GGNN: Graph-based GPU Nearest Neighbor Search

Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P.A. Lensch

Abstract—Approximate nearest neighbor (ANN) search in high dimensions is an integral part of several computer vision systems and gains importance in deep learning with explicit memory representations. Since PQT [1], FAISS [2], and SONG [3] started to leverage the massive parallelism offered by GPUs, GPU-based implementations are a crucial resource for today's state-of-the-art ANN methods. While most of these methods allow for faster queries, less emphasis is devoted to accelerating the construction of the underlying index structures. In this paper, we propose a novel GPU-friendly search structure based on nearest neighbor graphs and information propagation on graphs. Our method is designed to take advantage of GPU architectures to accelerate the hierarchical construction of the index structure and for performing the query. Empirical evaluation shows that GGNN significantly surpasses the state-of-the-art CPU- and GPU-based systems in terms of build-time, accuracy and search speed.

Index Terms—Nearest neighbor searches, Graph and tree search strategies, Information retrieval, Approximate search, Similarity search, Big data.

1 Introduction

Approximate nearest neighbor (ANN) search plays a crucial and long-standing role in various domains, including databases, computer vision, autonomous vehicles, personalized medicine, and machine learning. Since collecting large amounts of data became easier, the creation of a scalable and efficient data structure for retrieving similar items has become an active research topic. Despite all recent advances, the only available method for guaranteed retrieval of the exact nearest neighbor in high dimensions is still exhaustive search, due to the curse of dimensionality [4]. Even when leveraging modern hardware, it remains impractical to perform an exhaustive search over billions of high-dimensional data points. Instead, most popular methods relax the problem by searching for an entry that is likely to be the nearest neighbor, accepting a minimal loss in accuracy.

Besides designing a very fast GPU-based approximate kNN query algorithm, we address the efficient construction of a hierarchical graph-based search structure. Building time becomes more and more important in dynamically growing or changing datasets for on-the-fly analysis, e.g. in correspondence and feature matching for tracking and object recognition in videos, for newly embedded vectors of neural networks similar to DEEP1B [5], or in recommender systems [6]. Our graph-construction explicitly determines the true k nearest neighbors of each point in the dataset with high probability. Solving this for-all task is highly relevant in n-body problems like salient point estimation, kernel-density computation, cluster analysis, retrieval of more robust prototypes, or feature matching in embeddings.

To keep up with the scale of data that is produced day by day, modern approaches use index structures that are heavily tailored towards exploiting the massive parallelism of GPUs [1], [2], [7], [8] or custom hardware [9], as opposed to previous CPU-based methods [10], [11], [12], [13], [14], [15], [16], [17].

The quality of the recall heavily depends both on the choice of the search structure and the executed search (query) itself. Structures based on quantization or hashing/binning schemes [1], [2], [11], [12], [13], [14], [15], [16], [17], [18], [19] can be built efficiently, but typically suffer from relatively low recall rates as enumerating and visiting neighboring cells is exhaustive in high dimensions. Better recall rates are recently achieved by graph-based methods [8], [20], [21], [22], [23], [24], [25], [26], [27]. Existing methods for constructing effective search-graphs, e.g. [20], update varying-sized edge lists sequentially. They are highly dependent on global memory synchronization and difficult to parallelize effectively beyond a few cores. Hence, their construction times are not scaling well and are measured in hours or even days [19], [27].

Given a precomputed graph structure, a query traverses the edges of the graph to decrease the distance to a query point. It needs to compute the distances to all neighbors of the currently investigated node, move to the next-best point and store which points have already been visited. All decisions are made locally and independent for each query, rendering the query algorithm an ideal candidate for parallelization. However, a parallel implementation needs to carefully address a number of issues to be efficient. For one, the available memory bandwidth for loading each vector to compare with might become a limiting factor. Here, Optane memory [27] or GPUs [3] offer a solution. Secondly, on GPUs one needs to cope with the limited amount of available per-thread or per-block memory when storing the list of visited points.

Hence, we propose a GPU-friendly query design on thread-block level with high on-chip resource utilization through a fully

- *During the work of this paper, all authors were member of the Computer Graphics group at the University of Tübingen, Germany.*
- *Patrick Wieschollek and Fabian Groh are now with Amazon, Tübingen, Germany. This work has been done prior to joining Amazon.*
- *This is the author's version of the paper. The final published version can be found here: https://doi.org/10.1109/TBDATA.2022.3161156*
-

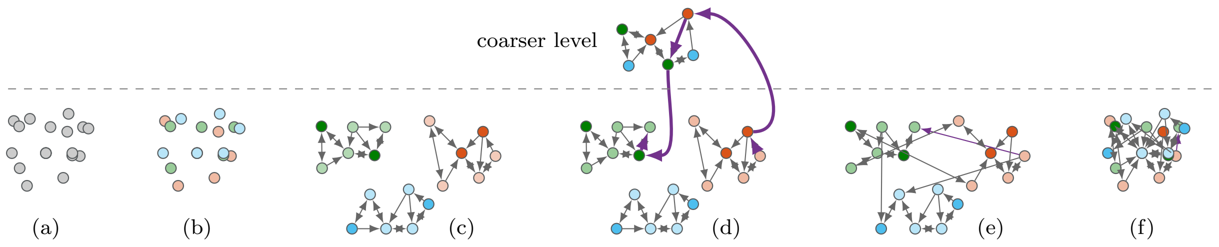


Fig. 1: Illustration of the bottom-up construction of the kNN-graph for a dataset (a). After random partitioning (b), a kNN-graph is constructed for each partition (c). A selection of nodes builds a coarser kNN-graph (d), which is used to propagate links between partitions. These links are used to merge several partitions (e) into a final kNN-graph (f).

parallel multi-purpose cache and a fixed number of neighbors per point. Further, we introduce a novel technique for very fast graph construction that utilizes the fast parallel query algorithm to iteratively merge multiple sub-graphs together. This bottom-up construction scheme is sketched in Figure 1. Hierarchical graphs are used as a global optimization substitution to overcome gaps in local connectivity, in particular during construction. Additionally, our local symmetric linking approach reduces the amount of redundant links in the graph to circumvent negative impacts of overflows in the memory-limited caches.

Our approach is also inherently suited for batch processing by already generating multiple sub-graphs at the same time. By forgoing the final merge of several sub-graphs into one combined graph, one can easily obtain independent yet effective sub-graphs for parts of large-scale datasets which would otherwise not fit in memory. These can easily be processed even on multiple GPUs at the same time where each query then needs to be executed once on each shard. This simple yet effective method allows for optimal multi GPU utilization.

To summarize our main contributions, we propose a method for extremely fast approximate kNN-search on GPUs for high-dimensional data. We focus not only on fast query time, but also on a very efficient construction of the index structure, while still achieving high recall rates of the true nearest neighbours.

As evidenced by our empirical evaluation, the presented scheme outperforms existing approaches concerning both the construction as well as the query time. At the same time, the recall rate is consistently high and can be traded in for even faster construction or query time. We present a multi-GPU scheme that is capable of achieving above 99% recall even for common benchmark datasets with billions of high-dimensional entries.

2 Related Work

A large amount of literature exists on designing structures that accelerate a nearest neighbor search. Besides traditional approaches [10], [28], most popular techniques rely either on data quantization in clusters [1], [2], [11], [12], [13], [14], [15], [16], [17], [18], [19] or building neighborhood graphs [8], [22], [23], [24], [25], [26], [27]. To achieve peak performance, most of these methods compute a compressed representation for each entry as large datasets will not fit into fast memory. There exist several strategies to compute such a compression. While hashing methods [29], [30], [31] produce compact binary codes, quantization-based methods reuse centroids by assigning each data point a unique identifier based on the centroid to which they belong. It has been empirically shown that quantization methods are more accurate than various hashing methods [10], [11].

Quantization methods for nearest neighbor search using clustering methods were popularized by Jégou *et al.* [11] while originally being introduced in [32]. Such index structures, like IVFADC [11], partition the high-dimensional search space into disjoint Voronoi cells described by a set of centroids obtained by Vector Quantization (VQ) [33]. The idea has been extended later by Babenko *et al.* [15], where the high-dimensional vector-space is factored in orthogonal subspaces. Hereby, each vector is assigned to a centroid independently for each subspace according to a separate codebook that resides there. Wieschollek *et al.* [1] proposed a hierarchical representation of the codebook besides demonstrating superior performance using a GPU. Johnson *et al.* [2] ported IVFADC [11] to the GPU in combination with a fast GPU-based implementation for k-selection, *i.e.*, returning the k lowest-valued elements from a given list (a crucial part of quantization based methods). They are the first employing multi-GPU parallelism by replication and sharding. Their work forms the library “FAISS”. Eventually, Chen *et al.* [18] proposed a GPU based method RobustiQ overcoming the memory limitations of FAISS by extending the idea of Line Quantization from [1] in a hierarchical fashion. Still, the reported distances are only an approximation of the true distance.

All hashing and quantization-based indexing schemes share the same problem that they partition the space into cells. While the containing cell for a query might be found very efficiently, the exact nearest neighbor might be across the boundary to one of the neighboring cells. Determining and visiting all neighboring cells in high dimension is a problem severely limiting these approaches.

kNN-graph based methods are another way to accelerate the query process. Our presented approach belongs to this category. The main idea is to link each point from the search space to k of its nearby points. Each query will start at a random guess in the dataset. Then, the guess itself is refined by replacing it with a better point from the k linked neighbor points. Chen *et al.* [23] propose a fast divide and conquer strategy for computing such kNN-graphs. Done *et al.* [22] introduced NN-descent for using kNN-graphs to accelerate NN-search. Hereby, each point maintains a list of its own nearest neighbors and points where itself is considered as a nearest neighbor. This has been later extended [24] to make use of MapReduce. EFANNA as a multiple hierarchical index structure uses a truncated KD-tree to build a kNN-graph [25].

In the ideal case, a kNN-graph augmented with additional links could guarantee that for an arbitrary starting point the NN-descent will converge to the correct solution. Computing such a graph with additional links at scale is not practicable. Therefore, several methods exist to at least approximate such a graph [8], [26]. Fu *et al.* [26] introduce NSG as an approximation. To reduce the overall number of edges, their optimization tries to lower the out-degree individually per node. Their method can scale beyond multiple cores, outperforming a GPU approach [2] on a benchmark dataset. Harwood *et al.* [8] suggest an alternative approach for constructing such a graph. Starting from a fairly dense graph, they remove "shadowed edges", which are redundant when considering traversing paths during a query. They showed promising results using the GPU but only on rather small datasets as their build time is rather high. Malkov *et al.* [20] construct a hierarchical graph structure to accelerate the nearest neighbor search.

For graph-based methods, memory throughput is often the limiting factor. Ren *et al.* [27] propose an optimized construction of HNSW-style search graphs on out-of-core datasets using heterogeneous memory. Using fast Optane memory allows them to quickly query even billion-scale datasets at high accuracy. Zhao *et al.* [3] propose GPU-based "search on graph" (SONG) using index structures built using either HNSW [20] or NSG [26], achieving significant speedup over CPU-based queries in most cases.

Our method is most similar to HNSW [20] with its hierarchical construction, but it is very carefully tuned for optimal parallelism of construction tasks and also differs during the query.

3 Background

In this section, we formally introduce the approximate nearest neighbor (ANN) problem statement and used notation.

3.1 Nearest Neighbor Search

The nearest neighbor problem retrieves a point $x^\star$ from a dataset $\mathcal{X} = \{x_1, \ldots, x_n\}$ that has the smallest distance to a query q. For the sake of simplicity, here, we assume an Euclidean space ($\mathcal{X} \subset \mathbb{R}^d, q \in \mathbb{R}^d$) and Euclidean distances ($\|\cdot\|_2$). The nearest neighbor $x^\star \in \mathcal{X}$ of q therefore is defined as

$$x^\star = \arg\min_{x \in \mathcal{X}} \|q - x\|_2 \,. \tag{1}$$

Similarly, the k-nearest neighbor search retrieves the k closest entries from $\mathcal{X}$ for a given query. As finding the exact nearest neighbor might be costly, we may accept points in $\mathcal{X}$ which are close to q and therefore deliver an approximate solution to Eq. (1).

3.2 KNN Graph

In a kNN-graph, each point x form the dataset $\mathcal{X}$ represents one node in the graph G. Further, we define $\mathcal{N}_x \subseteq \mathcal{X}$ as a local neighborhood of x with k elements and defer the details on how to construct $\mathcal{N}_x$ to the next section. The edges E of the graph are then defined as (x, y) where $y \in \mathcal{N}_x$. Note that the resulting graph is a directed graph $G = (\mathcal{X}, E)$, where $E = \{(x, y) \mid x \in \mathcal{X}, y \in \mathcal{N}_x\}$ and $(x, y) \in E \nRightarrow (y, x) \in E$.

One greedy algorithm to find the nearest neighbor for a query point q is *NN-descent* [22]. Starting from an initial guess $x \in \mathcal{X}$, the distance between q and each neighboring point $y \in \mathcal{N}_x$ is computed. If any $y \in \mathcal{N}_x$ is closer to q than x, the guess x is replaced by the closest point from $\mathcal{N}_x$ to q. This process iterates until no point in $\mathcal{N}_x$ has a smaller distance to q than x. However, as the current x might not provide an edge into the right search direction, this greedy algorithm might get stuck in a local minimum on a pure kNN-graph.

3.2.1 Common Pitfalls

Since the NN-descent is a greedy search, it offers no guarantee on finding the exact solution. Some of the reasons are listed below:

Connectivity: As a kNN-graph is a directed graph, y might be directly connected to x, being its nearest neighbor, hence $y \in \mathcal{N}_x$. But this does not imply that the inverse link exists ($y \in \mathcal{N}_x \nRightarrow x \in \mathcal{N}_y$). Therefore, the construction of an augmented (diversified) kNN search-graph has to deal with synchronizing outgoing and incoming (inverse) edges.

Gaps in high-dimensional spaces: As each point is only linked to a finite number of local neighbors, there exist pathological cases (even in 2D), where close-by points are not directly connected at all. Such a case is illustrated in Figure 3. Due to the gap, the true nearest neighbor will not be found. Computing an idealized monotonic relative neighborhood graph [26] (MRNG) would avoid this issue, but requires a strongly varying connectivity not suitable for parallel approaches – besides its additional computational burden.

Degree of nodes: There exists a trade-off when choosing the cardinality of $\mathcal{N}_x$ for any $x \in \mathcal{X}$. Having fewer edges amplifies the previously described issues, but also reduces the number of necessary comparisons at each step. Conversely, having many edges allows the greedy search to escape from local neighborhoods but increases the cost for each iteration.

4 GPU-Based Nearest Neighbor Search

Searching for the k nearest neighbors for multiple queries, given some graph structure, is the central operation for various applications and also the main building block for our graph construction (see Section 5). Achieving high recall rates at very short times is the primary goal of our parallel GPU-based search-algorithm on kNN-graph structures. The algorithm can, within bounds, be tuned for quality or speed, offering a solution to a broad range of use-cases, accommodating those requiring high recall rates as well as real-time algorithms with reduced quality constraints.

The main idea is to highly parallelize the search by utilizing one thread-block per query. In contrast to the naïve solution of a query per thread, the main advantage is to bundle enough on-chip resources that all meta-data can be kept on very fast memory. This is of utmost importance, since graph-based methods still need to load a large amount of neighborhood information as well as vectors from global memory for distance computations. In addition, the thread-block approach guarantees high memory bandwidth through very efficient coalesced memory accesses. It also allows to perform more computation in parallel since common tasks like distance computations or the maintenance of priority queues and visited lists can be parallelized. Furthermore, individual queries that vary in runtime can be scheduled independently.

4.1 GPU-based Search with Backtracking

Assuming a diversified (see Section 5.2) kNN-graph with given starting points $\mathbf{s} \subset \mathcal{X}$, for a query $q \in \mathbb{R}^d$, the simple greedy downhill search with backtracking from Algorithm 1 is executed. First, the cache structure (see Section 4.1.1) is initialized (`init`) with the query point and a slack factor τ (see Section 4.1.2). Then, the starting points $\mathbf{s}$ are introduced to the cache by `fetch`, which computes their distances to the query q and adds them to the priority queue (*prioq*). As starting points $\mathbf{s}$, we use the nodes on the top-layer of our hierarchical graph (see Section 5.4).

Until there are no more unexplored points in the priority queue, all neighbors $\mathcal{N}_a$ of the closest not yet visited point a are being fetched to the cache. Essentially, performing a depth-first search in the graph. Our cache structure manages a sorted list of the k closest points (*best*) to the query which are observed during the traversal. Hence, the *best* list is returned as the result of the search.

Algorithm 1: Basic query with backtracking.

```
Function query($G$, $\mathbf{s}$, $q$, $\tau$, $k$):
    Input: search-graph $G$ with start points $\mathbf{s}$, query
           point $q$, slack factor $\tau$ (see Section 4.1.2)
    Output: $k$ closest points to $q$
    $cache$.init($q$, $\tau$) /* (see Section 4.1.1)   */
    $cache$.fetch($\mathbf{s}$)
    while $(a \leftarrow cache.\texttt{pop}()) \neq \emptyset$:
        $cache$.fetch($\mathcal{N}_a$) /* (see Section 3.2) */
    return $cache.best$
```

4.1.1 Caching on GPUs

The centerpiece of our fast GPU-based kNN search is a multi-purpose cache (see Algorithms 1 and 2). One of the major deficits of GPUs is their limited on-chip memory (*register* and *shared memory*[1]) that can be accessed with low latency.

In particular, kNN-graph algorithms have a strong demand for highly dynamic structures like unpredictably growing lists of potential points to visit or of already visited points. These meta-data are perused and modified frequently during the query.

The cache's memory layout is shown in Figure 2. It consists of three major parts:

1) *best*: A sorted list of the points closest to the query and their distances.
2) *prioq*: A priority-queue that manages points to be visited as a distance-sorted ring-buffer.
3) *visited*: A ring-buffer that caches indices of already visited points in a first in first out (FIFO) fashion.

All three parts are handled in *shared memory*. The combined length is a multiple of the assigned threads, hence, every thread has multiple work items.

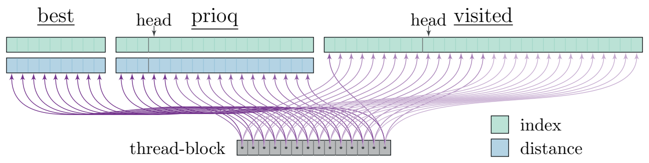


Fig. 2: Our cache consists of a *best* list, *prioq* (priority queue) and a *visited* list. It resides entirely within shared memory. The *best* list and *prioq* both contain indices and distances and are sorted by distance. The *visited* list contains only indices to preserve memory. The *prioq* and the *visited* list are implemented as ring buffers. This allows elements to be easily popped from the front of the *prioq* and to overwrite old elements in the *visited* list once it is full. The cache is accessed and maintained in parallel by the entire thread-block.

The main methods of our cache are shown in Algorithm 2 and described in the following paragraphs. Note that for ease of readability synchronization barriers preventing race conditions are omitted. We refer to our published open source code[2] for more information. For the purpose of visualization, *shared memory* variables are denoted with an underline, while normal variables can be assumed to live in *register* space.

Algorithm 2: Cache Operations.

```
Function fetch($\mathbf{p}$):
    Input: Point proposals $\mathbf{p}$.
    parfor $p_{c_i} \in \{\underline{best, prioq, visited}\}$ :
        for $p \in \underline{\mathbf{p}}$ :
            if $p_{c_i} = p$ :
                remove $p$ from $\underline{\mathbf{p}}$
    for $p \in \underline{\mathbf{p}}$ :
        $d \leftarrow$ dist ($p$) /* Compute in parallel.  */
        if criteria($d$) :
            push ($p$, $d$)
Function push($p$, $d$): /* Parallel insertion    */
    Input: Point proposal $p$ with distance $d$.
    parfor $p_{c_i}, d_{c_i} \in \{\underline{best, prioq}\}$ :
        /* Read $p_{c_i}$ and $d_{c_i}$ and sync.     */
        if $d_{c_i} \geq d$ :
            if is_not_end($c_i$) :
                $\underline{p_{c_{i+1}}} \leftarrow p_{c_i}$
                $\underline{d_{c_{i+1}}} \leftarrow d_{c_i}$
            if is_begin($c_i$) or $\underline{d_{c_{i-1}}} < d$ :
                $\underline{p_{c_i}} \leftarrow p$
                $\underline{d_{c_i}} \leftarrow d$
Function pop(): /* Single-threaded routine    */
    Output: Returns head of priority queue.
    $p, d \leftarrow \underline{prioq_{head}}$
    if not criteria($d$):
        return empty
    /* Remove point from prioq.          */
    $\underline{prioq_{head}} \leftarrow empty$
    move_head_forward($head_{prioq}$)
    /* Add point to visited list.          */
    $\underline{visited_{head}} \leftarrow p$
    move_head_forward($visited_{head}$)
    return $p$
Function criteria($d$):
    return $d \leq d_{best_k} + \xi$ /* (see Section 4.1.2) */
```

Fetch: Given a list of point proposals $\mathbf{p}$, first, known elements are removed from the proposal list. Each proposal is compared in parallel against the working items of each thread and removed in case of a match. The remaining proposals are processed iteratively. The distance to the query point is computed in parallel (dist). Every thread reads its respective vector element(s) from memory and computes the element-wise result. A subsequent reduction produces the complete distance. For parallel primitives like reduction, we use NVIDIA's CUB[3] library for highly optimized implementations. The parallel coalesced loading and processing is highly effective in terms of GPU utilization. Points that are inside our criteria are pushed to the cache structure.

Push: For a given pair of index p and distance d, the push method performs a parallel insertion into the distance-sorted lists of *best* and *prioq*. To perform the insertion, all items with index c_i that have a distance which is greater than d are temporarily copied into the thread-block's registers and are written back into

1. https://docs.nvidia.com/cuda/
2. https://github.com/cgtuebingen/ggnn
3. https://nvlabs.github.io/cub/

the subsequent index c_{i+1} unless the end of the list or ring-buffer is reached. The new element is inserted where the item to the left is closer than the new item, or at the beginning if the new item is closer than all previous items. Consequently, a point that is eligible for *best* and *prioq* will be inserted in both lists simultaneously.

Since *prioq* is a ring-buffer, the logical beginning and end positions are dependent on the head position. Hence, index computations on the physical borders need to be wrapped around.

Pop: This single-threaded routine returns the current head of the priority queue and manages the ring-buffers. Our `criteria` is monotonically decreasing over the course of a query (inspect Section 4.1.2). Hence, if the head of the *prioq* violates the `criteria`, we are able to safely terminate the query. In the valid case, the point is removed from the *prioq* and added to the *visited* list at the head position. Both head pointers are moved one step forward. Finally, the point is returned.

4.1.2 Stopping Criterion

On high-dimensional data, algorithms that solely rely on greedy downhill search will quickly get stuck in local minima. On the other hand, complete backtracking might visit the entire dataset.

A common stopping criterion, e.g. [20], is to terminate the search once the *best* list cannot be improved by adding the closest not yet visited element, i.e., once $d > d_{\text{best}_K}$, where d is the distance of the new element and d_{best_K} is the distance of the last element in the *best* list. To achieve higher recalls, the *best* list needs to be extended (K needs to be increased), potentially by hundreds of elements.

We propose an efficient approximation by adding an adaptive, monotonically decreasing slack ξ to the distance of the k-closest neighbor: $d_{\text{best}_K} \approx d_{\text{best}_k} + \xi$, where $k \ll K$, that allows us to limit the size of the *best* list to only the k closest neighbors which are to be queried (or at least 10). Instead of explicitly tracking the distance of further neighbors, the slack accounts for a safety margin to ensure that the path to the closest neighbor will likely be found, e.g. Figure 3. The search will terminate once

$$d > d_{\text{best}_k} + \xi, \quad \xi = \tau \cdot \min\{d_{\text{best}_1}, d^+_{\text{nn}_1}\}. \tag{2}$$

The slack factor τ controls the size of the safety margin. d_{best_1} is query-specific and relates the margin to the currently best found match while $d^+_{\text{nn}_1}$ correlates with the density of the given database nodes. Specifically, $d^+_{\text{nn}_1}$ provides a global limit (to catch outliers) that is calculated during graph construction. It denotes the maximum distance to the closest neighbor across all points within the currently processed subset of the graph $\mathcal{S}$:

$$d^+_{\text{nn}_1} = \max_{x \in \mathcal{S}} \left\{ \min_{y \in \mathcal{N}_x} \|x - y\|_2 \right\}, \text{where } \mathcal{S} \subseteq \mathcal{X}. \tag{3}$$

Using our stopping criterion keeps the *best* list at a small size and by that reduces the amount of shared memory required for performing the query. As demonstrated by Figure 3, our stopping criterion allows for high efficiency even at high accuracy.

Typical values for τ are above 0 and below 2. Larger values improve accuracy with diminishing returns.

5 Approximate Symmetric Nearest Neighbor Graph Construction

The construction of proper CPU-based kNN search-graphs is performed typically in a sequential procedure on global graphs. Edges are either attached (e.g., [20]), or pruned (e.g., [8], [21], [26]) one after the other. During the process, the edge-lists per node tend to vary heavily ranging from completely empty to complete lists of all possible points. While these global optimization approaches are able to produce high quality graph structures on CPUs, for fast GPU-based construction, they are impractical.

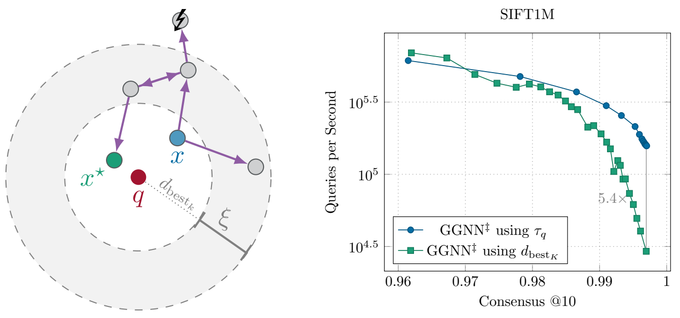


Fig. 3: Allowing a slack ξ makes it possible to escape a local minimum $x \in \mathcal{X}$, eventually reaching the solution $x^\star \in \mathcal{X}$ for a given query $q \in \mathbb{R}^d$. Points outside of this boundary are unlikely to provide helpful links and are discarded to reduce computational cost. (right) Comparison of using our stopping criterion (τ_q) to using a growing *best* list to terminate queries. With the *best* list based approach, high recalls can only be achieved in conjunction with high memory usage which slows down the query.

We propose a parallel graph construction based on parallel merging of hierarchical kNN search-graphs as sketched in Figure 1. By partitioning the task of constructing the search graph into small, parallelizable tasks, we are able to efficiently use the massive parallism offered by GPUs. In the following, we present our parallel graph construction process, followed by details on the hierarchical query process, and the graph-diversification and refinement by symmetric linking. Finally, we look at how to best perform the final queries and multi-GPU configurations.

5.1 Building the Hierarchical kNN-Graph

Our construction process (Algorithm 3) builds the kNN search-graph bottom-up by recursively merging smaller search-graphs.

To `init`ialize the process, we logically partition the entire dataset $\mathcal{X}$ into small batches of size s, e.g. 32. These represent hierarchical search-graphs of height 1. We initialize the neighbors within the bottom layer by first performing the `merge` operation which initializes the k outgoing edges per point with each point's nearest neighbors. Then, the `sym` operation replaces up to k_{sym} outgoing links in an effort to approximate an undirected graph, details in Section 5.2.

In order to merge the individual sub-graphs, we first `select` s points from the groups of g graphs which each are to be merged. These selected points now form batches representing the new top layer. There, we perform the same operations as with the bottom-layer before. We `merge`, i.e. query for the nearest neighbors to fill the outgoing edge list and perform graph-diversification (`sym`). As the nearest neighbor of any point might be found in any sub-graph, edges across batch boundaries will be introduced. These steps now repeat top-down until the entire hierarchical search-graph is interconnected.

Whenever the bottom layer is reached, we also save the distance to the first nearest neighbor d_{nn_1} for each point and compute the mean and max over the dataset (`stats`). This allows for an easy adaption of our stopping criterion to every dataset without any precomputation. Initially, the maximum is prone to

Algorithm 3: Parallel graph construction.

```
Function BuildGraph($\mathcal{X}$, $L$, $k$, $\tau_{build}$):
    Input: Dataset $\mathcal{X}$, number of layers $L$, number of
           neighbors per point $k$, slack factor $\tau_{build}$
    Result: search structure graph
    graph $\leftarrow$ init($\mathcal{X}$, $L$, $k$, $\tau_{build}$)
    for $l_{top} \leftarrow 0$ to $L-1$:
        if $l_{top} > 0$:
            graph.select($l_{top}$)
        for $l_m \leftarrow l_{top}$ to 0:
            graph.merge($l_{top}$, $l_m$)
            graph.sym($l_m$)

Function graph.merge($l_{top}$, $l_m$):
    parfor $z \in$ graph.layer($l_m$):
        /* Use brute force if $l_{top} = l_m$.          */
        $b_z \leftarrow$ graph.query($z$, $l_{top}$, $l_m$)
    graph.layer($l_m$) $\leftarrow b$
    if $l_m = 0$:
        graph.stats()

Function graph.sym($l_i$):
    parfor $z \in$ graph.layer($l_i$):
        graph.sym_links($z$)
```

Algorithm 4: Hierarchical query.

```
Function graph.query($z$, $l_{top}$, $l_m$):
    Input: query point $z$ from layer $l_m$, start layer $l_{top}$,
           and end layer $l_m$
    Output: $k$ closest points to $z$
    $cache$.init($z$, $\tau$)
    $\mathbf{s} \leftarrow$ getTopSegment($z$, $l_{top}$)
    $cache$.fetch($\mathbf{s}$)
    for $l_i \leftarrow l_{top} - 1$ to $l_m$:
        $cache$.transform($l_i$)
        while ($a \leftarrow cache$.pop()) $\neq \emptyset$:
            $cache$.fetch($\mathcal{N}_a$)
    return $cache.best$
```

outliers in the still coarse nearest neighbor graph. Therefore, during construction, we substitute the maximum distance to the closest neighbor $d^+_{\text{nn}_1}$ in Eq. 2 with the mean distance $\bar{d}_{\text{nn}_1}$:

$$\bar{d}_{\text{nn}_1} = \frac{1}{|\mathcal{S}|} \sum_{x \in \mathcal{S}} \left\{ \min_{y \in \mathcal{N}_x} \|x - y\|_2 \right\}, \text{where } \mathcal{S} \subseteq \mathcal{X}. \quad (4)$$

For selecting the points for the top layer, weighted reservoir sampling with d_{nn_1} as weights is used with the method of [34].

As all construction tasks can be performed in parallel for each point in each layer while requiring small, limited amounts of memory each, the construction can be performed quite quickly when exploiting the massive parallelism offered by GPUs.

Hierarchical kNN Graph Query: In a top layer, the `merge` operation can simply perform brute-force search between the s points per batch. With each lower layer, the number of batches grows with the factor g. To perform efficient searches for nearest neighbors in lower layers we perform a hierarchical query (Algorithm 4), which layer by layer utilizes the already merged upper layer to find entry points into the batches on the lower layers, and otherwise behaves just like the previously presented simple query (Section 4.1). The indices of the closest points found on the upper layer are then `transform`ed into the indices of the same points on the next lower layer and the search continues until the target layer l_m is reached. While distances to already visited points can be reused, their neighborhood on the lower layers changes. Therefore, we allow all points to be visited again after switching between layers.

As in HNSW [20], the hierarchy allows us to bridge gaps in the connectivity as multiple entry points from the top layer may approach the query on the bottom layer from different sides.

5.2 Graph Diversification

In order to reach any point from any direction, every edge of the graph, in principle, should be undirected. However, when every point should at least know its k_{nn} nearest neighbors, the total number of edges per point x would vary significantly for an undirected graph as the number of points which have x as their nearest neighbor will heavily depend on the local geometry.

In order to obtain a regular representation and to allow for simple parallelized traversal with constant workload per step, our graph only consists of exactly k directed (*i.e.*, outgoing) edges. We logically split the outgoing edges into at least $k_{\text{nn}} = k/2$ true nearest neighbors and up to $k_{\text{sym}} = k/2$ inverse links that are used to approximate an undirected graph.

As the number of representable inverse links is limited, one has to determine which of the inverse links are necessary. Harwood and Drummond [8] introduce the concept of shadowed links which are made redundant by the query's greedy exploration strategy. Optimizing the entire graph to remove all potentially shadowed edges requires a complex global optimization. The same holds for the construction of monotonic relative neighborhood graphs [26].

Our approach to graph diversification is to explicitly search for missing inverse links by querying for each point z from all its k_{nn} nearest neighbors $x_i \in \mathcal{N}_z^{nn}$ within a small search radius. Where a direct inverse link from x_i back to z exists, the query can trivially be skipped. When there is no path within the allowed range, the link is added (e.g., Figure 4). The searches are executed in parallel over all points $z \in \mathcal{X}$, allowing for faster construction.

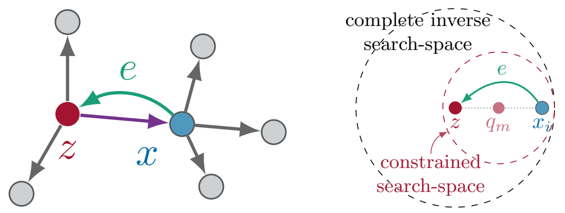


Fig. 4: Maintaining symmetric links. If there is no easily found connection from $x \in \mathcal{N}_z^{nn}$ back to $z \in \mathcal{X}$, the edge e is added to allow propagating nearest neighbor information between z and x. (right) Instead of considering the full search radius when probing whether to insert symmetric links, we constrain the maximum distance of points to be visited to the smaller sphere centered around the midpoint q_m.

The detailed symmetric linking operation is shown in Algorithm 5. Each point z launches queries for itself, starting from its k_{nn} nearest neighbors x_i. During the query, only the k_{nn} nearest neighbors of each visited point $\mathcal{N}_{x_i}^{nn}$ and potential already inserted symmetric/inverse links at that point $\mathcal{N}_{x_i}^{sym}$ are considered by the query as all others may still be overwritten. If a path back to z is found, the search continues with z's next neighbor. Otherwise, a

Algorithm 5: Symmetric linking query.

```
Function graph.sym_links($z$):
    Input: symmetric query point $z$
    for each $x_i \in \mathcal{N}_z^{nn}$:
        $cache$.init($z$, $\tau$)
        $cache$.fetch($x_i$)
        while $(a \leftarrow cache.\text{pop}()) \neq \emptyset$:
            $\mathbf{p} \leftarrow \mathcal{N}_a^{nn} \cup \mathcal{N}_a^{sym}$
            if $z \in \mathbf{p}$:
                skip $x_i$
            $cache$.fetch($\mathbf{p}$)
        for each $p \in cache.best$:
            if $\left|\mathcal{N}_p^{sym}\right| < k_{sym}$:
                $\mathcal{N}_p^{sym} \leftarrow \mathcal{N}_p^{sym} \cup \{z\}$
                break
```

link to z is inserted at the closest point to z encountered during the search which still has free capacities in its inverse link list.

To avoid race conditions on the insertions, we use atomics to keep track of the inverse link list sizes. On average, less than $k/4$ inverse links are necessary this way. If no candidate is found, the link is ignored. In our setting, this is, however, a very rare case and z might potentially still be reachable through either through hierarchy or multiple start points.

5.2.1 *Additional Symmetric Linking Constraint*

As demonstrated in Figure 4, the usual search-space would be defined depending on z and x_i. However, in order to generate a search structure that is able to find a path for all potential query points, we additionally constrain it on the worst-case query that still needs to find a path to z. Hence, a link is required to bridge the gap for a worst-case query on the midpoint q_m. Note, a path to x_i is given since it is directly known from z. For symmetric linking, only this search space is relevant, which is the circle defined by the center q_m and distance to x_i, but still encloses z. In practice, we set $q_m = z + 0.4 \cdot (x_i - z)$. This is slightly less conservative, but decreases the number of necessary symmetric links.

5.3 Graph Refinement

The initial graph construction might not always produce perfect results at the first trial. The reason for this is to some extent the large merge factor g that results when constructing search-graphs of low height that need to span over massive datasets.

In our experiments we found using a height of $L = 4$ for the hierarchical search-graph to provide the best performance. To create a single graph from the initially n/s disjoint sub-graphs consisting of single batches of size s (usually, $s = 32$), we need to merge groups of $g = \sqrt[L-1]{n/s}$ sub-graphs on each layer, where each new sub-graph's top-layer contains s points sampled from these g sub-graphs (see Section 5.1). While for $n = 256$, just 2 graphs need to be merged together each, for $n = 10^6$, about 32 graphs each need to be merged per layer. The higher g, the fewer entry points into the lower level exist, limiting the success rate of the initial hierarchical queries, as there are only s/g per merged sub-graph. For $g > s$, some points can only be reached after sub-graphs are bridged by symmetric linking.

To improve the graph quality, refinement steps, as described in Algorithm 6, can be performed which simply repeat the merging and symmetric linking steps throughout the search-graph. While refinement steps would also be beneficial during the actual sub-graph merging, we observed that performing them in the end is sufficient for the overall quality of the search structure.

Algorithm 6: Parallel graph refinement.

```
Function RefineGraph(graph, $l_{top}$):
    Input: search structure graph, top layer $l_{top}$
    Result: refined search structure graph
    for $l_m \leftarrow l_{top} - 1$ to $0$ :
        graph.merge($l_{top}$, $l_m$)
        graph.sym($l_m$)
```

5.4 Query Considerations

For our approach, during graph construction, it is crucial to perform a hierarchical query, which searches nearest neighbors layer by layer, due to the fact that the search index is not yet fully merged. Here, the coarse-to-fine methodology is able to bridge gaps between sub-graphs that are not yet connected and provides multiple routes to the area of interest. Starting from multiple spread-out points increases the quality significantly. In particular, if not only the 1-NN is of importance but also the neighbors at the far end of k. However, for a converged search-graph, this effort is no longer necessary as all points are interlinked well.

Searching in flat index structures does not have the hierarchical overhead. For example, Li *et al.* [21] start their query always from the centroid in a flat graph. Nevertheless, as the number of points grows, the number of hops to the centroid grows as well. In addition, with a single starting point, any goal point will be approached from just one direction. The search might be prone to gaps or linking problems inside the search-graph.

In our design, the bottom layer is actually a flat graph that has all points included. When searching the finished search-graph for nearest neighbors of novel points, it turned out to be more efficient in practice to skip the intermediate layers and continue the search directly on the bottom layer after exploring the s start points that make up the top layer. This allows to significantly cut back on the search iterations performed as otherwise the search would need to iterate on each intermediate layer. In practice, compared with the hierarchical approach, we did not observe any loss in recall but a significant saving in time.

5.5 Multi-GPU

The parallel construction and search algorithm can be extended to multiple GPUs. Johnson *et al.* [2] distinguish between two types of multi-GPU parallelism: *Replication* and *Sharding*. *Replication* copies the entire dataset $\mathcal{X}$ to multiple GPUs to parallelize the query process even further. The queries $\mathcal{Q}$ are divided equally among the available GPUs, resulting in linear speedups.

Sharding subdivides the dataset $\mathcal{X}$ into smaller, independent shards. This allows us to process billion-scale datasets which would otherwise not fit into GPU memory. When necessary, shards can also be swapped into main memory or onto disk. All GPUs construct the search-graphs for their shards in parallel. Since each shard has lower complexity due to the reduced number of points, super-linear speedups *w.r.t.* the same quality can be achieved.

The downside is that all shards need to be processed for each query. When querying multiple shards per GPU, we swap already processed shards for new ones in the background to hide memory latencies. Still, swapping shards puts a limit on the minimum run time per query batch. As some additional overhead, the individual query results per shard need to be merged. Within each GPU, we use a parallel radix sort. The final result across the GPUs is computed on the CPU using an efficient n-way merge.

6 Empirical Evaluation

In the following, we evaluate the performance of the proposed approach and its individual components on several publicly available benchmark datasets and report qualitative and quantitative results in terms of timings and accuracy.

Datasets. We ran experiments on datasets of varying sizes and dimensionality generated in different application contexts: SIFT1M [11] and SIFT1B [12] containing SIFT vectors of dimension 128, DEEP1B [5] with 1 billion 96-dimensional feature vectors encoding entire images, NyTimes [35], [36] and GloVe 200 [35], [37], which are two skewed and clustered datasets containig random projections of word embeddings compared using cosine-similarity, and finally the 960-dimensional dataset GIST [11]. Details on the datasets and construction details for our search graphs can be found in Table 4.

Hardware. We use a machine with 8x NVIDIA Tesla V100 and 2x Intel Xeon Gold 5218, where we usually use just a single GPU, except for SIFT1B and DEEP1B, where we use all 8 GPUs. We publish results for additional GPUs in the supplemental.

Performance Metrics. When comparing results to other approaches, one has to carefully look at the employed metric. The query performance is typically measured in recall (R@k). For quantization-based methods (e.g., [11]), the recall is understood as the ratio of queries which return the true first nearest neighbor among the first k results:

$$R@k = \frac{\left|\mathcal{N}_q^{\text{gt}}(1) \cap \mathcal{N}_q(k)\right|}{\left|\mathcal{N}_q^{\text{gt}}(1)\right|}. \tag{5}$$

For methods with exact distance computations – such as graph-based methods – the recall is instead understood as the overlap between the first k results and the first k true nearest neighbors. For disambiguation, we refer to it as consensus (C@k):

$$C@k = \frac{\left|\mathcal{N}_q^{\text{gt}}(k) \cap \mathcal{N}_q(k)\right|}{\left|\mathcal{N}_q^{\text{gt}}(k)\right|}. \tag{6}$$

As our algorithm uses exact distance calculations, the true nearest neighbor is either reported as the first element in an answer or not found at all. Therefore, we report only R@1 (=C@1).

Batch Size. The number of queries executed per batch is an important factor for GPU-based queries, which we analyze in Figure 8b. In our experiments, we measure the time for executing all 10000 queries (except 1000 queries for GIST [11]) in a batch and report the total execution time divided by the number of queries as µs/query or plot it as queries per second.

6.1 Performance Comparisons

Evaluation is performed on the default configuration GGNN, a faster, less accurate configuration GGNN‡, and a 8-GPU configuration GGNN^8 (see Table 4 for parameters).

As reference, we use several recent CPU-based methods [20], [21], [26], where we executed the single-threaded reference implementations on an Intel Core i7-9700K. We further compare against the GPU-based SONG [3] and mimic their approach of

TABLE 1: Million-scale performance comparison.

Approach	τ_q	SIFT1M [11] Query time µs/query ↓	Recall @1 ↑	Recall @10 ↑	Recall @100 ↑
LOPQ [13]		51 100	0.51	0.93	0.97
IVFPQ [11]		11 200	0.28	0.70	0.93
FLANN [10]		5320	0.97	-	-
HM-ANN [27]		855	0.9995	-	-
		236	0.99	-	-
PQT [1]		20	0.51	0.83	0.86
FAISS [2]		20	0.80	0.88	0.95
PQFPGA [9]		20	0.88	0.94	0.97
FANNG [8]		1.3	0.95	-	-
		0.8	0.90	-	-
GGNN	0.65	5.8	**0.9997**	-	-
	0.42	1.5	0.99	-	-
	0.30	0.7	0.95	-	-
	0.20	**0.5**	0.90	-	-
Brute Force (1x V100)		381.4	1.0	-	-

TABLE 2: Performance comparison on SIFT1B [12]. Our method GGNN^8 used 8 GPUs. RobustiQ [18] used 2 GPUs. Otherwise, single CPUs or GPUs were used. Our method achieves perfect recall@1. GGNN^8 is limited by the host to device memory bandwidth and could otherwise perform the query with 99% recall@1 in just 24.5µs/query (see Section 6.3.1). The smaller GGNN^{8‡} can be queried much faster, since it iterates over fewer shards and fits on the GPUs in one piece.

Approach	τ_q	SIFT1B [12] Query time µs/query ↓	Recall @1 ↑	Recall @10 ↑	Recall @100 ↑
LOPQ [13]		8000	-	0.20	-
IVFPQ [11]		74 000	0.08	0.37	0.73
Multi-D-ADC [15]		1603	0.33	0.80	0.98
HM-ANN [27]		1014	0.99	-	-
PQT [1]		150	0.14	0.35	0.57
FAISS [2]		17.7	-	0.37	-
PQFPGA [9]		20	-	0.55	-
RobustiQ [18]		33	0.33	0.76	0.90
GGNN^8	0.56	61.6	**1.0**	-	-
	0.50	46.4	0.999	-	-
	0.38	42.7	0.99	-	-
GGNN^{8‡}	0.61	7.1	0.99	-	-
	0.42	**3.9**	0.90	-	-
Brute Force (8x V100)		≈30 000	1.0	-	-

TABLE 3: Performance comparison on DEEP1B [5]. Our method GGNN^8 used 8 GPUs. FAISS [2] used 4 GPUs. RobustiQ [18] used 2 GPUs. Otherwise, single CPUs were used. GGNN^8 is limited by the host to device memory bandwidth to at least 233µs/query on the 10k queries set (see Section 6.3.1). To show the influence of τ_q and to compare with future hardware configurations, we report times without the currently necessary upload as GGNN^{8*}.

Approach	τ_q	DEEP1B [5] Query time µs/query ↓	Recall @1 ↑	Recall @10 ↑	Recall @100 ↑
Multi-D-ADC [15]		1065	0.37	0.74	0.92
HM-ANN [27]		1142	**0.99**	-	-
FAISS [2]		13.3	0.45	-	-
RobustiQ [18]		30	0.38	0.75	0.89
GGNN^{8*}	1.50	27.6	**0.99**	-	-
	0.44	8.3	0.95	-	-
	0.36	**7.2**	0.90	-	-

TABLE 4: Datasets and search graph construction details. For some datasets, we also demonstrate the construction-query trade-off with a faster construction version (‡). All configurations shown here are capable of reaching at least 99% recall@1.

Dataset			Search Graph Configuration						Construction Time and Graph Size			
Dataset	#Points	#Dim.	Config.	k	s	l	τ_b	r	GPU		Time	Size
SIFT1M [11]	1 000 000	128	GGNN‡	24	32	4	0.5	2	1x V100		9.2 s	95 MiB
			GGNN	40	32	4	0.5	2			21.8 s	158 MiB
NyTimes [35], [36]	290 000	256	GGNN‡	40	32	4	0.3	0			13.4 s	46 MiB
			GGNN	96	64	4	0.3	0			39.3 s	113 MiB
GloVe 200 [35], [37]	1 183 514	200	GGNN	96	64	4	0.5	2			6.0 min	450 MiB
GIST [11]	1 000 000	960	GGNN	96	64	4	0.4	2			11.9 min	382 MiB
SIFT1B [12]	1 000 000 000	128	GGNN^{8‡}	20	32	4	0.5	2	8x V100	16 shards	33.4 min	75 GiB
			GGNN^{8}	40	32	4	0.5	2		128 shards	78.6 min	152 GiB
DEEP1B [5]	1 000 000 000	96	GGNN^{8}	24	32	4	0.5	2		32 shards	47.5 min	90 GiB

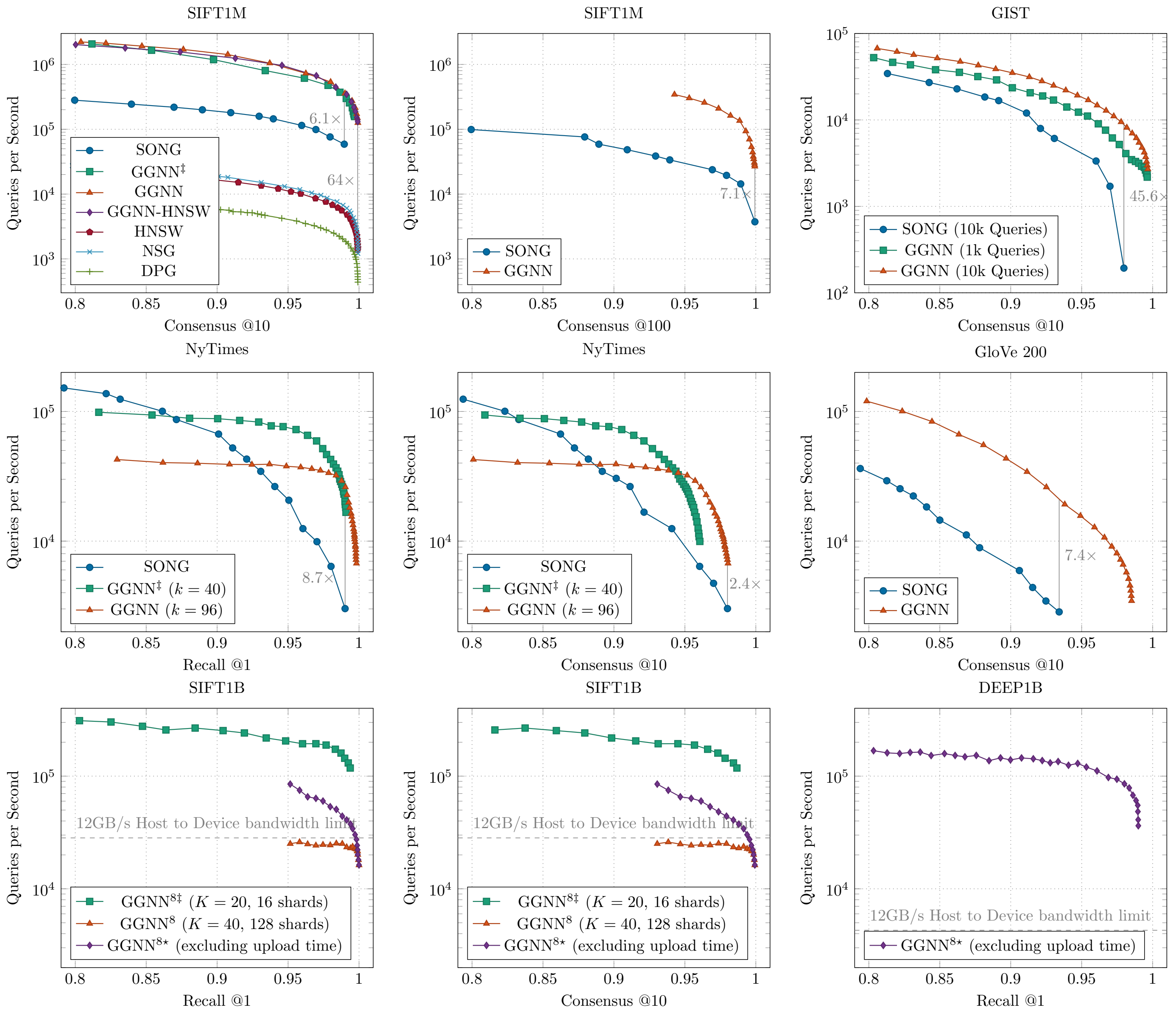


Fig. 5: Performance comparison across various datasets and top-k queries. Results in the top-right corner are preferable. We compare mainly against SONG [3], using the values from their paper and using the same GPU. On SIFT1M, we additionally compare against state-of-the-art single-core CPU-based methods [20], [21], [26] and include a configuration ”GGNN-HNSW”, where we use our method to query the graph built by HNSW [20], like SONG [3] does. Our query significantly outperforms SONG and achieves similar results as a query on a similarly-sized search-graph constructed in much less time using our method. High performance is also achieved when querying for the 100 nearest neighbors or more complicated datasets like GloVe 200 or very high dimensional datasets like GIST. On the rather small NyTimes dataset, our graph construction appears to be non-optimal but the highly efficient query still manages to outperform SONG by a large factor. Using 8 GPUs, even billion-scale datasets like SIFT1B or DEEP1B can be queried at rates around 100k queries per second with near-perfect recalls.

querying the bottom layer of a prebuilt HNSW [20] graph using our query as "GGNN-HNSW". Using our query is even faster. In addition, it performs very similar to querying a similarly sized graph constructed using our method in significantly less time. Detailed performance comparisons are shown in Figure 5 and Tables 1, 2, and 3. In the tables, we compare against previously published results, where GPU-based methods [1], [2], [8], [18] used NVIDIA GTX Titan X GPUs, and [9] used an Arria 10 GX1150 FPGA. Besides being significantly faster on all datasets (around 1µs per query on SIFT1M), our method can also achieve very high recall rates, close to perfect.

6.2 Search-Graph Construction

In the following, we inspect several aspects of the construction process, including the time spent on the individual construction steps, the influence of the dataset size, the resulting graph quality in terms of nearest neighbors per point, and how additional construction time can be traded in for even faster queries.

6.2.1 Construction Time

Construction times and index sizes of our method are listed in Table 4. For comparison, FAISS [2] report construction times between 4 and 24 hours for DEEP1B, yet reaches less than 50% R@1. HM-ANN [27] report construction times around 100 hours for effective billion-scale search-graphs. Using our hierarchical GPU-based graph-merge algorithm, 99% R@1 on DEEP1B and even perfect 100% R@1 SIFT1B can be reached with fast queries while the construction takes less than 2 hours.

6.2.2 Construction Time Composition

Figure 6 shows the time spent on the individual graph-construction operations with 2 refinement iterations. The majority of the construction time is spent on the `merge` operation, especially those merges which involve the bottom layer, where the merge operation searches for the k nearest neighbors for all points in the dataset.

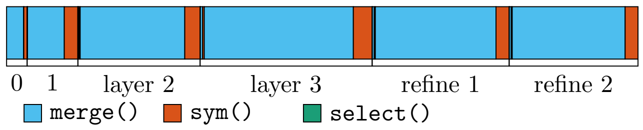


Fig. 6: Split-up of the construction time for the SIFT1M search graph. Most time is spent on the `merge` operation (86.92%), some time on the `sym` operation (13.06%) and almost no time on `select`ing the points for the upper layers (0.02%).

6.2.3 Dataset Size

Efficient graph construction needs to scale well with the size of the dataset. As demonstrated in Figure 7, the construction time scales almost linearly with the size of the dataset, $\approx O(n^{1.077})$. This can be explained by most of the construction time being spent on independently determining the neighbors of each bottom-level point (see Section 6.2.2). This almost linear behavior is significantly better than the complexity of brute-force kNN construction $O(n^2)$.

6.2.4 For-all Queries

The fast construction of the kNN search graph from scratch solves another interesting problem on the side, namely computing the k nearest neighbors for all points in the dataset, as frequently encountered in n-body problems. When we construct the full hierarchy, the final merging step effectively searches for the k neighbors of all points. In Table 5, we demonstrate the quality of the for-all problem considering the consensus. Referring to Figure 7, this for-all problem again shows almost a linear behavior. In case of sharding, though, all points would need to be tested once against every shard. As the number of shards linearly depends on the number of points one approaches $O(n^2)$ complexity again with a tiny constant though, e.g. 16 shards for the SIFT1B dataset.

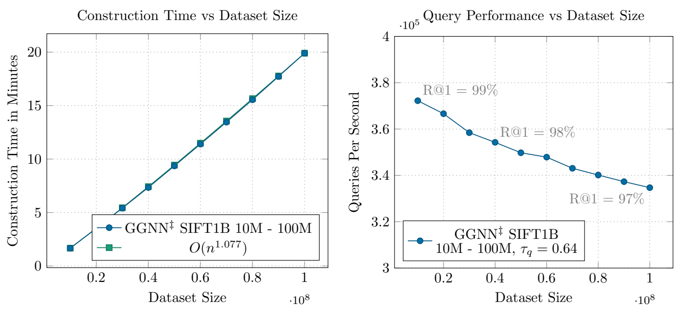


Fig. 7: Construction times and queries per second for increasing subsets of SIFT1B (up to 100M) on a single GPU. The construction time almost linearly depends on the dataset size. With increasing dataset size, the query performance suffers slightly in both execution time and accuracy for a constant slack factor τ_q.

TABLE 5: Accuracy of the nearest neighbors for each point in the dataset, as contained within our search graph.

Method	Dataset	C@1 ↑	C@10 ↑
GGNN‡	SIFT1M	0.996	0.981
GGNN		0.999	0.998

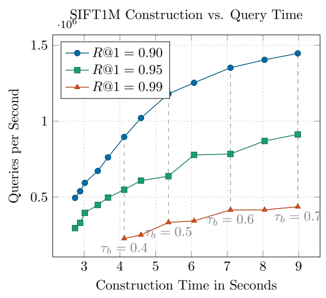
6.2.5 The Construction-Query Trade-off

Our graph construction algorithm and the query can be tuned for speed or accuracy. The build time is controlled by the slack factor τ_b and by how many refinement iterations r are carried out. The longer the construction time, the more precise the resulting graph. A quickly assembled graph will have worse edges and a query might need to visit more nodes until fulfilling the stopping criterion. In Figure 8a, the trade-off between build time and query time is visualized for fixed recall rates.

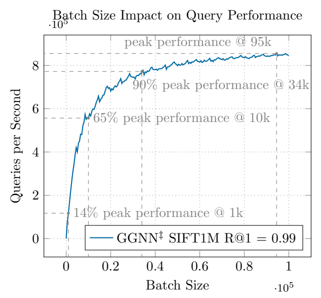
6.3 Query Behaviour

The runtime and the quality of a query can be controlled by adapting the stopping criterion through the slack factor τ_q. As can be seen e.g. in Table 1, by increasing τ_q, a larger safety margin is considered during query, resulting in better recall rates at the cost of visiting more points and consequently longer query time.

It is instructive to look at the behaviour of the query over time. In Figure 9, the initial distance of the query to the starting point, i.e. the closest point on the top layer, is drastically reduced already after very few iterations. Here, an iteration stands for fetching the best point from the priority queue and calculating the distance to all its neighbors. Most time is spent to carefully explore the neighborhood around the true nearest neighbor. Quickly after locating the best candidates, the query is terminated by the stopping criterion to prevent unnecessary iterations. This effect does not only show up in the small subset of plotted query distances, but also statistically in the histogram over all queries.

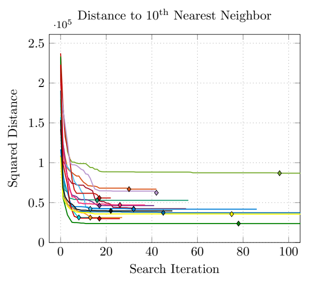


(a) Varying construction time.

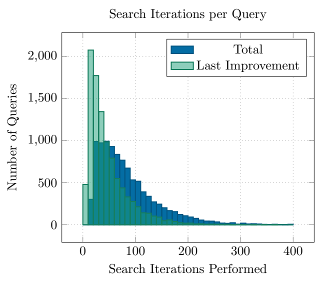


(b) Varying batch size.

Fig. 8: Query performance depends both on the quality of the search-graph and the number of queries executed simultaneously. Queries with the same accuracy can be executed faster if more time is spent on construction (a). Here, no refinement is performed ($r = 0$) to visualize the influence of τ_b on the initial construction. Generally, having a few refinement iterations can further boost query performance. In terms of query size (b), high performance is already reached for a few thousand queries, while further simultaneous queries yield even better performance with a peak at 95k queries.



(a) Development of the 10th best query point as the query progresses.



(b) Iterations performed total, and until the last improvement is found.

Fig. 9: Looking at the behavior of several queries (a), one can see that the general area is found very quickly by the search graph. Afterwards, the local neighborhood is explored and shortly after finding the last improvement (indicated by the marker), the query is terminated by the stopping criterion. The effectiveness of the stopping criterion also shows in the statistics (b). The distribution of total iterations closely follows the distribution of iterations necessary to make the final improvement, only lagging behind by a handful of iterations.

6.3.1 Out of GPU-memory Queries

When querying on billion-scale datasets, the amount of necessary GPU-memory will quickly exceed the available memory even on an 8 NVIDIA Tesla V100 system. While SIFT1B together with the low-profile graph GGNN^{8‡} still fits on the GPUs, with the GGNN^{8} graph, it exceeds the available memory by 2 shards (4.2 GiB). On DEEP1B, the dataset alone already surpasses the available memory, and with GGNN^{8} an additional 28 GiB per GPU need to be loaded per query pass. This results in a bound that is defined by the host to device memory throughput ($\approx$ 12 GiB/s [38]), *i.e.*, for 10k queries, this results in a minimum of 35µs/query on SIFT1B and 233µs/query on DEEP1B. These bounds are shown in Figure 5. Note that this loading time can be hidden completely by performing an overall costlier query, e.g. using more queries, increased τ_q, or larger caches. For easy comparison with future approaches on devices with sufficient memory, we also report the computation time without the memory transfers.

7 Conclusion

Our proposed parallel GPU-based search algorithm is surpassing all state-of-the-art algorithms in terms of speed by a large factor and represents the fastest ANN query technique while maintaining recall rates above 99%. The efficient query algorithm further accelerates the parallel construction and merging of sub-graphs. Our hierarchical construction scheme creates high-quality kNN graphs with graph-diversification links for very efficient traversal. It is easily deployed in multi-GPU systems to cope with very large-scale datasets. It is the first ANN search method capable of constructing effective search structures (R@1 $\geq$ 0.99) for billion-scale datasets in less than an hour. For million-scale datasets, search-graphs of sufficient quality can often be constructed within seconds, allowing for quick nearest-neighbor searches also on intermediate data structures as a part of larger GPU-based algorithms, e.g. in machine learning applications.

Currently, our method computes the exact distance to all visited high-dimensional points. As the number of distance calculations dominates the query time, a compressed representation of the data vectors could lead to further acceleration.

Acknowledgments

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645 and SFB 1233, TP 02 – Project number 276693517. It was supported by the German Federal Ministry of Education and Research (BMBF): Tübingen AI Center, FKZ: 01IS18039A.

Appendix A
Hierarchical Construction Example

To visualize the individual steps in the construction of the hierarchy, Figure 10 shows an example with growth rate $g = 3$ where layer 1 has already been constructed by merging 9 bottom layer batches into 3 sub-graphs of height 2. To merge the individual sub-graphs further, first, s points are selected equally from the current top layers of the g sub-graphs, creating a new top layer (`select`). Starting from the top, layer by layer, a hierarchical query for the k nearest neighbors of all points in the current layer is performed which can access all g sub-graphs via the new top layer. (On the new top layer, the k nearest neighbors can again be determined efficiently using brute-force.) The kNN lists within the layer are then updated using the result of these queries (`merge`). Afterwards, symmetric links are inserted (`sym`). Once all layers are processed, the g sub-graphs form one consistent search-graph, where each point can potentially reach all others.

Appendix B
Parallelization of Graph-based methods

Graph-based methods receive impressive recall numbers even for high-dimensional datasets, e.g. HNSW [20]. Nonetheless, there is a caveat to it. Every comparison between two points needs to load the complete vector from memory. Therefore, graph-based methods require a huge memory throughput, that gets especially prominent when considering parallelization. In contrast, quantization-based methods are already utilizing the huge advantages in memory throughput offered by GPUs, *e.g.*, 900

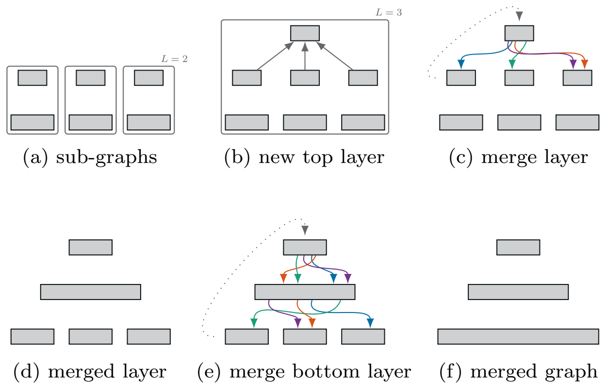


Fig. 10: Bottom-up-top-down construction of our graph hierarchy. a) Disjoint batches/sub-graphs provide a few sampled points which are fused into a common new top layer b). For the top layer, a symmetric kNN-graph is constructed by brute force. c) The next finer layer is fused by carrying out hierarchical kNN-searches from the top which will find and link neighborhoods across sub-graphs, d) establishing one consistent kNN-layer. This process of e) finding the nearest neighbors for all points in a layer and f) fusing this layer across the sub-graphs is recursively applied until all initial shards are merged.

GB/s NVIDIA Tesla V100 vs. 76.8 GB/s Intel Xeon E5-2650 v4. Figure 11 demonstrates the limits of CPU-based parallelization approaches. Hence, our work focuses on enabling graph-based kNN structures on GPUs, where we do not have the luxury of cheap global or sequential update steps, excessive resources, nor simple dynamic neighborhood lists.

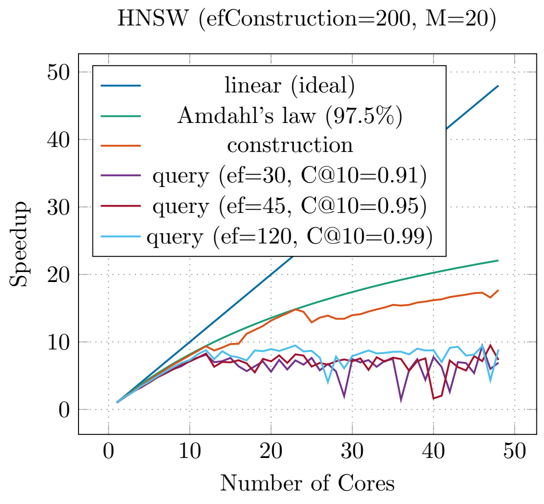


Fig. 11: Parallelization of the HNSW algorithm on two Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (2x12 cores with 24 threads each). First, all 12 physical cores on the first CPU were used, then the 12 cores of the second CPU were added. Finally, the logical cores were added. Speedup is computed as the execution time on a single core divided by the parallelized time. The trendline using Amdahl's law has been fitted optimistically.

Appendix C
Choosing Parameters

Our method can be tuned for optimal performance using various parameters. In the following, we provide guidelines on choosing good parameters for the construction and query process.

C.1 Choosing Search-Graph Construction Parameters

Our search graph construction process can be tuned by the following parameters:

$k = |\mathcal{N}_x|$ selects the number of neighbors, i.e., outgoing links, per point in the graph and is best kept low to prevent over-exploration during query. For highly skewed or very high-dimensional datasets, large neighborhoods seem necessary to bridge gaps and achieve high recalls.

s is the segment size in which points are grouped and nearest neighbors are determined in a brute-force fashion to fill the initial neighbor lists. As a result, $s > k/2$ is a requirement. Simultaneously, it is also the size of the top-layer which is used as the set of starting points for each query. Using multiples of 32 makes good use of the GPU's resources which always executes 32 threads simultaneously in a *warp*.

l chooses the number of layers in the hierarchy. Larger hierarchies can bridge wider gaps in the dataset more easily, but don't help much in the case of individual outliers. The larger l, the longer the construction will take. Empirically, we found $l = 4$ to be the ideal choice for most datasets.

r chooses the number of refinement iterations which repeat the merging and symmetric linking steps performed during construction. For most datasets, using $r = 2$ can significantly boost query performance, while additional refinements do not result in further improvements of query performance. For the NyTimes dataset, we did not observe any improvement, most likely due to its comparably small size.

C.1.1 Graph Size

In some applications, the size of the search graph can be a limiting factor. Especially for larger datasets, it can become problematic to fit the dataset and the search graph onto the GPU. In such cases, we resort to conventional sharding.

To determine the precise memory requirements, one must first determine the exact integer growth factor g and the resulting number of points on the upper layers N_{upper}. There are $N_{\text{upper}} = s \sum_{i=0}^{l-2} g^i$ points in the upper layers of the hierarchy where the growth factor $g \approx \sqrt[l-1]{\frac{N}{s}}$ may be rounded up or down such that the bottom-level segment size $s_0 \approx \frac{N}{g^{l-1}}$ is closest to s except to ensure that $s_0 > k$.

To determine the memory required by the search-graph, one can then simply sum up the following:

- $(N + N_{\text{upper}}) \cdot k \cdot 4$ Bytes for the graph's edge list.
- $2 \cdot N_{\text{upper}} \cdot 4$ Bytes for translating higher-level indices into the level below and into the base-level indices.
- $2 \cdot 4$ Bytes for $d^{+}_{\text{nn}_1}$ and $\bar{d}_{\text{nn}_1}$.

As an example, for SIFT1M with $k = 24$, the graph takes the following amount of memory: ($g = 32$)

$$(1033824 \cdot 24 + 2 \cdot 33824 + 2) \cdot 4\,\texttt{B} \approx 94.9\,\texttt{MiB}$$

During construction, additional temporary memory is required:

- $N \cdot k \cdot 4$ Bytes for temporary edge lists.
- $N \cdot 4$ Bytes for random numbers used for selection.
- $N \cdot (k_{sym} + 1) \cdot 4$ Bytes for temporary inverse link lists.
- $N \cdot 4$ Bytes for the nearest neighbor distance per point.
- temporary storage for reduction using CUB

TABLE 6: Query configurations for the NVIDIA Tesla V100 (CUDA compute capability 7.0).

Dataset	*best*	*prioq*	*visited*	max. iter.	block size	registers	shared memory	occupancy
SIFT1M	10	54	384	400	32	56	2200‡ / 2232	0.5
	100	28	896	1000	64	48	4796	0.625
NyTimes	10	22	2016	2000	128	51	8536‡ / 8760	0.5625
GloVe 200	10	22	2016	2000	128	51	8760	0.5625
GIST	10	22	2016	2000	128	56	8740	0.5625
SIFT1B	10	54	384	400	32	56	2200‡ / 2232	0.5
DEEP1B							2200	

As not all of these are used simultaneously, about $N \cdot (k+1) \cdot 4\,\texttt{Bytes}$ suffice in practice, which is roughly as much as is needed for the graph itself.

C.2 Choosing Query Parameters

To make the most use of the GPU's resources, the parameters of the query kernel can be carefully tuned. The query uses *registers* to cache the query vector for efficient distance computations, whereas our cache structure resides in *shared memory*.

As discussed in the paper, we can afford to use $best$ lists as short as the number of neighbors to be searched for. However, we always search for at least 10 neighbors. This comes at negligible additional cost for queries for fewer neighbors but keeps the stopping criterion stable.

When aiming for high accuracy, it turned out that large $visited$ lists are of utmost importance, as most time is spent to carefully explore the local neighborhood of a query. Being able to enumerate each visited point makes sure that the query does not perform any cycles. We also limit the maximum number of iterations performed by the query such that the visited list is not exceeded by more than a few points at most.

Concerning the $prioq$, we found that using larger priority queues is only beneficial up to a point and often very short priority queues perform surprisingly well, allowing us to only cache as little as 32 or 64 distances in our queries.

Finally, the *block size* can be tuned to achieve higher *occupancy* on the GPU when more registers are needed, e.g., for high-dimensional datasets, or more shared memory, when using a large cache. The ideal values will vary somewhat depending on the used GPU but small block sizes (e.g., 32) perform better during the reduction-based distance computation, especially for lower-dimensional (e.g., 128) datasets. For more costly queries, however, larger block sizes may be necessary to achieve higher occupancy which allows to hide latencies when accessing data in the GPU's main memory.

C.2.1 Used Query Parameters

The detailed query parameters used for evaluating the performance of our method on each dataset can be found in Table 6. This includes the GPU-specific register and shared memory usage, as well as the resulting occupation on the device.

We use the same parameters across the more complicated datasets. For SIFT1M, a cheaper query suffices. The query for GIST needs a few more registers to store the high-dimensional query vector. The queries for NyTimes and GloVe 200 have some shared memory overhead due to the cosine-similarity distance metric. There is almost no overhead due to the dataset size, so even the queries on SIFT1B and DEEP1B can use the same configuration as the query on SIFT1M with the same resulting register and shared memory usage. (There is a minor overhead when dealing with billion-scale datasets due to the necessity of 64-bit indices when addressing base-vectors or neighbors in the graph.) In all cases, our query reaches an occupancy of at least 50%, which allows to effectively hide memory-access latencies.

Appendix D
Performance on Different GPUs

In the main paper, we only show results using the NVIDIA Tesla V100, which is a high-end datacenter GPU. Figure 12 demonstrates the performance of our method on a wide range of GPUs, including the low-end GT 1030. All GPUs used the same query configuration shown in Table 6. On the GT 1030 and GTX 1080 Ti, increasing the blocksize to 64 resulted in slightly faster queries.

On all GPUs, our method performs equally well in terms of quality but the number of queries per second depends on the memory bandwidth and computational power that the GPU is able to provide. E.g., with about 5% of the V100's memory bandwidth, the GT 1030 is also only able to provide about 5% of the query performance. The construction time is similarly impacted, as shown in Table 7, which also shows the memory bandwidth of the compared GPUs.

Finally, the consistent query quality across the GPUs also demonstrates the stability of the search-graph construction process, despite its largely independent construction and random selection steps. While the shown results are each based on different, newly constructed search-graphs, their consensus at the fixed slack factor (τ_q) intervals is almost identical.

TABLE 7: Construction times for the GGNN search-graph for SIFT1M on various GPUs.

GPU	Memory Bandwidth	Construction Time
RTX 3090	936 GB/s	21.7 s
Tesla V100	900 GB/s	21.8 s
Titan RTX	672 GB/s	26.7 s
RTX 2080 Ti	616 GB/s	29.3 s
GTX 1080 Ti	484 GB/s	59.1 s
GTX Titan X	336 GB/s	102.1 s
GT 1030	48 GB/s	586.8 s

Appendix E
Query Starting Points

Every graph based search needs to start on at least one point in the graph. Our algorithm uses all the points at the top layer as **starting points**. These points are well connected in the graph due to the construction design, which is of particular interest during construction itself. However, In a well converged flat search-graph,

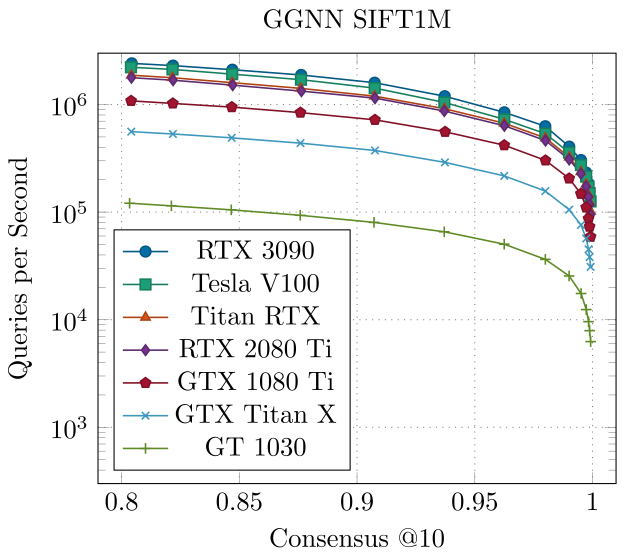


Fig. 12: Performance comparison across different GPUs.

the actual starting points are not that important in terms of quality. The option to start from the **centroid** point is proposed by Li *et al*. [21]. We conducted an experiment to compare the different methods. Additionally, we also test the performance when a search starts at the furthest point from the centroid, which is denoted as **worst**. The centroid or worst point is cheaply determined during construction with no measurable overhead. The experiment is performed on a lower spec NVIDIA GTX Titan X GPU, compared to the main paper. Table 8 shows the results for the 1, 2, and 5 million point subsets of the SIFT1B dataset. While c@10 is on par for all query start point options, we see some improvements in terms of speedup when using our **starting points**.

TABLE 8: Comparison of query start options for different slack factors τ and subsets of the SIFT1B. Execution times are evaluated on an NVIDIA GTX Titan X GPU.

τ	starting points		centroid point		worst point		SIFT1B
	c@10	us/query	c@10	us/query	c@10	us/query	subset
0.6	0.983	7.4	0.983	8.5	0.983	8.5	1M
	0.978	8.1	0.978	9.3	0.978	9.3	2M
	0.969	8.8	0.968	10.3	0.969	10.3	5M
0.7	0.993	12.2	0.994	13.7	0.993	13.7	1M
	0.990	13.2	0.990	14.9	0.990	14.8	2M
	0.985	14.0	0.984	16.1	0.985	17.0	5M

Appendix F
Impact of Partitioning

In our experiments, we are using all datasets as they are, i.e. without any argumentation or pre-analysis. All points in the dataset are read in as they are laid out in the files (*constant*). Our algorithm splits data into multiple different partitions throughout the construction process. When using multi-GPU variants, further partitions are introduced by sharding. Table 9 demonstrates the effect of constructing the search-graph on a randomly *shuffled* dataset to mix up the points grouped together in each otherwise deterministic partition. The experiment is conducted on on the 1 million subset of SIFT1B with 4 shards. The mean μ and standard deviation σ over the consensus@10 are computed across 100 iterations. While σ is increased in the *shuffled* case, μ is on par with the constant read in, which is expected due to our randomized selection process for upper-level points.

TABLE 9: Analysis of partitioning impact: Mean μ and standard deviation σ are computed on the 1M subset of SIFT1B with 4 shards over 100 iterations.

τ	c@10		
	μ	σ	
0.6	0.9956	1.8e-5	constant
	0.9959	2.0e-4	shuffled
0.7	0.9989	1.4e-5	constant
	0.9988	9.5e-5	shuffled

Appendix G
Impact of the Symmetric Linking Constraint and Refinement

To analyze the impact of our proposed symmetric linking constraint introduced in Section 5.2.1 and search-graph refinement introduced in Section 5.3, we inspect the query performance on SIFT1M for several configurations.

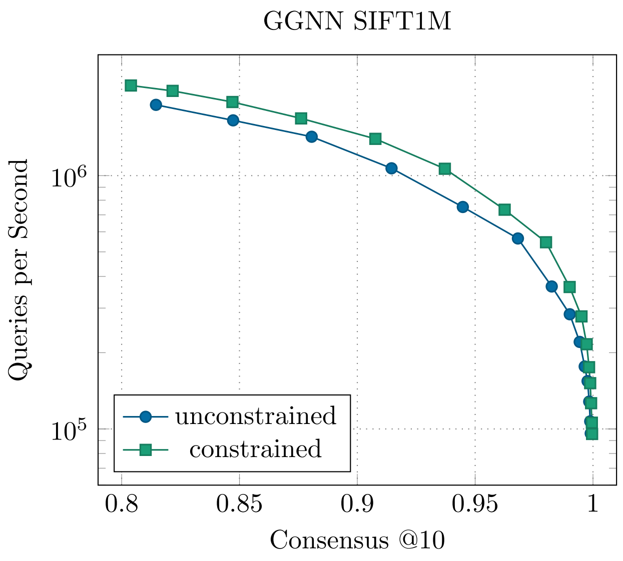


Fig. 13: Comparing query performance using symmetric linking without and with the additional constraint introduced in Section 5.2.1.

We inspect query performance without and with the additional symmetric linking constraint introduced in section 5.2.1 in Figure 13. Search-graph construction took 21.2 seconds without the constraint and 22.1 seconds with the constraint. Without the additional constraint, symmetric linking took about 250 ms more, but merging took about 1.1 seconds less construction time.

These results confirm that constraining the “symmetric” query for reverse paths from known nearest neighbors results in the addition of better links for graph diversification. With the additional constraint in place, following queries explore a richer set of candidate points, which takes slightly more time to explore but also results in increased accuracy. This is part of what allows our method to achieve (close to) perfect recall rates even on billion-scale datasets. At the same time, more queries can be processed per second at the same accuracy.

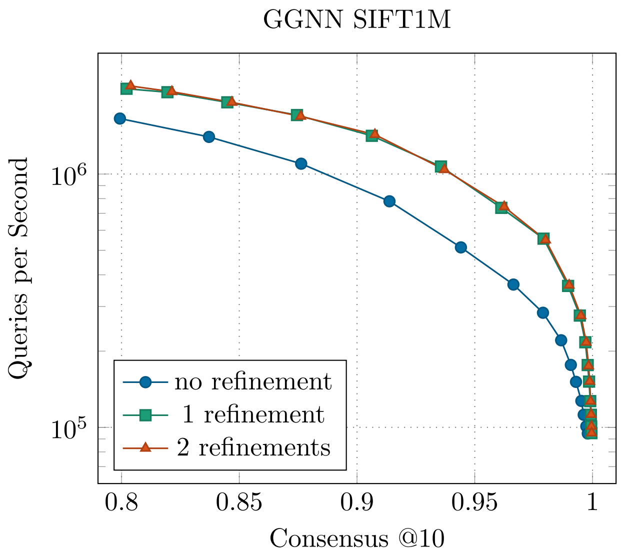


Fig. 14: Query performance at different levels of refinement.

When inspecting the query performance at different levels of refinement as shown in Figure 14, we see that query performance is increased significantly after the first refinement iteration. Refining the search-graph a second time can further boost query performance by a small amount in terms of both speed and accuracy. Further refinements result in no noticeable changes in query performance. Therefore, we generally perform two refinement iterations during search-graph construction to achieve optimal query performance while keeping construction time to a minimum.

References

[1] 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 (CVPR)*, 2016, pp. 2027–2035. 1, 2, 8, 10

[2] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019. 1, 2, 3, 7, 8, 10

[3] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1033–1044. 1, 3, 8, 9

[4] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, p. 194–205. 1

[5] A. Babenko and V. S. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2055–2063, 2016. 1, 8, 9

[6] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-based systems*, vol. 46, pp. 109–132, 2013. 1

[7] W. Chen, J. Chen, F. Zou, Y.-F. Li, P. Lu, Q. Wang, and W. Zhao, "Vector and line quantization for billion-scale similarity search on gpus," *Future Generation Computer Systems*, vol. 99, p. 295–307, Oct 2019. 1

[8] B. Harwood and T. Drummond, "Fanng: Fast approximate nearest neighbour graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 5713–5722. 1, 2, 3, 5, 6, 8, 10

[9] J. Zhang, S. Khoram, and J. Li, "Efficient large-scale approximate nearest neighbor search on opencl fpga," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4924–4932. 1, 8, 10

[10] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 36, 2014. 1, 2, 8

[11] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search." *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 33, no. 1, pp. 117–128, 2011. 1, 2, 8, 9

[12] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011, pp. 861–864. 1, 2, 8, 9

[13] 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 (CVPR)*. Columbus, Ohio: IEEE, June 2014. 1, 2, 8

[14] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, June 2013. 1, 2

[15] A. Babenko and V. S. Lempitsky, "The inverted multi-index," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2012, pp. 3069–3076. 1, 2, 8

[16] A. Babenko and V. Lempitsky, "Tree quantization for large-scale similarity search and classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 4240–4248. 1, 2

[17] A. Babenko and V. Lempitsky, "Additive quantization for extreme vector compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 931–938. 1, 2

[18] 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 International Conference on Multimedia Retrieval (ICMR)*, 2019, pp. 132–140. 1, 2, 8, 10

[19] Y. Matsui, T. Yamasaki, and K. Aizawa, "Pqtable: Fast exact asymmetric distance neighbor search for product quantization using hash tables," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1940–1948, 2015. 1, 2

[20] 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. 1, 3, 5, 6, 8, 9, 10, 11

[21] 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*, 2019. 1, 5, 7, 8, 9, 14

[22] 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*. ACM, 2011, pp. 577–586. 1, 2, 3

[23] J. Chen, H.-r. Fang, and Y. Saad, "Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection," *Journal of Machine Learning Research*, vol. 10, no. Sep, pp. 1989–2012, 2009. 1, 2

[24] T. Warashina, K. Aoyama, H. Sawada, and T. Hattori, "Efficient k-nearest neighbor graph construction using mapreduce for large-scale data sets," *IEICE Transactions*, vol. 97-D, pp. 3142–3154, 2014. 1, 2

[25] C. Fu and D. Cai, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv preprint arXiv:1609.07228*, 2016. 1, 2

[26] 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, pp. 461–474, 2019. 1, 2, 3, 5, 6, 8, 9

[27] J. Ren, M. Zhang, and D. Li, "Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, November 2020. 1, 2, 3, 8, 10

[28] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sep. 1977. 2

[29] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04. New York, NY, USA: ACM, 2004, pp. 253–262. 2

[30] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, p. 117, Jan. 2008. 2

[31] S. Korman and S. Avidan, "Coherency Sensitive Hashing," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1607–1614, Nov. 2011. 2

[32] D. S. Kim and N. B. Shroff, "Quantization based on a novel sample-adaptive product quantizer (sapq)." *IEEE Transactions on Information Theory*, vol. 45, no. 7, pp. 2306–2320, 1999. 2

[33] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Transactions on Communications*, vol. 28, no. 1, pp. 84–95, January 1980. 2

[34] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Information Processing Letters*, vol. 97, no. 5, pp. 181 – 185, 2006. 6

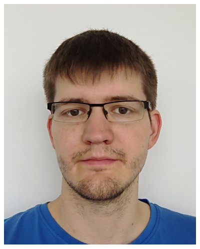
[35] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," 2018. 8, 9

[36] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml 8, 9

[37] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. 8, 9

[38] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018. 11

Fabian Groh received his diploma in computer science in 2013 from Ulm University, Germany. He is working towards a PhD degree at the computer graphics group of the University of Tübingen. His primary research topics are processing and learning on unstructured data in large-scale settings by utilizing GPU-based designs. In 2020 he joined Amazon Tübingen.



Lukas Ruppert received his master of science in computer science in 2019 from University of Tübingen. He is working on his Ph.D. in computer graphics with a focus on physically based rendering.

Patrick Wieschollek received his master of science in 2014 from Friedrich-Schiller University, Germany. He wrote his PhD thesis at the computer graphics group of the University of Tübingen and Max Planck Institute for Intelligent Systems in Tübingen. His primary research topic is developing data-driven approaches for multi-frame and unstructured data in large-scale settings. In 2019 he joined Amazon Tübingen.

Hendrik Lensch Hendrik P. A. Lensch holds the chair for computer graphics at Tübingen University. He received his diploma in computers science from the University of Erlangen in 1999. He worked as a research associate at the computer graphics group at the Max-Planck-Institut für Informatik in Saarbrücken, Germany, and received his PhD from Saarland University in 2003. Hendrik Lensch spent two years (2004-2006) as a visiting assistant professor at Stanford University, USA. From 2009 to 2011 he was a full professor at the Institute for Media Informatics at Ulm University, Germany. In his career, he received the Eurographics Young Researcher Award 2005 and was awarded an Emmy Noether Fellowship by the German Research Foundation (DFG) in 2007. His research interests include 3D appearance acquisition, computational photography, machine learning, global illumination and image-based rendering, and massively parallel programming.