VEX: Scaling HNSW-Based Vector Search with DPU Memory and Parallelism

Kihwan Kim*,†, Hyungsun Yoo*,†, Woojung Kim*, Donghyun Min*, Myungcheol Lee‡, Jihoon Yang*
Weikuan Yu§, Youngjae Kim*,¶
*Sogang University, ‡ETRI, §Florida State University
{lewis461, hsyoo, wjk216, mdh38112, yangjh, youkim}@sogang.ac.kr, mclee@etri.re.kr, wyu3@fsu.edu

***Abstract*—Vector similarity search is a core component of modern AI services, and HNSW is widely adopted due to its high recall and low latency. However, its memory-intensive design makes billion-scale deployment difficult, and performance collapses when relying on swapping or remote memory. This paper targets recent DPUs (SmartNICs) with substantially improved compute capability and onboard DRAM, and proposes VEX, a host–DPU integrated vector search system that uses the DPU as both an extended memory tier and a parallel search engine for HNSW. VEX (i) partitions and places independent HNSW indices on the host and DPU while preserving semantic structure, (ii) minimizes host–DPU overhead via a dual-path DMA-based communication design, and (iii) overlaps search, communication, and aggregation with heterogeneity-aware pipelining. Experiments show that under memory pressure requiring disk access, VEX delivers 5–10× higher throughput than DiskANN at stable Recall@100. Even in ideal settings where the index fully resides in memory, VEX outperforms in-memory HNSW by up to 1.9× in query throughput.**

***Index Terms*—Vector Search, ANN, HNSW, DPU, SmartNIC, DMA, Heterogeneous architecture, Memory disaggregation, Hardware-Software Co-Design**

I. INTRODUCTION

Vector similarity search plays a fundamental role in a wide range of modern applications, including retrieval-augmented generation (RAG) [1], intelligent recommendation systems, and semantic search. These services require fast and accurate identification of semantically similar items in high-dimensional vector spaces, for which approximate nearest neighbor search (ANN) techniques are widely adopted [2]. In particular, graph-based ANN indices that explicitly encode vector proximity as graph connectivity (e.g., HNSW [3]) have emerged as the industry standard for large-scale vector search workloads, as they achieve both high recall and low latency [4].

HNSW (Hierarchical Navigable Small World) organizes vectors into a multi-layer graph structure, where search progressively narrows candidates from upper layers and performs fine-grained exploration at lower layers. However, HNSW requires storing both graph connectivity information and full-precision embedding vectors, resulting in substantial memory consumption. In practice, datasets with tens of millions of vectors can yield HNSW indices occupying hundreds of gigabytes, which exceeds the capacity of main memory on a single server [5]–[8]. Moreover, ANN search over HNSW inherently involves irregular memory access patterns, causing the working set size to grow rapidly and further amplifying memory pressure [9].

As multimodal data continues to proliferate, large-scale vector search systems are increasingly moving beyond single-node ANN engines toward settings where multiple clients or service instances concurrently access a shared, massive vector dataset [10]. In such settings, maintaining a copy of the index at each serving node is inefficient, and a centralized remote shared storage is often preferred. This approach reduces storage and management overhead caused by index replication and enables horizontal scaling without rebuilding or redistributing the index as serving nodes are added or removed. However, graph-based ANN engines such as HNSW assume that the index resides in the host memory space of each serving node. As index sizes grow, it becomes increasingly difficult to accommodate them within node-local DRAM alone. Although disk swapping or memory-mapped I/O can be used as mitigation under memory pressure, their effectiveness is limited due to the high latency of disk accesses (Table I).

Meanwhile, SmartNICs such as NVIDIA BlueField DPUs [11]–[13] provide an important opportunity to revisit the aforementioned limitations in shared storage environments [14]. SmartNICs were originally introduced to offload data movement and protocol processing from the network and storage paths to the NIC, thereby reducing host CPU involvement and streamlining the data path. With the emergence of DPUs such as the BlueField family, which integrate independent ARM cores and onboard DRAM, the role of SmartNICs has expanded beyond simple packet processing to support on-device data processing. Accordingly, prior research has largely focused on offloading portions of data processing to DPUs in order to reduce host CPU involvement along network and storage stack paths [15]–[19].

However, the generational evolution of BlueField DPUs [11]–[13] has shifted the role of the DPU from a simple offloading device to a practical intermediate memory tier located between host DRAM and storage. For example, BlueField-3 [12] provides dozens of ARM cores and tens of gigabytes of onboard DRAM, enabling a portion of the working set to reside on the device. BlueField-4 [13] further strengthens this capability by offering larger onboard memory capacity, enhanced memory and I/O subsystems, and substantially increased compute resources (Table II). These advances support the use of DPUs

†K. Kim and H. Yoo contributed equally to this work.
¶Corresponding author.

as execution tiers that not only participate in data movement but also retain data in device memory and perform computation locally. In essence, by absorbing part of the working set into DPU memory and exploiting the DPU's internal cores for parallel processing, new opportunities emerge to alleviate host memory pressure and reduce storage access frequency.

In this work, we propose VEX, the first attempt to mitigate the memory pressure of HNSW-based ANN search in a three-tier memory architecture, where the DPU is utilized as an intermediate tier between host DRAM and storage (e.g., SSD). In particular, when the non-disk memory space is extended beyond host DRAM to include DPU memory, placing a portion of the HNSW index on the DPU reduces disk accesses and enables independent, parallel execution of distance computations and graph traversals using DPU cores, thereby improving overall throughput.

However, HNSW-based ANN search in host–DPU heterogeneous environments involves fundamental design challenges that go beyond merely increasing the available memory capacity. First, since HNSW represents graph connectivity in the vector space, naively distributing index data across different memory regions leads to frequent cross-region accesses during search, resulting in throughput degradation. Restricting search to a specific memory region to reduce such overhead can, in turn, induce disconnections in search paths when partitioning is performed without considering vector distributions, thereby degrading ANN accuracy. Second, interaction between the host and the DPU during similarity search incurs communication overhead. For instance, processing a query vector typically involves DMA-based transfers to the DPU, and the repeated DMA preparation costs can accumulate and increase overall communication latency. Finally, the host and the DPU are inherently asymmetric in compute and memory capacity, and the workload assigned to each device can vary dynamically during execution. As a result, completion times across the two devices are frequently skewed, resulting in idle periods in which one device waits for the other to finish. This imbalance increases resource idle time and reduces overall system utilization.

Key Insights and Technical Contributions. To address the aforementioned design challenges, VEX incorporates the following techniques.

- **First**, VEX partitions the dataset by explicitly preserving the semantic structure of the vector distribution and places independent HNSW sub-indices across the host and the DPU. To mitigate search path disconnections at partition boundaries, VEX selectively replicates boundary vectors to reinforce global connectivity, thereby maintaining high recall even under partitioned deployment.
- **Second**, since frequent host–DPU interactions accumulate communication latency and limit throughput, VEX routes each query to complete its search within a single partition. Data transfers are performed over a DMA-based PCIe path, and to minimize recurring DMA preparation overhead at runtime, VEX employs a pre-registered, ring-based send/receive buffer structure in host memory, effectively reducing data transfer overhead.
- **Third**, VEX overlaps the search, communication, and result aggregation stages in a pipelined manner, hiding system idle time caused by mismatched completion times across heterogeneous execution paths and maximizing overall system utilization.

Experimental results show that, in environments where large-scale vector indices cannot fully reside in host DRAM and disk-based access becomes inevitable, VEX achieves approximately 5-10× higher query throughput than the state-of-the-art disk-based ANN method, DiskANN, while stably maintaining Recall@100. Moreover, even in ideal settings where the entire index fits in host memory, VEX leverages the DPU's parallel search resources to achieve up to 1.9× higher throughput compared to *HNSW(mem)*. These results demonstrate that VEX effectively scales HNSW-based vector search systems in large-scale vector environments by exploiting the DPU's onboard memory and compute resources.

II. Background and Motivation

A. Hierarchical Navigable Small Worlds (HNSW)

HNSW is a widely used proximity graph-based indexing algorithm for approximate nearest neighbor (ANN) search. As illustrated in Fig. 1(a), HNSW organizes vector nodes hierarchically across multiple layers, forming neighbor relationships within each layer. Based on the layout shown in Fig. 2, each vector can be indexed as a single node consisting of a unique ID, the original vector values, and a list of neighbor nodes connected at each layer.

Only a subset of the entire vector set is sampled and placed in the upper layers, which helps reduce the search space. In the lower layers, more nodes are included and connections become denser, enabling fine-grained search. This hierarchical structure allows HNSW to achieve high search accuracy while effectively reducing search latency.

During index construction, the highest level at which each vector appears is assigned probabilistically. HNSW then greedily traverses from the global entry point, selecting the closest neighbors at each level and connecting the new vector to a subset of them. Increasing the connection density strengthens the graph connectivity and accuracy, but it also increases memory pressure to store neighbor information.

HNSW Search Procedure. HNSW search begins at the topmost layer by selecting a single entry node as the starting point, as shown in Fig. 1(a). During the search, both the entry node and its connected neighbors are included as candidate neighbors. The similarity distance between each candidate node and the query vector is then computed. Among these neighbors, the closest node is chosen as the new entry point for the next lower layer. This process continues sequentially until the lowest layer is reached.

At the lowest layer, the search evaluates the candidate set accumulated from the nodes visited during the traversal. The closest nodes to the query vector are selected, and distances to

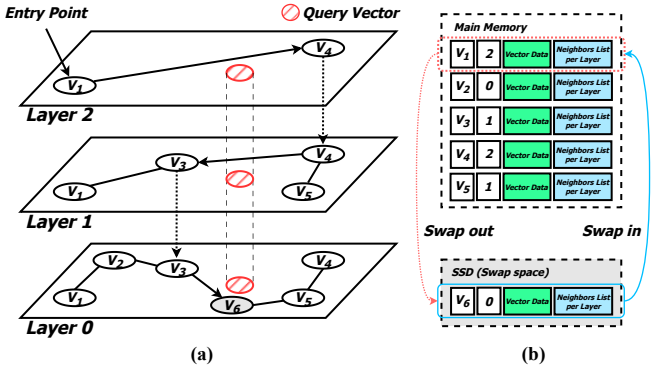

Fig. 1: (a) An architectural overview of HNSW and (b) Swap.

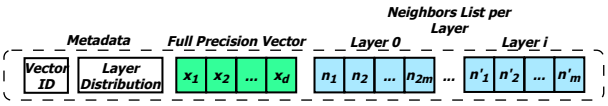

Fig. 2: HNSW Index Layout.

their connected neighbors are computed to expand and update the candidate set. This iterative process repeats until no further updates occur in the candidate set. Finally, the algorithm returns the top candidates (e.g., top-K) that are closest to the query vector.

B. Memory Pressure and Thrashing in Large HNSW Indices

HNSW exhibits highly irregular memory access patterns during the graph traversal process. In particular, during the search at the lowest layer, HNSW keeps multiple candidate nodes simultaneously and repeatedly expands the neighbors of these candidates. As a result, HNSW traversal inherently accesses nodes that are scattered across memory in a non-contiguous and irregular manner.

Due to these access patterns, HNSW reduces search latency by keeping as much of the index data as possible resident in memory. As illustrated in Fig. 2, each node stores not only its corresponding full precision vector but also maintains neighbor node IDs for each layer in the form of in-memory adjacency lists. Consequently, HNSW must simultaneously retain both full-precision vector data and multi-level graph metadata in memory, significantly increasing memory pressure. This effect becomes more pronounced as the vector dimensionality increases or as the graph connectivity becomes denser.

For example, in datasets such as TriviaQA (Table III), which contain tens of millions of high-dimensional embeddings, the accumulation of vector data and graph metadata results in HNSW index sizes reaching tens to hundreds of gigabytes, and potentially scaling to terabytes depending on the vector dimensionality and connectivity. When such large HNSW indices are deployed in memory-constrained environments, the operating system's virtual memory policy causes parts of the index to be evicted to the local disk swap space, as shown in Fig. 1(b).

However, the logical traversal order determined by the graph neighborhood structure in HNSW is not aligned with the physical memory layout of the nodes. As a result, the search process exhibits poor cache locality and irregular memory access patterns, leading to frequent swap-in/out operations whenever nodes residing in the swap space are accessed.

TABLE I: Impact of swapping on HNSW query throughput.

Configuration	QPS
HNSW (All in DRAM)	1413
HNSW (50% DRAM + 50% swap on local disk)	123

TABLE II: NVIDIA BlueField DPU specifications.

Specification	BF-2	BF-3	BF-4
CPU	Cortex-A72	Cortex-A78	Neoverse V2
Cores	8	16	64
Memory (GB)	16	32	128
PCIe Gen	Gen4 ×16	Gen5 ×16	Gen6 ×16
PCIe BW (GB/s)	32	64	128
Network BW (Gbps)	200	400	800

Since a single query may visit hundreds to thousands of nodes, the overhead of swapping accumulates rapidly, significantly increasing query latency. This issue becomes even more severe in batch-processing environments, where multiple queries are handled concurrently. As different queries simultaneously traverse different regions of the graph, the working set grows rapidly; once it exceeds available memory, required nodes are repeatedly evicted and swapped back in, causing severe thrashing. Consequently, both system throughput and response latency degrade sharply.

Table I demonstrates the drastic throughput degradation of HNSW under memory pressure when swapping occurs. While HNSW achieves 1413 QPS when the entire index resides in DRAM, the throughput sharply drops to 123 QPS when half of the index is evicted to local disk swap. These results clearly indicate that even with high-performance NVMe SSDs (Table V), severe performance degradation is unavoidable once swapping occurs during HNSW traversal.

Recognizing the practical constraint that large HNSW indices inevitably exceed physical memory capacity and partially spill into the storage tier, prior studies have redefined the problem by abandoning the assumption of fully in-memory indices and instead focusing on more efficient access to storage-resident index data [5], [20]–[22]. For instance, DiskANN [5] and Starling [20] depart from memory-centric designs by proposing SSD-based graph index traversal to reduce memory consumption. These approaches store the graph index on SSDs while retaining only compressed vectors in memory. However, due to unavoidable disk I/O during traversal, they suffer from increased search latency and reduced throughput compared to fully in-memory designs.

C. DPU-based Vector Search: Opportunities and Challenges

DPU as an Additional Compute and Memory Tier. In conventional SmartNICs and early DPU designs, network and I/O offloading was the primary use case, and the limited on-device resources made it impractical to directly execute user-level applications on the DPU. As a result, DPUs have traditionally been used only as auxiliary accelerators to the host CPU.

However, as shown in Table II, NVIDIA BlueField DPUs have continuously evolved across generations, with steady

improvements in the number of ARM cores, computational performance, parallel processing capability, and onboard memory capacity. Moreover, BlueField DPUs (DPU) support not only NIC mode but also DPU mode, depending on the operating configuration. In NIC mode, similar to traditional SmartNICs, the DPU focuses on offloading network- and I/O-related tasks such as packet processing, RDMA, and encryption/decryption. In contrast, in DPU mode, the internal ARM cores and onboard memory operate as an independent computing node, enabling user-level applications and portions of system software to execute directly on the DPU.

This evolution in hardware capabilities and execution models enables DPUs to move beyond their role as mere network and I/O accelerators and positions them as an additional compute and memory tier that can operate in parallel with the host. In particular, for workloads such as large-scale vector indexing, where memory access cost is a dominant performance bottleneck, this opens up new design opportunities in which the DPU's onboard memory can be leveraged as an extension of host memory while simultaneously executing search computation on the DPU.

Why DPU-based Vector Search is Non-Trivial. In this work, we leverage the DPU mode provided by NVIDIA BlueField to use the DPU's onboard memory as an extension of host memory. This enables the construction of a new intermediate memory tier between host memory and the disk tier, while simultaneously exploiting the DPU's ARM cores to execute vector search operations in parallel.

However, despite providing additional compute and memory resources, DPUs cannot be straightforwardly applied to ANN-based vector search. Unlike workloads with simple memory access patterns, graph-based vector indices such as HNSW strongly rely on the geometric structure of the vector space and the global connectivity across the entire index.

As a result, partitioning a large vector index across two heterogeneous memory regions—host memory and DPU onboard memory—introduces several fundamental challenges. First, naively placing a subset of the index on the DPU can break the global connectivity of the graph, leading to degraded search accuracy. Second, frequent back-and-forth accesses between the host and the DPU during search incur non-negligible communication overhead, offsetting the latency benefits of introducing an additional memory tier. Third, asymmetries in computation and memory access characteristics between the host and the DPU can reduce overall resource utilization, causing pipeline stalls during search execution.

III. Design of VeX

A. Design Principle

To address the challenges discussed in the previous section, VeX is designed based on the following three design principles.

- **Preserving Semantic Structure under Partitioning.** Graph-based vector indices strongly rely on both the geometric and semantic structure of the vector space, as well as on global connectivity across the index. Consequently, naive index partitioning that ignores these properties can lead to broken search paths and severe recall degradation. Therefore, even when the index is partitioned across host memory and DPU onboard memory, the design must preserve the semantic structure of the vector space. This principle ensures that search quality and connectivity are maintained across the distributed index.
- **Minimizing Communication Overhead across Heterogeneous Memory.** Vector search spanning multiple partitions inevitably involves request dispatch, partial result exchange, and synchronization for result aggregation. The communication overhead incurred in this process can become a dominant performance bottleneck. Accordingly, the design should minimize unnecessary host–DPU data movement and adopt DMA-centric communication mechanisms without CPU involvement. This approach reduces communication latency and synchronization costs, enabling efficient search across heterogeneous memory tiers.
- **Mitigating Load Imbalance under Heterogeneity.** The host and the DPU exhibit inherently different computational capabilities and execution latencies, causing imbalanced completion times among sub-batches after query routing. Since final result aggregation can proceed only after all partial searches are completed, synchronous execution that ignores such heterogeneity leads to increased idle time and reduced pipeline utilization. Therefore, the design must explicitly account for device heterogeneity by asynchronously coordinating and overlapping search, communication, and aggregation stages, thereby hiding execution delays and maximizing overall system efficiency.

B. Overview of VeX

VeX implements the above design principles through three concrete system mechanisms.

First, for semantic-aware index partitioning, VeX decomposes the embedding space into fine-grained sub-clusters using K-means, and then orders and selectively merges these sub-clusters along a principal axis extracted via PCA (PCA-guided sub-cluster merging). By distributing the merged sub-clusters across the host and the DPU, VeX partitions the index while preserving the geometric and semantic structure of the vector distribution. This approach mitigates graph connectivity loss and the resulting recall degradation. (Section III-C)

Second, VeX is designed to enable efficient and high-speed exchange of queries routed to the DPU and the corresponding search results returned to the host with minimal communication overhead. To this end, it separates control messages from data movement by constructing a lightweight control path and a low-latency DMA-based data path, and selects different transfer paths depending on the message type. To eliminate recurring DMA setup costs at runtime, VeX adopts pre-registered ring-based send/receive buffers in host memory, thereby minimizing host–DPU transfer overhead. (Section III-D)

Third, in batch query processing, the sub-batch workload and execution time assigned to the host and the DPU can vary across batches, often leading to skewed completion times

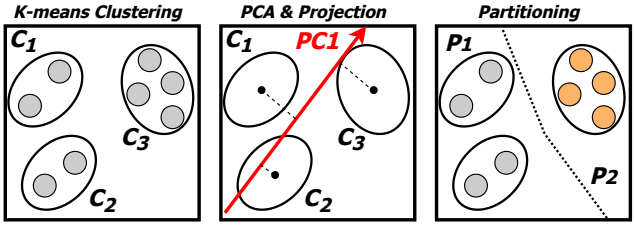

Fig. 3: Partitioning procedure of VEX.

between the two execution paths. To reduce idle time induced by such heterogeneity, VEX adopts a pipelined execution model that asynchronously coordinates and overlaps batch-level search, communication, and result aggregation stages. (Section III-E and III-F)

C. Semantic-Aware HNSW Index Partitioning

The index partitioning in VEX consists of three stages, as illustrated in Fig. 3: (1) K-means clustering, (2) PCA analysis and projection, and (3) partitioning. After partitioning, an independent HNSW index is constructed for each partition, and a Router Graph is built using the k centroids. To mitigate connectivity loss at partition boundaries, VEX further selects and replicates *boundary vectors* across adjacent partitions.

K-means Clustering: VEX first decomposes the entire dataset $X \in \mathbb{R}^{n\times d}$ into k sub-clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$ using K-means clustering, and obtains the corresponding centroids $\mathbf{c} = [c_1, \ldots, c_k]$. K-means iteratively assigns each vector to its nearest centroid, grouping semantically similar vectors into the same cluster. This fine-grained decomposition partitions the embedding space into small semantic units, providing flexibility to satisfy target memory ratios while preserving spatial locality during partition construction. Moreover, the k centroids serve as compact representatives of the overall data distribution and are directly reused for constructing the Router Graph.

PCA Analysis & Projection: The resulting k sub-clusters are then ordered using Principal Component Analysis (PCA) so that semantically adjacent clusters are grouped together. Specifically, given each cluster centroid $c_i \in \mathbb{R}^d$ and cluster size $n_i = |C_i|$, we first compute the weighted mean $\mu \in \mathbb{R}^d$ and perform centering as follows:

$$\mu = \frac{\sum_{i=1}^{k} n_i \cdot c_i}{\sum_{i=1}^{k} n_i}, \quad \tilde{c}_i = c_i - \mu \tag{1}$$

This centering step shifts the centroid distribution to the origin, clarifying relative positions and dominant directional trends.

Next, the first principal component (PC1) $v \in \mathbb{R}^d$, which captures the direction of maximum variance among the centered centroids, is approximated via power iteration:

$$v^{(t+1)} = \frac{\mathbf{C}^T\mathbf{C}v^{(t)}}{\|\mathbf{C}^T\mathbf{C}v^{(t)}\|} \tag{2}$$

where $\mathbf{C} \in \mathbb{R}^{k\times d}$ is the centered centroid matrix weighted by $\sqrt{n_i}$ (i.e., the i-th row is $\sqrt{n_i}\cdot\tilde{c}_i$), and t denotes the number of iterations. Power iteration typically converges within 10–25 iterations. The resulting PC1 represents the dominant semantic axis of the cluster distribution.

Each cluster centroid is then projected onto PC1 to obtain a scalar value $p_i = \tilde{c}_i \cdot v \in \mathbb{R}$, and the clusters are ordered according to these projection values. This projection embeds high-dimensional clusters onto a one-dimensional axis while preserving the principal semantic structure of the original space. Clusters that are adjacent along this axis tend to remain semantically close in the original embedding space.

Partitioning: The ordered clusters are sequentially scanned to identify split points that satisfy the target partition ratios $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, where $\sum_{j=1}^{m} \alpha_j = 1$:

$$\text{split}_j = \min\left\{\ell : \sum_{i=1}^{\ell} n_i \geq \left(\sum_{p=1}^{j} \alpha_p\right)\cdot N\right\}, \quad j = 1, \ldots, m-1 \tag{3}$$

Here, $N = \sum_{i=1}^{k} n_i$ denotes the total number of vectors, and split_j indicates the cluster index separating the j-th and $(j+1)$-th partitions. The partitions are then defined as:

$$P_j = \{C_i : \text{split}_{j-1} \leq i < \text{split}_j\}, \quad j = 1, \ldots, m \tag{4}$$

with $\text{split}_0 = 0$ and $\text{split}_m = k$. Because partitions correspond to contiguous segments along the projected axis, semantic disruption is largely confined to the partition boundaries, while intra-partition locality is preserved. By adjusting target partition size ratios$\{\alpha_1, \ldots, \alpha_m\}$, VEX flexibly adapts to diverse heterogeneous memory configurations.

Boundary Vector Replication: Cluster-based partitioning alone may fail to preserve global connectivity for vectors near partition boundaries, potentially degrading search accuracy. To address this issue, VEX performs boundary vector replication. For each vector $x \in X$, we compute the distance d_{own} to the centroid of its assigned partition P_j and the distance d_{neighbor} to the centroid of an adjacent partition P_{j+1}. The ratio is defined as $r = d_{\text{own}}/d_{\text{neighbor}}$. Vectors with r close to 1 are likely to lie near partition boundaries. By selecting vectors with $r > \tau$, where $\tau \in (0, 1)$ is a threshold, we identify the boundary vector set B for partition P_j, which are then replicated to the neighboring partition P_{j+1}. The same procedure is applied in the opposite direction to identify boundary vectors in P_{j+1} with respect to P_j, enabling bidirectional boundary replication.

HNSW Index Construction and Router Graph: After partitioning, independent HNSW indices are constructed for each partition P_{host} and P_{dpu}. At the same time, all k centroids obtained from the initial K-means clustering are used as representative vectors to build the Router Graph. Since each centroid captures the semantic center of its cluster, the Router Graph provides a fine-grained representation of the global data distribution while remaining compact (on the order of hundreds of KB to a few MB). The Router Graph resides in host memory and determines which partition should be searched for each incoming query.

D. Efficient Host–DPU Communication Path

In host–DPU integrated vector search, the communication path between the host and the DPU is a critical component that directly determines overall system throughput. The host and the DPU are directly connected via PCIe within the same

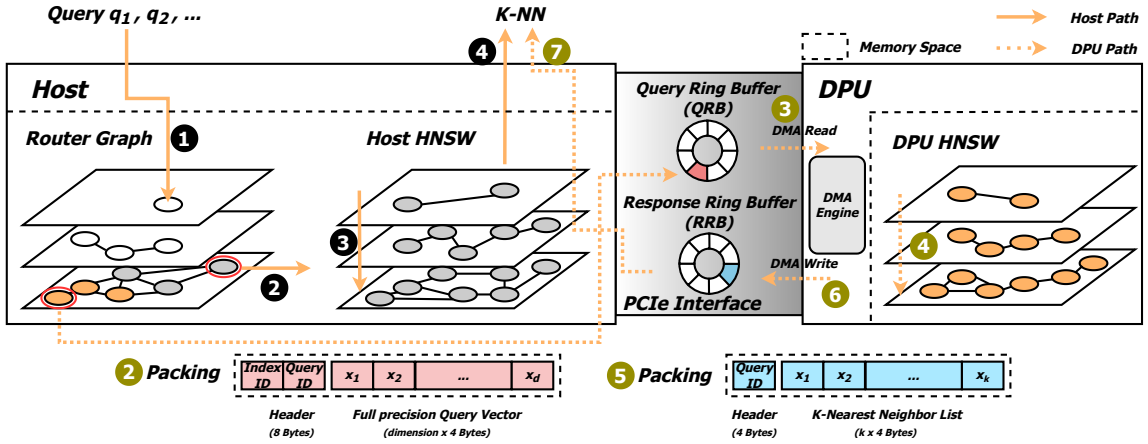

Fig. 4: Hardware–software architecture and execution flow of VEX.

node, and the DPU can read from and write to host memory with low latency using its hardware DMA engine [23].

DMA-based data transfers, however, require not only issuing transfer commands but also preparing the host memory through pinning/registration and establishing remote mappings on the DPU side. These preparation steps incur fixed overheads independent of payload size, which can dominate total communication latency when request/response round trips are repeatedly invoked in batch processing. Accordingly, VEX designs the host–DPU communication path such that PCIe/DMA is used for data movement, while pinning/registration and mapping overheads are confined to the initialization phase and are not repeated along the runtime execution path. To this end, VEX leverages DOCA [24] communication primitives, using COMCH (communication channel) [25] for control message exchange and DMA for data transfer.

During initialization, the host allocates reusable fixed-size circular buffers and registers them for DMA access, then shares the corresponding memory mapping information with the DPU once. Based on this information, the DPU establishes and caches the remote mappings, which are reused throughout runtime. At execution time, control messages and data transfers are handled via separate paths. Lightweight control messages are exchanged through COMCH, while query vectors and top-k results are transferred exclusively via DMA, with the DPU's DMA engine directly reading from and writing to host memory over PCIe. COMCH is used to manage request ordering, buffer state, and completion notification, whereas DMA performs only descriptor-specified memory transfers. This design eliminates per-request buffer preparation or remapping, enabling repeated reuse of fixed buffers along the critical execution path.

Concretely, the host manages a Query Ring Buffer (QRB) and a Response Ring Buffer (RRB), each divided into multiple slots, where a slot represents the minimum unit for request or response placement in the circular buffer. The host writes a request—consisting of a Query ID, target partition, and query vector array—into the next available QRB slot, and then sends a slot descriptor (buffer type, slot offset, payload length, and request identifier) via COMCH to trigger DPU processing. The DPU retrieves the request data by issuing a DMA read to the specified QRB slot, performs the search, and writes the results to the designated RRB slot via a DMA write. Upon completion, the DPU notifies the host through COMCH, after which the host collects the results from the RRB and merges them with the results obtained from the host-side HNSW index.

Overall, VEX concentrates pinning/registration and mapping overheads in the initialization phase, and constructs runtime request/response round trips using reusable slots and lightweight control-message exchange. This design keeps host–DPU communication overhead low while sustaining high throughput in heterogeneous vector search workloads.

E. Vector Search Procedure

Fig. 4 illustrates the overall architecture of VEX and the batch query processing flow in a heterogeneous host–DPU system. The system places distinct HNSW indices on the host and the DPU, and processes queries by exchanging data between the host and the DPU over the PCIe interface. In the following, the batch query processing flow of VEX is described, focusing on routing, sub-batch processing, and result aggregation.

Batch Routing via Router Graph: Given an input batch $Q = \{q_1, \ldots, q_n\}$, the dataset is distributed across host and DPU memory, and thus each query q_i must first be assigned to the appropriate index partition (host or DPU). To this end, ❶ The host routes all queries in the batch using a lightweight Router Graph residing in host memory, determining the search path for each q_i. Based on the routing results, the batch is split into a host processing set Q_{host} and a DPU processing set Q_{dpu}. Queries destined for the same path are then regrouped into host sub-batches and DPU sub-batches. Because the Router Graph is compact and incurs negligible traversal cost, routing overhead is minimal, and routing is completed entirely on the host without any host–DPU communication.

Host Sub-batch Processing: Host sub-batches that are routed to the host-resident HNSW index are processed along the host path. ❷ The host dispatches each host sub-batch to the corresponding host HNSW index for batch execution. ❸ Each query follows the standard HNSW search procedure, traversing the graph in a top-down manner to progressively

narrow the candidate set and expanding candidates at Layer 0 using dense neighborhood connections. ❹ The host produces k nearest-neighbor candidates for each query in the sub-batch and forwards the results to the aggregation stage for merging with DPU-side results.

DPU Sub-batch Processing: DPU sub-batches that are routed to the DPU-resident HNSW index are processed along the DPU path. ❷ The host packs one or more DPU sub-batches into request messages. As illustrated in the lower part of Fig. 4, each request message consists of a header (Index ID and Query ID) and full-precision query vectors (or an array of query vectors), which are written into a QRB slot in host memory. ❸ The DPU retrieves the request data from the corresponding QRB slot via a DMA read into DPU memory. ❹ DPU parses the request, extracts query vectors and partition information, and performs sub-batch search on the corresponding DPU HNSW index. Once the search completes, ❺ the DPU constructs a response message containing the Query ID and the k-nearest-neighbor list (K-NN). ❻ The response is written to the RRB slot in host memory via a DMA write. Finally, ❼ the host retrieves the response from the RRB slot and extracts the results for each query.

Batch Result Aggregation: At the final stage, the host aggregates the results obtained from the host path (step ❹) and the results received from the DPU path (step ❼) to construct the final K-NN results for each query in the batch. As described above, VEX performs batch processing by reorganizing queries into path- and partition-specific sub-batches based on Router Graph routing, executing searches on the host and DPU in parallel, and then merging the results. This design amortizes host–DPU data transfers and search overhead at the sub-batch granularity, enabling high throughput even in heterogeneous execution environments.

F. Heterogeneity-Aware Pipeline Overlap

In VEX, batch queries are first routed and then processed in parallel on the host-resident index and the DPU-resident index. However, depending on the routing outcome, the size and difficulty of each sub-batch can differ even within the same batch, and the two devices inherently exhibit asymmetric compute capabilities and memory access characteristics. As a result, the processing time of the host path and the DPU path can vary significantly across batches, and the batch completion time can be approximated as follows:

$$T_{\text{batch}} \approx T_{\text{route}} + \max\big(T_{\text{host}},\, T_{\text{dpu}} + T_{\text{comm}}\big) + T_{\text{merge}} \quad (5)$$

Here, T_{route} denotes the time for routing and request packing; T_{host} and T_{dpu} represent the sub-batch search time on the host and the DPU, respectively; T_{comm} corresponds to host–DPU data transfer and synchronization overhead; and T_{merge} accounts for result aggregation and reordering.

As shown in Equation 5, the execution time of each batch is dominated by $\max(T_{\text{host}},\, T_{\text{dpu}} + T_{\text{comm}})$. Consequently, even if one execution path finishes early, it must wait for the slower path, leading to idle periods that directly limit throughput. The faster path remains underutilized while waiting for the slower one, reducing overall device utilization and constraining system performance.

To mitigate this inefficiency, VEX pipelines batch processing such that host-side search, DPU-side search, and result aggregation are overlapped in time as much as possible. Instead of stalling execution until a single batch fully completes, VEX allows a bounded number of batches to be in flight simultaneously, enabling different batches to occupy different pipeline stages concurrently. With this design, even if one path is delayed for a particular batch, the other path can continue processing subsequent batches, structurally reducing batch-level waiting time.

G. Integrating VEX with the Storage Tier

VEX aims to mitigate the memory pressure of HNSW by leveraging both host DRAM and DPU onboard memory. However, for billion-scale datasets, the index size may exceed the combined capacity of host and DPU memory, requiring a portion of the index to be placed on auxiliary storage such as node-local disks or remote storage. To address this limitation, VEX can be extended to a three-tier memory hierarchy composed of host DRAM, DPU memory, and storage.

Specifically, VEX keeps the indices for a subset of partitions resident in the host and DPU memory tiers to enable low-latency search, while placing the remaining partitions in the storage tier and indexing them using DiskANN [5], an ANN indexing technique designed for disk-based access. The fraction of partitions assigned to the memory tiers is determined by the total available capacity of host DRAM and DPU memory, while the remaining partitions are placed in the storage tier.

At query time, VEX performs query routing using the Router Graph to determine the most relevant partition for each query. Based on the routing result, the query is dispatched to the tier that hosts the corresponding partition. Queries routed to in-memory partitions follow the VEX execution path described in Section III-E and are processed using the HNSW index residing in host or DPU memory. In contrast, queries routed to partitions in the storage tier are processed using DiskANN.

Importantly, each query is served by either the in-memory HNSW path or the storage-tier DiskANN path, without performing cross-tier graph traversal within a single query. This design avoids exposing HNSW's irregular memory access pattern directly to disk I/O, which would otherwise lead to severe thrashing and unpredictable latency.

In this architecture, the DPU serves as an intermediate memory tier between the host and storage by absorbing a working set of partitions, thereby reducing the frequency of storage accesses. Overall, this hybrid design reframes memory overflow from an OS-level swapping problem into a structured multi-tier index placement strategy, allowing VEX to achieve scalable and stable ANN search performance even when the index size exceeds the combined host and DPU memory capacity.

H. Implementation

VEX is implemented by extending FAISS [26] 1.12.0 with an offline index preprocessing pipeline and an online

routing/aggregation layer. In the offline stage, clustering and partition construction are performed on the input vectors; a dedicated HNSW sub-index is built for each partition; and the centroid metadata required to construct the Router Graph is persisted. In the online stage, a routing and aggregation module uses the stored Router Graph to dispatch batch queries to host/DPU target partitions and merges host-side and DPU-side results to produce the final top-k neighbors. Host–DPU control signaling and data exchange are implemented using NVIDIA DOCA SDK [24] 2.10.0: control messages are delivered via DOCA COMCH, while bulk data transfers are handled via DOCA DMA. A fixed-size ring-buffer-based communication path is further implemented so that runtime request/response exchanges reuse pre-registered buffers without repeating DMA setup overhead.

IV. Evaluation

A. Evaluation Setup

The experimental environment is configured as a disaggregated architecture consisting of a client server and a remote storage server. The two servers are connected via a 100 Gbps network based on NVIDIA ConnectX-7, and each server is equipped with an NVIDIA BlueField-3 DPU. On the client server, the DPU operates in *DPU mode* and is responsible for storing index partitions and executing VEX search operations, whereas the DPU on the remote storage server operates in *NIC mode* to provide the storage access path. The remote storage tier is configured using SPDK-based NVMe-oF. The detailed hardware and software specifications of each DPU and host system are summarized in Table IV and Table V, respectively.

Workloads. To capture a broad range of vector search characteristics, experiments are conducted using three datasets with different dimensionalities: SIFT, TriviaQA, and MS MARCO. SIFT is a widely used standard ANN benchmark, while TriviaQA and MS MARCO are text-based benchmarks that reflect realistic question-answering and search scenarios. For each dataset, the query sets provided by the benchmark are used directly.

To evaluate the effectiveness of VEX, the following configurations are compared.

TABLE III: Characteristics of Datasets Used.

Name	# Vector	# Dimension	Size	# Query
SIFT	50M	128	24GB	1000
TriviaQA	14M	384	22GB	1000
MS MARCO	7M	1024	27GB	1000

TABLE IV: Specifications of the DPU.

SoC	ARM Cortex-A78 16 Cores
Memory	32GB DDR5
OS	Ubuntu 22.04.05, Linux Kernel 5.15.0
Storage	KIOXIA BG4 128GB NVMe SSD

TABLE V: Specifications of the host system.

CPU	AMD Ryzen 9 3900X 12 cores
Memory	64GB DDR4
OS	Ubuntu 24.04.2, Linux Kernel 6.8.0
Storage	Samsung NVMe SSD 970 EVO Plus 500GB (Local) Samsung NVMe SSD 980 1TB (Remote)

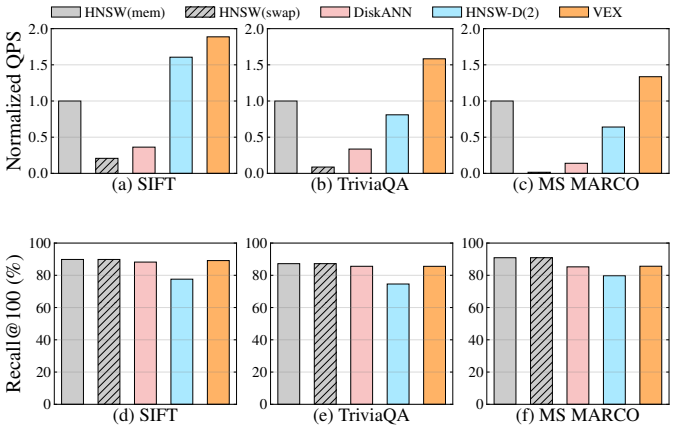

Fig. 5: Overall performance comparison of VEX and baselines in terms of normalized QPS and Recall@100 for three datasets.

- **HNSW**: *HNSW(mem)* keeps the entire index resident in host memory, whereas *HNSW(swap)* constrains the host DRAM budget based on the memory footprint of the baseline HNSW index, causing part of the index to be swapped out.
- **HNSW-D(k)**: This configuration ports disaggregated-memory ANN design ideas (e.g., [6], [8]) to a host–DPU environment. The dataset is partitioned into k partitions using K-means, and a separate HNSW sub-index is built for each partition. Queries are routed via the Router graph using random sampling across partitions, and host–DPU request/response exchanges are carried out using kernel-socket-based TCP communication. Unlike VEX, *HNSW-D(k)* does not perform cluster merging; instead, it places k partitions across the host and DPU according to the heterogeneous memory ratio.
- **DiskANN**: The index is built on remote NVMe storage accessed via NVMe-oF.
- **VEX**: The approach proposed in this paper constructs two HNSW indices via semantic-aware partitioning and employs Router Graph based query routing together with PCIe/DMA-based host–DPU request/response exchanges.

Search accuracy and system throughput are evaluated using Recall@100 and queries per second (QPS). Recall@100 is defined as the fraction of ground-truth nearest neighbors that appear in the top-100 results returned for each query, averaged over all queries. In all experiments, distances between vectors are computed using the L2 (Euclidean) metric. Following prior work [27], we conduct all experiments with a batch size of 16, which is commonly used in realistic ANN serving environments.

B. Performance under Memory-Constrained Settings

1) Overall Throughput–Recall@100 Comparison: Fig. 5 presents a comparison of normalized QPS and Recall@100 across three datasets under a memory-constrained setting, where the host DRAM budget is limited to 50% of the workload working set. This configuration reflects memory pressure

scenarios that commonly arise in large-scale vector search systems.

Overall, VEX achieves the highest QPS across all datasets without any significant degradation in Recall@100. This improvement stems from leveraging the DPU's onboard DRAM as an additional memory tier to absorb the working set, rather than relying solely on disk-based accesses to bypass HNSW's memory pressure. By offloading part of the index to DPU memory and executing vector search in parallel using the DPU's ARM cores, VEX both reduces disk I/O dependency and introduces additional compute parallelism. This allows VEX to achieve higher QPS than *HNSW(mem)*, improving throughput by approximately 1.9×, 1.6×, and 1.3× on SIFT, TriviaQA, and MS MARCO, respectively.

2) Comparison with Disk-Based ANN (DiskANN): Under the same memory constraints, VEX consistently outperforms the disk-based ANN indexing method *DiskANN* in terms of QPS. Although *DiskANN* is explicitly designed for SSD-based access, repeated storage accesses during graph traversal become a performance bottleneck that limits throughput. In contrast, VEX executes searches on HNSW partitions resident in DPU memory, significantly reducing disk accesses and achieving 5–10× higher QPS than *DiskANN* across all evaluated datasets.

3) Comparison with Distributed Memory Expansion (HNSW-D(k)): When compared with the distributed memory expansion approach *HNSW-D(2)*, the advantages of VEX become more pronounced in terms of communication mechanisms, routing policies, and boundary connectivity handling. *HNSW-D(2)* relies on kernel-socket-based TCP communication for data exchange with remote partitions, causing network stack overhead and accumulated latency as cross-partition interactions increase, thereby limiting QPS. In contrast, VEX employs PCIe/DMA-based host–DPU communication within the same node, minimizing data movement overhead that limits QPS. Moreover, *HNSW-D(2)* uses random-sampling-based routing, which may fail to direct queries to the most semantically relevant partition, leading to reduced accuracy. VEX, on the other hand, routes queries to semantically closest partitions using a Router Graph constructed from K-means centroids, concentrating the search space and achieving higher throughput with stable accuracy even under identical memory constraints.

In addition, unlike *HNSW-D(2)*, VEX reinforces connectivity near partition boundaries through boundary vector replication. Fig. 6 illustrates the impact of the boundary vector replication ratio on performance. As shown in the figure, increasing the replication ratio strengthens connectivity across partition boundaries, leading to improved Recall@100, while QPS remains largely unchanged. However, boundary vector replication introduces additional memory overhead due to index size expansion, increasing the total index size by 7.8%, 15.6%, and 23.7% as the boundary threshold is progressively lowered. Based on this trade-off analysis, VEX selects a boundary threshold of 0.99, which provides the highest recall improvement per unit of memory overhead among the evaluated thresholds. With these design choices, VEX achieves up to approximately 2.1× higher QPS than *HNSW-D(2)* while maintaining stable accuracy.

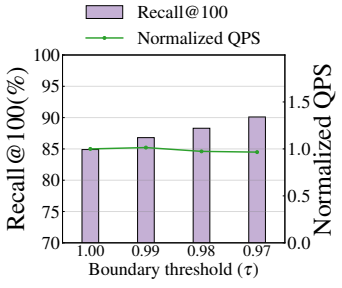

Fig. 6: Impact of the boundary threshold on search performance.

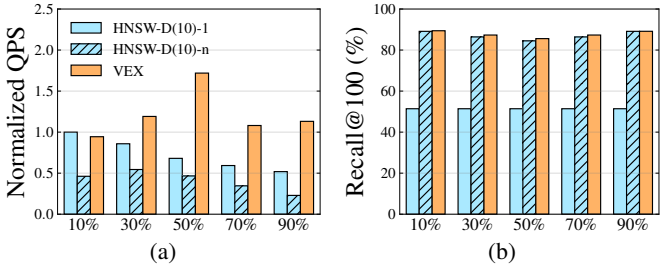

Fig. 7: Search performance as the index offloading ratio to the DPU varies. The figure reports normalized QPS and Recall@100 of VEX and comparison configurations based on disaggregated-memory approaches as the offloading ratio ranges from 10% to 90%.

4) Effect of Vector Dimensionality: Meanwhile, as the dimensionality of the dataset increases, the degree of QPS improvement achieved by VEX tends to diminish. This behavior arises because distance computation becomes increasingly dominant for higher-dimensional datasets. In such cases, the host CPU performs distance calculations using AVX-based SIMD instructions, whereas the DPU's ARM cores rely on NEON-based SIMD, resulting in a larger efficiency gap for identical operations. Consequently, for workloads with a high proportion of high-dimensional distance computations, the performance disparity between the host and DPU widens, partially attenuating the QPS gains provided by VEX.

C. Sensitivity to Offloading Ratio

Fig. 7 compares the changes in QPS and Recall@100 of VEX and *HNSW-D(10)* as the index offloading ratio to the DPU is varied.

First, *HNSW-D(10)-1*, which explores only a single partition per query, exhibits a severe drop in Recall@100 to approximately 51%. This degradation occurs because *HNSW-D(10)* lacks a merging process that preserves semantic continuity across partitions, and searching only a single partition is insufficient to cover a meaningful neighborhood in the vector space. Alternatively, accuracy can be recovered by increasing the number of searched partitions per query. In our experiments, we denote this configuration as *HNSW-D(10)-n*, where n represents the number of partitions explored among the ten partitions constructed during index building, and is selected such that the resulting Recall@100 is comparable to that of VEX. However, this improvement comes at the cost of reduced QPS due to query amplification caused by the expanded per-query search scope. Moreover, since partitions are not semantically aligned and are scattered across the host and DPU, exploring multiple partitions per query incurs

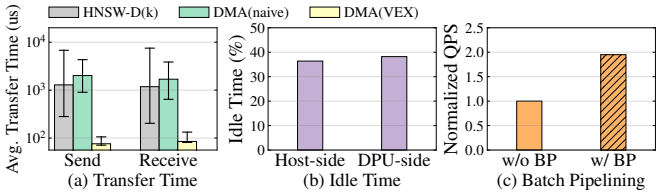

Fig. 8: Impact of host–DPU communication optimization and batch pipelining on system performance. The figure reports (a) average host–DPU transfer latency, (b) host-side and DPU-side idle time, and (c) normalized QPS.

accumulated host–DPU cross-dependencies and data transfers, further exacerbating the QPS degradation.

In contrast, VEX leverages PCA-guided ordering and merging following K-means clustering, enabling a single-partition search to sufficiently cover the relevant neighborhood for each query. As a result, VEX maintains high Recall@100 and stable QPS without query amplification or host–DPU cross-dependencies, even as the offloading ratio varies.

D. Host–DPU Communication Overhead Analysis

Fig. 8(a) compares the host–DPU data transfer latency across three communication mechanisms: socket-based TCP used in *HNSW-D(k)*, *DMA(naive)* with repeated memory registration and mapping overhead for every transfer, and *DMA(*VEX*)*, the optimized DMA design used in VEX. *HNSW-D(k)* exhibits high transfer latency and large variance, making it unfavorable in terms of both latency and stability, while *DMA(naive)* also suffers from significantly increased transfer latency due to repeated preparation overhead. In contrast, VEX reuses pre-registered ring buffers at runtime, thereby substantially reducing both transfer latency and variance. As a result, *HNSW-D(k)* and *DMA(naive)* observe significantly longer transfer times of approximately 1.2–2.0 ms, whereas *DMA(*VEX*)* maintains a stable transfer latency of approximately 0.08 ms.

E. Impact of Batch Pipelining on Resource Util. & Throughput

Fig. 8(b),(c) illustrate the impact of batch pipelining in VEX on idle time and system throughput in a heterogeneous environment. The time breakdowns for the host and the DPU in Fig. 8(b) show that, without pipelining, approximately 37.3% of the total execution time remains idle due to workload imbalance and asymmetric compute capabilities across heterogeneous devices, significantly reducing device utilization. Batch pipelining mitigates this inefficiency by overlapping stages across batches (search, communication, and merge/routing), thereby reducing waiting periods and enabling continuous execution on both the host and the DPU. Consequently, as shown in Fig. 8(c), overall QPS increases by 95% compared to the non-pipelined baseline.

F. Large-scale Datasets Analysis

Fig. 9 demonstrates how VEX sustains and scales performance by integrating with *DiskANN* when the index size exceeds the combined capacity of host DRAM and DPU onboard memory. In this experiment, only a portion of the index

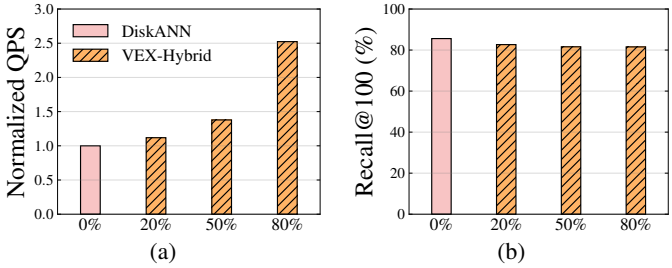

Fig. 9: System performance as the in-memory index fraction in the VEX-Hybrid configuration varies. (a) Normalized QPS and (b) Recall@100 for different fractions of the index resident in host and DPU memory.

corresponding to the total available host and DPU memory resides in memory, while the remaining data are placed in the storage tier and indexed using *DiskANN*, enabling cooperative ANN search across tiers.

In Fig. 9(a), VEX-Hybrid allows in-memory partitions on the host and DPU to absorb a substantial fraction of the working set, thereby progressively reducing remote storage access frequency. As a result, as the fraction of the index resident in the memory tiers increases from 20% to 50% and 80%, the QPS of VEX-Hybrid improves to approximately 1.1×, 1.4×, and 2.5× that of *DiskANN*, respectively. In terms of accuracy, VEX-Hybrid maintains stable performance. As shown in Fig. 9(b), appropriate query routing to in-memory partitions enables Recall@100 to remain within a 2–3% margin of *DiskANN* across all configurations.

V. Related Work

To overcome the DRAM capacity limits of a single server, prior work explores disaggregated and extended-memory approaches for ANN search [6], [8], [28]. Pyramid [8] distributes similarity search across memory nodes, while d-HNSW [6] reorganizes HNSW data placement and execution to reduce remote memory access. More recent CXL-based studies extend the memory hierarchy within a server using caching and prefetching [29]–[31], but the limited availability of commercial CXL devices restricts their practicality. In parallel, disk-based approaches incorporate SSDs into the search path to reduce memory footprints [5], [20], [21]. DiskANN [5] enables large-scale search on a single node using SSD-resident graphs, while Starling [20] and SPANN [21] improve I/O efficiency and memory–accuracy trade-offs. However, existing work focuses on memory expansion and storage offloading, leaving SmartNICs (DPUs) largely unexplored as a means to simultaneously extend memory capacity and enable parallel ANN search.

VI. Conclusion

This paper shows that DPUs can act as an effective memory and compute tier for scaling HNSW-based vector search. Scalable ANN search under memory pressure requires holistic co-design across partitioning, communication, and execution to preserve vector locality, minimize host–DPU dependencies, and mask imbalance caused by heterogeneous compute capabilities

and routing variability. VEX achieves high throughput and stable Recall in both memory-constrained settings for real workloads, and naturally extends to multi-tier architectures that incorporate disk-based ANN indices.

ACKNOWLEDGEMENT

This work was partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2024-00453929, RS-2024-00416666) and by the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-00136).

REFERENCES

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Proceedings of the Conference on Neural Information Processing Systems*, NeurIPS'20.

[2] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC'98.

[3] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[4] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2020.

[5] S. J. Subramanya, R. Krishnaswamy, H. V. Simhadri, S. Tirthapura, Y. Chen, and B. Krishnamurthy, "DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node," in *Proceedings of the Conference on Neural Information Processing Systems*, NeurIPS'19.

[6] T. Wang, Z. Zhu, L. Chen, and J. X. Yu, "Efficient Vector Search on Disaggregated Memory with d-HNSW," in *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage'25.

[7] Y. Jeong, H. Cho, K. Park, Y. Kim, and S. Park, "CALL: Context-Aware Low-Latency Retrieval in Disk-Based Vector Databases," in *Proceedings of the IEEE International Conference on High Performance Computing, Data, and Analytics*, HiPC'25.

[8] Z. Zhu, T. Wang, L. Chen, and J. X. Yu, "Pyramid: A General Framework for Distributed Similarity Search on Large-Scale Datasets," in *Proceedings of the IEEE International Conference on Big Data*, Big Data'19.

[9] J. Shim, J. Oh, H. Roh, J. Do, and S.-W. Lee, "Turbocharging Vector Databases using Modern SSDs," in *Proceedings of the VLDB Endowment*, VLDB'25.

[10] Zilliz, "Milvus: The High-Performance Vector Database Built for Scale." https://milvus.io/docs/v2.4.x/architecture_overview.md, 2020. Accessed: Dec. 10, 2025.

[11] NVIDIA, "NVIDIA BlueField-2 DPU: Programmable data center infrastructure on-a-chip." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2020. Accessed: Dec. 10, 2025.

[12] NVIDIA, "NVIDIA BlueField-3 DPU: Programmable data center infrastructure on-a-chip." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2022. Accessed: Dec. 10, 2025.

[13] NVIDIA, "NVIDIA BlueField-4 DPU: Programmable data center infrastructure on-a-chip." https://resources.nvidia.com/en-us-accelerated-networking-resource-library/bluefield-4-dpu-datasheet, 2025. Accessed: Dec. 10, 2025.

[14] NVIDIA, "NVIDIA ConnectX-7 Adapters: Accelerated networking for modern data center infrastructures." https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet, 2024. Accessed: Dec. 10, 2025.

[15] C. Ding, J. Zhou, K. Lu, S. Li, Y. Xiong, J. Wan, and L. Zhan, "D2Comp: Efficient Offload of LSM-tree Compaction with Data Processing Units on Disaggregated Storage," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 3, pp. 1–22, 2024.

[16] C. Ding, K. Lu, Q. Zhang, Z. Ye, T. Yao, D. Wang, H. Wu, and J. Wan, "DFlush: DPU-Offloaded Flush for Disaggregated LSM-based Key-Value Stores," in *Proceedings of the ACM on Management of Data*, SIGMOD'25.

[17] Q. Zhang, P. A. Bernstein, D. S. Berger, B. Chandramouli, J. Hu, and Y. Zheng, "DDS: DPU-Optimized Disaggregated Storage," in *Proceedings of the VLDB Endowment*, VLDB'24.

[18] W. Zhu, Z. Shen, Q. Wei, R. Chen, X. Yao, D. Yu, and Z. Shao, "HiDPU: A DPU-Oriented hybrid indexing scheme for disaggregated storage systems," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'25.

[19] A. Kashyap, Y. Li, and X. Lu, "DPU-KV: On the Benefits of DPU Offloading for In-Memory Key-Value Stores at the Edge," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'25.

[20] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment," in *Proceedings of the ACM on Management of Data*, SIGMOD'24.

[21] Q. Chen, J. Huang, Z. Zhu, J. X. Yu, and L. Chen, "SPANN: Highly-Efficient Billion-Scale Approximate Nearest Neighbor Search," in *Proceedings of the Conference on Neural Information Processing Systems*, NeurIPS'21.

[22] B. Tian, H. Liu, Z. Duan, X. Liao, H. Jin, and Y. Zhang, "Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs," in *Proceedings of the USENIX Annual Technical Conference*, ATC'24.

[23] A. Kashyap, Y. Li, D. Ng, and X. Lu, "Understanding the Idiosyncrasies of Emerging BlueField DPUs," in *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS'25.

[24] NVIDIA, "DOCA SDK." https://developer.nvidia.com/networking/doca, 2024. Accessed: Dec. 10, 2025.

[25] NVIDIA, "DOCA Communication Channel." https://docs.nvidia.com/doca/sdk/doca+comch/index.html, 2024. Accessed: Dec. 10, 2025.

[26] Meta, "A library for efficient similarity search and clustering of dense vectors." https://github.com/facebookresearch/faiss, 2025. Accessed: Dec. 10, 2025.

[27] D. Quinn, E. E. Yücel, M. Prammer, Z. Fan, K. Skadron, J. M. Patel, J. F. Martínez, and M. Alian, "DReX: Accurate and Scalable Dense Retrieval Acceleration via Algorithmic-Hardware Codesign," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA'25.

[28] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo, "Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs," in *Proceedings of the VLDB Endowment*, VLDB'20.

[29] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search," in *Proceedings of the USENIX Annual Technical Conference*, ATC'23.

[30] S. Ko, H. Shim, W. Doh, S. Yun, J. So, Y. Kwon, S.-S. Park, S.-D. Roh, M. Yoon, T. Song, and J. H. Ahn, "Cosmos: A CXL-Based Full In-Memory System for Approximate Nearest Neighbor Search," *IEEE Computer Architecture Letters*, vol. 24, no. 1, pp. 173–176, 2025.

[31] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, *et al.*, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'23.