Fast Graph Vector Search via Hardware Acceleration and Delayed-Synchronization Traversal

Wenqi Jiang
Systems Group, ETH Zurich
wenqi.jiang@inf.ethz.ch

Hang Hu
Systems Group, ETH Zurich
hanghu@student.ethz.ch

Torsten Hoefler
SPCL, ETH Zurich
torsten.hoefler@inf.ethz.ch

Gustavo Alonso
Systems Group, ETH Zurich
alonso@inf.ethz.ch

ABSTRACT

Vector search systems are indispensable in large language model (LLM) serving, search engines, and recommender systems, where minimizing online search latency is essential. Among various algorithms, graph-based vector search (GVS) is particularly popular due to its high search performance and quality. However, reducing GVS latency by intra-query parallelization remains challenging due to limitations imposed by both existing hardware architectures (CPUs and GPUs) and the inherent difficulty of parallelizing graph traversals. To efficiently serve low-latency GVS, we co-design hardware and algorithm by proposing Falcon and Delayed-Synchronization Traversal (DST). Falcon is a hardware GVS accelerator that implements efficient GVS operators, pipelines these operators, and reduces memory accesses by tracking search states with an on-chip Bloom filter. DST is an efficient graph traversal algorithm that simultaneously improves search performance and quality by relaxing traversal orders to maximize accelerator utilization. Evaluation across various graphs and datasets shows that Falcon, prototyped on FPGAs, together with DST, achieves up to 4.3× and 19.5× lower latency and up to 8.0× and 26.9× improvements in energy efficiency over CPU- and GPU-based GVS systems.

PVLDB Reference Format:
Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. Fast Graph Vector Search via Hardware Acceleration and Delayed-Synchronization Traversal. PVLDB, 18(11): 3797 - 3811, 2025.
doi:10.14778/3749646.3749655

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fpgasystems/Falcon-accelerate-graph-vector-search.

1 INTRODUCTION

Vector search is essential in large language model (LLM) serving systems [16, 37, 62], recommender systems [23, 87], and search engines [18, 54, 98]. Upon receiving a query vector, a vector search system retrieves the most similar vectors from a database approximately, a process known as approximate nearest neighbor (ANN) search. For example, search engines represent web pages as database vectors, and user's textual queries are encoded as query vectors [18, 40, 54, 55, 98]. Similarly, recommender systems identify advertisements that are potentially appealing to users by searching through encoded advertisement vectors [23, 87]. More recently, LLM systems have also adopted ANN search to improve content generation quality by retrieving reliable textual knowledge, an approach known as *Retrieval-Augmented Generation (RAG)* [16, 37, 62].

Among various ANN search algorithms, *graph-based vector search (GVS)* algorithms are particularly popular due to their high search performance and quality [28, 63, 73], with the latter measured by recall, the percentage of true nearest neighbors correctly identified by the search. The key idea of GVS is to construct a proximity graph on database vectors: each vector is a node, and similar vectors are linked by edges. During a search, the query vector is compared to a subset of database vectors by iteratively traversing the graph using best-first-search (BFS), which greedily selects the best candidate node to evaluate for each search iteration.

Given the rising adoption of ANN search in online systems, an ideal GVS system should *achieve low search latency for real-time query batches*, while being cost- and energy-efficient. However, reducing GVS latency remains challenging due to limitations imposed by common hardware architectures (CPUs and GPUs) and the inherent difficulty of parallelizing graph traversals. First, CPUs and GPUs operate on a time-multiplexed basis, executing GVS operations sequentially, thus leading to accumulated latency across traversal iterations. Second, if intra-query parallelization is employed, the synchronization overhead among CPU cores or GPU streaming multi-processors [58, 105] is disproportionately high relative to a single iteration of graph traversal, which typically takes only microseconds and involves just dozens of distance computations.

While previous research has explored hardware accelerator designs for GVS based on FPGA prototyping [80, 103], these approaches have three main limitations. Firstly, they only support the Hierarchical Navigable Small World (HNSW) graph. While HNSW is widely used today, more efficient graph construction algorithms are emerging that offer improved recall [28, 70, 72, 73, 81, 107, 111]. Secondly, directly implementing the software-oriented BFS algorithm on these accelerators results in sub-optimal search latency, because it significantly under-utilizes the accelerators, as we will further explain in conjunction with the hardware designs. Thirdly, existing architectures are mainly throughput-oriented and either do not support [80] or suboptimally support intra-query parallelism for low-latency search [103].

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749655

Table 1: Comparison between Falcon + DST and vector search on existing hardware platforms, including CPUs, GPUs, and prior FPGA-based accelerators. ✓ and × indicate supported and unsupported features, respectively.

Feature	CPU [28, 72, 73]	GPU [34, 106]	Prior FPGA [80, 103]	Falcon + DST (ours)
Minimize Memory Access (§3.2)	×	×	×	✓
Intra-query Parallelism (§3.3)	×	×	×	✓
Support Various Graphs (§3.4)	✓	×	×	✓
Hardware-Efficient Traversal Algorithm (§4.3)	×	×	×	✓
Latency	Moderate	High	Low to Moderate	**Low**
Throughput	High	**Very High**	High	High
Throughput / Bandwidth	High	High	High	**High**
Energy Efficiency	Low	Low	Moderate	**High**

To achieve low-latency GVS while supporting various graphs, we argue that both algorithm-level and hardware-level optimizations are essential. To this end, we propose a hardware-algorithm co-design solution including *Falcon*, a specialized GVS accelerator, and *delayed-synchronization traversal (DST)*, an accelerator-optimized graph traversal algorithm designed to simultaneously improve accelerator search performance and recall. We summarize the advantages of Falcon and DST over existing solutions in Table 1, and elaborate on the key features of our approach below.

Falcon is an in-memory GVS accelerator with four key features. Firstly, Falcon involves fast distance computations and sorting units, and minimizes off-chip memory accesses by using an on-chip Bloom filter to track visited nodes. Secondly, Falcon supports both intra-query parallelism, utilizing all compute and memory resources to process a single query, and across-query parallelism, handling multiple queries through separate processing pipelines. Thirdly, Falcon supports general GVS, allowing it to leverage emerging algorithms offering better recall and performance. Finally, Falcon functions as a networked service with an integrated TCP/IP stack, thus reducing end-to-end service latency by bypassing the accelerator's host server from the communication path.

Delayed-synchronization traversal (DST) relaxes the greedy graph traversal order to improve accelerator utilization. The design of the algorithm is motivated by two key observations. First, from a system performance perspective, the synchronous and greedy nature of the software-oriented best-first search (BFS) limits the amount of parallelism the accelerator can exploit and thus leads to significant accelerator under-utilization. Second, from a traversal-pattern perspective, we found that relaxing the order of candidate evaluations does not compromise recall. Building on these observations and drawing inspiration from label-correcting algorithms for parallel shortest path computation on graphs [14, 75], DST relaxes synchronizations that enforce the greedy traversal order, thereby increasing the amount of parallel workloads that Falcon can handle. Consequently, DST both reduces search latency by improving accelerator utilization and improves recall by allowing the exploration of search paths that the greedy BFS would otherwise overlook.

We prototype Falcon on FPGAs and evaluate it on various vector search benchmarks across different types of graphs. In combination with DST, Falcon achieves up to 4.3× and 19.5× lower online search latency and up to 8.0× and 26.9× better energy efficiency compared to CPU and GPU-based GVS systems, respectively. Besides, the proposed DST algorithm outperforms the classic BFS by 1.7~2.9× in terms of latency on Falcon and simultaneously improves recall.

The paper makes the following **contributions**:

- We identify the hardware primitives essential for efficient GVS, design Falcon, a specialized GVS accelerator, prototype it on FPGAs, and expose it as a networked service.
- We analyze the graph traversal patterns of best-first search and propose DST, an accelerator-optimized graph traversal algorithm that reduces GVS latency by relaxing traversal order.
- We evaluate Falcon and DST across diverse graphs and datasets, demonstrating their high performance and energy efficiency.

2 BACKGROUND AND MOTIVATION

In this section, we define the vector search problem (§2.1), introduce GVS algorithms (§2.2), discuss the limitations of existing processors for online GVS (§2.3), and motivate the need for an algorithm-hardware co-design solution for low-latency GVS (§2.4).

2.1 Vector Search: Problem Definition

A k nearest neighbor (*kNN*) search takes a d-dimensional query vector q as input and retrieves the k most similar vectors from a database Y containing d-dimensional vectors, based on metrics such as L2 distances, dot product, or cosine similarity.

Real-world vector search systems typically adopt *approximate nearest neighbor (ANN) search* instead of exact kNN search to boost search performance (latency and throughput) by avoiding exhaustive scans of all database vectors. The quality of an ANN search is measured by the recall at k ($R@k$). Let $NN_k(q)$ be the set of true k nearest neighbors to a query q and $ANN_k(q)$ be the set of k results returned by the ANN search, recall at k measures the proportion of the true k nearest neighbors that are successfully retrieved by the ANN search: $R@k = \frac{|ANN_k(q) \cap NN_k(q)|}{|NN_k(q)|}$.

2.2 Graph-based Vector Search

Graph-based vector search (GVS) is among the most popular ANN search methods, renowned for its high search performance and quality [28, 29, 70, 72, 73, 107, 111]. It involves constructing a proximity graph $G(V, E)$, where V represents the set of nodes, each is a database vector, and E represents the set of edges between nodes, with each edge indicating high similarity between the two connected nodes. Some notable examples of graph construction algorithms include HNSW[73], NSG[28], and DiskANN [44].

Algorithm 1 Best-First Search (BFS)

Require: graph G, entry node p, query vector q, maximum result queue size l, number of results to return k ($k \leq l$)
Ensure: k approximate nearest neighbors of query q

```
C ← {p}, R ← {p}, Visited ← {p}
while C ≠ ∅ and Min(C.dist) ≤ Max(R.dist) do
    c ← Extract-Min(C)                    ▷ pop the nearest candidate
    for all neighbors n of c do
        if n ∉ Visited then
            dist ← Compute-Dist(q, n)
            Visited.add(n), C.add(n, dist), R.add(n, dist)
    R.resize(l)                           ▷ keep only the closest l elements
return Sort(R)[: k]                       ▷ return the first k elements
```

2.2.1 Best-First Search (BFS) for Query Processing. Once the graph is constructed, query vectors can traverse the graph to find their nearest neighbors. While various graph construction algorithms exist [28, 70, 72, 73, 107, 111], the traversal on those constructed graphs all converges to the classic best-first search (BFS) algorithm.

BFS traverses a graph by greedily evaluating the best candidate node in each search iteration. As illustrated in Algorithm 1, BFS begins by adding the typically fixed entry node p to the candidate queue C, which stores nodes for potential exploration; the result queue R, which holds the nearest neighbors found so far; and the visited set $Visited$, which tracks nodes that have already been visited. It then searches on the graph iteratively as long as there is at least one candidate that is reasonably close to the query q. Here, reasonably close means that the minimum distance from the candidates in C to q is less than the maximum distance of the nodes currently in R. The algorithm then pops and evaluates the best candidate c by visiting all of its neighbors. Each neighbor that has not been visited is added to the visited set, the candidate queue, and the result queue, ensuring that no node is processed more than once. Following the exploration of neighbors, R is adjusted to maintain only the closest l elements.

The maximum size of the result queue l ($k \leq l$) controls the trade-off between search performance and quality. A larger l increases the threshold distance for considering a candidate, thereby expanding the number of candidate nodes evaluated during the search. Although visiting more nodes increases the likelihood of finding the nearest neighbors, it also leads to higher search latency.

2.3 Limitations of Existing Processors for GVS

Existing GVS systems have been mostly CPU-based, and recent research has explored their deployments on GPUs and FPGAs. All these systems adopt the classic BFS algorithm. However, current solutions remain sub-optimal for latency-sensitive online GVS.

2.3.1 Search on CPU. CPUs have several limitations in online GVS systems. Firstly, CPUs operate on a time-multiplexing basis, executing GVS operators such as fetching, computing, and insertion sequentially, with only limited timeline overlaps due to data prefetching. This sequential processing leads to cumulative search latency for each operator. Secondly, software implementations typically employ a byte array to track visited nodes for each query [28, 73], resulting in additional read and write operations per visited node. Thirdly, CPUs struggle with random memory accesses to fetch vectors, which are typically less than 1 KB, and to update the visited arrays (one byte per read or write).

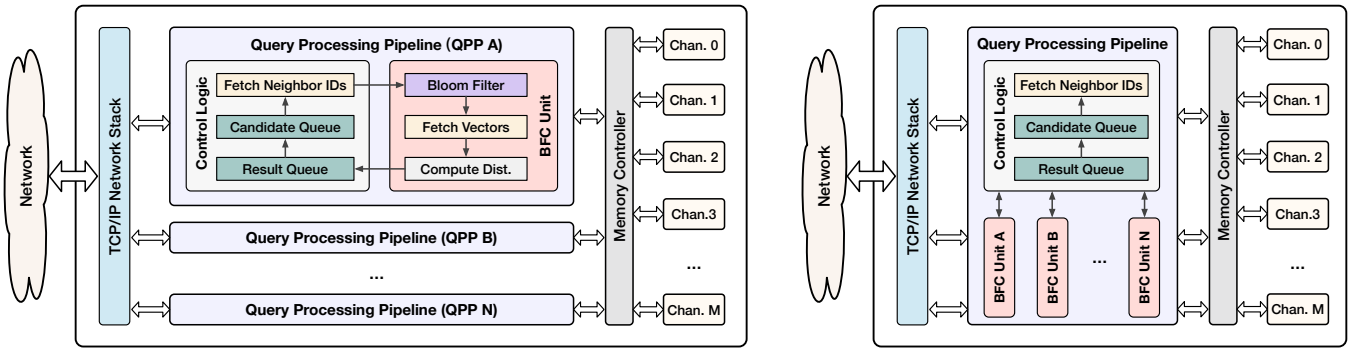
2.3.2 High-throughput GVS on GPUs. GPUs are known for their massive parallelism, featuring thousands of cores [22]. Thus, GPUs are well-suited for high-throughput GVS applications, as evidenced by recent studies [34, 106]. However, GPUs exhibit two shortcomings for online GVS. Firstly, GPUs show much higher GVS latency than CPUs as shown in our evaluation, because the limited amount of workload per search iteration makes it infeasible to effectively parallelize one query across multiple streaming multi-processors. Secondly, the scale of graphs that GPUs can efficiently serve is constrained by memory capacity. GPUs typically use high-bandwidth memory (HBM), which offers high bandwidth but several times less capacity compared to DDR memory given the same cost [4]. Although utilizing CPU-side memory is a potential option, search performance remains a concern: the throughput of fast CPU-GPU interconnects like the NVLink in NVIDIA Grace Hopper [7] is still an order of magnitude lower than that of GPU memory.

2.3.3 Specialized GVS Accelerators. Two recent studies [80, 103] implemented HNSW, a popular GVS algorithm, on FPGAs. Peng et al.[80] presented the first FPGA-based implementation, while Zeng et al.[103] further optimized the design by introducing data prefetching and enabling multi-FPGA search.

However, they are still not optimal for online GVS for the following reasons. Firstly, supporting only one type of graph (HNSW) may be inadequate given the rapid emergence of efficient GVS algorithms [28, 70, 72, 73, 107, 111]. For example, NSG [28], given longer graph construction time, can achieve better performance-recall trade-offs than HNSW. Specializing the accelerator for HNSW [80, 103] restricts the accelerator's flexibility in supporting various types of graphs: HNSW has a unique multi-level architecture, while the vast majority of graphs in GVS do not incorporate a leveled structure. Secondly, applying the software-friendly BFS on the accelerators leads to sub-optimal search performance. This is because BFS can cause significant under-utilization of the accelerators, as we will specify in §4.3. Thirdly, although Zeng et al. [103] supports intra-query parallelism, an improvement over Peng et al. [80], the parallel strategy remains suboptimal. Specifically, the method of partitioning the graph into several sub-graphs and searching all sub-graphs in parallel [103] leads to significantly more nodes being visited per query compared to traversing a single, larger graph, as we will explain further in §3.3.

2.4 Motivation: Algorithm-Hardware Co-Design

In this paper, we aim to achieve low-latency, energy-efficient, and general GVS. Given the insufficient hardware support (§2.3) and the inherent difficulty of parallelizing BFS (§2.2), we argue that achieving this goal requires both algorithm-level and hardware-level optimizations. In the following sections, we present a hardware-algorithm co-design solution, which includes *Falcon* (§3), a hardware accelerator for GVS, and *delayed-synchronization traversal (DST)* (§4), an accelerator-optimized graph traversal algorithm that simultaneously improves search performance and recall.



(a) Across-query Parallelism

(b) Intra-query Parallelism

Figure 1: Falcon overview. It has two architecture variants supporting across-query and intra-query parallelisms.

3 FALCON FOR GVS ACCELERATION

We present Falcon, a low-latency GVS accelerator that we prototype on FPGAs but also applicable to ASICs (§3.1). Falcon consists of various high-performance hardware processing elements (§3.2). It has two variants supporting across-query and intra-query parallelisms, optimized for processing batches of queries and individual queries, respectively (§3.3). The accelerator is directly accessible as a networked service and supports various types of graphs (§3.4).

3.1 Design Overview

Accelerator components. Figure 1 shows Falcon, a spatial dataflow accelerator for GVS. Each *query processing pipeline (QPP)* handles one query at a time, containing both control logics and *Bloom-fetch-compute (BFC) units*. Falcon is composed of various processing elements (PEs) interconnected via FIFOs, including systolic priority queues for storing candidate nodes and search results, Bloom filters to avoid revisiting nodes, and compute PEs for efficient distance calculations between query and database vectors.

Parallel modes. Falcon has two variants that support *across-query parallelism* and *intra-query parallelism*, as shown in Figure 1(a) and (b), respectively. Across-query parallelism processes different queries across QPPs, while intra-query parallelism minimizes per-query latency by utilizing all compute and memory resources (multiple BFC units) to process one query at a time.

Differences compared to existing accelerators. Falcon distinguishes itself from previous GVS accelerators [80, 103] in four aspects, as summarized in Table 1. Firstly, Falcon utilizes on-chip Bloom filters to manage the list of visited nodes, thereby minimizing memory accesses (§3.2). Secondly, Falcon's intra-query parallel design utilizes all compute and memory resources to traverse a single graph rather than partitioned sub-graphs (§3.3). Thirdly, Falcon supports various GVS algorithms, rather than being limited to a specific one such as HNSW, allowing it to benefit from emerging algorithms that offer improved search quality and performance (§3.4). Finally, Falcon employs the proposed accelerator-optimized traversal algorithm that significantly reduces vector search latency (§4).

3.2 Hardware Processing Elements

We now introduce the main types of PEs in the order of their appearance in Algorithm 1.

3.2.1 Priority Queues. We adopt the systolic priority queue architecture [41, 60] for the candidate and result queues in Algorithm 1. A systolic priority queue is a register array of s elements interconnected by $s-1$ compare-swap units. It enables high-throughput input ingestion of one insertion per two clock cycles by comparing and swapping neighboring elements in parallel in alternating odd and even cycles. The queue can be sorted in $s-1$ cycles.

3.2.2 Bloom Filters. Once the candidate queue pops a candidate to be explored, the next step is to check whether each of the candidate's neighbors is already visited.

Previous software and specialized hardware implementations either maintain a visited array or a hash table, but neither is ideal for Falcon. For example, software-based implementations [28, 73] maintain an array with a length as the number of nodes in the graph. Node IDs are used as the array addresses to access the visited tags. However, this approach leads to extra memory accesses, requiring one read operation per check and one extra write operation to update the array for unvisited nodes. Zeng et al. [103] developed on-chip hash tables as part of the accelerators to track the visited nodes to avoid off-chip memory accesses. Each entry of the hash table stores up to four visited node IDs. However, given the limited on-chip SRAM, it is unlikely to instantiate large hash tables, and thus collisions would appear during the search. A collision would not only lead to redundant node visits, but those visited nodes will be inserted into the candidate and result queues repetitively, thus eventually degrading recall.

Falcon, in contrast to existing solutions, adopts on-chip Bloom filters to track visited nodes. A Bloom filter is a space-efficient probabilistic data structure designed to test whether an element is a member of a set, e.g., determining whether a node has been visited based on its ID. A Bloom filter uses multiple (h) hash functions to map each input to several positions in a b-bit array. To check if a node has been visited, the same hash functions are used to check the status of these specific positions: if any of the bits are not set, the node is definitely not visited; if all are set, the node is highly likely visited (but not guaranteed, a scenario known as false positive). Given m inserted elements, the false positive rates can be calculated by $\left(1-e^{-\frac{hm}{b}}\right)^h$ [15].

Compared to hash tables, Bloom filters are significantly more space efficient for identifying visited nodes. For example, instantiating a

hash table with 1K slots for 4-byte node IDs requires 32Kbit SRAM. Using a chaining strategy to resolve hash collisions [74], where collided elements are moved to DRAM, the collision probability for a new incoming node ID is as high as 63.2% when 1K nodes have already been visited. In contrast, using the same amount of SRAM, a Bloom filter can provide 32K slots. With an equivalent number of nodes visited, the false positive rate for a new node ID is only 3.0% and 0.07% using a single hash function and three hash functions, respectively. As we will show in our experiments, the very few false positives, meaning that an unvisited node is reported as visited, will not visibly degrade recall. This is because a well-constructed graph typically offers multiple paths from the query vector to the nearest neighbors, mitigating the effects of these very few false positives.

Falcon implements Bloom filters in the following manner. Both the number of hash functions and the size of the Bloom filters are configurable. Currently, Falcon uses three Murmur2 hashes [5] per filter. These hash functions are computed in parallel, and each hash function pipeline can yield a hash code every clock cycle. The size of the bitmap is set to 256Kbit, which translates to low false positive rates — only one in 600K for 1K visited nodes.

3.2.3 Fetching Vectors. Upon identifying nodes to visit, the next step is reading the vectors for each node.

Falcon optimizes bandwidth utilization by pipelining vector fetches. Rather than waiting for the first vector to return before issuing a second read, each fetch unit pipelines up to 64 read requests (configurable), thus improving read throughput by hiding the latency associated with memory and the memory controller. The data width of the FIFO connecting a fetch unit to the memory controller is set to 64 bytes.

3.2.4 Distance Computations. Each vector fetch unit is connected to a compute PE that calculates L2 distances, dot product, or cosine similarity between queries and database vectors. A compute PE instantiates multiple multipliers and adders and pipelines different compute stages, such that the compute throughput can match the maximum read throughput of a vector fetch unit.

3.3 Intra-query and Across-query Parallelism

While across-query parallelism for batched queries can be straightforwardly implemented by instantiating multiple query processing pipelines (QPP) on the accelerator, there are two design choices for intra-query parallelism, which aim to minimize latency for individual queries. One option involves adopting the architecture of across-query parallelism by partitioning the dataset into multiple subsets, querying each subset with an individual QPP, and aggregating the results, as Zeng et al. [103] described.

Alternatively, our choice is to *speed up the traversal of a single graph* by instantiating multiple BFC units in a single QPP to utilize all the compute and memory resources for a single query (Figure 1(b)). This decision stems from the observation that traversing several sub-graphs significantly increases the total amount of workload per query compared to traversing a single graph. Figure 2 shows that, to achieve a recall of $R@10 = 90\%$ on the SPACEV natural language embedding dataset [9], the total number of visited nodes per query when using eight subgraphs is 4.2× of that for a single graph. Thus, the maximum speedup (assuming perfect load balancing) that eight partitions and eight QPPs can achieve is only 1.9× that of traversing a single graph with one QPP.

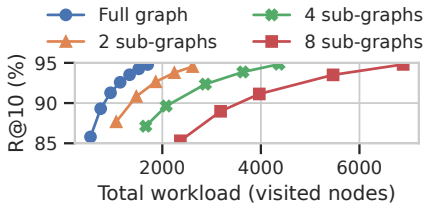


Figure 2: Traversing one graph versus several sub-graphs.

When traversing a single graph using intra-query parallelism, Falcon leverages its direct message-passing mechanism via FIFOs to enable low-overhead, fine-grained task dispatching among different BFC units. This is a significant architectural advantage compared to CPUs and GPUs, where synchronization overhead among CPU cores or GPU streaming processors [58, 105] is too high compared to a single iteration of graph traversal, which only takes microseconds typically involving dozens of distance computations.

3.4 Accelerator-as-a-Service Implementation

Falcon is implemented with a total of 6.6K lines of code, including 3.6K lines of High-Level Synthesis (HLS) code for the accelerator kernel, developed using Vitis 2022.1, and 3K lines of C++ code for the CPU host and client programs. We instantiate and evaluate Falcon on AMD Alveo U250 FPGAs, but the architecture is portable to arbitrary FPGA platforms. Falcon operates as a networked service through a TCP/IP stack and supports various types of graphs.

3.4.1 Network Stack Integration. Vector search systems are typically wrapped as services for real-time LLM serving or recommender systems. To minimize service latency, we integrate a TCP/IP network stack [38] into Falcon, as shown in Figure 1. This integration allows Falcon to function as a networked accelerator service in data centers [27, 83], facilitating direct communication with clients. This approach differs from common setups where the accelerator operates as a PCIe-based operator offloading engine, which involves additional latency including CPU handling requests from the network, accelerator kernel invocation, and data copying between the CPU and the accelerator. Compared to CPU and GPU-based services, Falcon can partially overlap communication and query latency: for a batch of queries, it begins processing the first query upon its arrival rather than waiting for the entire batch to be received.

3.4.2 Supporting Various Graphs. Falcon supports arbitrary graphs by representing them with a unified graph format, accommodating common graph elements including nodes, edges, entry nodes, and degrees. This approach is naturally compatible with the vast majority of graphs [28, 72, 107, 111], except for HNSW [73] that has a unique multiple-layer structure. The upper layers of HNSW are designed to identify a high-quality entry point into the base layer, which contains all the database vectors — thus the base layer is comparable to the entire graph in other GVS algorithms [28, 72]. Instead of customizing the accelerator for this case, we prioritize the Falcon's versatility by initiating searches from a fixed entry point on the base layer of HNSW, which is the same (and only) entry point used at the top level. We found that this approach, without

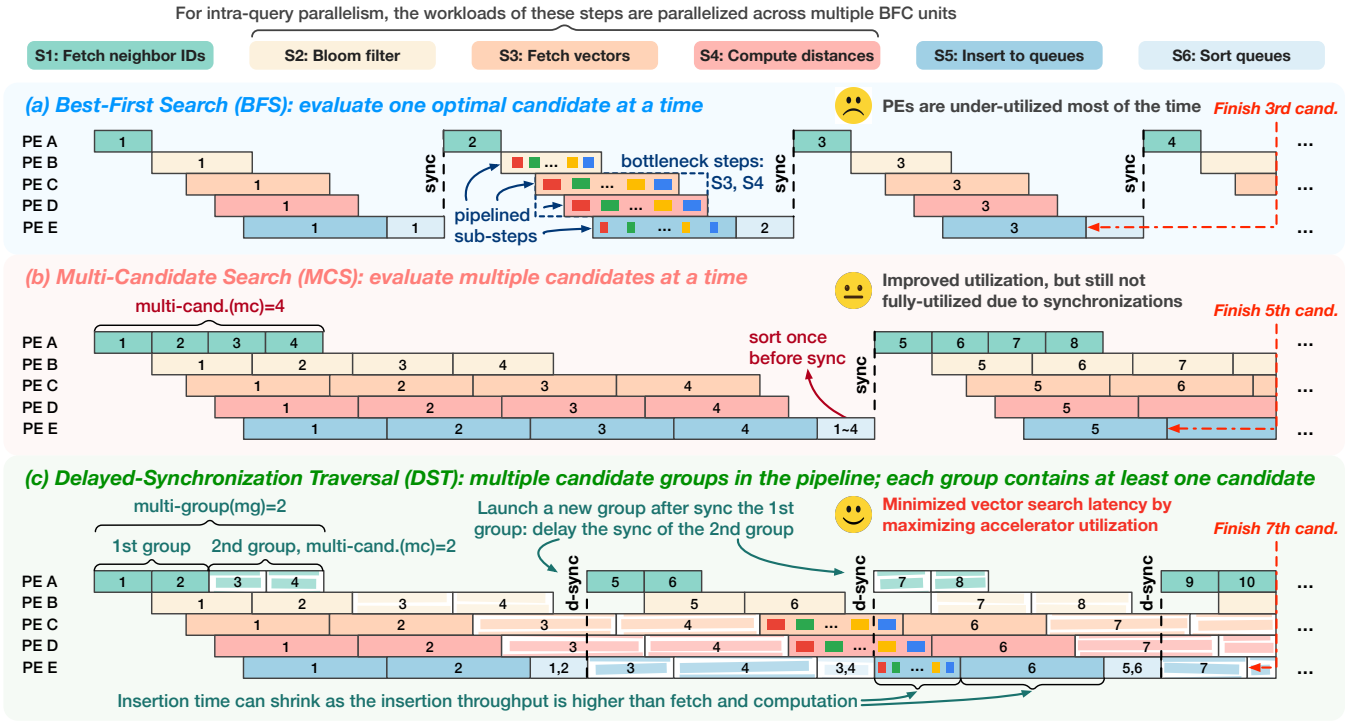


Figure 3: The proposed Delayed-Synchronization Traversal (DST) reduces vector search latency by maximizing accelerator utilization. It delays synchronizations and allows multiple candidates to be evaluated simultaneously in the processing pipeline.

starting from the optimal entry node at the base layer for each query, does not lead to recall degrades (Figure 7), although more hops might be necessary to reach the nearest neighbors, a finding also supported by existing research [66, 92].

3.4.3 Memory Management. To leverage multiple memory channels, we partition the data — including links and vectors — evenly across all channels in a round-robin fashion based on node IDs. This contrasts with the approach of splitting a single vector across multiple channels, which provides limited bandwidth benefits when fetching a vector due to the random access latency incurred by each channel. The number of BFC units is set equal to the number of memory channels, as each compute unit in a BFC is designed to fully utilize the maximum bandwidth of a single memory channel.

4 DELAYED-SYNCHRONIZATION TRAVERSAL

Realizing the inefficiencies of BFS on Falcon (§4.1), we investigate its graph traversal patterns (§4.2) and propose DST, an accelerator-optimized traversal algorithm applicable for both intra- and across-query parallelisms (§4.3).

4.1 Inefficiency of BFS on Accelerators

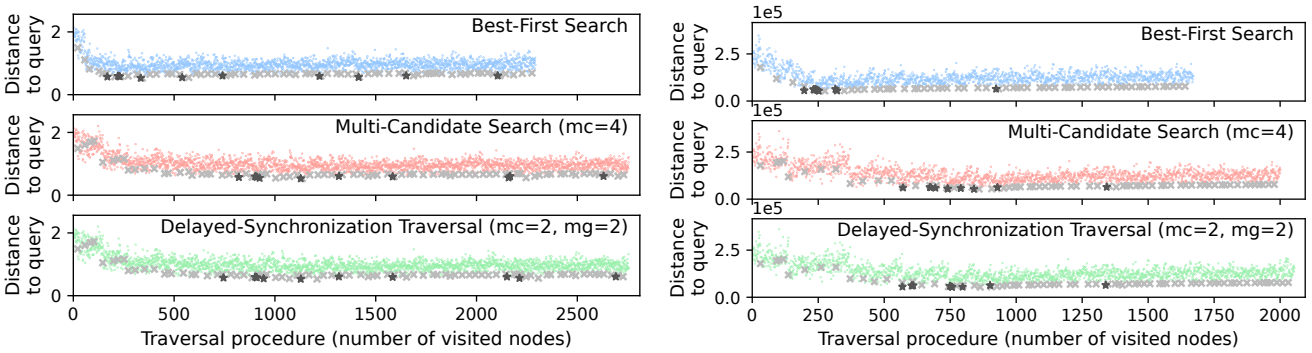
Figure 3(a) visualizes the timeline of BFS on Falcon, where each unique color represents one of the six search steps (S1~S6), and each PE handles a specific step, except for the priority queues that manage two steps, including distance insertions and sorting (S5 and S6). Some steps must wait for the previous step to complete: sorting only begins after all distances are inserted into the queues. Other steps like filtering, fetching vectors, computing distances, and insertions can partially overlap because these PEs pipeline the execution of sub-steps, where each sub-step involves one of the neighbors of the candidate being evaluated. Between search iterations, an *implicit synchronization* between all of the PEs ensures that the queues are sorted, such that the best candidate can be popped for evaluation in the next iteration.

Unfortunately, directly implementing the software-oriented BFS algorithm on a parallel GVS accelerator like Falcon can lead to sub-optimal search performance due to under-utilization of the accelerator. As shown in Figure 3(a), only a fraction of the PEs are utilized simultaneously because of the inherently greedy nature of BFS, which processes only one candidate at a time, offering little opportunity for parallelization.

4.2 Goal: Improving Accelerator Performance through Traversal Algorithm Redesign

A natural idea to optimize accelerator performance is to *maximize accelerator utilization by minimizing PE idleness.* Given the imbalanced workloads across different search steps, this approach does not necessitate all PEs to be always active but rather focuses on keeping those PEs involved in bottleneck steps consistently busy. In the context of GVS, the bottleneck steps usually include fetching neighbor vectors (S3) and calculating their distances relative to the query vectors (S4).

4.2.1 Algorithm-specific Observations. Given the critical role of accelerator utilization in search performance, we ask: *Is it necessary to strictly follow the BFS traversal order and synchronization pattern to achieve high search quality?*



(a) HNSW graph on Deep1M dataset

(b) NSG graph on SIFT1M dataset

Figure 4: Traversal convergence of BFS, MCS, and DST across graphs (HNSW and NSG) and datasets (Deep and SIFT). Each grey cross is an evaluated candidate, each colored dot a visited neighbor node, and each dark star one of the ten nearest neighbors.

To answer this question, we examine the traversal patterns of GVS. To address this question, we analyze the traversal patterns of GVS. The upper section of Figure 4 illustrates the BFS traversal process for a sample query on the Deep1M dataset [12] using HNSW (Figure 4a) and on the SIFT1M dataset [8] using NSG (Figure 4b). Each grey cross represents an evaluated candidate node, colored dots denote its neighbor nodes, and black stars mark the ten nearest neighbors. Notably, while the node distances to the query decrease at the beginning of the traversal, most subsequent candidates maintain similar distances rather than showing a monotonically decreasing trend — an observation consistent across queries, datasets (SIFT, Deep, and SPACEV), and graphs (HNSW and NSG), although not all data are visualized due to space limits.

This observation suggests that **traversals in GVS do not have to adhere to a strictly greedy approach — relaxing the traversal order of different candidate nodes should result in comparable search quality**, assuming the same or a similar set of candidates is evaluated.

4.2.2 Naive Solution: MCS. Leveraging the intuition above, one straightforward way to improve accelerator utilization is increasing the number of candidates evaluated per iteration, a strategy we term *multi-candidate search (MCS)*. As illustrated in Figure 3(b), each iteration evaluates $mc = 4$ candidates instead of just the closest one, because the second to the fourth best candidates per iteration may also be close to the query and could be on the search path of BFS.

However, the PE utilization is not yet optimal due to the implicit synchronization operations required between iterations, where the candidate queue must be sorted before evaluating the next mc nearest candidates. While increasingmc could push PE utilization rates towards 100%, this approach can potentially degrade end-to-end search performance as we will show in the evaluation, because evaluating many candidates per iteration means potentially processing irrelevant candidates.

4.3 Low-latency GVS via DST

To maximize accelerator utilization with minimal overhead (the number of extra nodes visited), we propose *Delayed-Synchronization Traversal (DST)*, a parallel, low latency graph traversal algorithm for GVS. **The key idea of DST is to allow on-the-fly processing of multiple groups of candidates within the query processing pipeline by delaying synchronizations between search iterations, thereby improving the utilization of compute units and memory bandwidth.** Here, each candidate group can contain one or multiple candidate nodes.

4.3.1 DST Procedure. Figure 3(c) demonstrates how DST enhances accelerator utilization. In this example, there are two candidate groups ($mg = 2$), each with two candidates ($mc = 2$), thus allowing four candidates to be processed simultaneously in the pipeline, mirroring the MCS setup ($mc = 4$) in Figure 3(b). Unlike MCS, DST introduces *delayed synchronization*: as the evaluation of the candidate group containing the 5th and 6th candidates begins, only the first group, containing the 1st and 2nd candidates, has been fully evaluated — the delayed synchronization sorts the existing results, while the synchronization of the second group (with 3rd and 4th candidates) is deferred. This strategy ensures that the processing pipeline remains filled and that the bottleneck-step PEs for fetching vectors and computing distances are fully utilized, thereby avoiding the periods of idleness around synchronizations as shown in Figure 3(a) and (b). When applying DST to intra-query parallelism, steps S2∼S4 can be parallelized across multiple BFC units, unlike across-query parallelism, which utilizes one BFC unit per QPP.

Algorithm 2 details the procedure of DST from the accelerator controller's perspective. DST starts by evaluating the entry node as the first candidate group. As soon as a candidate group is evaluated, DST tries to fill the accelerator pipeline by launching the evaluation of additional candidate groups, where both the number of groups in the pipeline (mg) and the number of candidates per group (mc) can be set by the user. DST terminates when there are no active groups in the pipeline and there are no more valid candidates.

4.3.2 Performance Benefits. DST achieves significantly higher throughput than BFS and MCS in terms of the number of candidates processed per unit of time. Figure 3 marks the count of processed candidates by the end of the timeline on the right side. In this example, BFS completes only three candidates, meaning that the results for the 3rd candidate have been inserted into the candidate queue. MCS shows improved throughput, managing to finish processing five candidates in the same time frame. DST, given an equivalent number of candidates in the pipeline as MCS (four), achieves the highest throughput by completing seven candidates

Algorithm 2 Delayed-Synchronization Traversal (DST)

Require: graph G, entry node p, query vector q, result queue size l, number of candidate groups mg, number of candidates per group mc, number of results k ($k \leq l$)
Ensure: k approximate nearest neighbors of query q

$C \leftarrow \{p\}, R \leftarrow \{p\}, Visited \leftarrow \{p\}$
LAUNCH-EVAL-NON-BLOCK($\{p\}$), $GroupCnt \leftarrow 1$
while $GroupCnt > 0$ **or** MIN($C.dist$) $\leq$ MAX($R.dist$) **do** ▷ stop if no active groups and qualified candidates
  if EARLIEST-EVAL-DONE **then** ▷ check task status
    $GroupCnt \leftarrow GroupCnt - 1$
    while $GroupCnt < mg$ **do** ▷ fill the pipeline
      $threshold \leftarrow$ MAX($R.dist$)
      $Group \leftarrow$ EXTRACT-MIN($C, mc, threshold$)
      if SIZE($Group$) > 0 **then**
        LAUNCH-EVAL-NON-BLOCK($Group$)
        $GroupCnt \leftarrow GroupCnt + 1$
return SORT(R)[: k] ▷ return the first k elements

by the end of the timeline. Notably, DST fully utilizes the critical PEs for vector fetching and distance computations, thanks to the delayed-synchronization mechanism.

4.3.3 Search Quality. Given the algorithmic relaxations in DST compared to BFS, one might immediately question: *Will the reordered traversal in DST degrade recall?* Contrary to this concern, DST can actually improve recall while lowering search latency as our experiments will demonstrate (Figure 7) for the following reasons. On one hand, BFS traverses the graph in a greedy manner, striving to avoid visiting nodes that are not sufficiently close to the query. On the other, DST, by delaying synchronizations and allowing multiple candidates to be processed in the pipeline, relaxes the threshold for node evaluation. Considering that the termination condition remains consistent with BFS (when there is no qualified candidate left), DST likely evaluates the high-quality candidates on the search path of BFS and additionally explores other potentially relevant candidates. Thus, the evaluation of these extra sub-optimal candidates (a) does not prevent the evaluation of better candidates close to the queries and (b) may uncover extra paths leading to the nearest neighbors, thereby potentially improving recall.

Figure 4 compares the search convergence of BFS, MCS, and DST. All of them find the nearest neighbors in this example, with DST and MCS visiting more nodes than BFS. This trend remains consistent across various datasets (SIFT, Deep, and SPACEV) and graph structures (HNSW and NSG), though Figure 4 only visualized a subset of experiments due to space constraints.

4.3.4 Parameter Configuration. DST introduces two additional runtime configurable parameters compared to BFS: the number of candidate groups in the pipeline (*mg*) and candidates per group (*mc*). The optimal configuration depends on several factors, including vector dimensionalities, data distributions, and degrees (number of neighbors per node). We found it challenging to determine the optimal parameters by performance modeling due to (a) the significant variance in node degrees and (b) the unpredictable proportion of visited nodes as traversal progresses. Thus, to ensure optimal search performance, it is advisable to perform an empirical parameter search using a set of sample queries before system deployment. Typically, this process only takes minutes, as the search space is relatively small, with both *mg* and *mc* usually not exceeding ten according to our experiments.

5 EVALUATION

Our evaluation aims to answer the following questions:

- How does Falcon's search performance and energy efficiency compare to that of CPUs and GPUs? § 5.2
- How much speedup and recall improvement can DST achieve on Falcon over BFS? § 5.3
- Where is the performance cross-over point between intra-query and across-query parallelism? § 5.4

5.1 Experimental Setup

Baseline systems. For CPUs, we evaluate two popular graphs, namely HNSW [72] and NSG [28], using their official implementations. For GPUs, we evaluate GGNN [34], an approximate version of HNSW optimized for GPU architectures. Additionally, we evaluate the inverted-file (IVF) index [86], a clustering-based index, using the Faiss library [2] for both CPUs and GPUs. As the previous FPGA GVS implementations [80, 103] are not open-sourced, we mainly compare their traversal strategies with DST based on Falcon in §5.3.

Hardware. We use server-class hardware manufactured in similar generations of technology (12~16 nm), where the CPU and GPU hold advantages over the FPGA in terms of bandwidth. We develop Falcon using Vitis HLS 2022.1, instantiate it on the AMD Alveo U250 FPGA (16 nm) with 64 GB of DDR4 memory (four channels x 16 GB, 77 GB/s in total), and set the accelerator frequency to 200 MHz. We use a CPU server with 48 cores of Intel Xeon Platinum 8259CL operating at 2.5 GHz and 384 GB DDR4 memory (12 channels, 256 GB/s). GPU evaluations are performed on NVIDIA V100 with 16 GB HBM2 memory (900 GB/s). Both the CPU and GPU servers are equipped with Mellanox ConnectX-4 NICs, providing network bandwidth equivalent to the FPGA at 100 Gbps.

Datasets. We use the SIFT [8], Deep [12], and SPACEV [9] datasets, containing 128, 96, and 100-dimensional vectors, respectively, thus covering both vision features (SIFT and Deep) and text embeddings (SPACEV). We evaluate their subsets of the first ten million vectors, such that the constructed graphs can fit within the GPU and FPGA memory.

Algorithm settings. Unless specified otherwise, we set the maximum degree of the graphs to 64, balancing between graph size and search quality. We set the candidate queue size as 64, which ensures at least 90% recall for ten nearest neighbors across datasets. Falcon uses the best-performing DST parameters unless otherwise specified. For IVF indexes, we set the number of IVF lists as 4096, approximately the square root of the number of vectors as a common practice.

5.2 End-to-end Performance and Efficiency

We compare Falcon with baseline systems on the six combinations between datasets and graphs. The software recall of these experiments is noted in Figure 5: NSG consistently achieves better recall than HNSW. Falcon always achieves better recall than software

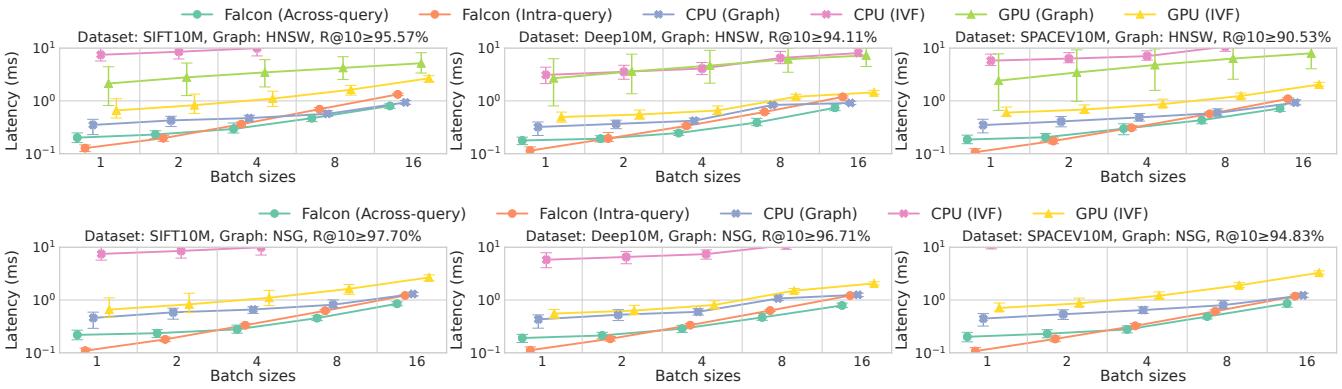


Figure 5: End-to-end GVS latency distribution of CPU, GPU, and Falcon across various graphs (rows) and datasets (columns). The error bar shows the range within which 95% of query latencies fall; CPU latency with IVF may surpass the y-axis limit.

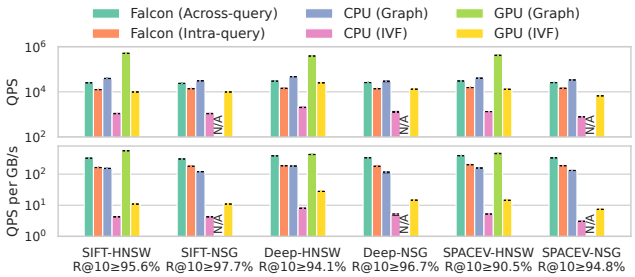


Figure 6: GVS throughput across processors, without latency constraints, is strongly related to memory bandwidth.

because DST explores more search paths per query than BFS, as we will analyze in §5.3.

5.2.1 End-to-end Online Search Latency. For online search, we treat all systems as a service where both the client and the server are connected to the same network switch. The network transmission time between CPU servers and between CPUs and FPGAs are similar — around 50μs given a batch size of one, only a tiny fraction of the end-to-end query latency. Figure 5 shows the distributions of vector search latency for various batch sizes across six graph-dataset combinations. We set the IVF-based index parameters for each scenario to achieve at least the same recall as GVS.

Falcon consistently outperforms all baselines in median latency, achieving speedups of up to 4.3× over CPU with graphs, 19.5× over GPU with graphs, 102.1× over CPU with IVF, and 6.5× over GPU with IVF. Falcon achieves the lowest search latency among the compared systems, with its intra-query and across-query parallel modes preferable for different batch sizes as we will discuss in §5.4. For CPUs, GVS outperforms the IVF index as the latter requires more database vectors to scan to achieve comparable recall [29, 63]. As batch sizes increase, CPU GVS latency becomes closer to that of Falcon, mainly benefiting from the CPU server's 3.3× higher bandwidth than the FPGA, whose bandwidth is saturated at a batch size of four. On GPUs, the embarrassingly parallel scan pattern of IVF results in better latency than GVS. Despite their high bandwidth and numerous cores, GPUs struggle to efficiently handle queries with small batch sizes due to the GPU's throughput-oriented architecture, which prioritizes parallel processing of many queries but results in high latency for individual queries.

5.2.2 Throughput without Latency Constraints. Figure 6 presents search throughput in queries-per-second (QPS) without latency constraints by setting the batch size as 10K.

Unsurprisingly, when latency constraints are removed, GVS throughput primarily becomes a competition of memory bandwidth. For both CPUs and GPUs, graph-based indexes outperform IVF, which necessitates scanning more database vectors to reach the same recall [29, 63]. For GVS, the GPU exhibits superior throughput thanks to its 12× memory bandwidth over the FPGA, as shown in the upper half of Figure 6. Upon normalization by bandwidth (Figure 6 lower), the performance of Falcon and GPUs becomes comparable, with GPUs showing a slight edge for SIFT. This is because the GPU adopts the greedy BFS algorithm, whereas Falcon uses DST that trades off additional nodes to visit for reduced latency, as we will analyze in §5.3. The CPU performs the worst in QPS per unit bandwidth due to additional memory accesses required to check and update the visit status array.

5.2.3 Energy Efficiency. We measure the power consumption (in Watt) of CPU, GPU, and Falcon using *Intel RAPL*, *NVIDIA System Management Interface*, and *AMD's Vitis Analyzer*. The energy consumption per query batch (in Joule) is calculated by multiplying power with batch latency.

Falcon is energy efficient, achieving up to 8.0×, 26.9×, 231.1×, and 5.5× better energy efficiency than CPU graph, GPU graph, CPU IVF, and GPU IVF, respectively. For online GVS with batch sizes up to 16, the power consumption of CPU, GPU, and Falcon ranges from 136.9~209.2W, 183.4~324.2W, and 55.2~62.3W, respectively. Considering energy consumption per batch, Falcon achieves 2.2~8.0× and 11.9~26.9× better energy efficiency than CPUs and GPUs. For offline GVS without latency constraints (using batch size of 10K), Falcon still achieves 1.9~3.9× energy efficiency over CPUs, but is outperformed by GPUs by 5.3~11.1×, indicating that GPUs remain the preferred option for scenarios requiring high-throughput thanks to their superior memory bandwidth.

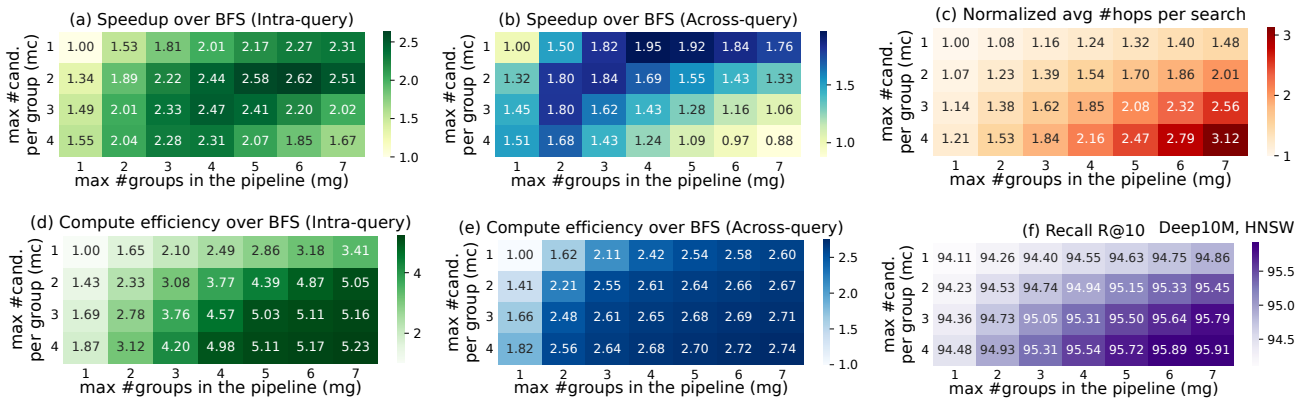


Figure 7: The speedup, number of evaluated nodes, compute efficiency, and recall given various traversal configurations (HNSW on the Deep10M dataset). The x- and y-axes represent DST parameters *mg* and *mc*, where BFS corresponds to $mg = 1$ and $mc = 1$.

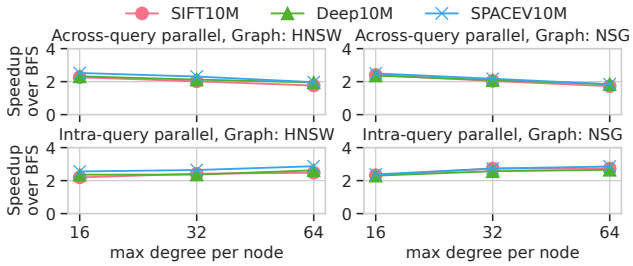


Figure 8: DST consistently outperforms BFS across various datasets, graph configurations, and parallel modes.

5.3 DST Efficiency on Accelerators

5.3.1 Performance Benefits. We now discuss the speedup achieved with different DST parameters and the maximum speedup across various experimental setups.

The impact of DST configurations on performance. We evaluate the impact of the numbers of candidate groups in the pipeline (*mg*) and candidates per group (*mc*) on DST performance. Figure 7 (a) and (b) shows the throughput speedup achieved by DST compared to BFS on the Deep10M dataset with HNSW, across both the intra-query and across-query parallel versions of Falcon. BFS is equivalent to $mg = 1, mc = 1$ (upper-left corner), whereas MCS, evaluating multiple candidates per iteration without delayed synchronization, is shown in the first column ($mg = 1, mc \geq 1$). All the other setups are considered as DST. Note that previous FPGA designs [80, 103] adopts BFS, with Zeng et al. [103] implementing a prefetching strategy on BFS that, at best (zero miss rate), matches the performance of MCS with $mc = 2$.

The optimal configuration for DST varies across use cases, with intra-query parallelism typically requiring higher parameter values than across-query parallelism. As shown Figure 7, the optimal parameters are $mg = 6, mc = 2$ for intra-query parallelism (Figure 7 (a)) and $mg = 4, mc = 1$ for across-query parallelism (Figure 7 (b)). This is because the intra-query version parallelizes the distance computations, thus achieving a higher throughput of workload processing per query, leading to a higher throughput of processing nodes and thus necessitating a greater workload intensity to fully utilize the accelerator. However, higher $mg = 6$ and $mc = 2$ also lead to a greater amount of query-wise workloads as more hops are needed before the search terminates, as shown in Figure 7 (c). Thus, the maximum speedup is determined by the balance between accelerator utilization and the number of extra hops per query. Figures 7 (d) and (e) show the compute efficiency—measured as the number of nodes evaluated per unit time—under different DST configurations. As *mg* and *mc* increase, compute efficiency improves but eventually plateaus. For instance, in Figure 7 (d), setting $mg = 5$ and $mc = 3$ yields a 5.03× improvement over BFS. However, further increasing parallelism to $mg = 7$ and $mc = 4$ results in only a marginal gain, reaching 5.23× over BFS.

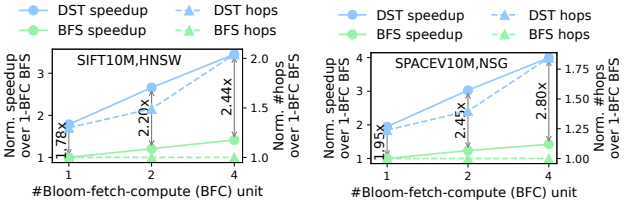
Maximum speedup in various experimental setups. Figure 8 shows the speedup of DST over BFS across various settings, including parallel modes, datasets, and graph types, and the maximum degrees of each graph.

DST consistently outperforms BFS across all setups, achieving speedups from 1.7~2.9×. DST is particularly advantageous in intra-query parallelism: with a maximum degree size of 64, it achieves speedups of 2.5~2.9× over BFS for intra-query parallelism, compared to 1.7~2.5× for across-query parallelism. This is because intra-query parallelism utilizes more BFC units for a single query and thus benefits more from increased workloads in the pipeline when adopting DST. This reason is that lower node degrees reduce the time required to fetch neighbors and compute distances when applying BFS, leading to significant accelerator under-utilization, as we explained in Figure 3.

5.3.2 Recall Benefits. The rightmost heatmap in Figure 7 shows the improvements in search quality achieved by DST.

In general, larger numbers of candidates in the processing pipeline (higher mg and mc) lead to increased recall. This is due to the evaluation of a broader range of candidates. Although some candidates may not be on the optimal search path, they could still lead to paths that reach the nearest neighbors.

DST consistently achieves better recall than BFS across all experiments. In Figure 7(f), employing the performance-optimal DST configurations enhances R@10 from 94.11% to 94.55% and 95.33% for across-query and intra-query parallelism, respectively. Given



(a) HNSW on SIFT (b) NSG on SPACEV

Figure 9: DST achieves significantly better performance scalability than BFS given intra-query parallelism.

various experimental setups as in Figure 8, the R@10 improvements range from 0.14% to 4.93%.

5.4 Across-query and Intra-query Parallelism

5.4.1 Scalability of Intra-query Parallelism. Figure 9 compares the scalability of DST and BFS given various numbers of Bloom-fetch-compute (BFC) units across datasets, with all units sharing a common control unit to form a query processing pipeline (QPP). For DST, we use *mc* and *mg* that achieve the highest performance.

DST demonstrates better performance scalability than BFS. For example, for HNSW on the SIFT dataset (Figure 9a), the speedup of DST over BFS increases from 1.78× to 2.44× as the number of BFC units grows from one to four. BFS, with four BFC units, achieves only a speedup of 1.41× over the single BFC version. This limited scalability of BFS stems from its greedy traversal pattern, which processes only one candidate at a time, resulting in minimal parallelizable workloads per iteration while the control overhead associated with the queues remains constant. In contrast, DST expands the workloads in the pipeline, ensuring that each BFC unit has sufficient workload to work with. A similar speedup trend of DST over BFS is observed across various datasets and graphs (e.g., Figure 9b), although not all results are shown here.

5.4.2 Performance Trade-offs between Intra-query and Across-query Parallelism. Figure 5 compares the performance of the two types of parallelism, where each accelerator contains four BFC units, configured as either a single QPP for intra-query parallelism or four QPPs for across-query parallelism.

The optimal choice of parallel mode is related to batch sizes. As shown in as shown in Figure 5, intra-query parallelism is always advantageous for a query size of one. However, since the latency speedup from intra-query parallelism does not scale linearly with the number of BFC units (Figure 9), across-query parallelism performs better for queries with batch sizes at least equal to the number of QPPs (four in our case). For batch sizes that fall between these two scenarios, the preferred parallel mode depends on the dataset, vector dimensionality, and graph construction parameters.

5.5 Speedup in Recommender and RAG Systems

We also evaluate the end-to-end speedup achieved by deploying Falcon in recommender and RAG systems. *Due to the wide range of possible model configurations, the proportion of time spent on retrieval can vary significantly (and consequently, the achievable speedup), as we demonstrate below.*

Recommender Systems. We instantiate two DLRM models of different sizes, as summarized in Table 2. The smaller model (RM-S) is

Table 2: Recommendation model configurations.

Model	Embedding Table Num	Embedding Dimension	Bot. Layers	Top Layers
RM-S	26	64	256,128,64	512,512,256,128,1
RM-L	104	64	512,256,128,64	1024,1024,512,256,1

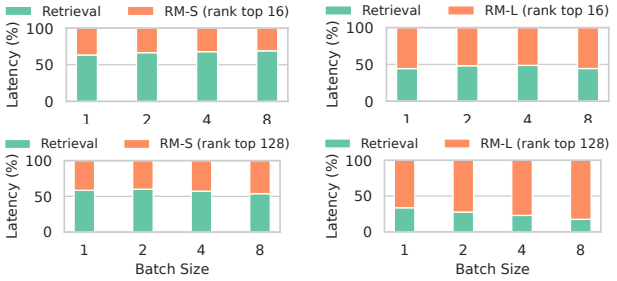


Figure 10: Latency breakdown of end-to-end recommendation using a CPU for retrieval and a GPU for inference, across different model sizes and numbers of candidates to rank.

based on the Criteo TB dataset and includes 26 embedding tables, while the larger model (RM-L) comprises over 100 tables, aligning with recent industry-scale recommender systems [36, 47, 48]. We assume that each recommendation request first performs an ANN search to identify candidate items, followed by ranking the top 16 or 128 candidates via model inference. We adopt NVIDIA's inference implementation [6] and evaluate it on a V100 GPU (same as the vector search baseline), assuming that the embedding tables fit in GPU memory. For the ANN search, we use the latency on the SIFT dataset with HNSW as a reference. Figure 10 shows the latency breakdown when using a typical CPU-GPU architecture, where the CPU is responsible for ANN search. Depending on the model sizes and the number of candidates to rank per request, the percentage of time spent on retrieval can range from 17.56~68.92%. Thus, by replacing the CPU with Falcon, the end-to-end speedup for recommender system ranges from 1.03~1.60×.

RAG Systems. Similar to recommender systems, RAG pipelines can be built using various LLMs [50, 52]. We evaluate the performance of LLaMA models of varying sizes (ranging from 1B to 13B parameters) across different GPUs (from NVIDIA V100 to B100), using the Generative LLM Analyzer [13]. As in the recommender setting, we use the latency of vector search on the SIFT dataset with HNSW as a reference for retrieval time. We assume the prompt, including both the query and retrieved documents, has a total length of 512 tokens [50]. Prompt computation is done with a batch size of one, as it already performs token-level batching inherently and can fully utilize the GPU without request-level batching [79, 109]. The left side of Figure 11 shows the inference latency of prompt computation given a single GPU, while the right side illustrates the percentage of time-to-first-token (TTFT) latency spent on CPU-based retrieval. As GPU capability improves, inference latency decreases significantly. For example, for the 1B model, TTFT latency drops from 7 ms on a V100 to just 0.25 ms on a B100. As a result, the proportion of retrieval time in the overall TTFT latency increases—from 4.83% to 59.20%. Consequently, the end-to-end TTFT

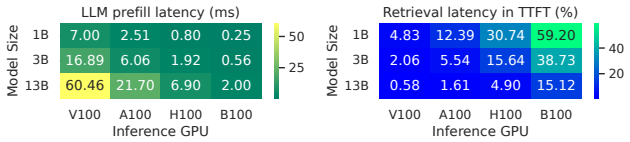


Figure 11: RAG latency across inference and retrieval stages.

speedup by deploying Falcon depends on both model size and GPU backend, and can reach up to 1.60× given advanced GPUs.

6 DISCUSSION

We have shown the performance advantages of Falcon and DST over CPUs and GPUs through FPGA-prototyping. We now discuss potential future extensions of the prototype to enable broader deployments, including adding more functionalities, supporting larger-scale searches, and achieving even higher efficiency.

Handling insertions and updates. To support data insertions, deletions, or updates in Falcon, one could refer to the designs of software vector search systems. They typically manage a primary index for a dataset snapshot, an incremental (smaller) index for newly added vectors since the last snapshot, and a bitmap marking deleted vectors [95]. These two indexes are merged periodically, e.g., daily, into a new primary index. Falcon can adopt this approach by focusing on serving the primary index, while the incremental index remains small enough to be efficiently managed by CPUs.

Scale-out the system. We have not yet scaled out Falcon due to the limited number of FPGAs available. However, we expect the scale-out design to be similar to software-based GVS systems [26]. Specifically, the dataset is partitioned into subsets, each associated with a graph managed by a separate Falcon node. Queries are then directed to one or several of these partitions, with the results subsequently aggregated.

Effectiveness on various memory and storage backends. Although Falcon is evaluated using an FPGA prototype with DDR memory, DST is applicable to a wide range of memory and storage backends. This is because its key idea — improving the utilization of compute units and memory bandwidth — is not tied to any specific hardware. For instance, SSDs exhibit access latencies that are over an order of magnitude higher (e.g., 20µs [57]) than DRAM (e.g., <100ns [17]). Under the high-latency condition and given the BFS traversal strategy, compute units often remain idle while waiting for data, as illustrated in stages S1 and S3 of Figure 3. By contrast, DST's aggressive node exploration strategy enables on-the-fly processing of many nodes, thereby improving compute unit utilization and reducing overall search latency.

Extensions for alternative hardware. The Falcon architecture is not specific to the FPGA platform used in our evaluation and can be applied to other hardware backends, including ASICs and various systems containing FPGAs. For example, Falcon can be implemented on FPGA-based data processing units (DPUs) acting as SmartNICs [1, 3]. In such deployments, the FPGA may access not only its local device memory, but also host server memory and potentially remote memory, enabling larger-scale vector search. ASIC-based instantiation will not only offer higher energy efficiency, but also provide flexibility in memory technology choices. For instance, integrating HBM into an ASIC-based Falcon design can deliver both low latency and high throughput that is comparable to GPU-based systems.

7 RELATED WORK

ANN Search Algorithms. Researchers have developed various ANN search algorithms [24, 30, 42, 46, 71, 77, 88, 93, 100, 108, 110] and vector data management systems [35, 67, 76, 78, 90, 95, 101]. Many variants of graph construction algorithms for GVS have also been proposed [11, 28, 29, 70, 72, 73, 81, 97, 107, 111], as GVS can achieve high recall with low latency. Apart from GVS, other ANN search indexes offer different trade-offs between indexing cost and search performance. For example, locality-sensitive hashing (LSH) and inverted-file (IVF) indexes are indexing techniques that partition the vector space. Locality-sensitive hashing (LSH) [25, 32] offers theoretical guarantees for ANN search, but empirically does not perform as well as graph-based algorithms. IVF indexes empirically outperform LSH, but still require scanning more database vectors than GVS to achieve the same recall [29, 63]. Beyond indexing, product quantization (PQ) [31, 45] is a widely adopted approach to compress high-dimensional vectors into compact byte codes. Often combined with IVF [31, 45] or graph indexes [33, 44], PQ is particularly prevalent in large-scale ANN search, where reducing memory footprint is crucial, although this lossy compression technique can degrade recall beyond that caused by the indexes.

Vector search on modern hardware. Beyond software optimizations [10, 82], researchers have proposed various hardware-based solutions for vector search. Exact kNN search can be accelerated by TPUs [21] or FPGAs [69, 102]. For ANN search, Faiss is a popular GPU-accelerated library [53], and there are several other implementations for PQ-based vector search [19, 20, 68, 96] and GVS [34, 106]. Lee et al. [59] study ASIC designs for IVF-PQ, and several works [49, 51, 104] implement IVF-PQ on an FPGA, although their designs are constrained by either the limited HBM capacity or the speed of the CPU-FPGA interconnect. Several works propose to push down vector search to storage to improve performance by reducing data movements [39, 56, 64, 65, 94, 99]. Besides, the database vectors can be stored in non-volatile memory [84] or CXL [43] to scale up GVS, while on-disk GVS must carefully manage I/O costs [18, 44, 61, 91], similar to other graph processing workloads [85, 89].

8 CONCLUSION

To meet the surging demands of online GVS, we propose Falcon, a high-performance GVS accelerator, and DST, an accelerator-optimized traversal algorithm. Evaluated across various graphs and datasets, they shows up to 4.3× and 19.5× speedup in online search latency compared to CPUs and GPUs, while being up to 8.0× and 26.9× more energy efficient. These compelling results show the potential for Falcon and DST to become the standard solutions for GVS acceleration.

ACKNOWLEDGMENTS

We thank AMD for their generous donation of the Heterogeneous Accelerated Compute Clusters (HACC) at ETH Zurich (https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html), on which the experiments were conducted.

REFERENCES

[1] [n.d.]. AMD Alveo SN1000 SmartNIC Accelerator Card. https://www.amd.com/en/products/accelerators/alveo/sn1000/a-sn1022-p4.html.

[2] [n.d.]. Faiss. https://github.com/facebookresearch/faiss/.

[3] [n.d.]. Intel FPGA SmartNIC N6000-PL Platform. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic/n6000-pl-platform.html.

[4] [n.d.]. The Memory Wall: Past, Present, and Future of DRAM. https://semianalysis.com/2024/09/03/the-memory-wall/.

[5] [n.d.]. The MurmurHash family. https://github.com/aappleby/smhasher.

[6] [n.d.]. NVIDIA Deep Learning Recommender Model Implementation. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Recommendation/DLRM.

[7] [n.d.]. The NVIDIA GH200 Grace Hopper Superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip.

[8] [n.d.]. SIFT ANNS dataset. http://corpus-texmex.irisa.fr/

[9] [n.d.]. The SPACEV Web Embedding Dataset. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B.

[10] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. 12.

[11] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[12] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[13] Abhimanyu Bambhaniya, Ritik Raj, Geonhwa Jeong, Souvik Kundu, Sudarshan Srinivasan, Midhilesh Elavazhagan, Madhu Kumar, and Tushar Krishna. 2024. Demystifying Platform Requirements for Diverse LLM Inference Use Cases. arXiv:2406.01698 [cs.AR]

[14] Dimitri P Bertsekas. 1993. A simple and fast label correcting algorithm for shortest paths. *Networks* 23, 8 (1993), 703–709.

[15] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[16] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.

[17] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. 323–336.

[18] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2111.08566* (2021).

[19] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. 2019. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems* 99 (2019), 295–307.

[20] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. 2019. Robustiq: A robust ann search method for billion-scale similarity search on gpus. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*. 132–140.

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

[22] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 2021. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 48–50.

[23] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[24] Michele Dallachiesa, Themis Palpanas, and Ihab F Ilyas. 2014. Top-k nearest neighbor search in uncertain data series. *Proceedings of the VLDB Endowment* 8, 1 (2014), 13–24.

[25] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[26] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2020. LANNS: a web-scale approximate nearest neighbor lookup system. *Proceedings of the VLDB Endowment* (2020).

[27] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

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

[29] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[30] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[31] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

[32] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[33] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[34] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[35] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: A Cloud Native Vector Database Management System. *arXiv preprint arXiv:2206.13843* (2022).

[36] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. 2020. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 982–995.

[37] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: Retrieval-augmented language model pre-training. *arXiv preprint arXiv:2002.08909* (2020).

[38] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *2021 31th International Conference on Field Programmable Logic and Applications (FPL)*.

[39] Han-Wen Hu, Wei-Chen Wang, Yuan-Hao Chang, Yung-Chun Lee, Bo-Rong Lin, Huai-Mu Wang, Yen-Po Lin, Yu-Ming Huang, Chong-Ying Lee, Tzu-Hsiang Su, et al. 2022. ICE: An Intelligent Cognition Engine with 3D NAND-based In-Memory Computing for Vector Similarity Search Acceleration. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 763–783.

[40] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.

[41] Muhuan Huang, Kevin Lim, and Jason Cong. 2014. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.

[42] Qiang Huang, Yifan Lei, and Anthony KH Tung. 2021. Point-to-Hyperplane Nearest Neighbor Search Beyond the Unit Hypersphere. In *Proceedings of the 2021 International Conference on Management of Data*. 777–789.

[43] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} Collaborative Memory Disaggregation and Computation for {Billion-Scale} Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 585–600.

[44] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[45] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[46] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2015. Exact top-k nearest keyword search in large networks. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 393–404.

[47] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. 2021. MicroRec: efficient recommendation inference by hardware and data structure solutions. *Proceedings of Machine Learning and Systems* 3 (2021), 845–859.

[48] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, et al. 2021. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3097–3105.

[49] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. 2023. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[50] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. In *Proceedings of the 52th Annual International Symposium on Computer Architecture*.

[51] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2025. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *Proceedings of the VLDB Endowment* 18 (2025).

[52] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2025. Piperag: Fast retrieval-augmented generation via adaptive pipeline parallelism. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2025).

[53] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019).

[54] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906* (2020).

[55] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[56] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. 2022. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Trans. Comput.* 72, 1 (2022), 278–290.

[57] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. 2018. Exploring system challenges of {ultra-low} latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[58] James LaGrone, Ayodunni Aribuki, and Barbara Chapman. 2011. A set of microbenchmarks for measuring OpenMP task overheads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Citeseer, 1.

[59] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W Lee, and Tae Jun Ham. 2022. ANNA: Specialized Architecture for Approximate Nearest Neighbor Search. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 169–183.

[60] Charles E Leiserson. 1979. *Systolic Priority Queues*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[61] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. 2008. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 5 (2008), 869–883.

[62] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[63] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[64] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive {SSD}: A deep learning engine for {In-Storage} data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 395–410.

[65] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. 2022. VStore: in-storage graph based vector search accelerator. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 997–1002.

[66] Peng-Cheng Lin and Wan-Lei Zhao. 2019. Graph based nearest neighbor search: Promises and failures. *arXiv preprint arXiv:1904.02077* (2019).

[67] Shige Liu, Zhifang Zeng, Li Chen, Adil Ainihaer, Arun Ramasami, Songting Chen, Yu Xu, Mingxi Wu, and Jianguo Wang. 2025. TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs. *arXiv preprint arXiv:2501.11216* (2025).

[68] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2023. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. *arXiv preprint arXiv:2312.01712* (2023).

[69] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. 2020. CHIP-KNN: A configurable and high-performance k-nearest neighbors accelerator on cloud FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 139–147.

[70] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 15, 2 (2021), 246–258.

[71] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient processing of k nearest neighbor joins using mapreduce. *arXiv preprint arXiv:1207.0141* (2012).

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

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

[74] Dinesh P Mehta and Sartaj Sahni. 2004. *Handbook of data structures and applications*. Chapman and Hall/CRC.

[75] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[76] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[77] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive top-k nearest neighbors search in large road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1781–1795.

[78] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).

[79] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677* (2023).

[80] Hongwu Peng, Shiyang Chen, Zhepeng Wang, Junhuan Yang, Scott A Weitze, Tong Geng, Ang Li, Jinbo Bi, Minghu Song, Weiwen Jiang, et al. 2021. Optimizing fpga-based accelerator design for large-scale molecular similarity search (special session paper). In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–7.

[81] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[82] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 313–328.

[83] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[84] Jie Ren, Minjia Zhang, and Dong Li. 2020. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems* 33 (2020), 10672–10684.

[85] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[86] Josef Sivic and Andrew Zisserman. 2003. Video Google: A text retrieval approach to object matching in videos. In *Computer Vision, IEEE International Conference on*, Vol. 3. IEEE Computer Society, 1470–1470.

[87] Ján Suchal and Pavol Návrat. 2010. Full text search engine as scalable k-nearest neighbor recommendation system. In *IFIP International Conference on Artificial Intelligence in Theory and Practice*. Springer, 165–173.

[88] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[89] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic {I/O} Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 507–522.

[90] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[91] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional

vector similarity search on data segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[92] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment* (2021).

[93] Xiaoyang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Muhammad Aamir Cheema. 2015. Optimal spatial dominance: an effective search of nearest neighbor candidates. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 923–938.

[94] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai Li, Yiran Chen, et al. 2023. In-Storage Acceleration of Graph-Traversal-Based Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2312.03141* (2023).

[95] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[96] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. 2016. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2027–2035.

[97] Yubao Wu, Ruoming Jin, and Xiang Zhang. 2014. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. 1139–1150.

[98] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate nearest neighbor negative contrastive learning for dense text retrieval. *arXiv preprint arXiv:2007.00808* (2020).

[99] Weihong Xu, Junwei Chen, Po-Kai Hsu, Jaeyoung Kang, Minxuan Zhou, Sumukh Pinge, Shimeng Yu, and Tajana Rosing. 2023. Proxima: Near-storage Acceleration for Graph-based Approximate Nearest Neighbor Search in 3D NAND. *arXiv preprint arXiv:2312.04257* (2023).

[100] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. 2015. Reverse k nearest neighbors query processing: experiments and analysis. *Proceedings of the VLDB Endowment* 8, 5 (2015), 605–616.

[101] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.

[102] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. 2022. {FAERY}: An {FPGA-accelerated} Embedding-based Retrieval System. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 841–856.

[103] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. 2023. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. (2023).

[104] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2018. Efficient large-scale approximate nearest neighbor search on OpenCL FPGA. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4924–4932.

[105] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. 2020. A study of single and multi-device synchronization methods in Nvidia GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 483–493.

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

[107] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.

[108] Yuxin Zheng, Qi Guo, Anthony KH Tung, and Sai Wu. 2016. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *Proceedings of the 2016 International Conference on Management of Data*. 2023–2037.

[109] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670* (2024).

[110] Huaijie Zhu, Xiaochun Yang, Bin Wang, and Wang-Chien Lee. 2016. Range-based obstructed nearest neighbor queries. In *Proceedings of the 2016 International Conference on Management of Data*. 2053–2068.

[111] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2645–2658.