SONG: Approximate Nearest Neighbor Search on GPU

Weijie Zhao, Shulong Tan and Ping Li
Cognitive Computing Lab
Baidu Research USA
1195 Bordeaux Dr. Sunnyvale, CA 94089
10900 NE 8th St. Bellevue, WA 98004
{weijiezhao,shulongtan,liping11}@baidu.com

***Abstract*—Approximate nearest neighbor (ANN) searching is a fundamental problem in computer science with numerous applications in (e.g.,) machine learning and data mining. Recent studies show that graph-based ANN methods often outperform other types of ANN algorithms. For typical graph-based methods, the searching algorithm is executed iteratively and the execution dependency prohibits GPU adaptations. In this paper, we present a novel framework that decouples the searching on graph algorithm into 3 stages, in order to parallel the performance-crucial distance computation. Furthermore, to obtain better parallelism on GPU, we propose novel ANN-specific optimization methods that eliminate dynamic GPU memory allocations and trade computations for less GPU memory consumption. The proposed system is empirically compared against HNSW–the state-of-the-art ANN method on CPU–and Faiss–the popular GPU-accelerated ANN platform–on 6 datasets. The results confirm the effectiveness: SONG has around 50-180x speedup compared with single-thread HNSW, while it substantially outperforms Faiss.**

I. Introduction

Nearest neighbor (NN) searching is a fundamental problem since the early days of computer science [16], [17], with numerous practical applications in many fields such as machine learning, computer vision, data mining, information retrieval, etc. The challenge of the NN task is to find the nearest neighbor without scanning all data points in the repository of data. In recent years, methods for approximate near neighbor (ANN) search become popular, because many applications [2], [10], [12], [30], [59], [61], [66] only require finding a close enough neighbor instead of the exact nearest solution.

ANN searching methods. In the ANN searching paradigm, each query is compared with a subset of data points instead of the entire dataset. To obtain the subset, a vareity of index structures have been proposed, including probabilistic hashing [7], [9], [24], [31], [39], [41], [56], quantization [23], [32], [33], [63], ball tree or KD tree variants [8], [11], [52], and graph-based searching [19], [26], [46], [47], [64], etc.

Graph-based ANN methods. Recently, graph-based methods draw great attentions. In the literature, extensive experiments show that graph-based methods outperform other types of ANN methods in common metrics [19], [47], [50]. Typically, these methods build a graph index referred as *Proximity Graph*. Vertices of proximity graph represent the points in the dataset. Edges in the graph illustrate neighborhood relationships between the connecting nodes. The neighborhood relationship is defined on various constraints to make graphs applicable for the ANN problem. For example, some graph constraints like Delaunay Graphs [5] and Monotonic Search Networks [13] guarantee that there exists a path with monotonic decreasing distance to the query point from any starting vertex. GNNS [26], IEH [35], EFANNA [18], NSG [19], NSW [46] and HNSW [47] approximate the Delaunay Graph or Relative Neighborhood Graph [60], to reduce the proximity graph construction complexity to subquadratic time. Those approximations make graph-based ANN methods applicable to massive data and become popular tools in industry practice.

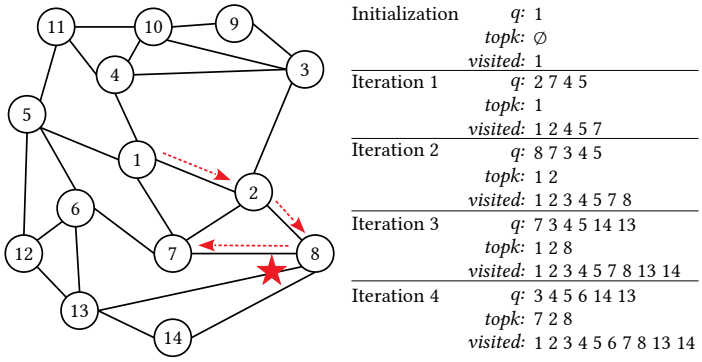
Algorithm 1 Searching algorithm on the proximity graph.

Input: Graph index $G(V, E)$; a query point p;
Number of output candidates K
Output: Top K candidates for each query *topk*

1. Initialize a binary min-heap as priority queue q and a hash set *visited* with the default starting point. Construct an empty binary max-heap as priority queue *topk*
2. **while** $q \neq \emptyset$ **do**
3. (*now_dist*,*now_idx*) $\leftarrow$ q.pop_min()
4. **if** *topk*.size=K **and** *topk*.peek_max()<*now_dist* **then**
5. **break**
6. **else**
7. *topk*.push_heap((*now_dist*,*now_idx*))
8. **end if**
9. **for each** (*now_idx*,v) $\in E$ **do**
10. **if** *visited*.exist(v) $\neq$ **true then**
11. $d \leftarrow$ dist(p,v)
12. *visited*.insert(v)
13. q.push_heap((d,v))
14. **end if**
15. **end for**
16. **end while**
17. **return** *topk*

Search on graph. Although these methods have diverged constraints on building the graph indices, most of the graph-based methods [18], [19], [26], [35], [46], [47], [57], [58] share the similar heuristic searching algorithm (Algorithm 1) — a variant of A^* heuristic search [27]. Its workflow is similar to Breadth-First Search (BFS), except the queue in the BFS is replaced with a priority queue q. The priority queue orders vertices ascendingly by the distance to the query point p. The searching process starts from a default starting point. Then it

extracts a vertex from the priority queue q (line 3), updates the top-K candidates (line 4-8), and inserts the neighbors of the vertex into q for future exploration (line 13). The main searching loop (line 2-16) stops when the extracted vertex from the priority queue is worse than the searched top-K result candidates (line 4-5). Abstractly, the algorithm greedily follows a path to reach the nearest neighbor of the query point. Fig. 1 illustrates an example for Algorithm 1.



Initialization	*q:*	1
	topk:	∅
	visited:	1
Iteration 1	*q:*	2 7 4 5
	topk:	1
	visited:	1 2 4 5 7
Iteration 2	*q:*	8 7 3 4 5
	topk:	1 2
	visited:	1 2 3 4 5 7 8
Iteration 3	*q:*	7 3 4 5 14 13
	topk:	1 2 8
	visited:	1 2 3 4 5 7 8 13 14
Iteration 4	*q:*	3 4 5 6 14 13
	topk:	7 2 8
	visited:	1 2 3 4 5 6 7 8 13 14

Fig. 1: An example to illustrate the searching algorithm on the proximity graph (Algorithm 1). The star represents the query point p. In this example, we target to find the top $K = 3$ nearest neighbors of p. Vertex 1 is the default starting point. The searching path $1 \rightarrow 2 \rightarrow 8 \rightarrow 7$ is highlighted by the dashed arrows. The right part shows the states of two priority queues–q and *topk*–and the hash table *visited* in each iteration.

Parallelism. The searching procedure is executed iteratively—each iteration relies on the progress of the previous one. The execution dependency prohibits any embarrassingly-parallel solutions for this algorithm. The conventional parallel solution for graph-based ANN is to execute different queries concurrently on multi-core CPUs. There are few studies discussing the GPU adaptation for graph-based methods. However, without considering the architecture of GPUs, a query-parallel solution does not scale on GPU platforms. In the meanwhile, there are an increasing number of researches [21], [22], [36], [61] investigating the ANN system on GPUs. For example, Faiss [36] is a popular GPU-accelerated quantization-based ANN system. As the quantization method has low instruction dependencies, Faiss can fully parallel the execution and shows superior performance over dense data compared with other CPU-based methods. On the other hand, despite that graph-based methods provide better results on CPUs, complex graph structures and high execution dependencies of graph searching make the GPU adaptation a challenging task. In this paper, we investigate the GPU adaptation problem and propose a combination of optimizations, for graph-based ANN methods.

Summary of Contributions:

- We introduce SONG (acronym of "Search ON Graph")—an ANN search system that performs graph-based ANN searching on GPUs. To the best of our knowledge, SONG is the first graph-based ANN system designed for GPUs.
- We develop a novel framework that decouples the searching on graph algorithm into 3 stages: candidates locating, bulk distance computation and data structures maintenance to parallel the performance-crucial distance computation. Unlike GPU Breadth-First search methods, our proposed framework is optimized specifically for the graph-based ANN searching by addressing the heavy high-dimensional distance computation of ANN problems.
- We propose a combination of data structures and optimizations for GPU ANN graph searching. We employ open addressing hash table, Bloom filter and Cuckoo filter to serve as a high-performance hash table. We adopt a series of optimizations: bounded priority queue, selected insertion and visited deletion, to eliminate dynamic memory allocations and trade computations for less memory consumption.
- We present a parameterized graph searching algorithm–multi-query and multi-step probing–in the candidate locating stage that enables fine-tuning for the performance.
- We evaluate experimentally the proposed system and compare it against HNSW (the state-of-the-art ANN method on CPU platform) and Faiss (the popular GPU ANN system) on 6 datasets. The results confirm that SONG has around 50-180x speedup compared with single-thread HNSW, while it substantially outperforms GPU-version Faiss.

II. GPU Architecture

In this section, we introduce the GPU architecture that is the foundation for the GPU-specific optimizations of SONG.

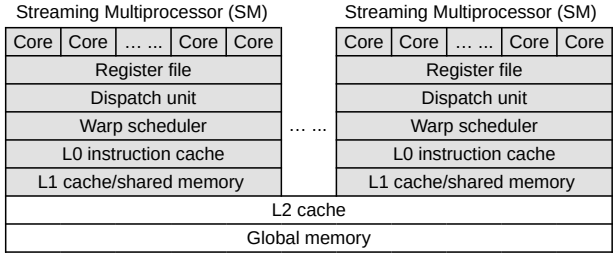


Fig. 2: GPU architecture.

Hardware architecture and memory hierarchy. Fig. 2 illustrates the GPU hardware architecture and memory hierarchy. A GPU contains multiple streaming multiprocessors (SM). Each SM has dozens of cores, two dispatch units, a warp scheduler, a register file and a configurable L1 cache and shared memory. All SMs share an L2 cache and a global memory. The cores in SM are targeted at a limited subset of computation instructions. The dispatch units and the warp scheduler in the SM issue instructions and schedule execution of the cores. Register file is an array of processor registers that can be directly accessed by the cores. L0 instruction cache is introduced recently in the NVIDIA Volta architecture [34]. L0 instruction cache is employed to provide higher efficiency than the instruction buffers used in prior GPUs. Unlike CPU platforms, the GPU L1 cache is configurable—we can allocate a portion of the L1 cache as shared memory. Shared memory can be manipulated explicitly and is accessible to all cores

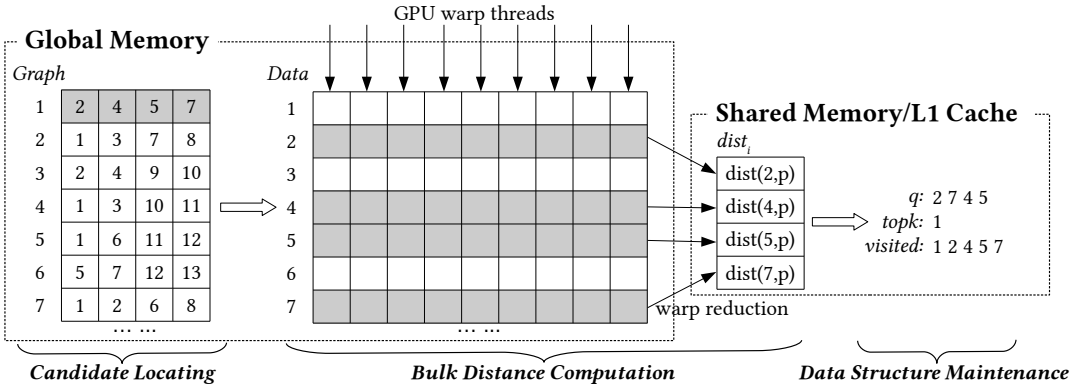


Fig. 3: An example to deonstrate the GPU warp proximity graph searching workflow. The graph (adjacency list) on the left inherits from the example in Fig. 1. The shadowed cells represents the memory access pattern of the first iteration in Fig. 1.

inside the same SM. All SMs can access the global memory and the L2 cache acts as a cache for the global memory I/Os.

Global memory can be accessed from all the cores across SMs. Although the largest in size, global memory has the lowest bandwidth and the highest access latency. Shared memory is a high-bandwidth and low-latency memory. It is also shared by cores within an SM. The two levels of cache L1 and L2 are leveraged to improve the global memory access latency. When global memory addresses are requested, aligned continuous addresses are merged into a single memory transaction. Cores have to access consecutive global memory addresses to access data efficiently from global memory.

GPU programming model. In the CUDA programming model, workloads are issued to the GPU in the form of a function—the function is referred as a *kernel*. Similar to the definition in CPU platforms, a logical instance of the kernel is called a *thread*. Threads are grouped in *blocks*. Physically, all the threads within a block locate on the same SM. Threads can access the various units of the deep memory hierarchy explicitly in the CUDA code. To manage thousands of concurrent threads running on different parts of the data, the SM employs SIMT (single-instruction, multiple-thread) or SIMD (Single Instruction Multiple Data) parallelism by grouping consecutive threads of a block into a *warp*. All the threads in a warp have to perform the same instruction at a time. If-else branch in the code blocks the execution of some threads in the same warp. Threads working on different code branches are executed sequentially.

III. SONG System Overview

In this section, we identify the performance bottlenecks of Algorithm 1, and present the main modules of SONG that optimize the algorithm from a high-level view.

Distance computation. The profiling result of the searching algorithm on CPU platforms shows that more than 95% time is spent on distance computations (line 11) for most common ANN benchmark datasets [47]—the performance is dominated by distance computations. Compared with the time-consuming high-dimensional distance computation, other operations like priority queue maintenance and memory accesses take less than 5% execution time. In contrast, GPUs are capable to efficiently parallel the high-dimensional distance computation with its large number of cores. The major 95% cost on CPUs can be significantly improved on GPUs. Although the major cost is substantially reduced, other unnoticeable problems on CPUs arise on GPUs. As opposed to the CPUs, GPU memory accesses become the dominant factor.

Memory access. The CPU graph searching algorithm introduces dynamic memory allocations and occupies a large working memory in the searching. It is not scalable on GPU because there are much fewer available memory budgets per thread on GPU. In order to efficiently parallel the searching, we have to limit the memory consumption of the priority queues and the hash table in Algorithm 1.

High-level approach. Here we introduce the high-level approach for solving one query in the GPU graph searching algorithm. Since the major time of non-parallel approximate nearest neighbor searching on high-dimensional datasets is spent on computing distances [47], our GPU graph searching algorithm focuses on exploiting GPU to accelerate the distance computation. In order to fully utilize the GPU computing bandwidth, we have to compute distances in a batch instead of a large number of independent pair-wise distance computation function calls. Therefore, we extract the distance computation part from the searching workflow by decoupling the graph searching algorithm into 3 stages: candidates locating, bulk distance computation and data structures updating.

Fig. 3 illustrates the decoupled workflow. The graph (adjacency list) on the left inherits from the example in Fig. 1. The i^{th} row stores the adjacent vertices to vertex i. For example, the first row in the graph shows that 1 connects to 2, 4, 5 and 7. As Algorithm 1, in each iteration, we first extract a vertex from the priority queue q to begin the search. Consider we are searching on vertex 1 in this example. The candidate locating stage fetches the vertices $(2, 4, 5, 7)$ that connect to 1 from the graph. Then, the bulk distance computation stage reads the vector values of these vertices from the data matrix and employs GPU warp reduction to compute the distances to the query point p. The toy example in Fig. 1 is only for illustrating the idea of the searching algorithm. In the

real ANN applications, the dimensionality goes from a few hundred to a thousand—we can fully utilize the GPU threads in the bulk distance computation stage. After that, the data structure maintenance stage uses the distances to update the priority queues and the hash table for the next iteration.

This 3-stage workflow decouples the distance computation and queue maintenance dependency (Algorithm 1 line 9-15) into a batch processing pipeline. Therefore, we can accelerate the distance computation through GPUs. However, these 3 stages are still executed sequentially. We optimize each stage for the GPU architecture to achieve better performance.

IV. Data Structure Maintenance

A. Data Structures on GPU

The data structures are designed specifically for the graph searching task on GPU. We introduce the fixed degree graph for graph storage and the hash table alternatives.

Fixed degree graph storage. The proximity graph used in graph-based ANN searching has the following property: the degree of each vertex is bounded by a fixed constant K [47]. Storing the graph as an adjacency list requires us to keep an index in the GPU memory that tracks the offset of each vertex. Index look-up is inefficient since it requires an additional memory operation to load the index. Storing the graph as a fixed-degree adjacency list eliminates the additional index look-up in adjacency lists. We can locate a vertex by multiplying its index with the fixed size of a vertex, because each vertex takes the same fixed amount of memory. The fixed degree graph is stored in the GPU global memory (Fig. 3).

Query. While the proximity graph and the dataset can be persisted on the GPU global memory before the queries come, the query points have to be transferred from the host CPU main memory to the GPU global memory at runtime. During the searching process of a query, the queried point is frequently accessed to compute the distance to other vertices in the graph. We explicitly copy the query point into the fast on-chip shared memory to reduce GPU global memory reading.

Concurrency control. Although there are a few studies introducing the lock-free concurrent priority queue and hash table, those data structures are designed as a substitute for host-side CPU data structures—not for the threads in a CUDA kernel. Furthermore, only dozens of insertion are required in an iteration. The sequential operations outperform the complicated concurrent data structures. Therefore, the priority queues and the hash tables are maintained by one single thread in our proposed system.

Memory access patterns. GPU shared memory can be accessed by all threads in the same block in low latency. As we discussed above, p is copied to the shared memory because p is frequently accessed in distance computations. In addition, *candidate* and *dist* are allocated as fixed-length array in the shared memory. The lengths of *candidate* and *dist* are at most the fixed degree of the graph index. Allocating them as a fixed-length array is more efficient than dynamic allocation. Putting them into the shared memory eliminates additional communication cost in the warp reduction. Because the priority queues and hash tables are maintained by only one thread, *q*, *topk* and *visited* are allocated as the local memory of its host thread—no other threads access these data structures. The graph index and the data are kept in the global memory.

B. Hash Table Alternatives

We discuss the design of the hash table (*visited*) in the searching algorithm and its alternatives in this section.

Open addressing hash table. One of the most popular hash table implementations is separate chaining, e.g., the hash table (unordered_set) implementation of GNU GCC uses separate chaining to resolve the hash collision. However, the chaining solution requires dynamic memory allocation, such as linked list, destroy the GPU computation performance. We employ another hash collision method–open addressing [43]–in our GPU graph searching implementation. Open addressing probes through alternate locations in the array until either the target record is found, or an unused array slot is found. We allocate a fixed-length array in the shared memory for each thread block. The length is proportional to the searching parameter K and can be pre-computed. The linear probing step can be paralleled in the warp level—all threads in a warp probe the memory and locate the insertion/deletion location by a warp reduction. We limit the parallel probing in the warp level because the linear probing does not need to probe many locations. Probing one memory location for each thread in a warp (32 threads) is usually sufficient to find a valid insertion/deletion location.

Bloom filter. We observe that the visit test does not have to be answered precisely—false positives can be tolerated while false negatives may incur heavy computation overhead. False positives (*visited* tells us an element is visited but actually not) prevent us to search some unvisited vertices—we may lose some search accuracy when the skipped vertices are the only path to the queried nearest neighbor. On the other hand, false negatives (*visited* tells us a vertex is not visited but actually we have searched this vertex before) lead us to search the visited vertices again—it introduces significant overhead when the revisited vertices are searched and inserted into the priority queue multiple times. The data integrity is also influenced, because additional checks are required to avoid inserting one vertex multiple times in the priority queue. To occupy less memory, we employ a probabilistic data structure–Bloom filter [49]–to replace the hash table. Bloom filter takes a constant small memory footprint that can be implemented efficiently in GPU settings. In addition, Bloom filter guarantees no false negatives and the false negatives are theoretically constrained. Bloom filter works as a counterpart of the hash table by slightly trades-off the accuracy introduced by the false positives. The false positive rate is related to the number of elements inserted into the Bloom filter. In practical, a Bloom filter with around 300 32-bit integers has less than 1% false positives when inserting $1,000$ vertices—the accuracy loss introduced by the Bloom filter is ignorable. Suppose the key of a data point is also a 32-bit integer, the Bloom filter method takes at least 3x less memory than the hash table.

Initialization	*q:* 1
	topk: ∅
	visited: 1
Iteration 1	*q:* 2 7 4
	topk: 1
	visited: 1 2 4 5 7
Iteration 2	*q:* 8 7 3
	topk: 1 2
	visited: 1 2 3 4 5 7 8
Iteration 3	*q:* 7 3 4
	topk: 1 2 8
	visited: 1 2 3 4 5 7 8 13 14
Iteration 4	*q:* 3 4 6
	topk: 7 2 8
	visited: 1 2 3 4 5 6 7 8 13 14

Bounded Priority Queue

(a)

Initialization	*q:* 1
	topk: ∅
	visited: 1
Iteration 1	*q:* 2 7 4
	topk: 1
	visited: 1 2 4 5 7
Iteration 2	*q:* 8 7 3
	topk: 1 2
	visited: 1 2 3 4 5 7 8
Iteration 3	*q:* 7 3 4
	topk: 1 2 8
	visited: 1 2 3 4 5 7 8
Iteration 4	*q:* 3 4
	topk: 7 2 8
	visited: 1 2 3 4 5 7 8

+ Selected Insertion

(b)

Initialization	*q:* 1
	topk: ∅
	visited: 1
Iteration 1	*q:* 2 7 4
	topk: 1
	visited: 1 2 4 7
Iteration 2	*q:* 8 7 3
	topk: 1 2
	visited: 1 2 3 7 8
Iteration 3	*q:* 7 3 4
	topk: 1 2 8
	visited: 1 2 3 4 7 8
Iteration 4	*q:* 3 4
	topk: 7 2 8
	visited: 2 3 4 7 8

+ Visited Deletion

(c)

Fig. 4: Examples for (a) bounded priority queue; (b) selected insertion; and (c) visited deletion. Each step is built on top of the previous optimization. The proximity graph referred in this example is in Fig. 1.

C. Bounded Priority Queue Optimization

The searching algorithm (Algorithm 1) maintains a priority queue that stores the current found top-k candidates for the query. It is efficient to implement the priority queue as a binary heap on CPU. However, straightforwardly transplanting the binary heap to the GPU is problematic. The binary heap implementation consumes unbounded memory—we keep adding vertices into the queue during the search—the size of q can grow much larger than K. Unbounded memory allocation is catastrophic to GPU performance. In order to utilize GPU effectively to store q, we have the following observation:

Observation 1: The searching on graph algorithm (Algorithm 1) only utilizes the first K elements in q.

Proof. After the size of q grows to $K+1$, denote the $(K+1)^{th}$ element in q as x. If we will not extract x from q in future iterations, x will not be included in the top-K result. Otherwise, we will extract x from q later. At that moment, *topk* has already been filled by K elements. Meanwhile, since there are at least K processed element better than the x, the condition in Algorithm 1 line 4 is satisfied—the algorithm exits and x is not included in the top-K result. □

Therefore, we can pop out the worse candidate from q when its size grows to $K+1$. Fig. 4(a) depicts the iterations of this optimization. Comparing with the original algorithm (Fig. 1), the bounded priority queue optimization eliminates the insertions of $5, 13, 14$ and bounds q within 3 elements.

We implement a GPU version of symmetric min-max heap [3] to act as the bounded priority queue. Insertion and popping the min/max element both take logarithmic time.

D. Selected Insertion Optimization

Since the data structure maintenance is dominated by memory bandwidth and latency, this stage can be improved when the data structures have smaller memory footprints. The memory consumption of both hash table and Bloom filter is proportional to the number of insertions. We propose the selected insertion optimization to reduce the insertions to *visited*—the memory consumption is decreased as a result.

After a vertex's distance to the query point is computed, the vertex is marked as visited in the original algorithm (Algorithm 1 line 12). We propose a selection before the insertion: the vertices that have larger distances than all the K elements in the *topk* are filtered out—a vertex is marked as visited and pushed into q *only* when it is among the current top-K closest vertices to the query point.

Fig. 4(b) illustrates an example when applying the selected insertion on top of the bounded priority queue optimization in (a). In the 3^{rd} iteration, vertex 13 and 14 are not marked as visited nor pushed into q if we apply the selected insertion optimization—because *topk* is fully filled with K candidates while 13 and 14 have larger distances than all the vertices in *topk*. In the 4^{th} iteration, 6 is not inserted into q nor *visited* because 4 is worse than all the candidates in *topk*. Since the filtered vertex is not marked as visited, we may compute its distance again when its neighbor is processed in the future iterations. Note that the top-K candidates (in the *topk* priority queue) are guaranteed to become closer and closer to the query point during iterations. Therefore, a filtered-out vertex is guaranteed to be filtered out again in future iterations. With this selected insertion optimization, the correctness and integrity of the searching algorithm are still preserved—no vertex is inserted multiple times in the hash tables or priority queues. The selected insertion method reduces insertions and GPU memory usages at the cost of computing some distance multiple times. In this example, the final *visited* size is 6—it stores 4 less elements than Fig. 4(a). The distance computation can be fully paralleled, while the hash table and priority queue maintenance are sequential. Thus, this computation-space trade-off is able to improve query performance.

E. Visited Deletion Optimization

We follow the idea of selected insertion and take one step further to save more GPU memory aggressively—in order to ensure the correctness of the searching algorithm, we only need to keep the *visited* as a hash table that shows whether a vertex is in the priority queues q and *topk*. Specifically, after a vertex is extracted from the priority q and processed, we can delete it from *visited* if the vertex does not update the *topk*. Also, when the *topk* is updated, the popped-out vertex can also be deleted from the *visited* hash table. The intuition is similar to the selected insertion optimization: the deleted vertices (logically re-marked as unvisited) have larger distances than the current top-K candidates—they will not update the top-K candidates nor be inserted into the priority queue q in future iterations. Fig. 4(c) shows an example. When applying the visited deletion optimization, the hash table *visited* is exactly the union of q (size at most K) and *topk* (size at most K). Therefore, the size of *visited* is bounded by $2K$.

The visited deletion optimization requires the deletion operation on *visited*. The deletion operation of hash table can be performed in constant time. While our hash table alternative–Bloom filter–does not support deletion. We choose Cuckoo filter [14] as the hash table probabilistic data structure alternative to validate the visited deletion optimization.

Comparing with the original algorithm in Fig. 1 and the running example in Figure4(c), we can observe that we utilize almost 50% less memory by applying our 3 optimizations. In real applications, the searching takes hundreds of iterations—it yields more memory savings than this toy example.

V. Candidate Locating

Basic candidate locating. In the candidate locating stage, one thread in the warp (say, thread 0) is responsible to extract the vertex id that is closest to the query point, and the thread adds the current unvisited neighbor vertices into a list *candidate*. In Figures 3, $2, 4, 5, 7$ are located and stored in *candidate* for the distance computation stage. Since this stage is simple and does not involve complicated computations, one thread can handle the task efficiently.

Multi-query in a warp. In the basic candidate locating method, other threads are idle when thread 0 extracting the vertex from the priority queue. We can process multiple queries in a warp to improve thread utilization. For example, consider we are processing 4 queries in a warp. We construct priority queues and hash tables for each query. 4 active threads (say, threads 0, 1, 2 and 3) extract the vertex id from their corresponding q. Although we have more active threads, the disadvantage is that we have to create a separate set of data structures (priority queues and hash tables) for each query processed in this warp. It is not clear to choose the best number of queries in a warp. We discuss this problem in the experimental evaluations (Section VIII-C).

Multi-step probing. The recent GPU BFS studies [44], [45] suggest a strategy that expands adjacent neighbors in parallel. In our graph searching problem, it corresponds to extract multiple vertices from q instead of only the first vertex. Multi-step probing fills more vertices into the candidate list. Unlike the general BFS problem, searching on the proximity graph usually goes along the direction to the query point in a small number of steps. Therefore, the neighbors of the current processing vertex are more likely to be the head of the priority queue. The multi-step probing may waste the probing memory accesses and distance computations on suboptimal candidates. Its effectiveness is evaluated in Section VIII-C.

VI. Bulk Distance Computation

We now address the issue with heady distance computations in graph-based ANN methods. The bulk distance computation stage takes the vertices in the *candidate* list as input, fetches the corresponding data from the dataset, computes their distances to the query point, and outputs the results into an array in the shared memory (Fig. 3). All threads in the block are involved in this stage–each thread takes charge of a subset of dimensions to compute a partial distance. Afterwards, thread 0 aggregates the partial distances of all warps into one value through a shfl_down warp reduction. The aggregated distance for the i^{th} element in *candidate* is stored in $dist_i$.

Instead of simply computing the distance for each candidate concurrently, the above parallel strategy is more cache-friendly. In the proposed parallel reduction, 32 threads are organized to access contiguous memory addresses. If we process the candidates concurrently, the memory access pattern of each thread is independent—more cache misses are generated.

This parallel reduction distance computation method can be applied to common popular ANN distance measures such as *p-norm* distance, Cosine similarity, and inner product.

VII. Out-of-GPU-Memory Datasets

In this section, we discuss the solution to tackle out-of-GPU-memory datasets. This problem is particularly emergent for storing high-dimensional data. Often the size of the graph index is much smaller—it is proportional to the *degree*$\times$*#data*, where *degree* is the degree limit of the graph index and *#data* is the number of data points in the dataset. Empirically, it is sufficient to use 16 for the *degree*—the graph index is under 1 GB for millions of data points. For example, the 16-degree graph index size of 8 million 784-dimensional data points takes 988 MB, while the dataset size is 24 GB. For using GPU to accelerate graph searching, we need to reduce the dataset size to store it in the GPU. Accordingly, we employ random hashing techniques that encode high-dimensional data into a bit vector. Then the hashed dataset may fit in GPU memory, and distances are computed on the low-dimensional bits.

1-bit random projection. Among numerous probabilistic hashing methods, in this paper we focus on introducing a popular method named "1-bit random projections" [9], [25], [40]. Formally, for two data vectors $u, v \in \mathbb{R}^d$, we generate a random vector $r \in \mathbb{R}^d$ with entries in iid standard normal. Then $Pr\left(sgn(< u, r >) = sgn(< v, r >)\right) = 1 - \frac{\theta(u,v)}{\pi}$, where $\theta(u, v)$ is the angle between u and v. If entries of r are sampled from iid cauchy instead of normal, then this collision probability is closely related to the χ^2 similarity [40]. With h independent random vectors, each data point is mapped into an h-bit vector. The hamming distance between the bit-vectors becomes an good estimate of the similarity in the original data (if h is not too small). In the implementation, h can be set as a multiple of 32 so that the bit vector can be stored as several 32-bit unsigned integers. This way, the memory footprint of a bit vector equals to the space of $h/32$ single-precision floating values. We investigate the hashing performance for out-of-GPU-memory dataset scenario in Section VIII-H.

Although they are not studied here, other techniques such as sharding are also applicable to the scalability challenge. For example, when multiple GPUs are considered, we can shard the data for each GPU, build a graph index for each shard, perform graph search on each GPU and merge the results.

VIII. Experiments

In this section, we provide a detailed investigation on 6 real datasets to analyze the effectiveness of the proposed system.

Implementation. We implement SONG as a `C++11` prototype. The code is compiled with g++-6.4.0 enabling the "O3" optimization. The GPU CUDA code is complied with nvcc from the CUDA 10.0 that enables "-Xptxas -O3" optimization. SONG loads pre-built graph index generated by NSW [46]. We choose NSW since it is a general and flexible proximity graph

construction algorithm. Other alternative graph construction algorithms are also adaptable to our acceleration framework.

Hardware System. We execute the experiments on a single node server. The server has one Intel Xeon Processor E5-2660 (64 bit)–8 cores 16 threads–and 128 GB of memory. Ubuntu 16.04.4 LTS 64-bit is the operating system. The GPU we use on the server is NVIDIA TESLA V100.

GPU Memory Hierarchy. The GPU L1 cache is configurable: we can allocate a portion of the L1 cache as shared memory. The L1 cache has lower latency and higher bandwidth than the GPU global memory. Meanwhile, the L1 cache capacity is limited: 96 KB per SM. We allocate shared memory from the L1 cache for the heaps that store the searching candidates and the top-k results, the working query point, and the bulk computed distances. These data structures are frequently accessed during the graph searching algorithm. Moreover, their sizes are bounded and fit in the L1 cache. The *visited* hash table is not bounded when we do not apply any proposed optimization. Since its size can grow beyond the L1 cache capacity, we can only put them in the global memory. With our proposed selected insertion and visited deletion optimization, the hash table size is bounded by the searching parameter K so that we can store it in the shared memory to accelerate the hash table probing and updating. The dataset and the graph index cannot fit in the L1 cache, thus, we store them in the global memory.

Compared algorithms. The compared algorithms are HNSW—the state-of-the-art ANN method on CPUs, and Faiss—the top GPU ANN system for large scale data. Other studies [4], [36] have shown that other types of algorithms such as tree-based, hashing-based approaches have inferior performance. We do not include them as competitors in this paper. We use the code of Faiss and HNSW from their GitHub repository. Especially, there are multiple implementations of HNSW, we choose the implementation from NMSLIB (the one with the best performance in ANN Benchmark[1]). To make a fair comparison, we vary parameters of HNSW and Faiss over a fine grid. Then their best results are considered.

Methodology. The index of each algorithm is pre-built. The index construction time is not included in the experiments. Especially, SONG does not generate its own index. Instead, we use the same graph index as NSW (similar to HNSW but no hierarchical structures) for SONG. We run HNSW with one single thread: since HNSW supports inter-query parallel, we can assume the performance of HNSW is *linearly scalable to the number of threads*. For Faiss and SONG, we execute the queries in one batch on one single GPU. Similar to HNSW, Faiss and SONG can also *linearly scale with multiple GPU cards*. Comparing with the single-threaded HNSW gives us a factor that shows how many CPU threads can be replaced with one GPU. The performance are evaluated by searching time (*Query Per Second*) and retrieval quality (*recall*).

Searching Time. We measure wall-clock time of each algorithm and present the number of answered Query Per Second (throughput) as the execution time measurement. We choose Query Per Second as the metric instead of the execution time of a query batch, because Query Per Second can be compared without normalizing query batch to the same size. All experiments are performed at least 3 times. We report the average value as the result.

Retrieval Quality. Recall is a widely-used retrieval quality measurement for ANN algorithms. Suppose the candidate point set returned by an algorithm is A, and the correct K nearest neighbor set of the query is B, then the recall is defined as: $Recall(A) = \frac{|A \cap B|}{|B|}$. A higher recall corresponds to a better approximation to the correct nearest neighbor result.

Dataset	Dim	#Data	#Query	Size in HDF5
NYTimes	256	289,761	10,000	301 MB
SIFT	128	1,000,000	10,000	501 MB
GloVe200	200	1,183,514	10,000	918 MB
UQ_V	256	3,295,525	10,000	3.2 GB
GIST	960	1,000,000	10,000	3.6 GB
MNIST8m	784	8,090,000	10,000	24 GB

TABLE I: Specifications of the datasets.

Data. We use 6 ANN benchmark datasets for experiments: NYTimes[2], SIFT[3], GloVe200[4], UQ_V[5], GIST[3] and MNIST8m[6]. The specification of the datasets are shown in Table I. The dimensions of our test datasets varies from 128 (SIFT) to 960 (GIST), while the number of data points ranges from 290,000 (NYTimes) to 8,090,000 (MNIST8m). MNIST8m is the largest dataset (24 GB) and NYTimes is the smallest (301 MB). The distribution of the datasets also diverges—NYTimes and GloVe200 are heavily skewed while SIFT, UQ_V, GIST and MNIST8m have less skewness. We use MNIST8m for the study on out-of-GPU-memory scenarios.[7].

A. ANN Search Performance Comparisons

ANN performance comparison results of SONG, Faiss and HNSW are shown in Fig. 5. Table II presents the detailed results of the speedup over Faiss.

	0.5	0.6	0.7	0.8	0.9	0.95
SIFT	5.9	5.6	6.6	7.8	6.8	N/A
GloVe200	14.0	10.7	N/A	N/A	N/A	N/A
NYTimes	20.2	N/A	N/A	N/A	N/A	N/A
GIST	4.8	6.2	7.7	N/A	N/A	N/A
UQ_V	16.4	13.7	14.2	14.2	N/A	N/A

TABLE II: Speedup over Faiss from the moderate recall (0.5) to the high recall (0.95) for top-10. The N/A means Faiss cannot reach the given recall. It is consistent with previous studies [4], [6], [19], [36], [61]. The top-1 speedup is omitted. It has a similar trend and slightly better speedup in general.

[1]https://github.com/erikbern/ann-benchmarks

[2]https://archive.ics.uci.edu/ml/datasets/bag+of+words

[3]http://corpus-texmex.irisa.fr/

[4]https://nlp.stanford.edu/projects/glove/

[5]http://staff.itee.uq.edu.au/shenht/UQ_VIDEO/

[6]https://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/multiclass.html

[7]The number of queries in the original GIST dataset is 1,000. In order to eliminate the difference of query batch size with other datasets, we duplicate the GIST queries 9 more times to scale the queries to 10,000

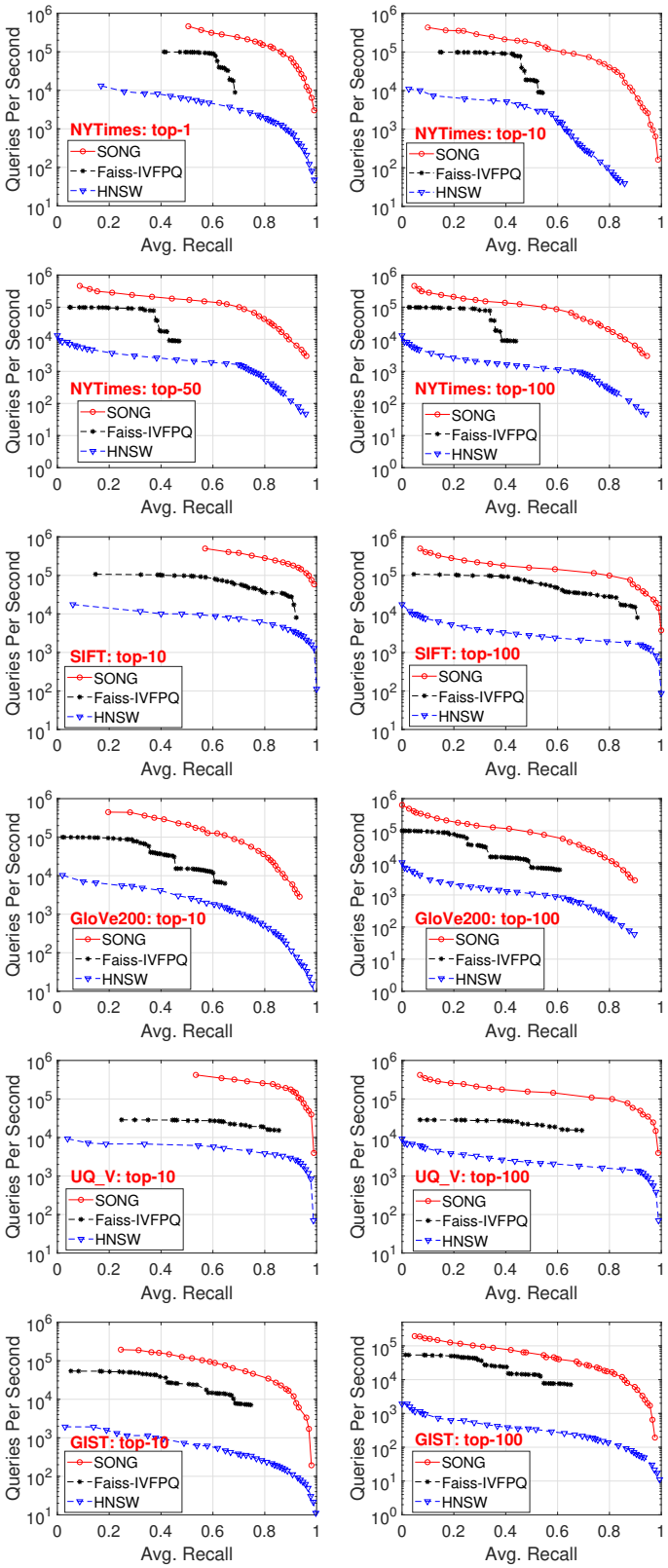


Fig. 5: Performance comparison of SONG, Faiss, and HNSW on five datasets. The results for top-1, 10, 50, and 100 are shown for NYTimes dataset, and top-10, 100 for other datasets. Y-axis (Queries Per Second, or QPS) is in logarithmic scale. In each figure, curves closer to top-right are better.

Table II shows the speedup over Faiss from moderate recalls (0.5) to high recalls (0.95) for top-10. SONG is $4.8 - 20.2$ times faster than Faiss at the same recall. Besides, Faiss cannot reach high recalls for GloVe200, NYTimes and GIST. Compared with single-thread HNSW, SONG is around 50-180x faster—it implies SONG accelerated by 1-GPU can obtain about 3-11x speedup over HNSW on a 16-thread CPU server.

ANN searching on NYTimes and GloVe200 is difficult—the data points are skewed and clustered. Faiss has a competitive performance with SONG in the low recall ranges (recall $< 60\%$). As a quantization-based method, Faiss is limited by the quality of its generated quantization code. For NYTimes and GloVe200, the Query Per Second drops dramatically at higher recalls. With increasing the size of searching priority queues, SONG can achieve more than 95% recall. Meanwhile, SONG performs around 100 times better than HNSW when the recall is around 80%. SIFT and UQ_V are "friendly" to the ANN searching—because of the un-clustered distribution of the dataset, ANN methods can quickly locate the neighboring points of a query point. SONG achieves 99% recall with a priority queue size around 100 on SIFT. GIST has the largest number of dimensions (i.e., 960) among these 5 datasets. Faiss has a closer gap to SONG because the quantization method of Faiss encodes the high dimensional data into codes in a much shorter length. Thus Faiss is required to perform fewer computations. Despite this, SONG still outperforms Faiss owing to its massively paralleled distance computation. In most recall ranges, SONG is about 180 times faster than HNSW.

Top-K nearest neighbors. The trends of the compared algorithms are consistent when K increases—the lines in the figure shifts to the left. Intuitively, finding 99% of top-1 is easier than obtaining 99% of the correct candidates of all top-10 data points. By exploring the same searching space, the recall drops when the problem becomes more difficult.

	SIFT	GloVe200	NYTimes	GIST	UQ_V
SONG	123 MB	145 MB	36 MB	123 MB	403 MB
Faiss	32 MB	38 MB	10 MB	32 MB	106 MB

TABLE III: Index memory size.

Index memory size. Table III presents the index memory size comparison. Due to the complex structure of the graph, the graph index of SONG consumes more memory than the inverted index of Faiss. This (relatively small) difference is acceptable for the GPU memory capacity.

Speedup over HNSW. We list the SONG speedup ratio to HNSW in Fig. 6 for top-10 and top-100 results. For SIFT and GloVe200, the speedup ranges from 50 to 100 for most recall values. NYTimes is an interesting case: with our optimizations that let SONG use less memory, the Query Per Second of SONG drops slower than HNSW in NYTimes dataset and thus the speedup of SONG keeps increasing when having larger recalls. The speedup of GIST is more significant than SIFT and GloVe200 because GIST has more dimensions—SONG has more chances to parallel the distance computations.

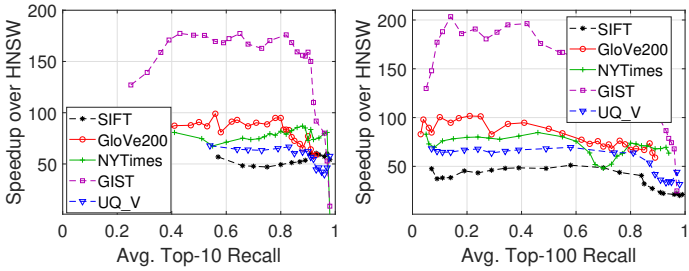


Fig. 6: Speedup over HNSW.

B. Selected Insertion and Visited Deletion Optimizations

Fig. 7 depicts the behaviors of the proposed selected insertion and visited deletion optimizations over the hash table and its alternatives. We compare performance among the basic hash table (SONG-hashtable), hash table with selected insertion (SONG-hashtable-sel), hash table with both selected insertion and visited deletion (SONG-hashtable-sel-del), and two probabilistic data structure alternatives (SONG-bloomfilter and SONG-cuckoofilter).

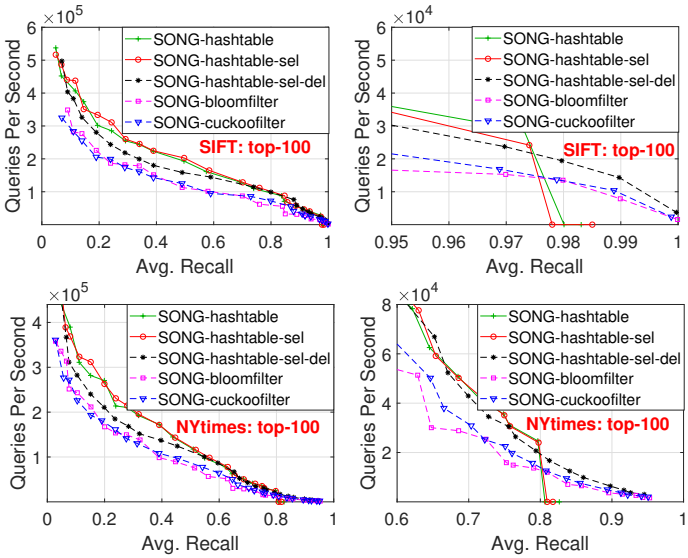


Fig. 7: Performance comparison for multiple hash table alternatives at top-100 on SIFT and NYTimes. The right column is the zoomed-in view of the first column.

For SIFT, the hash table enabled both selected insertion and visited deletion optimizations has the best performance. Selected insertion does not make a significant difference from the basic hash table solution, because we only need a small priority queue size to achieve high recall in SIFT. The visited deletion optimization works well in SIFT. The Bloom filter and Cuckoo filter solution reside between basic hashtable and hashtable-sel-del. Although they are inferior to hashtable-sel-del, they consume less GPU memory—both of them can be an alternative solution when we are short-hand of GPU memory.

When the data distribution is "unfriendly" in NYTimes, we have to enlarge the priority queue size to a few thousand to obtain high recalls. We can observe a similar trends. In addition, because of the overhead of hash table deletion operations, hashtable-sel outperforms others in the beginning. However, it runs out of memory after the recall reaches around 81%—its performance drops dramatically. At the same time, hashtable-sel-del uses much less memory. Therefore, hashtable-sel-del becomes the fastest solution among other methods. The two probabilistic data structures have competitive performance in the high recall region because they consume less memory.

C. Searching Parameter

We investigate the effectiveness of multiple searching parameters of the candidate locating stage in this section. We vary one searching parameter, setting other parameters as 1 and fix the hash table alternative as *hashtable-sel-del*. Due to the page limitation, we only show a subset of the experiment results. The omitted experiments tell similar trends.

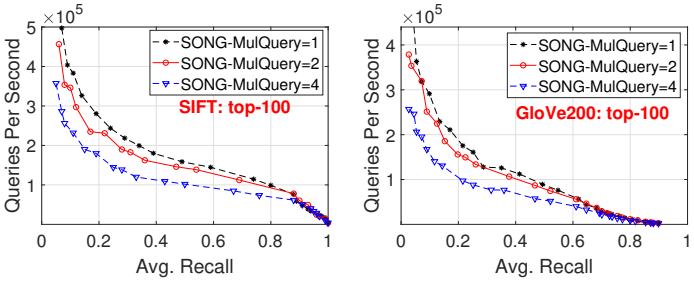


Fig. 8: The effect of multi-query in a warp.

Multi-query in a warp. As depicted in Fig. 8, we vary the number of queries in a warp among 1, 2 and 4. Although the number of active threads increases when we process more queries in a warp, the query performance goes down. The major time spent in the candidate locating stage is to loads the graph data from the global memory—it is bounded by the memory instead of the computation. Therefore, having more active threads does not improve performance. Accessing multiple parts of the graph makes the memory access pattern more unpredictable. Meanwhile, processing multiple queries in a warp also constructs multiple copies of priority queues and hash tables—it consumes more GPU memory. Thus, inferior performance is observed when solving multi-queries in a warp.

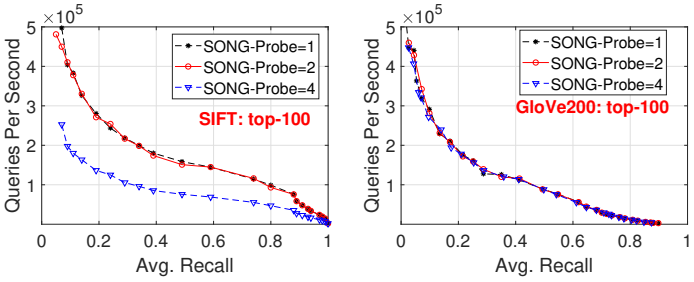


Fig. 9: The effect of multi-step probing.

Multi-step probing. The effect of multi-step probing is presented in Fig. 9. As a common parallel solution in GPU BFS, multi-step probing does not improve the performance in ANN graph searching. The reason is that the neighbors of the current processing vertex are likely to be the head of the priority queue. The multi-step probing wastes the probing

memory access and distance computations on unrelated candidates. In high recall ranges, the performance gap is much smaller because we have to probe a lot of steps to find very accurate nearest neighbor candidates—in this case, multi-step probing does not waste the operations.

D. Where Does The Time Go?

We analyze the time percentage consumed by each component on GloVe200 and GIST in Fig. 10.

Data transfer overhead. In order to use a GPU to process queries, we have to first transfer the query data from the CPU memory (host) to the GPU card (device). After the queries are completed on the GPU, we have to copy the result stored on GPU back to the CPU memory. These two memory transfer are referred as *HtoD* and *DtoH*, respectively. The left part of Fig. 10 shows the time distribution of data transfers and kernel execution. We can observe that the kernel execution takes the major execution time (more than 96% on GloVe200 and more than 89% on GIST). Since the HtoD memory transfer cost a constant time, the percentage HtoD takes decreases when the kernel execution becomes more time-consuming—we use larger priority queue size. On the other hand, more candidates are returned when we set a larger K. Thus, the DtoH time percentage slightly increases.

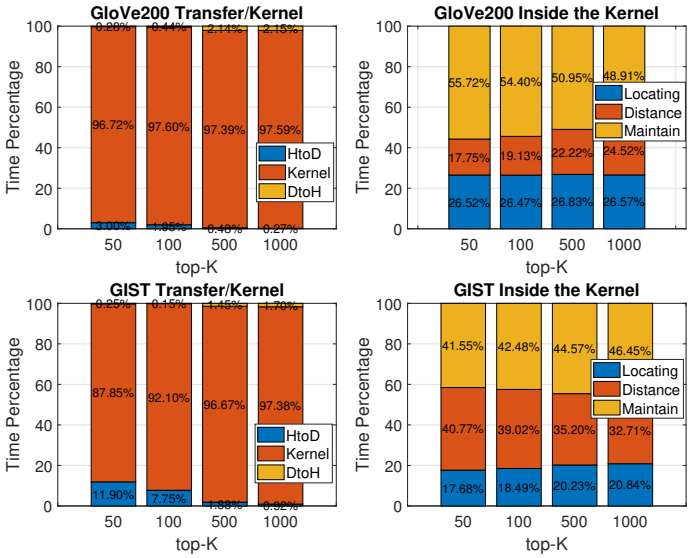


Fig. 10: Time distributions for GPU memory transfer/kernel execution and for each stage inside a kernel.

Time distribution over 3 stages. The right part of Fig. 10 depicts the time distribution of candidate locating, bulk distance computation and data structure maintenance. For both GloVe200 and GIST, data structure maintenance takes the major execution time. Since GloVe200 has 200 dimensions and GIST has 960 dimensions, the distance computation on GIST takes 8%-20% more time than the one on GloVe200. However, the maintenance time occupies a larger ratio on GIST. The selected insertion filters more vertices on GloVe200—more distance computations are required to save the GPU memory and data structure maintenance cost.

E. Query Batch Size

Fig. 11 illustrates the searching performance on different batch sizes. We sample 100 and 1k queries from the SIFT queries to construct small query batches. On the other hand, in order to investigate the searching performance on large batch sizes, we duplicate SIFT query dataset to 100k and 1m queries. As expected, the Query Per Second increases when we have a larger batch. The data transferring overhead within CPU memory and GPU is not negligible when we have a small batch. In addition, we cannot fully use the thousands of cores on GPU with a small number of queries. The performance gets better with larger batch since the overhead is amortized to the number of queries in the batch and there are sufficient queries for the massive parallelism on GPU. The Query Per Second reaches the top after having a batch with 100k. Even larger batch (1m) does not improve the performance anymore. The query batch size is 10k in previous experiments—the speedup to HNSW can be larger when we use larger query batches.

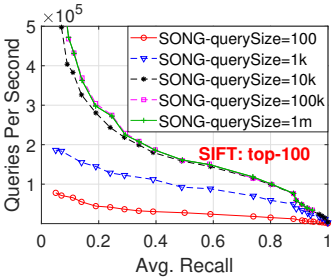


Fig. 11: Batch size impacts.

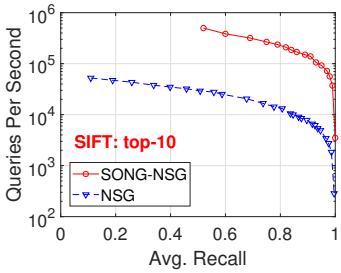


Fig. 12: SONG for NSG.

F. Generalization to Other Graph Methods

SONG is a general GPU framework for graph-based methods that accelerates the graph searching algorithm. Besides HNSW, SONG can be applied to other graph-based methods. Here we show the generalization to Navigating Spreading-out Graph [19] (NSG) as an example. We extract the graph index built by NSG and employ SONG to answer the queries on the extracted NSG index. The result is depicted in Fig. 12. For high recalls (>0.8), SONG has a 30-37x speedup to NSG.

G. Performance on Various GPUs

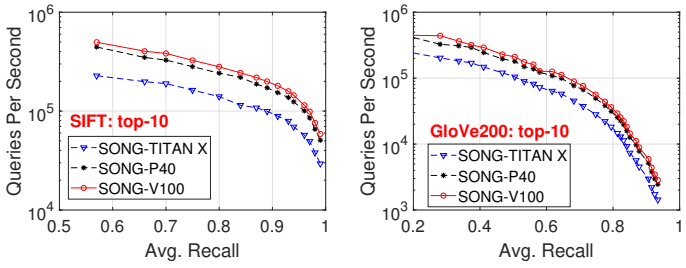


Fig. 13: SONG on different GPUs: V100, P40 and TITAN X.

Fig. 13 demonstrates the performance of SONG on various GPUs. We include 3 GPUs in the comparison: NVIDIA TESLA V100 (5,120 cores, 32 GB memory), NVIDIA TESLA

P40 (3,840 cores, 24 GB memory) and NVIDIA TITAN X (3,584 cores, 12 GB memory). The performance of SONG follows the same trend on different GPUs—the same trends are shown in the figure. The gaps of these lines are consistent with the computation power of the GPUs.

H. Out-of-GPU-Memory Dataset

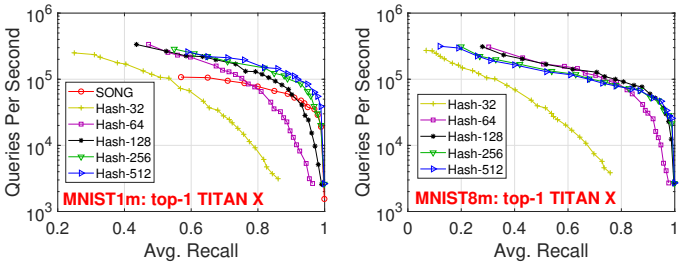


Fig. 14: Hashing on MNIST8m. We sample 1,000,000 data (MNIST1m) to demonstrate the hashing effectiveness.

We investigate the effect of hashing dimension reduction in Fig. 14. We perform the experiment on TITAN X to show the out-of-GPU memory scenario—TITAN X has the smallest memory (12 GB) among our 3 GPUs. MNIST8m (24 GB) cannot fit in the GPU memory of TITAN X. We first sample 1 million data points from MNIST8m to verify the performance of the hashing method. As illustrated in the left part of Fig. 14, we can observe that the searching performance on the 128-bit hashed dataset is comparable to the original full 784 dimensional data. For the recall ranges less than 0.9, the performance of the hashed dataset is better than the original one because distance computations of hashed datasets are faster—the distance is computed in much lower dimensions than the original 784 dimensions.

Hash bits	32	64	128	256	512	Original
Size (MB)	31	62	124	247	494	2.4×10^4

TABLE IV: Hashed dataset size of MNIST8M.

Table IV presents the hashed dataset size. With hashing, the size of the dataset becomes hundreds of times smaller and fit in the GPU memory. For example, 128-bit hashing makes the original dataset more than 190 times smaller while having comparable query performance to the original dataset. After applying the hash technique on MNIST8m, it can fit in the GPU memory. The performance on hashed MNIST8m has a consistent trend with MNIST1m, as shown in Fig. 14.

I. CPU Implementation of SONG

Besides the GPU optimizations, we implement a CPU version of SONG that is heavily engineered to improve the performance. As shown in Fig. 15, our CPU implementation outperforms HNSW on NYTimes and UQ_V.

IX. Related Work

ANN methods. Flann [51] is an ANN library based on composite tree algorithm. Annoy is based on a binary search

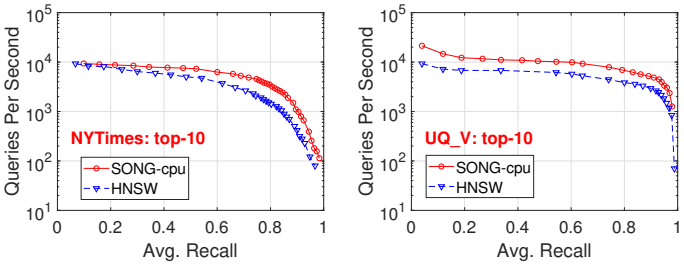


Fig. 15: SONG on CPU vs. HNSW (on CPU).

forest. FALCONN [1] is a multi-probe hashing ANN method. As the representative implementations in the tree-based and hash-based algorithms, their performance is inferior to graph-based methods [19]. For graph-based methods, HNSW [47] is based on a hierarchical graph and DPG [42] is based on an undirected graph selected from a kNN graph. NSG [19] contains only one graph with a navigating node where the search always starts. They share the same ANN graph searching. SONG can accelerate most of the algorithms in the graph-based ANN family. Faiss [36] is the fastest GPU quantization-based ANN library. Our GPU ANN system is graph-based.

GPU graph searching. Data layouts in GPU graph searching is investigated in [37], [38], [48], [53]–[55]. They partition or stream the graph to fit into the GPU memory. In our ANN searching applications, we consider the case that there is sufficient GPU memory. iBFS [44], GunRock [62], Enterprise [45], in-cache query [28] and GTS [38] constructs multiple frontiers and search them concurrently. Our ANN graph searching is different from the normal BFS—we extract the vertex from the priority queue for the next iteration. Virtual Warp [29], CuSha [37], Fine Par [65] and MapGraph [20] propose algorithm to schedule tasks and reduce warp divergence. In our ANN application, the searching architecture has to be re-designed to compute high-dimensional distances.

Maximum inner product search (MIPS). The MIPS problem has attracted a lot of attentions these days as researchers and practitioners have identified a wide range of related applications, for example, matching users with ads in sponsored search using ANN by considering a weight (such as the bid value for an ad) to each vector [15]. The recent MIPS method [67] has adopted SONG as the underlying algorithm.

X. Conclusion

In this paper, we introduce SONG—an ANN search system that performs graph-based ANN searching on GPUs. We show a novel framework that decouples the searching on graph algorithm into 3 stages to parallel the high-dimensional distance computation. We propose a combination of data structures and optimizations for GPU ANN graph searching. We present selected insertion and visited deletion optimizations to reduce the GPU memory consumption. We evaluate experimentally SONG and compare it against HNSW and Faiss on 6 real-world datasets. The results confirm the effectiveness of SONG, which has 50-180x speedup compared with single-thread HNSW, while it substantially outperforms Faiss.

REFERENCES

[1] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance. In *NIPS 2015*.

[2] A. Arora et al. HD-Index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. *VLDB 2018*.

[3] A. Arvind and C. P. Rangan. Symmetric min-max heap: a simpler data structure for double-ended priority queue. *Information Processing Letters*, 69(4):197–199, 1999.

[4] M. Aumüller et al. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *SISAP 2017*.

[5] F. Aurenhammer. Voronoi diagramsa survey of a fundamental geometric data structure. *CSUR 1991*.

[6] D. Baranchuk et al. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *ECCV 2018*.

[7] A. Z. Broder. On the resemblance and containment of documents. In *the Compression and Complexity of Sequences 1997*.

[8] L. Cayton. Fast nearest neighbor retrieval for bregman divergences. In *ICML 2008*.

[9] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *STOC 2002*.

[10] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD 2005*.

[11] R. R. Curtin et al. Fast exact max-kernel search. In *SDM 2013*.

[12] A. P. de Vries et al. Efficient k-nn search on vertically decomposed data. In *SIGMOD 2002*.

[13] D. Dearholt, N. Gonzales, and G. Kurup. Monotonic search networks for computer vision databases. In *ACSSC 1988*.

[14] B. Fan et al. Cuckoo filter: Practically better than bloom. In *CoNEXT 2014*.

[15] M. Fan, J. Guo, S. Zhu, S. Miao, M. Sun, and P. Li. MOBIUS: towards the next generation of query-ad matching in baidu's sponsored search. In *KDD 2019*.

[16] J. H. Friedman, F. Baskett, and L. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24:1000–1006, 1975.

[17] J. H. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

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

[19] C. Fu et al. Fast approximate nearest neighbor search with the navigating spreading-out graph. *VLDB 2019*.

[20] Z. Fu et al. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *GRADES 2014*.

[21] V. Garcia et al. Fast k nearest neighbor search using gpu. In *CVPR 2008*.

[22] V. Garcia et al. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *ICIP 2010*.

[23] T. Ge et al. Optimized product quantization for approximate nearest neighbor search. In *CVPR 2013*.

[24] A. Gionis et al. Similarity search in high dimensions via hashing. In *VLDB 1999*.

[25] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of ACM*, 42(6):1115–1145, 1995.

[26] K. Hajebi et al. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI 2011*.

[27] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[28] J. He et al. In-cache query co-processing on coupled cpu-gpu architectures. *VLDB 2014*.

[29] S. Hong et al. Accelerating cuda graph algorithms at maximum warp. In *SIGPLAN 2011*.

[30] Q. Huang et al. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *VLDB 2015*.

[31] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC 1998*.

[32] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[33] H. Jegou et al. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV 2008*.

[34] Z. Jia et al. Dissecting the nvidia volta GPU architecture via microbenchmarking. *arXiv:1804.06826*, 2018.

[35] Z. Jin et al. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE Trans. Cybernetics*, 44(11):2167–2177, 2014.

[36] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[37] F. Khorasani et al. Cusha: Vertex-centric graph processing on GPUs. In *HPDC 2014*.

[38] M.-S. Kim et al. Gts: A fast and scalable graph processing method based on streaming topology to GPUs. In *SIGMOD 2016*.

[39] P. Li. Linearized GMM kernels and normalized random Fourier features. In *KDD 2017*, 2017.

[40] P. Li, G. Samorodnitsky, and J. Hopcroft. Sign cauchy projections and chi-square kernel. In *NIPS 2013*.

[41] P. Li, A. Shrivastava, and C. A. Konig. GPU-based minwise hashing: Gpu-based minwise hashing. In *WWW 2012*, 2012.

[42] W. Li et al. Approximate nearest neighbor search on high dimensional data: Experiments, analyses, and improvement. *arXiv:1610.02455*, 2016.

[43] W. Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB 1980*.

[44] H. Liu et al. ibfs: Concurrent Breadth-First Search on GPUs. In *SIGMOD 2016*.

[45] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *SC 2015*.

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

[47] 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.

[48] D. Merrill et al. Scalable GPU graph traversal. In *SIGPLAN 2012*.

[49] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[50] S. Morozov and A. Babenko. Non-metric similarity graphs for maximum inner product search. In *NIPS 2018*.

[51] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[52] P. Ram and A. G. Gray. Maximum inner-product search using cone trees. In *KDD 2012*.

[53] H. Seo et al. Gstream: A graph streaming processing method for large-scale graphs on GPUs. In *SIGPLAN 2015*.

[54] X. Shi et al. Graph processing on gpus: A survey. *CSUR 2018*.

[55] X. Shi et al. Optimization of asynchronous graph processing on GPU with hybrid coloring model. *SIGPLAN 2015*.

[56] A. Shrivastava and P. Li. Fast near neighbor search in high-dimensional binary data. In *ECML 2012*.

[57] S. Tan, Z. Zhou, Z. Xu, and P. Li. Fast item ranking under neural network based measures. In *WSDM 2020*.

[58] S. Tan, Z. Zhou, Z. Xu, and P. Li. On efficient retrieval of top similarity vectors. In *EMNLP 2019*.

[59] G. Teodoro et al. Approximate similarity search for online multimedia services on distributed cpu—gpu platforms. *The International Journal on Very Large Data Bases*, 23(3):427–448, 2014.

[60] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.

[61] Y. Wang et al. FLASH: Randomized algorithms accelerated over CPU-GPU for ultra-high dimensional similarity search. *SIGMOD 2018*.

[62] Y. Wang et al. Gunrock: A high-performance graph processing library on the gpu. In *SIGPLAN 2016*.

[63] X. Wu et al. Multiscale quantization for fast similarity search. In *NIPS 2017*.

[64] Y. Wu et al. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *SIGMOD 2014*.

[65] F. Zhang et al. Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *CGO 2017*.

[66] Y. Zheng et al. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD 2016*.

[67] Z. Zhou, S. Tan, Z. Xu, and P. Li. Möbius transformation for fast inner product search on graph. In *NeurIPS 2019*.