together.

Remarks. (1) The sequential method is inefficient because of the waste of the GPU computing bandwidth, i.e., there is no inter-block level parallelism. (2) The parallel method suffers from the quality of graphs. This is because all other points in the same batch are ignored during the graph construction for each point v_i, e.g., they will not appear in its adjacency list.

The pitfalls of the above two methods prompt us to design a new NSW graph construction algorithm such that we can fully exploit the GPU computing bandwidth without sacrificing the quality of the constructed graphs. In a nutshell, we use the divide-and-conquer strategy. As shown in Figure 5, we partition the points P into a set of disjoint groups $\{P_0, \ldots, P_t\}$

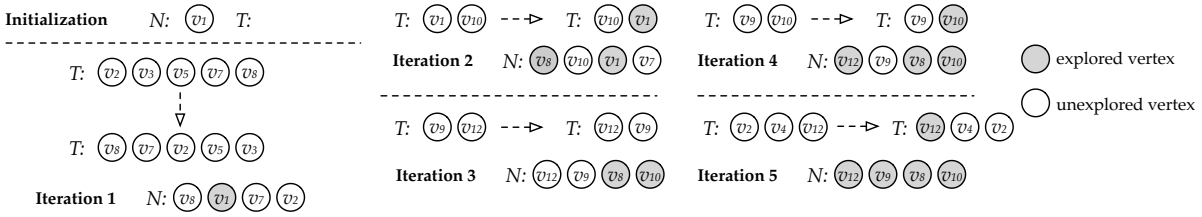


Fig. 4: An example of search algorithm GANNS

of the same size. In the first phase, we process each group P_i by one thread block and build a *local* NSW graph G_i by sequentially processing the points. Then the local graphs will be successively merged to the first local graph G_0. We carefully design the implementation to ensure that (1) all operations in the graph construction are friendly to GPU; (2) both inter-block level parallelism and intra-block level parallelism are achieved; and (3) the quality of the resulting graphs is the same as the NSW graph constructed by sequential insertions.

B. GPU-based NSW Graph Construction

We first demonstrate data structures as well as their locations in the memory hierarchy. Notice that the search process is also included in the construction process, and we only introduce new data structures here.

Data Structures and Memory Allocation. (1) The proximity graph G is pre-allocated in the global memory to store neighborhood information of all points. (2) Graph G' is also pre-allocated in the global memory. It is used to store intermediate neighborhood information for the optimization of our GPU solution. (3) An edge list E is located in the global memory, and is used to record nearest neighbors for inserted points. (4) An array I is used for the processing of the edge list E, which is allocated in the global memory.

Construction Algorithm. Without loss of generality, we assume that all points are assigned a unique ID, representing the order in which they are inserted. Let $v.N$ denote the adjacency list of the vertex (point) v in the NSW graph. for each vertex $u \in v.N$, we say the edge $v \rightarrow u$ is a "forward edge" if u.ID < v.ID. Otherwise, $v \rightarrow u$ is a "backward edge".

Generally, a forward edge $v \rightarrow u$ is generated in an aggressive way when v issues a KNN search in NSW graph construction, and u is included in the search result. A backward edge $v \rightarrow u$ is created in a passive way when u issues the kNN search, and v is one of the search results.

Algorithm 2 presents the pseudo-code of the NSW graph construction. The term **parallel do across blocks** indicates the *inter-block level parallelism*, i.e., the task is parallel processed by multiple thread blocks. We first partition points in P into $t+1$ disjoint groups P_0, P_1, ..., P_t (Line 1). The following construction algorithm consists of two phases: *local graphs construction* and *local graphs merge*.

(1) *Local graph construction.* (Lines 2-8) We assign each group P_i to a thread block for local graph construction (Line 2). The computation of each local graph is independent so the inter-block level parallelism can be immediately achieved. These points in the group P_i will be sequentially

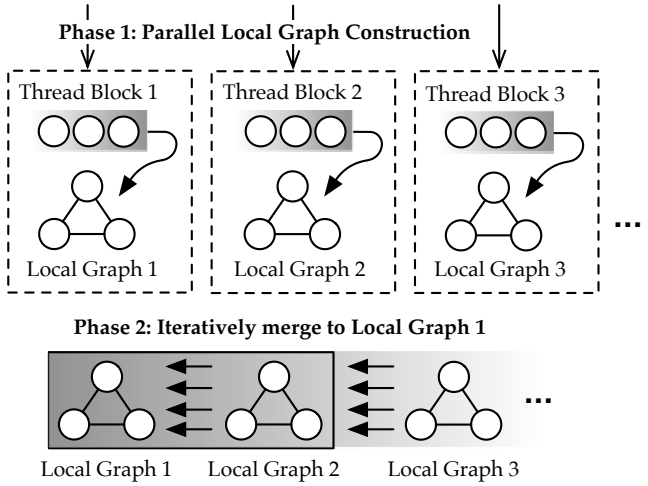


Fig. 5: Our Strategy

inserted into the local graph G_i using the following two steps (Lines 4-8).

Step 1. For each inserted vertex v_{ij} in P_i, we search d_{min} nearest neighbors on G_i. The search results of each new point v_{ij} are stored in their corresponding adjacency lists in both G and G', denoted by $v_{ij}.N$ and $v_{ij}.N'$ respectively. Note that $v_{ij}.N'$ uses memory space allocated for G'.

Step 2. For each point u_{ij} in the adjacency list of v_{ij}, its adjacency list in G is updated. Specifically, we insert the vertex v_{ij} into the adjacency list of u_{ij}. We then locate the position by binary search and write the vertex id of v_{ij} and the distance between u_{ij} and v_{ij}. Note that the adjacency list of each vertex is an array with fixed size d_{max} where elements are ordered by distance. The last element is discarded if the list is already full.

(2) *Local graphs merge.* (Lines 9-22) After constructing local graphs, the last t graphs are sequentially merged into the first local graph G_0. It requires t iterations, and there are three steps in each iteration. Assume we are in the i-th iteration.

Step 1. For each vertex v_{ij} in G_i, it is processed by one thread block. The computations of these vertices are independent of each other in this step, hence the inter-block level parallelism is immediate.

First, d_{min} nearest neighbors of v_{ij} is retrieved against the graph G_0 which includes points in $P_0 \dots P_{i-1}$ (Line 12), and these nearest neighbors will merge with $v_{ij}.N'$ in G' to form new $v_{ij}.N$ (Line 14). Recall that $v_{ij}.N'$ stores nearest neighbors of v_{ij} among points in P_i which have smaller ID than v_{ij}. Consequently, the d_{min} nearest neighbors of v_{ij} among points accessed before v_{ij} are kept in $v_{ij}.N$, i.e., the forward edges of v_{ij} are now readily available.

Next, we update the backward edges. Note that the update

of backward edges for points u_{ij} in $\bigcup_{v_{ij} \in P_i} v_{ij}.N$ is a bit tricky because the vertex u_{ij} might be included as one of the nearest neighbors in several adjacency lists, i.e., it could appear across multiple thread blocks. Hence, it might lead to inconsistent results if we do not have any concurrency control among blocks. Therefore, we put these backward edges into an edge list E for subsequent processing (Line 17).

Step 2. Here, we aim to organize these backward edges in compressed sparse raw (CSR) format ((Line 19)). (1) We employ bitonic sorting to organize edges in E by the IDs of the starting vertices (i.e., u_{ij}), with the ties broken by the distances. (2) $I[i]$ is set to 1 if the i-th edge in E is the first edge of a particular starting vertex; otherwise, $I[i] = 0$. (3) The prefix sum of I is computed, and we update I such that $I[i]$ is the beginning position of i-th starting vertex in E.

Step 3. Now, we update the backward edges of starting vertices in E. We assign each starting vertex to one thread block. Assume the vertex u_{ij} is assigned to the i-th thread block. (1) We use $I[i]$ and $I[i+1]-1$ to obtain the corresponding edges of u_{ij} in E, and load these edges from the global memory to the shared memory. (2) We load the current adjacency list of u_{ij} in G_0 from the global memory to the shared memory. (3) These two adjacency lists are merged, and we use the first d_{max} elements as the adjacency list of u_{ij}.

Consequently, we update G_0 by merging G_i, and G_0 is returned after all local graphs are processed.

Remarks. Though we propose Algorithm 2 in the context of GPU, it is essentially independent of hardware substrate. That is to say, it can also be applied to other system settings that have multiple working units such as multi-core CPU systems and distributed systems. In these system settings, each working unit can be individually responsible for the construction of one local graph and the search of nearest neighbors of one point in the merged local graph in each iteration during the merging of local graphs.

Algorithm 2: GPU-based NSW Graph Construction

Input: A point set P, the minimum degree d_{min} in G and the maximum degree d_{max} in G
Output: A proximity graph $G = (V, E)$

```
1  partition P into disjoint sets P_0, P_1, ..., P_t;
2  parallel do across blocks
3      G_i := ∅;
4      for each v_ij in P_i do
5          v_ij.N, v_ij.N' := Search(G_i, v_ij, d_min);
6          for each u_ij in v_ij.N do
7              parallel do within block
8                  u_ij.N := u_ij.N ∪ {v_ij};
9  for i = 1 to t do
10     E := ∅; I := ∅;
11     parallel do across blocks
           // v_ij ∈ G_i
12         v_ij.N := Search(G_0, v_ij, d_min);
13         parallel do within block
14             v_ij.N := v_ij.N ∪ v_ij.N';
15         for each u_ij ∈ v_ij.N do
16             parallel do within block
17                 E := E ∪ (u_ij → v_ij, δ(u_ij, v_ij));
18     parallel do across blocks
19         E, I := GatherScatter(E);
20     parallel do across blocks
21         parallel do within block
22             u_ij.N := u_ij.N ∪ {E[I[i]], E[I[i]+1], ...};
23 return G_0;
```

C. Analysis

Quality of graphs. Given exact nearest neighbors, Algorithm 2 can generate the NSW graph, which is the same as that constructed by sequential insertions.

Proof sketch. Suppose the NSW graph is constructed by the sequential insertion, the outgoing edges of a vertex v consists of forward edges and backward edges, denoted by E_f^v and E_b^v, respectively. Specifically, the ending vertices in E_f^v are nearest neighbors ahead of v, and v is one of the nearest neighbors of each ending point u in E_b^v which is behind v.

We can prove that if nearest neighbors set η_v among points ahead of v can be found for each point v, we can construct NSW graphs. It is self-explanatory that E_f^v has been found. Consider any u in E_b^v, v must appear in η_u. Otherwise, u will not appear in E_b^v because we can find the nearest neighbors set η for each point including u, we can update E_b^v after u searches its nearest neighbors.

In Algorithm 2, $v.N$ initially records the nearest neighbors of v within $P_0 \cup \ldots P_{i-1}$, and $v.N'$ keeps the nearest neighbors of v from P_i which are ranked before v. Therefore, we have $v.N \cup v.N' = \eta_v$ for each vertex v. □

Memory Usage. Global memory and shared memory are used in other parts besides search. (1) The size of G is $O(n_p \times d_{max})$ where n_p is the number of points in P and the user-defined parameter d_{max} represents the maximum degree in G. (2) The size of G' is $O(n_p \times d_{min})$ where d_{min} denotes the number of returned nearest neighbors while searching. (3) The size of E is $O(n_b \times d_{min})$ where n_b denotes the number of points in a batch. (4) The size of I is also $O(n_b \times d_{min})$. (5) For each thread block, the usage of shared memory is $O(d_{max})$ for merging.

Time complexity. We analyze the time complexity of steps excluding search. Let n_t and n_b be the number of threads in one thread block and the number of thread blocks respectively. Note that n_t may vary in different kernels. (1) The complexity of Step 2 in the local graph construction phase is $O(d_{max} \times \frac{t}{n_b} \times \frac{d_{min}}{n_t})$ where t represents the number of local graphs. Specifically, there is the factor d_{max} because we must place neighbors in new positions. (2) For Step 2 in the local graph merge phase, the complexity is $O(\log^2 |E| \times \frac{|E|}{n_b \times n_t})$ and $O(\log |E| \times \frac{|E|}{n_b \times n_t})$ for bitonic sorting and prefix sum computation, respectively. (3) For Step 3 in the local graph merge phase, the complexity is $O(\log d_{max} \times \frac{|P_i|}{n_b} \times \frac{d_{max}}{n_t})$ where $|P_i|$ represents the number of points in the local graph P_i. This implies that, theoretically, the speedup of the NSW graph construction is linear to the number of threads within a thread block (n_t) as well as the number of thread blocks (n_b).

D. Extension

Though the proposed construction algorithm (Algorithm 2) is tailored for the NSW graph, with minor modifications it can also be used to construct other proximity graphs. Next, we show how to extend the above NSW graph construction algorithm to support two popular proximity graphs: the HNSW and KNN graphs.

HNSW graph. HNSW graph [14] is a hierarchical organization of the NSW graph where each layer is an NSW graph for a subset of the point set P that is randomly selected. The higher the level of a layer is, the fewer points it contains. The bottom layer includes all points in P.

The adaptation of Algorithm 2 to support the construction of the HNSW graph is natural. A straightforward method is to construct an NSW graph for the subset of points on each layer respectively. However, when the construction of each NSW graph is independent of each other, search of nearest neighbors of points during construction can not benefit from the hierarchical structure that can shorten search path. To avoid this drawback, we determine to construct HNSW graph level-by-level.

However, there is still one problem even though we sequentially construct NSW graph on each layer: the random selection of subsets on each layer is such that we can not directly access adjacency lists of points on some layers according to their vertex IDs because some points might not be on this layer. A possible method is to maintain an index for each layer that records the position of each vertex it has. However, it assigns an index for each vertex on each layer, which consumes additional memory. A better method might be that (1) we shuffle IDs of vertices and record the mapping. (2) during construction, these vertices are inserted into each layer in turn, i.e., vertices with smaller IDs can reach higher levels. This means that we can access the adjacency list of one vertex according to its vertex ID because points that have a smaller ID than this must be also on this layer. (3) vertex IDs are recovered based on the stored mapping after construction. Consequently, we only need to maintain a shuffled order while keeping the random selection of points on each layer.

KNN Graph. The difference between a KNN graph and an NSW graph is that the former needs to maintain global nearest neighbors for each point. That is to say, when a set of new points P' are added into a KNN graph constructed on the point set P, we need to find not only the nearest neighbors on P for each point in P', but also the nearest neighbors on P' for each point in P. Then, for each point in P (resp. P'), returned nearest neighbors on P' (resp. P) are merged into its original adjacency list on P (resp. P').

A straightforward adaptation of Algorithm 2 is that (1) during local graph construction, we maintain nearest neighbors in the current local graph for each point. (2) while merging local graphs, we search the nearest neighbors for vertices in both G_0 and G_i. (3) we update the corresponding adjacency lists in both graphs, comparable to the update of the backward edges. The main drawback of this solution is that multiple NN searches will be invoked for each point.

To overcome it, we turn to an iterative method proposed by [9]. This method follows the property that the neighbors of neighbors are likely to be neighbors. Initially, this method randomly generates the adjacency list of each point. Then, it is iteratively improved. In each iteration, each pair of neighbors $\{u_1, u_2\}$ of each vertex v will form two new edges $u_1 \rightarrow u_2$ and $u_2 \rightarrow u_1$ for u_1 and u_2, respectively. Then, these newly generated edges are used to update the adjacency lists of vertices. This process terminates when the adjacency lists of all points cease to change. We can see that the key to this framework is distance computation between each pair of neighbors of each vertex and the update of adjacency lists. They can be implemented naturally as shown in Figure 3 (BulkDistanceComputation) and Algorithm 2 (Step 3 of local graphs merge phase).

TABLE I: Real-life Datasets

Dataset	Type	Dimension	Vertices	Metric
SIFT1M [15]	Image	128	1M	Euclidean
GIST [15]	Image	960	1M	Euclidean
NYTimes [16]	Text	256	0.29M	Cosine Similarity
GloVe200 [17]	Text	200	1.18M	Cosine Similarity
UQ_V [18]	Video	256	3.03M	Euclidean
MSong [19]	Audio	420	0.99M	Euclidean
Notre [20]	Image	128	0.33M	Euclidean
UKBench [21]	Image	128	1.1M	Euclidean
DEEP [22]	Image	96	8M	Euclidean
SIFT10M [15]	Image	32	10M	Euclidean

V. Experiments

We conduct experiments to evaluate (1) the efficiency of our GPU-accelerated nearest neighbors search algorithm GANNS; (2) the impact of the number k of returned nearest neighbors and the number e of explored vertices; (3) the efficiency of our GPU-based graph construction framework GGraphCon; and (4) scalability.

Datasets. We use ten real-world datasets as summarized in Table I. In particular, (1) NYTimes and GloVe200 are heavily skewed while the dimension of GIST is relatively high. This makes them hard, compared to other datasets. (2) SIFT10M consists of ten million vectors randomly selected from SIFT1B [15]. Here, we only use the first 32 dimensions of each vector. Similarly, DEEP comprises eight millions points randomly selected from the original dataset DEEP1B[1].

Algorithms. We implement the following algorithms in C++ and CUDA C. (1) GANNS is shown in Section III-B for GPU-based nearest neighbors search. In our implementation, we set l_n to the power of 2 for ease of GPU memory management. Here, we introduce another parameter e to achieve a fine-grained trade-off between efficiency and accuracy, where we only considere the first e vertices in N for exploration. (2) GSerial and GNaiveParallel (Section IV-A) are straightforward GPU-based graph construction algorithms that use SONG for searching. (3) The GPU-based graph construction framework GGraphCon is shown in Section IV-B. It includes (3.1) GGraphCon$_{\text{GANNS}}$ that uses GANNS for searching, (3.2) GGraphCon$_{\text{SONG}}$ that uses SONG for searching.

Baselines. (1) For ANN search, we compare with the state-of-the-art graph-based ANN search algorithm SONG [5]. The code is from original authors[2]. (2) For graph construction, apart from straightforward methods GSerial and GNaiveParallel, we also compare with algorithms GraphCon$_{\text{NSW}}$ and

[1]https://research.yandex.com/datasets/biganns

[2]https://github.com/sunbelbd/song

GraphCon$_{\text{HNSW}}$ on CPU. GraphCon$_{\text{NSW}}$ establishes navigable small world graphs with degree limitation, which is implemented by [5][2]. GraphCon$_{\text{HNSW}}$ constructs hierarchical navigable small world graphs [14], which is publicly available[3].

Evaluation. (1) For nearest neighbors search, we measured the accuracy by recall, which is defined as the ratio of correct nearest neighbors to returned neighbors. The accuracy is evaluated over the test set. Specifically, each test set comprises 2000 vertices. Search time is denoted by "Queries Per Second" which represents the average number of completed queries per second. (2) For graph construction, we report running time and measure the quality of graphs by recall, which can be achieved given the same search algorithm.

Configuration. The experiments are conducted on a Linux Server powered by a 26-core Intel Xeon Gold 6238R 2.2GHz CPU and NVIDIA Quadro P5000 GPU with 2560 cores and 16GB memory. We compile all codes with NVCC (CUDA 10.0), GCC 5.4.0 and the -O3 flag.

A. Search Performance

Efficiency. Fixing $k = 10$, we evaluate the efficiency of GANNS and SONG. We test the number of queries per second when varying the recall. As shown in Figure 6, we have the following observations. (1) The ranges of recall achieved by GANNS and SONG are the same on all datasets. This shows that the parallelization scheme of GANNS does not change the quality of results. (2) High recall values (larger than 0.95) can be achieved on all datasets except GloVe200. For instance, the highest recall on the hard dataset GIST is 0.97 while the highest recall values on some datasets like UKBench and UQ_V are close to 1. This validates the superior search accuracy of the graph-based search methods. (3) GANNS consistently outperforms SONG on all datasets. When the recall is not very high (around 0.8), GANNS is 1.5-5 times faster than SONG. (a) On some datasets, GANNS can achieve about 5x speedup. For instance, the throughput of GANNS is 458.5k queries per second on SIFT1M when the recall is 0.795. By contrast, the throughput of SONG is 88.5k queries per second while achieving the same recall. (b) On the hard datasets with moderate dimension NYTimes and GloVe200, the speedup is reduced to around 2. (c) For the hard dataset GIST with a high dimension, GANNS achieves around 1.5x speedup.

A time breakdown of GANNS and SONG is presented in Figure 7 when the recall is around 0.8. It demonstrates that compared with SONG, the cost of data maintenance of GANNS decreases. This is consistent with Figure 6. In particular, on hard datasets, the cost of data maintenance of GANNS is little higher than that on other datasets. This is because more vertices as candidates need to be maintained on hard datasets.

Effect of Parameters. We evaluate the impact of the number k of returned vertices and the number e of explored vertices. Varying k. Fixing $recall = 0.8$, we vary k from 1 to 100. We report the results on SIFT1M and GIST here. As shown in Figure 8, the speedup remains relatively stable as k increases. On SIFT1M, the largest speedup is 5.3, and the smallest speedup is about 5; On GIST, the largest speedup is 2, and the smallest speedup is 1.5.

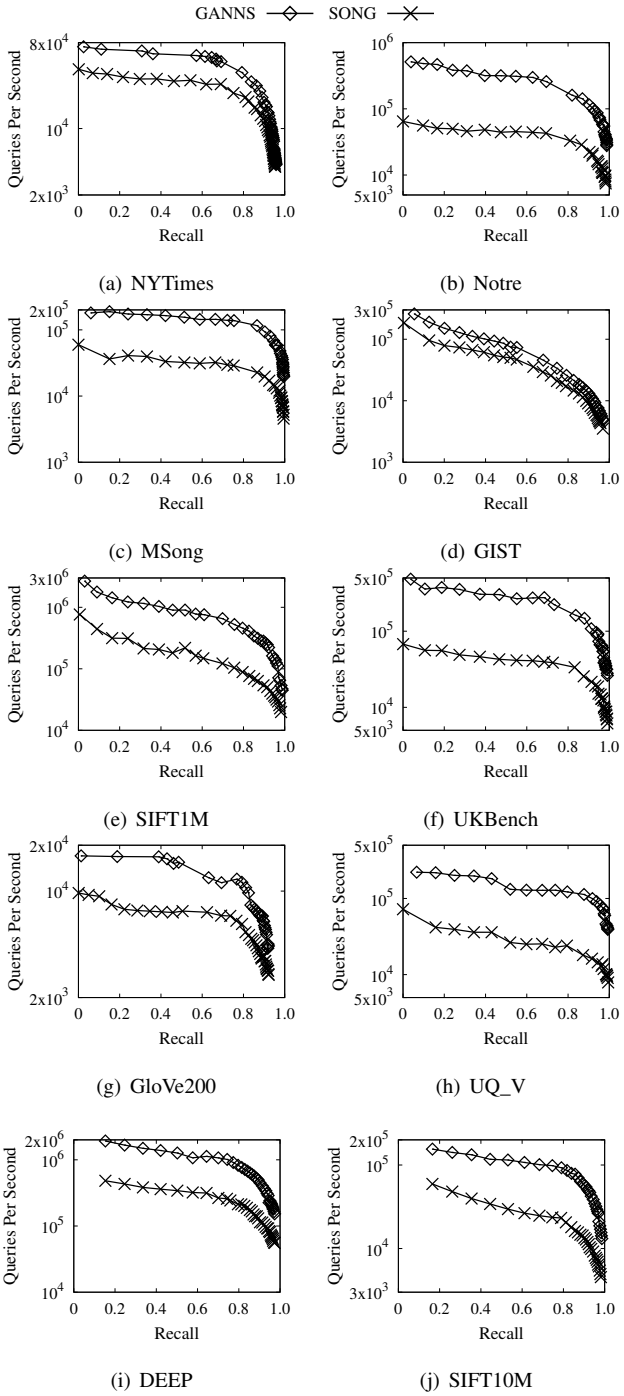


Fig. 6: Throughput on different recall

Varying n_d. Fixing $k = 10$, we vary n_d from 960 to 60 on dataset GIST to demonstrate the effect of dimension on query performance of GANNS and SONG. Here, we report throughput when $recall = 0.8$. As shown in Figure 9, query performance of both algorithms improve when the dimensionality n_d decreases mainly because the cost of bulk distance computation becomes low for low dimensional data. It is shown that the performance gap between GANNS and SONG becomes more significant when the dimensionality is

[3]https://github.com/nmslib/nmslib

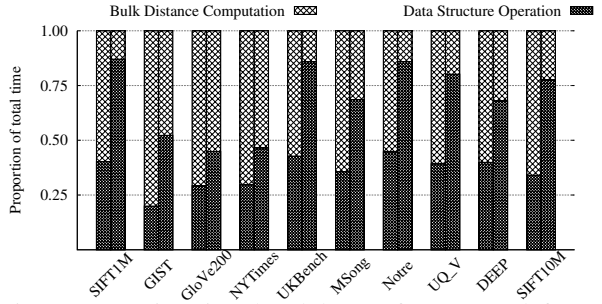
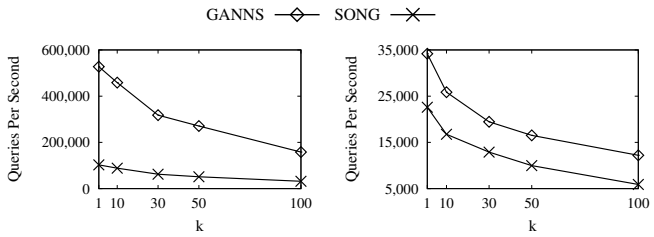


Fig. 7: Execution time breakdown of GANNS (left) and SONG (right)



(a) SIFT1M (b) GIST

Fig. 8: Throughput on different parameter k

low, e.g., the speedup enlarges from 1.5x to 6x as dimension n_d decreases from 960 to 60. This is because the percentage of the data structure operation cost becomes larger when the dimensionality decreases, and SONG cannot take advantages of the parallelism of GPU for this cost within a thread block.

Parallelism. To evaluate parallelism of GANNS and SONG, we vary the number of threads in each thread block (i.e., query) from 4 to 32, and report the average distance computation time and data structure operation time of two algorithms, SONG and GANNS, on SIFT1M dataset in Figure 10. Regarding the distance computation, two algorithms take similar time and both enjoy a significant speedup. For instance, they spend around 100 ms and 24 ms when the number of thread is 4 and 32, respectively. Regarding the data structure operation, GANNS still demonstrates a good speedup, e.g., takes around 71 ms and 12.3 ms when the number of threads is 4 and 32 respectively, while SONG cannot take advantage of the parallelism within the thread block.

B. Construction Performance

Parallelization scheme. We evaluate our GPU-based graph construction. The upper bound d_{max} and the lower bound d_{min} are fixed as 32 and 16, respectively, unless otherwise stated. CPU algorithms run in a single thread.

Efficiency. We compare the running time of GGraphCon$_{\text{GANNS}}$, GGraphCon$_{\text{SONG}}$, GNaiveParallel and GSerial. As shown in Figure 11, (1) Given the same search kernel, the straightforward method GNaiveParallel only slightly outperforms our proposed scheme GGraphCon$_{\text{SONG}}$. This shows that the overhead of our scheme is minor, which is derived from considering links between vertices in the same batch. (2) The search process dominates graph construction. By using GANNS, GGraphCon$_{\text{GANNS}}$ achieves the apparent speedup over GGraphCon$_{\text{SONG}}$. On some datasets, GGraphCon$_{\text{GANNS}}$ has a 2x-3.3x speedup. For instance, the running time of GGraphCon$_{\text{GANNS}}$ on UQ_V is only 43s, while GGraphCon$_{\text{SONG}}$ needs to spend 145s. On hard datasets, the speedup is between 1.4-2.2. (3) We also conduct experiments to evaluate GSerial. The result tells us that its running time is very long, e.g. $3810s$ on SIFT1M (not shown).

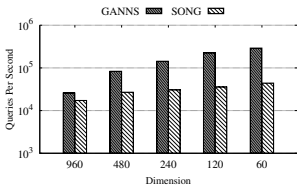


Fig. 9: The effect of n_d

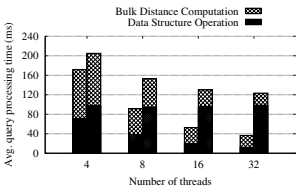


Fig. 10: The effect of n_t

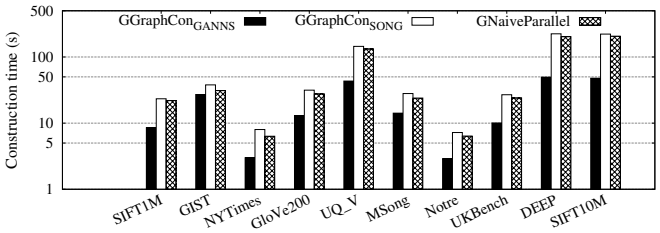
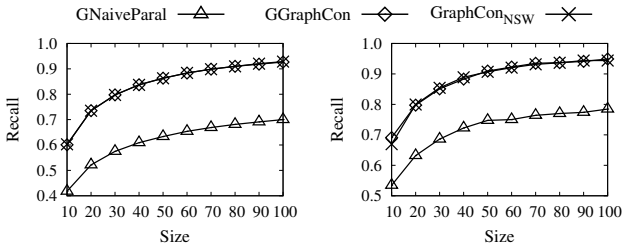


Fig. 11: Graph construction time

TABLE II: Comparison with CPU algorithm (NSW)

Dataset	GraphCon$_{\text{NSW}}$	GGC$_{\text{GANNS}}$	GGC$_{\text{SONG}}$
SIFT1M	355s	8.5s (41.8x)	23s (15.4x)
GIST	1335s	27s (49.4x)	38s (35.1x)
NYTimes	249s	3s (83x)	8s (31.1x)
GloVe200	531s	13s (41x)	31.5s (16.9x)
UQ_V	1720s	43s (40x)	145s (11.9x)
MSong	620s	14s (44.3x)	28s (22.1x)
Notre	87s	3s (29x)	7s (12.4x)
UKBench	375s	10s (37.5x)	27s (13.9x)
DEEP	4135s	49.5s (83.5x)	224s (18.5x)
SIFT10M	2986s	48s (62x)	222s (13.5x)

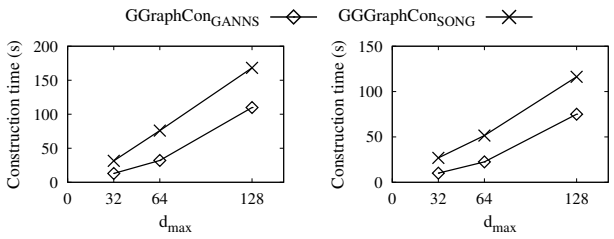


(a) SIFT1M (b) UKBench

Fig. 12: Graph quality

For reference, we also compare with GraphCon$_{\text{NSW}}$. The results are reported in Table II (GGC denotes GGraphCon). GGraphCon$_{\text{GANNS}}$ and GGraphCon$_{\text{SONG}}$ both have significant speedup. In particular, GGraphCon$_{\text{GANNS}}$ could achieve 40-50x speedup on most datasets.

Quality. We evaluate the quality of proximity graphs from the straightforward parallelization scheme GNaiveParallel, our proposed scheme GGraphCon and the serial CPU method GraphCon$_{\text{NSW}}$. We report the results on SIFT1M and UKBench. We form the following observations from Figure 12. (1) The recall achieved on the graph by GNaiveParallel is much lower than that of other graphs. For instance, on SIFT1M, the recall of GNaiveParallel is only 0.7 even though e increases to 100, whereas recalls of the other two al-



(a) GloVe200 (b) UKBench

Fig. 13: Construction time by varying d_{max}

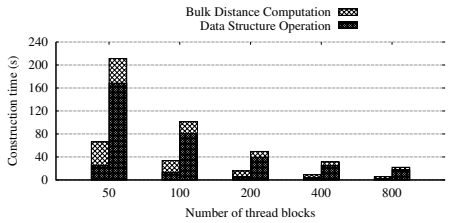


Fig. 14: Performance scaling of GGC$_{GANNS}$ (left) and GGC$_{SONG}$ (right) while varying the number of thread blocks

gorithms could be 0.92. This demonstrates that the graph quality constructed by GNaiveParallel is poor. (2) The graph quality produced by GGraphCon is almost the same as that by GraphCon$_{NSW}$ among all values of e on two datasets. This shows that the quality of the graph from GGraphCon is as good as the graph from the serial algorithm. The result is consistent with our analysis in Section IV-C.

Scalability. We evaluate the scalability of our proposed scheme GGraphCon. We vary the bound d_{max} from 32 to 128. Correspondingly, d_{min} varies from 16 to 64. We report the results on GloVe200 and UKBench here. The results are shown in Figure 13. The running time gently increase when increasing d_{max}. The increase of running times of GGraphCon$_{GANNS}$ and GGraphCon$_{SONG}$ are both almost linear.

Parallelism. We evaluate parallelism of GGraphCon$_{GANNS}$ and GGraphCon$_{SONG}$ on the dataset SIFT1M. Setting d_{max} and d_{min} to 32 and 16 respectively, we vary the number of thread blocks from 50 to 800 for the construction of NSW graphs where the number of threads per thread block is set to the default value (32). Here, we report the graph construction time of two algorithms: GGraphCon$_{GANNS}$ and GGraphCon$_{SONG}$. It includes distance computation time and data structure operation time. As shown in Figure 14, Though we still cannot achieve the theoretical maximal speedup, it is reported that around 10x-13x speedup can be achieved for the distance computation and data structure operations of two algorithms when the number of thread blocks grows from 50 to 800 (i.e., 16x speedup theoretically). Note that although SONG cannot utilize the parallelism of the threads within the thread blocks, it can immediately take advantage of multiple thread blocks since the search processes are independent to each other.

Extension. We show that our proposed scheme can be extended to construct graphs with other formats. Here, we implement the construction of HNSW where d_{max} and d_{min} are 32 and 16 respectively. The running time is shown in

TABLE III: Comparison with CPU algorithm (HNSW)

Dataset	GraphCon$_{HNSW}$	GGC$_{GANNS}$	GGC$_{SONG}$
SIFT1M	313s	11s (28.5x)	37s (8.5x)
GIST	2138s	48s (44.5x)	68s (31.4x)
NYTimes	324s	4s (81x)	12s (27x)
GloVe200	5255s	17s (309x)	52s (101x)
UQ_V	1737s	47s (37x)	215s (8x)
MSong	823s	20s (41x)	48s (17.1x)
Notre	85s	3.2s (26.6x)	11s (7.7x)
UKBench	342s	11s (31.1x)	38s (9x)
DEEP	4550s	70.2s (65x)	308s (15x)
SIFT10M	2823s	82s (34.4x)	338s (8.4x)

Table III (GGC denotes GGraphCon). The result is consistent with Table II.

VI. Related Works

CPU-based ANN methods. For a good trade-off between accuracy and efficiency, many works have been proposed from multiple perspectives. As suggested in [2], existing techniques can be roughly classified into three categories: *LSH-based*, *space partitioning-based* and *graph-based* methods.

LSH-based methods. These approaches (e.g., [23], [24], [25], [26], [27]) are typically based on locality-sensitive hashing (LSH), which maps high-dimensional points to low-dimensional points via a set of appropriately chosen random projection functions.

Space partitioning-based methods. These works (e.g., [28], [29], [30], [31], [32], [33], [34]) decompose the original vector space into the cartesian product of lower dimensional subspaces and quantize each subspace separately.

Graph-based methods. These methods receive much attentions due to their superior search performance. Many efforts have been devoted to constructing a variety of proximity graphs such as kNN graphs, small world graphs, relative graphs as well as their variants (e.g., k-DR [35], DPG [2], PANNG [36], EFANNA [37], FANNG [38], NSG [39] and Vamana [40]).

Recently, benchmark and survey studies (e.g., [2], [41]) have also been presented for researchers and practitioners.

GPU-based ANN methods. Due to the promising performance of GPU on general-purpose data-parallel applications, some researchers try to accelerate ANN problem by using GPU.

Quantization-based methods. Methods of this kind including Faiss [4], Robustiq [42] and PQT [43] have achieved significant speedups on GPU due to their high parallelism. In particular, Faiss[4] extends the idea of IVFADC with a fast k-selection implementation on GPUs and successfully achieves billion-scale similarity search.

Graph-based methods. SONG [5] proposes an efficient search algorithm on proximity graphs, outperforming Faiss in terms of search time and accuracy.

VII. Conclusion

We propose a GPU-based algorithm GANNS to accelerate the ANN search on proximity graphs by re-designing the classical CPU-based search algorithm and using GPU-friendly data structures. Given that there is no existing study on GPU-based proximity graph construction, we fill this gap by developing novel algorithms and ensuring the quality of the resulting proximity graph. Extensive experiments are conducted to show the outstanding performance of our techniques on both ANN search and proximity graph construction.

REFERENCES

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[2] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2019.

[3] "Proxima," "https://www.alibabacloud.com/blog/proxima-a-vector-retrieval-engine-independently-developed-by-alibaba-damo-academy_597699", 2021.

[4] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[5] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1033–1044.

[6] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[7] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

[8] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 163–170.

[9] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 577–586.

[10] Y. Chen, F. Hua, C. Huang, J. Bierema, C. Zhang, and E. Z. Zhang, "Accelerating concurrent heap on gpus," *arXiv preprint arXiv:1906.06504*, 2019.

[11] B. Lessley and H. Childs, "Data-parallel hashing techniques for gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 237–250, 2019.

[12] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[13] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 420–432.

[14] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[15] "SIFT and GIST," http://corpus-texmex.irisa.fr/, 2010.

[16] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[17] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[18] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," in *Proceedings of the 19th ACM international conference on Multimedia*, 2011, pp. 423–432.

[19] "MSong," http://www.ifs.tuwien.ac.at/mir/msd/download.html.

[20] "Notre," http://phototour.cs.washington.edu/datasets/.

[21] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. Ieee, 2006, pp. 2161–2168.

[22] A. B. Yandex and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *IEEE Conference on Computer Vision & Pattern Recognition*, 2016, pp. 2055–2063.

[23] L. Gong, H. Wang, M. Ogihara, and J. Xu, "idec: indexable distance estimating codes for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, pp. 1483–1497, 2020.

[24] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 1–12, 2015.

[25] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *Proceedings of the VLDB Endowment*, 2014.

[26] W. Liu, C. Mu, S. Kumar, and S.-F. Chang, "Discrete graph hashing," in *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*, 2014, pp. 3419–3427.

[27] Z. Jin, C. Li, Y. Lin, and D. Cai, "Density sensitive hashing," *IEEE transactions on cybernetics*, vol. 44, no. 8, pp. 1362–1371, 2013.

[28] D. Xu, I. W. Tsang, and Y. Zhang, "Online product quantization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 11, pp. 2185–2198, 2018.

[29] J. Martinez, J. Clement, H. H. Hoos, and J. J. Little, "Revisiting additive quantization," in *European Conference on Computer Vision*. Springer, 2016, pp. 137–153.

[30] J. Wang, J. Wang, J. Song, X.-S. Xu, H. T. Shen, and S. Li, "Optimized cartesian k-means," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 1, pp. 180–192, 2014.

[31] A. Babenko and V. Lempitsky, "Additive quantization for extreme vector compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 931–938.

[32] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2321–2328.

[33] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 4, pp. 744–755, 2013.

[34] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[35] K. Aoyama, K. Saito, H. Sawada, and N. Ueda, "Fast approximate similarity search based on degree-reduced neighborhood graphs," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1055–1063.

[36] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, vol. 98, 1998, pp. 194–205.

[37] C. Fu and D. Cai, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv preprint arXiv:1609.07228*, 2016.

[38] B. Harwood and T. Drummond, "Fanng: Fast approximate nearest neighbour graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5713–5722.

[39] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proceedings of the VLDB Endowment*, vol. 12, no. 5.

[40] S. J. Subramanya, F. Devvrit, H. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, pp. 13 771–13 781, 2019.

[41] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[42] W. Chen, J. Chen, F. Zou, Y.-F. Li, P. Lu, and W. Zhao, "Robustiq: A robust ann search method for billion-scale similarity search on gpus," in *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, 2019, pp. 132–140.

[43] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. Lensch, "Efficient large-scale approximate nearest neighbor search on the gpu," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2027–2035.