CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search

Junhyeok Jang, *Computer Architecture and Memory Systems Laboratory, KAIST;* Hanjin Choi, *Computer Architecture and Memory Systems Laboratory, KAIST and Panmnesia, Inc.;* Hanyeoreum Bae and Seungjun Lee, *Computer Architecture and Memory Systems Laboratory, KAIST;* Miryeong Kwon and Myoungsoo Jung, *Computer Architecture and Memory Systems Laboratory, KAIST and Panmnesia, Inc.*

https://www.usenix.org/conference/atc23/presentation/jang

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search

Junhyeok Jang*, Hanjin Choi*†, Hanyeoreum Bae*, Seungjun Lee*, Miryeong Kwon*†, Myoungsoo Jung*†
*Computer Architecture and Memory Systems Laboratory, KAIST
†Panmnesia, Inc.

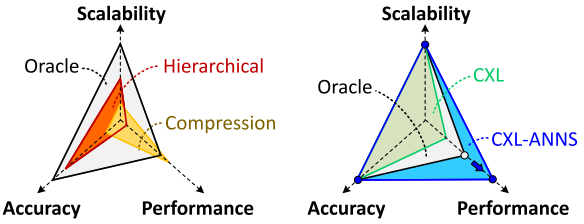
Abstract

We propose *CXL-ANNS*, a software-hardware collaborative approach to enable highly scalable approximate nearest neighbor search (ANNS) services. To this end, we first disaggregate DRAM from the host via compute express link (CXL) and place all essential datasets into its memory pool. While this CXL memory pool can make ANNS feasible to handle billion-point graphs without an accuracy loss, we observe that the search performance significantly degrades because of CXL's far-memory-like characteristics. To address this, CXL-ANNS considers the node-level relationship and caches the neighbors in local memory, which are expected to visit most frequently. For the uncached nodes, CXL-ANNS prefetches a set of nodes most likely to visit soon by understanding the graph traversing behaviors of ANNS. CXL-ANNS is also aware of the architectural structures of the CXL interconnect network and lets different hardware components therein collaboratively search for nearest neighbors in parallel. To improve the performance further, it relaxes the execution dependency of neighbor search tasks and maximizes the degree of search parallelism by fully utilizing all hardware in the CXL network.

Our empirical evaluation results show that CXL-ANNS exhibits 111.1× higher QPS with 93.3% lower query latency than state-of-the-art ANNS platforms that we tested. CXL-ANNS also outperforms an oracle ANNS system that has DRAM-only (with unlimited storage capacity) by 68.0% and 3.8×, in terms of latency and throughput, respectively.

1 Introduction

Dense retrieval (also known as nearest neighbor search) has taken on an important role and provides fundamental support for various search engines, data mining, databases, and machine learning applications such as recommendation systems [1–8]. In contrast to the classic pattern/string-based search, dense retrieval compares the similarity across different objects using their distance and retrieves a given number of objects, similar to the query object, referred to as *k*-nearest neighbor (*kNN*) [9–11]. To this end, dense retrieval embeds input information into a few thousand dimensional spaces of each object, called a feature *vector*. Since these vectors can encode a wide spectrum of data formats (e.g., images, documents, sounds, etc.), dense retrieval understands an input query's semantics, resulting in more context-aware and accurate results than traditional search [6, 12, 13].



(a) Previous studies. (b) CXL-based approaches.

Figure 1: Various billion-scale ANNS characterizations.

Even though kNN is one of the most frequently used search paradigms in various applications, it is a costly operation taking linear time to scan data [14, 15]. This computation complexity unfortunately makes dense retrieval with a billion-point dataset infeasible. To make the kNN search more practical, *approximate nearest neighbor search* (ANNS) restricts a query vector to search only a subset of neighbors with a high chance of being the nearest ones [15–17]. ANNS exhibits good vector searching speed and accuracy, but it significantly increases memory requirement and pressure. For example, many production-level recommendation systems already adopt billion-point datasets, which require tens of TB of working memory space for ANNS; Microsoft search engines (used in Bing/Outlook) require 100B+ vectors, each being explained by 100 dimensions, which consume more than 40TB memory space [18]. Similarly, several of Alibaba's e-commerce platforms need TB-scale memory spaces to accommodate their 2B+ vectors (128 dimensions) [19].

To address these memory pressure issues, modern ANNS techniques leverage lossy compression methods or employ persistent storage, such as solid state disks (SSDs) and persistent memory (PMEM), for their memory expansion. For example, [20–23] split large datasets and group them into multiple clusters in an offline time. This compression approach only has product quantized vectors for each cluster's centroid and searches kNN based on the quantized information, making billion-scale ANNS feasible. On the other hand, the hierarchical approach [24–28] accommodates the datasets to SSD/PMEM, but reduces target search spaces by referring to a summary in its local memory (DRAM). As shown in Figure 1a, these compression and hierarchical approaches can achieve the best kNN search performance and scalabil-

ity similar to or slightly worse than what an *oracle*[1] system offers. However, these approaches suffer from a lack of accuracy and/or performance, which unfortunately hinders their practicality in achieving billion-scale ANNS services.

In this work, we propose *CXL-ANNS*, a software-hardware collaborative approach that enables scalable approximate nearest neighbor search (ANNS). As shown in Figure 1b, the main goal of CXL-ANNS is to offer the latency of billion-point kNN search even shorter than the oracle system mentioned above while achieving high throughput without a loss of accuracy. To this end, we disaggregate DRAM from the host resources via compute express link (CXL) and place all essential datasets into its memory pool; CXL is an open-industry interconnect technology that allows the underlying working memory to be highly scalable and composable with a low cost. Since a CXL network can expand its memory capacity by having more *endpoint devices* (EPs) in a scalable manner, a host's *root-complex* (RC) can map the network's large memory pool (up to 4PB) into its system memory space and use it just like a locally-attached conventional DRAM.

While this CXL memory pool can make ANNS feasible to handle billion-point graphs without a loss of accuracy, we observe that the search performance degrades compared to the oracle by as high as 3.9× (§3.1). This is due to CXL's far-memory-like characteristics; every memory request needs a CXL protocol conversion (from CPU instructions to one or more CXL flits), which takes a time similar to or longer than a DRAM access itself. To address this, we consider the relationship of different nodes in a given graph and cache the neighbors in the local memory, which are expected to visit frequently. For the uncached nodes, CXL-ANNS prefetches a set of nodes most likely to be touched soon by understanding the unique behaviors of the ANNS graph traversing algorithm. CXL-ANNS is also aware of the architectural structures of the CXL interconnect network and allows different hardware components therein to simultaneously search for nearest neighbors in a collaborative manner. To improve the performance further, we relax the execution dependency in the KNN search and maximize the degree of search parallelism by fully utilizing all our hardware in the CXL network.

We summarize the main contribution as follows:

• *Relationship-aware graph caching.* Since ANNS traverses a given graph from its entry-node [10, 19], we observe that the graph data accesses, associated with the innermost edge hops, account for most of the point accesses (§3.2). Inspired by this, we selectively locate the graph and feature vectors in different places of the CXL memory network. Specifically, CXL-ANNS allocates the node information closer to the entry node in the locally-attached DRAMs while placing the other datasets in the CXL memory pool.

• *Hiding the latency of CXL memory pool.* If it needs to traverse (uncached) outer nodes, CXL-ANNS prefetches the datasets of neighbors, most likely to be processed in the next step of kNN queries from the CXL memory pool. However, it is non-trivial to figure out which node will be the next to visit because of ANNS's procedural data processing dependency. We propose a simple foreseeing technique that exploits a unique graph traversing characteristic of ANNS and prefetches the next neighbor's dataset during the current kNN candidate update phase.

• *Collaborative kNN search design in CXL.* CXL-ANNS significantly reduces the time wasted for transferring the feature vectors back and forth by designing EP controllers to calculate distances. On the other hand, it utilizes the computation power of the CXL host for non-beneficial operations in processing data near memory (e.g., graph traverse and candidate update). This collaborative search includes an efficient design of RC-EP interfaces and a sharding method being aware of the hardware configurations of the CXL memory pool.

• *Dependency relaxation and scheduling.* The computation sequences of ANNS are all connected in a serial order, which makes them unfortunately dependent on execution. We examine all the activities of kNN query requests and classify them into urgent/deferrable subtasks. CXL-ANNS then relaxes the dependency of ANN computation sequences and schedules their subtasks in a finer granular manner.

We validate all the functionalities of CXL-ANNS's software and hardware (including the CXL memory pool) by prototyping them using Linux 5.15.36 and 16nm FPGA, respectively. To explore the full design spaces of ANNS, we also implement the hardware-validated CXL-ANNS in gem5 [29] and perform full-system simulations using six billion-point datasets [30]. Our evaluation results show that CXL-ANNS exhibits 111.1× higher bandwidth (QPS) with 93.3% lower query latency, compared to the state-of-the-art billion-scale ANNS methods [20, 24, 25]. The latency and throughput behaviors of CXL-ANNS are even better than those of the oracle system (DRAM-only) by 68.0% and 3.8×, respectively.

2 Background

2.1 Approximate Nearest Neighbor Search

The most accurate method to get k-nearest neighbors (kNN) in a graph is to compare an input query vector with all data vectors in a brute-force manner [9, 31]. Obviously, this simple dense retrieval technique is impractical mainly due to its time complexity [10, 24]. In contrast, approximate nearest neighbor search (ANNS) restricts the query vector to retrieve only a subset of neighbors that can be kNN with a high probability. To meet diverse accuracy and performance requirements, several ANNS algorithms such as tree-structure based [32, 33], hashing based [11, 34, 35] and quantization based approaches [20–23] have been proposed over the past decades. Among the various techniques, ANNS algorithms using graphs [10, 19, 24] are considered as the most promising

[1] In this paper, the term "Oracle" refers to a system that utilizes ample DRAM resources with an unrestricted memory capacity.

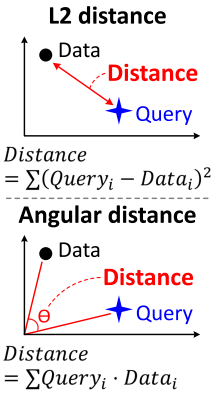


Figure 2: Distance.

Algorithm 1: Best-first Search

Input: *query*,
k := number of nearest neighbor to find
Output: *k* nearest neighbor of *query*

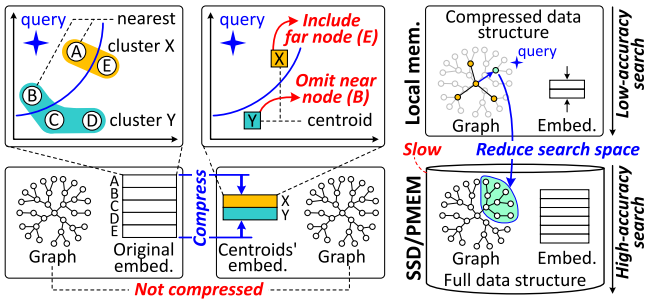
```
1  distance = calcDist(query, startNode)
2  CandidateArr = {startNode, distance}
3  curNode = startNode
4  CandidateArr.markVisited(curNode)
   /* Main iteration (line 5~10) */
5  while !candidateArr.allVisited() do
6      neighbors = Graph.neighbors(curNode)
7      distances = calcDist(query, neighbors)
8      CandidateArr.insert(neighbors, distances)
9      curNode = CandidateArr.getNextNode()
10     CandidateArr.markVisited(curNode)
11 return CandidateArr[: k]
```

solution, with great potential [2]. This is because graph-based approaches can better describe neighbor relationships and traverse fewer points than the other approaches that operate in a Euclidean space [19, 36–39].

Distance calculations. While there are various graph construction algorithms for ANNS [10, 19, 24], the goal of their query search algorithms is all the same or similar to each other; it is simply to find *k* numbers of neighbors in the target graph, which are expected to have the shortest distance from a given feature vector, called *query vector*. There are two most common methods to define such a distance between the query vector and neighbor's feature vector (called *data vector*): i) L2 (Euclidean) distance and ii) angular distance. As shown in Figure 2, these methods map the nodes that we compare into a temporal dimension space using their own vector's feature elements. Let us suppose that there are n numbers of features for each vector. Then, L2 and angular distances are calculated by $\sum_i (Query_i - Data_i)^2$ and $\sum_i (Query_i \cdot Data_i)$, respectively; where $Query_i$ and $Data_i$ are the i^{th} feature of a given query and data vectors, respectively ($i \leq n$). These distance definitions are simplified to reduce their calculation latency, which differs from the actual distances in a multi-dimensional vector space. This simplification works well since ANNS uses the distances only for a relative comparison to search kNN.

Approximate kNN query search. Algorithm 1 explains the graph traversing method that most ANNS employs [10, 19, 24]. The method, best-first search (BFS) [38, 40], traverses from an *entry-node* (line ❸) and moves to neighbors getting closer to the given query vector (lines ❺~❿). While the brute-force search explores a full space of the graph by systematically enumerating all the nodes, ANNS uses a preprocessed graph and visits a limited number of nodes for each hop. The graph is constructed (preprocessed) to have the entry-node that arrives all the nodes of its original graph within the minimum number of average edge hops; this preprocessed graph guarantees that there exists a path between the entry-node and any of the given nodes. To minimize the overhead of graph traverse, BFS employs a *candidate array* that includes the neighbors whose

[2] For the sake of the brevity, we use "graph-based approximate kNN methods" and "ANNS" interchangeably.



(a) Compression-based approach. (b) Hierarchical approach.

Figure 3: Existing billion-scale ANNS methods.

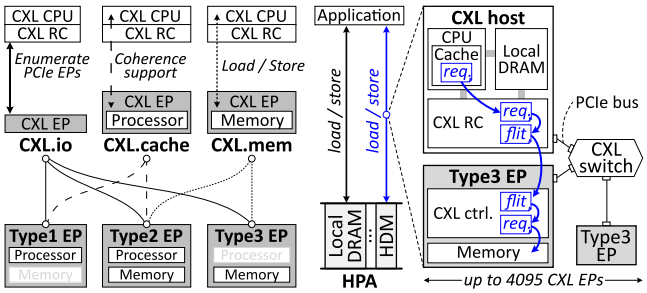
distances (from the query vector) are expected to be shorter than others. For each node visiting, BFS checks this candidate array and retrieves unvisited node from the array (line ❻, ❾). It then calculates the distances of the node's neighbors (line ❼) by retrieving their vectors from the embedding table. After this distance calculation, BFS updates the candidate array with the new information, neighbors and distances (line ❽). All these activities are iterated (line ❺) until there is no unvisited node in the candidate array. BFS finally returns the *k* number of neighbors in the candidate array.

2.2 Towards Billion-scale ANNS

While ANNS can achieve good search speed and reasonable accuracy (as it only visits the nodes in the candidate array), it still requires maintaining all the original graph and vectors in its embedding table. This renders ANNS difficult to have billion-point graphs that exhibit high memory demands in many production-level services [18, 19]. To address this issue, there have been many studies proposed [20–27], but we can classify them into two as shown in Figures 3a and 3b.

Compression approaches. These approaches [20–23] reduce the embedding table by compressing its vectors. As shown in Figure 3a, they logically split the given graph into multiple sub-groups, called *clusters*; the nodes A and E are classified in the cluster X whereas the others are grouped as the cluster Y. For each cluster, these approaches then encode the corresponding vectors into a single, representative vector (called centroid) by averaging all the vectors in the cluster. They then replace all the vectors in the embedding table with their cluster ID. Since the distances are calculated by the compressed centroid vectors (rather than original data vectors), it exhibits a low accuracy for the search. For example, the node E can be selected as one of kNN although the node B sits closer to the query vector. Another issue of these compression approaches is the limited reduction rate in the size of the graph datasets. Since they quantize only the embedding table, their billion-point graph data have no benefit of the compression or even get slightly bigger to add a set of shortcuts into the original graph.

Hierarchical approaches. These approaches [24–27] store all the graph and vectors (embedding table) to the underlying SSD/PMEM (Figure 3b). Since SSD/PMEM are prac-



(a) Types of CXL EP. (b) CXL-based memory pool.

Figure 4: CXL's sub-protocols and endpoint types.

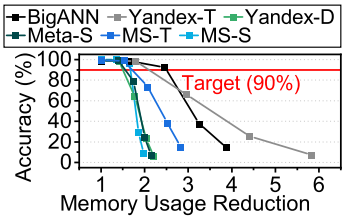


Figure 5: Accuracy.

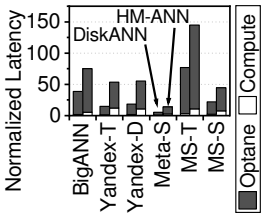


Figure 6: Latency.

tically slower than DRAM by many orders of magnitude, these methods process kNN queries in two separate phases: i) *low-accuracy search* and ii) *high-accuracy search*. The former only refers to compressed or simplified datasets, similar to the datasets that the compression approaches use. The low-accuracy search quickly finds out one or more nearest neighbor candidates (without a storage access) thereby reducing the search space that the latter needs to process. Once it has been completed, the high-accuracy search refers to the original datasets associated with the candidates and processes the actual kNN queries. For example, DiskANN [24] 's low accuracy search finds the kNN candidates using the compressed datasets in DRAM. The high-accuracy search then re-examines and re-ranks the order of kNN candidates by visiting their actual vectors stored in SSD/PMEM. On the other hand, HM-ANN [25] simplifies the target graph by adding several shortcuts (across multiple edge hops) into the graph. HM-ANN's low-accuracy search scans a candidate closer to the given query vector from the simplified graph. Once HM-ANN detects the candidate, the high-accuracy search checks its kNN by referring to all the graph and data vectors recorded in SSD/PMEM.

2.3 Compute Express Link for Memory Pool

CXL is an open standard interconnect which can expand memory over the existing PCIe physical layers in a scalable option [41–43]. As shown in Figure 4a, CXL consists of three sub-protocols: i) *CXL.io*, ii) *CXL.cache*, and iii) *CXL.mem*. Based on which sub-protocols are used for the main communication, CXL EPs can be classified as Types.

Sub-protocols and endpoint types. CXL.io is basically the same as the PCIe standard, which is aimed at enumerating the underlying EPs and performing transaction controls. It is thus used for all the CXL types of EPs to be interconnected to the CXL CPU's root-complex (*RC*) through PCIe. On the other hand, CXL.cache is for an underlying EP to make its states coherent with those of a CXL host CPU, whereas CXL.mem supports simple memory operations (load/store) over PCIe. Type 1 is considered by a co-processor or accelerator that does not have memory exposed to CXL RC while Type 2 employs internal memory, accessible from CXL RC. Thus, Type 1 only uses CXL.cache (in addition to CXL.io), but Type 2 needs to use both CXL.cache and CXL.mem. A potential example of Type 1 and 2 can be FPGAs and GPUs, respectively. On the other hand, Type 3 only uses CXL.mem (read/write), which means that there is no interface for a device-side compute unit to update its calculation results to CXL CPU's RC and/or get a non-memory request from the RC.

CXL endpoint disaggregation. Figure 4b shows how we can disaggregate DRAM from host resources using CXL EPs, in particular, Type 3; we will discuss why Type 3 is the best device type for the design of CXL-ANNS, shortly. Type 3's internal memory is exposed as a *host-managed device memory* (HDM), which can be mapped to the CXL CPU's host physical address (HPA) in the system memory just like DRAM. Therefore, applications running on the CXL CPU can access HDM (EP's internal memory) through conventional memory instructions (loads/stores). Thanks to this characteristic, HDM requests are treated as traditional memory requests in CXL CPU's memory hierarchy; the requests are first cached in CPU cache(s). Once its cache controller evicts a line associated with the address space of HDM, the request goes through to the system's CXL RC. RC then converts one or more memory requests into a CXL packet (called *flit*) that can deal with a request or response of CXL.mem/CXL.cache. RC passes the flit to the target EP using CXL.mem's read or write interfaces. The destination EP's PCIe and CXL controllers take the flit over, convert it to one or more memory requests, and serve the request with the EP's internal memory (HDM).

Type consideration for scaling-out. To expand the memory capacity, the target CXL network can have one or more switches that have multiple ports, each being able to connect a CXL EP. This switch-based network configuration allows an RC to employ many EPs (upto 4K), but only for Type 3. This is because CXL.cache uses virtual addresses for its cache coherence management unlike CXL.mem. As the virtual addresses (brought by CXL flits) are not directly matched with the physical address of each underlying EP's HDM, the CXL switches cannot understand where the exact destination is.

3 A High-level Viewpoint of CXL-ANNS

3.1 Challenge Analysis of Billion-scale ANNS

Memory expansion with compression. While compression methods allow us to have larger datasets, it is not scalable since their quantized data significantly degrades the kNN search accuracy. Figure 5 analyzes the search accuracy of billion-point ANNS that uses the quantization-based com-

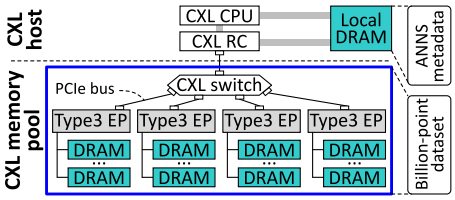


Figure 7: CXL baseline architecture.

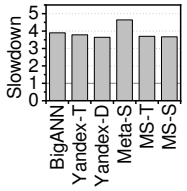


Figure 8: CXL.

pression described in §2.2. In this analysis, it reduces the embedding table by 2×∼16×. We use six billion-point datasets from [30]; the details of these datasets and evaluation environment are the same as what we used in §6.1. As the density of the quantized data vectors varies across different datasets, the compression method exhibits different search accuracies. While the search accuracies are in a reasonable range to service with low compression rates, they significantly drop as the compression rate of the dataset increases. It cannot even reach the threshold accuracy that ANNS needs to support (90%, recommended by [30]) after having 45.8% less data than the original. This unfortunately makes the compression impractical for billion-scale ANNS at high accuracy.

Hierarchical data processing. Hierarchical approaches can overcome this low accuracy issue by adding one more search step to re-rank the results of kNN search. This high-accuracy search however increases the search latency significantly as it eventually requires traversing the storage-side graph and accessing the corresponding data vectors (in storage) entirely. Figure 6 shows the latency behaviors of hierarchical approaches, DiskANN [24] and HM-ANN [25]. In this test, we use 480GB Optane PMEM [44] for DiskANN/HM-ANN and compare their performance with the performance of an oracle ANNS that has DRAM-only (with unlimited storage capacity). One can observe from this figure that the storage accesses of the high-accuracy search account for 87.6% of the total kNN query latency, which makes the search latency of DiskANN and HM-ANN worse than that of the oracle ANNS by 29.4× and 64.6×, respectively, on average.

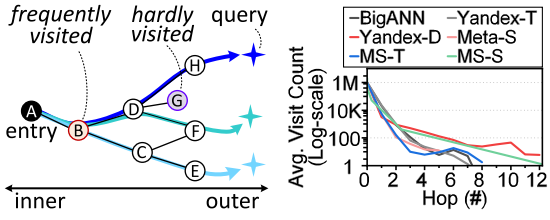
CXL-augmented ANNS. To avoid the accuracy drop and performance depletion, this work advocates to directly have billion-point datasets in a scalable memory pool, disaggregated using CXL. Figure 7 shows our baseline architecture that consists of a CXL CPU, a CXL switch, and four 1TB Type 3 EPs that we prototype (§6.1). We locate all the billion-point graphs and corresponding vectors to the underlying Type 3 EPs (memory pool) while having ANNS metadata (e.g. candidate array) in the local DRAM. This baseline allows ANNS to access the billion-point datasets on the remote-side memory pool just like conventional DRAMs thanks to CXL's instruction-level compatibility. Nevertheless, it is not yet an appropriate option for practical billion-scale ANNS due to CXL's architectural characteristics that exhibit lower performance than the local DRAM.

To be precise, we compare the kNN search latency of the baseline with the oracle ANNS, and the results are shown in Figure 8. In this analysis, we normalize the latency of the baseline to that of the oracle for better understanding. Even though our baseline does not show severe performance depletion like what DiskANN/HM-ANNS suffer from, it exhibits 3.9× slower search latency than the oracle, on average. This is because all the memory accesses associated with HDM(s) insist the host RC convert them to a CXL flit and revert the flit to memory requests at the EP-side. The corresponding responses also requires this *memory-to-flit* conversion in a reverse order thereby exhibiting the long latency for graph/vector accesses. Note that this 3.6∼4.6× performance degradation is not acceptable in many production-level ANNS applications such as recommendation systems [45] or search engines [46].

3.2 Design Consideration and Motivation

The main goal of this work is to make the CXL-augmented kNN search faster than in-memory ANNS services working only with locally-attached DRAMs (cf. CXL-ANNS vs. Oracle as shown in Figure 8). To achieve this goal, we propose CXL-ANNS, a software-hardware collaborative approach, which considers the following three observations: i) node-level relationship, ii) distance calculation, and iii) vector reduction.

Node-level relationship. While there are diverse graph structures [10, 19, 24] for the best-first search traverses (cf. Algorithm 1), all of the graphs starts their traverses from a unique, single entry-node as described in §2.1. This implies that the graph traverse of ANNS visits the nodes closer to the entry-node much more frequently. For example, as shown in Figure 9a, the node B is always accessed to serve a given set of kNN queries targeting other nodes listed in the graph branch while the node G is difficult to visit. To be precise, we examine the average count to visit nodes in all the billion-point graphs that this work evaluate when there are a million kNN query



(a) Traverse. (b) Nodes' visit count.

Figure 9: Graph traverse.

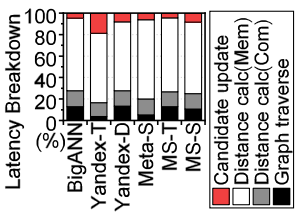
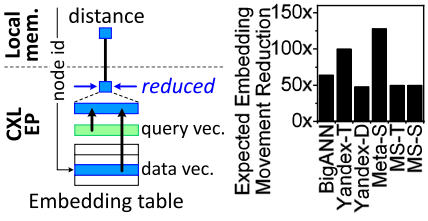


Figure 10: End-to-end breakdown analysis.



(a) Reduction example. (b) Reduction ratio.

Figure 11: Data reduction.

requests. The results are shown in Figure 9b. One can observe from this analysis that the nodes most frequently accessed during the 1M kNN searches reside in the 2∼3 edge hops. By appreciating this node-level relationship, we will locate the graph and vector data regarding inner-most nodes (from the entry-node) to locally-attached DRAMs while allocating all the others to the underlying CXL EPs.

Distance calculation. To analyze the critical path of billion-point ANNS, we decompose the end-to-end kNN search task into four different sub-tasks, i) candidate update, ii) memory access and iii) computing fractions of distance calculation, and iv) graph traverse. We then measure the latency of each sub-tasks on use in-memory, oracle system, which are shown in Figure 10. As can be seen from the figure, ANNS distance calculation significantly contributes to the total execution time, constituting an average of 81.8%. This observation stands in contrast to the widely held belief that graph traversal is among the most resource-intensive operations [47–49]. The underlying reason for this discrepancy is that distance calculation necessitates intensive embedding table lookups to determine the data vectors of all nodes visited by ANNS. Notably, while these lookup operations have the same frequency and pattern as graph traversal, the length of the data vectors employed by ANNS is 2.0× greater than that of the graph data due to their high dimensionality. Importantly, although distance calculation exhibits considerable latency, it does not require substantial computational resources, thus making it a good candidate for acceleration using straightforward hardware solutions.

Reducing data vector transfers. We can take the overhead brought by distance calculations off the critical path in the kNN search by bringing only the distance that ANNS needs to check for each iteration its algorithm visits. As shown in Figure 11a, let's suppose that CXL EPs can compute a distance between a given query vector and data vectors that ANNS is in visit. Since ANNS needs the distance, a simple scalar value, instead of all the full features of each data vector, the amount of data that the underlying EPs transfer can be reduced as many as each vector's dimensional degrees. Figure 11b analyzes how much we can reduce the vector transfers during services of the 1M kNN queries. While the vector dimensions of each dataset varies (96∼256), we can reduce the amount of data to load from the EPs by 73.3×, on average.

3.3 Collaborative Approach Overview

Motivated by the aforementioned observations, CXL-ANNS first caches datasets considering a given graph's inter-node relationship and performs ANNS algorithm-aware CXL prefetches (§4.1). This makes the performance of a naive CXL-augmented kNN search comparable with that of the oracle ANNS. To go beyond, CXL-ANNS reduces the vector transferring latency significantly by letting the underlying EPs to calculate all the ANNS distances near memory (§5.1). As this near-data processing is achieved in a collaborative manner between EP controllers and RC-side ANNS algorithm handler, the performance can be limited by the kNN query service sequences. CXL-ANNS thus schedules kNN search activities in a fine-grained manner by relaxing their execution dependency (§5.3). Putting all together, CXL-ANNS is designed for offering high-performance even better than the oracle ANNS without an accuracy loss.

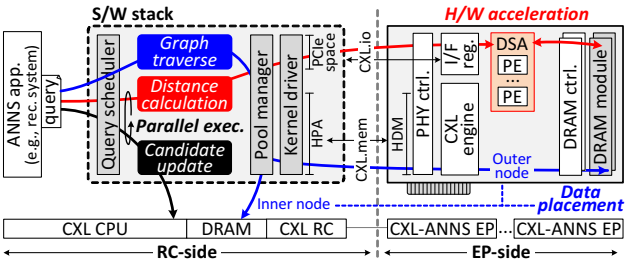


Figure 12: Overview.

Figure 12 shows the high-level viewpoint of our CXL-ANNS architecture, which mainly consists of i) RC-side software stack and ii) EP-side data processing hardware stack.

RC-side software stack. This RC-side software stack is composed of i) query scheduler, ii) pool manager, and iii) kernel driver. At the top of CXL-ANNS, the query scheduler handles all kNN searches requested from its applications such as recommendation systems. It splits each query into three subtasks (graph traverse, distance calculation, and candidate update) and assign them in different places. Specifically, the graph traverse and candidate update subtasks are performed at the CXL CPU side whereas the scheduler allocates the distance calculation to the underlying EP by collaborating with the underlying pool manager. The pool manager handles CXL's HPA for the graph and data vectors by considering edge hop counts, such that it can differentiate graph accesses based on the node-level relationship. Lastly, the kernel driver manages the underlying EPs and their address spaces; it enumerates the EPs and maps their HDMs into the system memory's HPA that the pool manager uses. Since all memory requests for HPA are cached at the CXL CPU, the driver maps EP-side interface registers to RC's PCIe address space using CXL.io instead of CXL.mem. Note that, as the PCIe spaces where the memory-mapped registers exist is in non-cacheable area, the underlying EP can immediately recognize what the host-side application lets the EPs know.

EP-side hardware stack. EP-side hardware stack includes a domain specific accelerator (DSA) for distance calculation in addition to all essential hardware components to build a CXL-based memory expander. At the front of our EPs, a physical layer (PHY) controller and CXL engine are implemented, which are responsible for the PCIe/CXL communication control and flit-to-memory request conversion, respectively. The converted memory request is forwarded to the underlying memory controller that connects multiple DRAM modules at its backend; in our prototype, an EP has four memory controllers, each having a DIMM channel that has 256GB DRAM

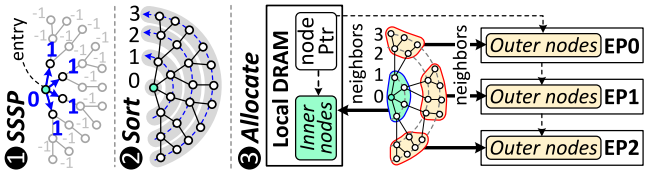


Figure 13: Data placement.

modules. On the other hand, the DSA is located between the CXL engine and memory controllers. It can read data vectors using the memory controllers while checking up the operation commands through CXL engine's interface registers. These interface registers are mapped to the host non-cacheable PCIe space such that all the commands that the host writes can be immediately visible to DSA. DSA calculates the approximate distance for multiple data vectors using multiple processing elements (PEs), each having simple arithmetic units such as adder/subtractor and multiplier.

4 Software Stack Design and Implementation

From the memory pool management viewpoint, we have to consider two different system aspects: i) graph structuring technique for the local memory and ii) efficient space mapping method between HDM and graph. We will explain the design and implementation details of each method in this section.

4.1 Local Caching for Graph

Graph construction for local caching. While the pool manager allocates most graph data and all data vectors to the underlying CXL memory pool, it caches the nodes, expected to be most frequently accessed, in local DRAMs as much as the system memory capacity can accommodate. To this end, the pool manager considers how many edge hops (i.e., calculating the number of edge hops) exist from the fixed entry-node to each node for its relationship-aware graph cache. Figure 13 explains how the pool manager allocates the nodes in a given graph to different places (local memory vs. CXL memory pool). When constructing the graph, the pool manager calculates per-node hop counts by leveraging a single source shortest path (SSSP) algorithm [50, 51]; it first lets all the nodes in the graph have a negative hop count (e.g., -1). Starting from the entry-node, the pool manager checks all the nodes in one edge hop and increases its hop count. It visits each of the nodes and iterates this process for them until there is no node to visit in a breadth-first search manner. Once each node has its own hop count, the pool manager sorts them based on the hop count in an ascending order and allocates the nodes from the top (having the smallest hop count) to local DRAMs as many as it can. The available size of the local DRAMs can be simply estimated by referring to system configuration variables (`sysconf()`) of the total number of pages (`_SC_AVPHYS_PAGES`) and the size of each page (`_SC_PAGESIZE`). It's important to mention that in this study, the pool manager uses several threads within the user space to execute SSSP, aiming to reduce the construction time

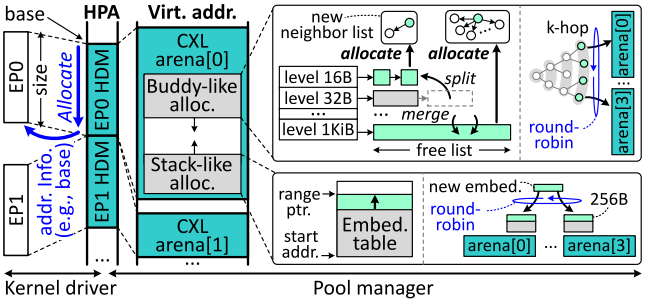


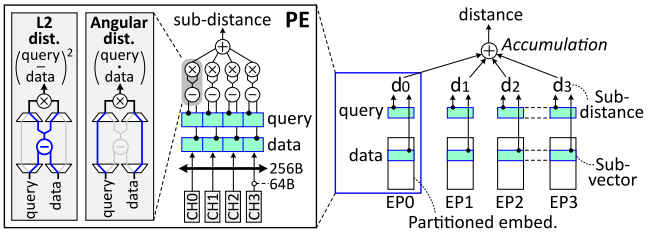
Figure 14: Memory management.

to a minimum. Once the construction is done, the threads are terminated to make sure they do not consume CPU resources when a query is given.

4.2 Data Placement on the CXL Memory Pool

Preparing CXL for user-level memory. When mapping HDM to the system memory's HPA, CXL CPU should be capable of recognizing different HDMs and their size whereas each EP needs to know where its HDM is assigned in HPA. As shown in Figure 14, our kernel driver checks PCIe configuration space and figures out CXL devices at the PCIe enumeration time. The driver then checks RC information from the system's data structure describing the hardware components that show where the CXL HPA begins (base), such as device tree [52] or ACPI [53]. From the base, our kernel driver allocates each HDM as much as it defines in a contiguous space. It lets the underlying EPs know where each of corresponding HDM is mapped in HPA, such that they can convert the address of memory requests (HPA) to its original HDM address. Once all the HDMs are successfully mapped to HPA, the pool manager allocates each HDM to different places of user-level virtual address space that the query scheduler operates on. This memory-mapped HDM, called *CXL arena*, guarantees per-arena continuous memory space and allows the pool manager to distinguish different EPs at the user-level.

Pool management for vectors/graph. While CXL arenas directly expose the underlying HDMs of CXL EPs to user-level space, it should be well managed to accommodate all the billion-point datasets appreciating their memory usage behaviors. The pool manager considers two aspects of the datasets; the data vectors (i.e., embedding table) should be located in a substantially large and consecutive memory space while the graph structure requires taking many neighbor lists with variable length (16B∼1KB). The pool manager employs stack-like and buddy-like memory allocators, which grow upward and downward in each CXL arena, respectively. The former allocator has a range pointer and manages memory for the embedding table, similar to stack. The pool manager allocates the data vectors across multiple CXL arenas in a round-robin manner by considering the underlying EP architecture. This vector sharding method will be explained in §5.1. In contrast, the buddy-like allocator employs a level pointer,



(a) PE architecture. (b) Sharding.

Figure 15: Distance calculation.

each level consisting of a linked list, which connects data chunks with different size (from 16B to 1KB). Like Linux buddy memory manager [54], it allocates the CXL memory spaces as much as each neighbor list exactly requires and merge/split the chunk(s) based on the workload behaviors. To make each EP balanced, the pool manager allocates the neighbor lists for each hop in round-robin manner across different CXL arenas.

5 Collaborative Query Service Acceleration

5.1 Accelerating Distance Calculation

Distance computing in EP. As shown in Figure 15a, a processing element (PE) of DSA has arithmetic logic tree connecting a multiplier and subtractor at each terminal for element-wise operations. Depending on how the dataset's features are encoded, the query and data vectors are routed differently to the two units as input. If the features are encoded for the Euclidean space, the vectors are supplied to the subtractor for L2 distance calculation. Otherwise, the multiplexing logics directly deliver the input vectors to the multiplier by bypassing the subtractor such that it can calculate the angular distance. Each terminal simultaneously calculates individual elements of the approximate distance, and the results are accumulated by going through the arithmetic logic tree network from the terminal to its root. In addition, each PE's terminal reads data from all four different DIMM channels in parallel, thus maximizing the EP's backend DRAM bandwidth.

Vector sharding. Even though each EP has many PEs (10 in our prototype), if we locate the embedding table from the start address of an EP in a consecutive order, EP's backend DRAM bandwidth can be bottleneck in our design. This is because each feature vector in the embedding table is encoded by high dimensional information (∼256 dimensions, taking around 1KB). To address this, our pool manager shards the embedding table in a column wise and stores different parts of the table across the different EPs. As shown in Figure 15b, this *vector sharding* splits each vector into multiple sub-vectors based on each EP's I/O granularity (256B). Each EP simultaneously computes its sub-distance from the split data vector that the EP accommodates. Later, the CXL CPU accumulates the sub-distances to get the final distance value. Note that, since the L2 and angular distances are calculated by accumulating the output of element-wise operations, the final distance is the same as the results of the sub-distance accumulation using vector sharding.

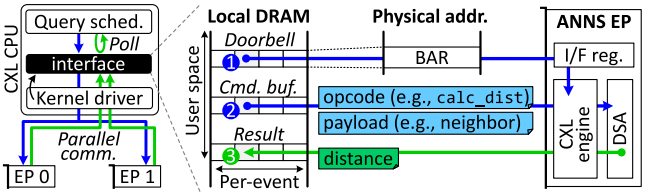


Figure 16: Interface.

Interfacing with EP-level acceleration. Figure 16 shows how the interface registers are managed to let the underlying EPs compute a distance where the data vectors exist. There are two considerations for the interface design and implementation. First, multiple EPs perform the distance calculation for the same neighbors in parallel thanks to vector sharding. While the neighbor list contains many node ids (≤200), it is thus shared by the underlying EPs. Second, handling interface registers using CXL.io is an expensive operation as CPU should be involved in all the data copies. Considering these two, the interface registers handle only the event of command arrivals, called *doorbell* whereas each EP's CXL engine pulls the corresponding operation type and neighbor list from the CPU-side local DRAM (called a command buffer) in an active manner. This method can save the time for CPU to move the neighbor list to each EP's interface registers one by one as the CXL engine brings all the information if there is any doorbell update. The CXL engine also pushes results of distance calculation to the local DRAM such that the RC-side software directly accesses the results without an access of the underlying CXL memory pool. Note that all these communication buffers and registers are directly mapped to the user-level virtual addresses in our design such that we can minimize the number of context switches between user and kernel mode.

5.2 Prefetching for CXL Memory Pool

Figure 17a shows our baseline of collaborative query service acceleration, which lets EPs compute sub-distances while ANNS's graph traverse and candidate update (including sub-distance accumulation) are handled at the CPU-side. This scheduling pattern is iterated until there is no kNN candidates to visit futher (Algorithm 1). A challenge of this baseline approach is traversing graph can be started once the all node information is ready at the CPU-side. While local caching

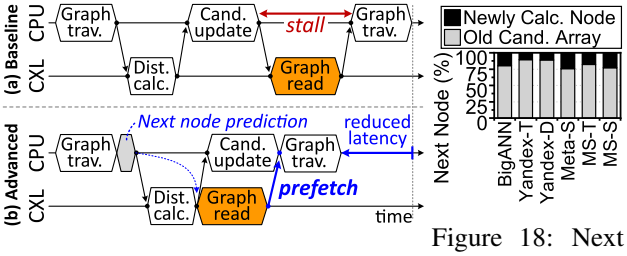


Figure 17: Prefetching.

Figure 18: Next node's source.

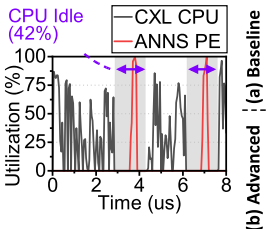


Figure 18: Resource utilization.

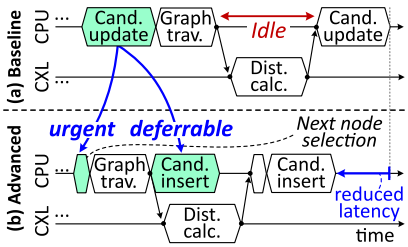


Figure 19: Query scheduling.

of our pool manager addreses this, it yet shows a limited performance. It is required to go through the CXL memory pool to get nodes, which does not sit in the innermost edge hops. As its latency to access the underlying memory pool is long, the graph traverse can be postponed comparably. To this end, our query scheduler prefetches the graph information earlier than the actual traverse subtask needs, as shown in Figure 17b.

While this prefetch can hide the long latency imposed by the CXL memory pool accesses, it is non-trivial as the prefetch requires knowing the nodes that the next (future) iteration of the ANNS algorithm will visit. Our query scheduler speculates the nodes to visit and brings their neighbor information by referring to the candidate array, which is inspired by an observation that we have. Figure 18 shows which nodes are accessed in the graph traverse of the next iteration across all the datasets that we tested. We can see that 82.3% of the total visiting nodes are coming from the candidate array (even though its information is not yet updated for the next step).

5.3 Fine-Granular Query Scheduling

Our collaborative search query acceleration can reduce the amount of data to transfer significantly and successfully hide the long latency imposed by the CXL memory pool. However, computing kNN search in different places makes the RC-side ANNS subtasks pending until EPs complete their distance calculation. Figure 18 shows how much the RC-side subtasks (CXL CPU) stay idle, waiting for the distance results. In this evaluation, we use Yandex-D as a representative of the datasets, and its time series are analyzed for the time visiting only first two nodes for their neighbor search. The CXL CPU performs nothing while EPs calculate the distances, which take 42% of the total execution time for processing those two nodes. This idle time cannot be easily removed as candidate update cannot be processed without having their distance.

To address this, our query scheduler relaxes the execution dependency on the candidate update and separates such an update into urgent and deferrable procedures. Specifically, the candidate update consists of i) inserting (updating) the array with the candidates, ii) sorting kNN candidates based on their distance, and iii) node selection to visit. The node selection is an important process because the following graph traverse requires knowing the nodes to visit (urgent). However, sorting/inserting kNN candidates maintain the k numbers of neighbors in the candidate array, which are not to be done immediately. Thus, as shown in Figure 19, the query scheduler performs the node selection before the graph traverse, but it executes the deferrable operations during the distance calculation time by delaying them in a fine-granular manner.

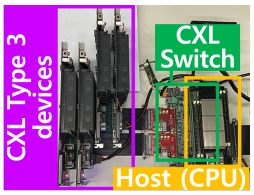


Figure 21: Prototype.

CPU	40 O3 cores, ARM v8, 3.6GHz L1/L2 $: 64KiB/2MiB per core
Local memory	128GiB, DDR4-3200
CXL memory pool	1 CXL switch 256GiB/device, DDR4-3200
Storage	4× Intel Optane 900P 480 GB
CXL-ANNS	1 GHz, 10 ANNS PE/device, 2 distance calc. unit/PE

Table 1: Simulation setup.

6 Evaluation

6.1 Evaluation Setup

Prototype and Methodology. Given the lack of a publicly available, fully functional CXL system, we constructed and validated the CXL-ANNS software and hardware in an operational real system (Figure 21). This hardware prototype is based on a 16nm FPGA. To develop our CXL CPU prototype, we adapted the RISC-V CPU [55]. The prototype integrates 4 ANNS EPs, each equipped with four memory controllers, linked to the CXL CPU via a CXL switch. The system's software for the prototype, including the kernel driver, is compatible with Linux 5.15.36. For the ANNS execution, we adjusted Meta's open ANNS library, FAISS v1.7.2 [56].

Unfortunately, the prototype system does not offer the flexibility needed to explore various ANNS design spaces. As a remedy, we also established a hardware-validated full-system simulator [29] that represents CXL-ANNS, which was utilized for evaluation. This model replicates all operational cycles extracted from the hardware prototype and is cross-validated with our real system at the cycle level. We conducted simulation-based studies in this evaluation, the system details of which are outlined in Table 1. Notably, the system emulates the server utilized in Meta's production environment [57]. Although our system by default uses 4 EPs, our system increases their count for specific workloads (e.g., Meta-S) that necessitate larger memory spaces (more than 2TB) compared to others.

Workloads. We use billion-scale ANNS datasets from BigANN benchmark [30], a public ANNS benchmark that multiple companies (e.g., Microsoft, Meta) participate in. Their important characteristics are summarized in Table 2. In addition, since ANNS-based services often need different number

Dataset	Dist.	Num. vecs.	Emb. dim.	Avg. num. neighbors	Candidate arr. size k=1	k=5	k=10	Num. devices.
BigANN	L2	1B	128	31.6	30	75	150	4
Yandex-T	Ang.	1B	200	29.0	440	900	2500	4
Yandex-D	L2	1B	96	66.9	300	700	1700	4
Meta-S	L2	1B	256	190	1200	2800	5600	8
MS-T	L2	1B	100	43.1	60	130	250	4
MS-S	L2	1B	100	87.4	580	100	200	4

Table 2: Workloads.

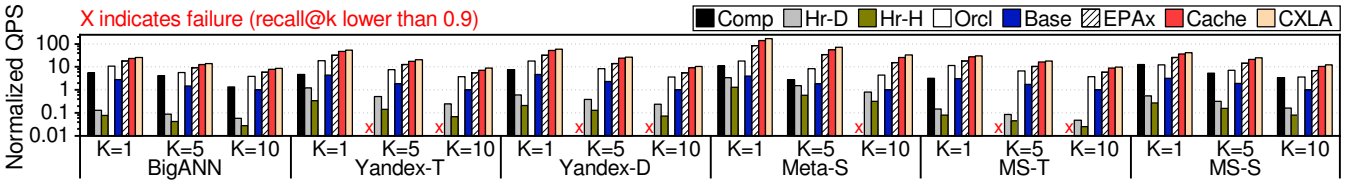
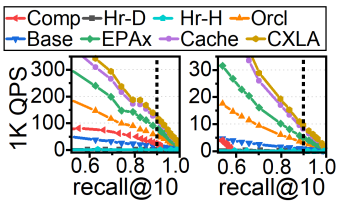


Figure 22: Throughput (queries per second).



(a) BigANN. (b) Yandex-D.

Figure 23: Recall-QPS curve.

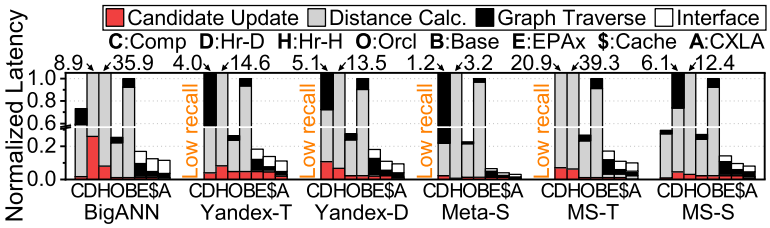


Figure 24: Single query latency (k = 10).

Dataset	Base	CXL-ANNS
BigANN	3.0	0.3
Yandex-T	66.0	7.4
Yandex-D	55.7	5.3
Meta-S	1121.2	34.2
MS-T	6.0	0.6
MS-S	107.2	8.6

* unit: ms

Table 3: Latency.

of nearest neighbors [3, 19], we evaluated our system on the various k (e.g., 1, 5, 10). We generated the graph for ANNS by using state-of-the-art algorithm, NSG, employed by production search services in Alibaba [19]. Since the accuracy and performance of BFS can vary on the size of the candidate array, we only show the performance behavior of our system when its accuracy is 90%, as recommended by the BigANN benchmark. The accuracy is defined as *recall@k*; the ratio of the exact k number of neighbors that are included in the k number of output nearest neighbors of ANNS.

Configurations. We compare CXL-ANNS with 3 state-of-the-art large-scale ANNS systems. For the compression approach, we use a representative algorithm, product quantization [20] (***Comp***). It compresses the data vector by replacing the vector with the centroid of its closest cluster (see §2.2). For the hierarchical approach, we use DiskANN [24] (***Hr-D***) and HM-ANN [25] (***Hr-H***) for the evaluation. The two methods employ compressed embedding table and simplified graphs to reduce the number of SSD/PMEM accesses, respectively. For fair comparison, we use the same storage device, Intel Optane [44], for both Hr-D/H. For CXL-ANNS, we evaluated its multiple variants to distinguish the effect of each method we propose. Specifically, ***Base*** places the graph and embedding table in CXL memory pool and lets CXL CPU execute the subtasks of ANNS. Compared to Base, ***EPAx*** performs distance calculation by using DSA inside the ANNS EP. Compared to EPAx, ***Cache*** employs relationship-aware graph caching and prefetching. Lastly, ***CXLA*** employs all the methods we propose, including fine-granular query scheduling. In addition, we compare oracle system (***Orcl***) that uses unlimited local DRAM. We will show that CXL-ANNS makes the CXL-augmented kNN search faster than Orcl.

6.2 Overall Performance

We first compare the throughput and latency of various systems we evaluated. We measured the systems' throughput by counting the number of processed queries per second (QPS, in short). Figure 22 shows the QPS of all ks, while Figure 24 digs the performance behavior deeper for k=10 by breaking down the latency. We chose k=10 following the guide from BigANN benchmark. The performance behavior for k=1,5 are largely same with when k=10. For both figures, we normalized the values by that of *Base* when k=10. The original latencies are summarized in Table 3.

As shown in Figure 22, the QPS gets lower when the k increases for all the systems we tested. This is because, the BFS visits more nodes to find more nearest neighbors. On the other hand, while Comp exhibits comparable QPS to Orcl, it fails to reach the target recall@k (0.9) for 7 workloads. This is because Comp cannot calculate the exact distance since it replaces the original vector with the centroid of a cluster nearby. This can also be observed in Figure 23. The figure shows the accuracy and QPS when we vary the size of candidate array for two representative workloads. BigANN represents the workloads that Comp does reach the target recall, while Yandex-D represents the opposite. We can see that Comp converges at low recall@10 of 0.92 and 0.58, respectively, while other systems reach the maximum recall@10.

In contrast, hierarchical approaches (Hr-D/H) reaches the target recall@k for all the workloads we tested, by re-ranking the search result. However, they suffer from the long latency of underlying SSD/PMEM while accessing their uncompressed graph and embedding table. Such long latency significantly depletes the QPS of Hr-D and Hr-H by 35.9× and 77.6× compared to Orcl, respectively. Consider Figure 24 to better understand; Since Hr-D only calculates the distance for the limited number of nodes in the candidate array, it exhibits 20.1× shorter distance calculation time compared to Hr-H, which starts a new BFS on original graph stored in SSD/PMEM. However, Hr-D's graph traverse takes longer time than that of Hr-H by 16.6×. This is because, Hr-D accesses the original graph in SSD/PMEM for both low/high-accuracy search while Hr-H accesses the original graph only for their high-accuracy search.

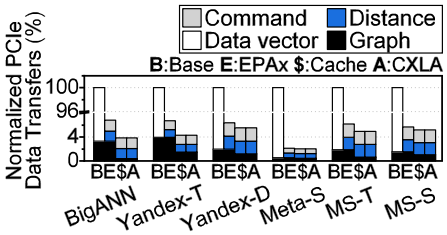


Figure 25: Data transfer.

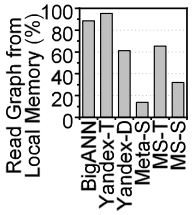


Figure 26: Local caching.

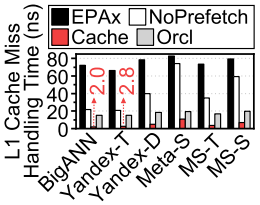


Figure 27: Cache miss handling time.

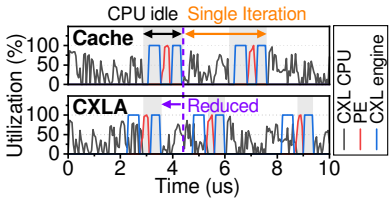


Figure 28: CPU/PE utilization (Yandex-D).

As shown in Figure 22, Base does not suffer from accuracy drop or the performance depletion of Hr-D/H since it employs a scalable memory pool that CXL offers. Therefore, it significantly improves the QPS by 9.4× and 20.3×, compared to Hr-D/H, respectively. However, Base still exhibits 3.9× lower throughput than Orcl. This is because Base experiences the long latency of memory-to-flit conversion while accessing the graph/embedding in CXL memory pool. Such conversion makes Base's graph traverse and distance calculation longer by 2.6× and 4.3×, respectively, compared to Orcl.

Compared to Base, EPAx significantly diminishes the distance calculation time by a factor of 119.4×, achieved by reducing data vector transfer through the acceleration of distance calculation within the EPs. While this EP-level acceleration introduces an interface overhead, this overhead only represents 5.4% of the Base's distance calculation latency. Hence, EPAx reduces the query latency by 7.5× on average, relative to Base. It's important to highlight that EPAx's latency is 1.9× lower than Orcl's, which has unlimited DRAM. This discrepancy stems from Orcl's insufficient grasp of the ANNS algorithm and its behaviour, which results in considerable data movement overhead during data transfer between local memory and the processor complex. Additional details can be found in Figure 25, depicting the volume of data transfer via PCIe for the CXL-based systems. The figure shows that EPAx eliminates data vector transfer, thereby cutting down data transfer by 21.1×.

Further, Cache improves EPAx's graph traversal time by 3.3×, thereby enhancing the query latency by an average of 32.7%. This improvement arises because Cache retains information about nodes anticipated to be accessed frequently in the local DRAM, thereby handling 59.4% of graph traversal within the local DRAM (Figure 26). The figure reveals a particularly high ratio for BigANN and Yandex-T, at 92.0%. As indicated in Table 2, their graphs have a relatively small number of neighbors (31.6 and 29.0, respectively), resulting in their graphs being compact at an average of 129.3GB. In contrast, merely 13.8% of Meta-S's graph accesses are serviced from local memory, attributable to its extensive graph. Nevertheless, even for Meta-S, Cache enhances graph traversal performance by prefetching graph information before actual visitation. As depicted in Figure 24, this prefetching can conceal CXL's prolonged latency, reducing Meta-S's graph traversal latency by 72.8%. While prefetching would introduce overhead in speculating the next node visit, it is insignificant, accounting for only 1.3% of the query latency. These caching and prefetching techniques yield graph processing performance similar to that of Orcl. We will explain the details of prefetching shortly.

Lastly, as depicted in Figure 22, CXLA boosts the QPS by 15.5% in comparison to Cache. This is due to CXLA's enhancement of hardware resource utilization by executing deferrable subtasks and distance calculations concurrently in the CXL CPU and PE, respectively. As illustrated in Figure 24, such scheduling benefits Yandex-T, Yandex-D, and Meta-S more so than others. This is attributable to their use of a candidate array that is, on average, 16.3× larger than others, which allows for the overlap of updates with distance calculation time. Overall, CXLA attains a significantly higher QPS than Orcl, surpassing it by an average factor of 3.8×.

6.3 Collaborative Query Service Analysis

Prefetching. Figure 27 compares the L1 cache miss handling latency while accessing the graph for the CXL-based systems we tested. We measured the latency by dividing the total L1 cache miss handling time of CXL CPU by the number of L1 cache access. The new system, NoPrefetch, disables the prefetching from Cache. As shown in Figure 27, EPAx's latency is as long as 75.4ns since it accesses slow CXL memory whenever there is a cache miss. NoPrefetch alleviates such problem thanks to local caching, shortening the latency by 45.8%. However, when the dataset uses a large graph (e.g. Meta-S, MS-S), only 24.5% of the graph can be cached in local memory. This makes NoPrefetch's latency 2.3× higher than that of Orcl. In contrast, Cache significantly shortens the latency by 8.5× which is even shorter than that of Orcl. This is because Cache can foresee the next visiting nodes and loads the graph information in the cache in advance. Note that, Orcl accesses local DRAM on demand on cache miss.

Utilization. Figure 28 shows the utilization of CXL CPU, PE, CXL engine on a representative dataset (Yandex-D). To clearly provide the behavior of our fine-granule scheduling, we composed a CXL-ANNS with single-core CXL CPU and single PE per device and show their behavior in a timeline. The upper part of the figure shows the behavior of Cache that does not employ the proposed scheduling. We plot CXL CPU's utilization as 0 when it polls the distance calculation results of PE, since it does not perform any useful job during that time. As shown in the figure, CXL CPU idles for 42.0% of the total time waiting for the distance calculation result. In

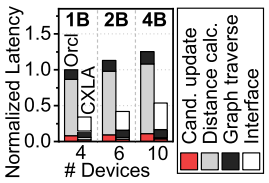


Figure 29: Device scaling (Yandex-D).

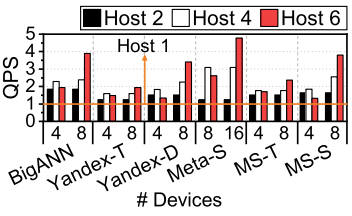


Figure 30: Host Sensitivity.

contrast, CXLA reduces the idle time by 1.3×, relaxing the dependency between ANNS subtasks. In the figure, we can see that the CXL CPU's candidate update time overlaps with the time CXL engine and PE handling the command. As a result, CXLA improves the utilization of hardware resources in CXL network by 20.9%, compared to Cache.

6.4 Scalability Test

Bigger Dataset. To evaluate the scalability of CXL-ANNS, we increase the number of data vectors in Yandex-D by 4B, and connect more EP to CXL CPU to accommodate their data. Since there is no publicly available dataset that is as large as 4B, we synthetically generated additional 3B vectors by adding noise to original 1B vectors. As shown in Figure 29, we can see that the latency of Orcl increases as we increase the scale of dataset. This is because larger dataset makes BFS visit more nodes to maintain the same level of recall. On the other hand, we can see the interface overhead of CXLA increases as we employ more devices to accommodate bigger dataset. This is because the CXL CPU should notify more devices for the command arrival by ringing the doorbell. Despite such overhead, CXLA exhibits 2.7× lower latency than Orcl thanks to its efficient collaborative approach.

Multi-host. In a disaggregated system, a natural way to increase the system's performance is to employ more host CPUs. Thus, we evaluate the CXL-ANNS that supports multiple hosts in the CXL network. Specifically, we split EP's resources such as HDM and PEs and then allocate each of them to one the the CXL hosts in the network. For ANNS, we partition the embedding table and make each host responsible for finding kNN from different partitions. Once all the CXL hosts find the kNN, the system gathers them all and rerank the neighbors to finally select kNN among them.

Figure 30 shows the QPS of multi-host ANNS. The QPS is normalized to that when we use single CXL host with the same number of EPs that we used before. Note that we also show the QPS when we employ more number of EPs than we used before. When the number of EPs stays the same, the QPS increases until we connect 4 CXL hosts in the system. However the QPS drops when the number of CXL hosts is 6. This is because the distance calculation by limited number of PEs became the bottleneck; the commands from the host pends since there is no availabe PE. Such problem can be addressed by having more EPs in the system, thereby distributing the computation load. As we can see in the figure, when we double the number of EPs in the network, we can improve the QPS when we have 6 CXL hosts in the system.

7 Discussion and Acknowledgments

GPU-based distance calculation. Recent research has begun to leverage the massive parallel processing capabilities of GPUs to enhance the efficiency of graph-based ANNS services [58, 59]. While GPUs generally exhibit high performance, our argument is that it's not feasible for CPU+GPU memory to handle the entirety of ANNS data and tasks, as detailed in Section 1. Even under the assumption that ANNS is functioning within an optimal in-memory computing environment, there are two elements to consider when delegating distance computation to GPUs. The first point is that GPUs require interaction with the host's software and/or hardware layers, which incurs a data transfer overhead for computation. Secondly, ANNS distance computations can be carried out using a few uncomplicated, lightweight vector processing units, making GPUs a less cost-efficient choice for these distance calculation tasks.

In contrast, CXL-ANNS avoids the burden of data movement overhead, as it processes data in close proximity to its actual location and returns only a compact result set. This approach to data processing is well established and has been validated through numerous application studies [48, 60–66]. Moreover, CXL-ANNS effectively utilizes the cache hierarchy and can even decrease the frequency of accesses to the underlying CXL memory pool. It accomplishes this through its CXL-aware and ANNS-aware prefetching scheme, which notably enhances performance.

Acknowledgments. The authors thank anonymous reviewers for their constructive feedback as well as Panmnesia for their technical support. The authors also thank Sudarsun Kannan for shepherding this paper. This work is supported by Panmnesia and protected by one or more patents. Myoungsoo Jung is the corresponding author (mj@camelab.org)

8 Conclusion

We propose CXL-ANNS, a software-hardware collaborative approach for scalable ANNS. CXL-ANNS places all the dataset into its CXL memory pool to handle billion-point graphs while making the performance of the kNN search comparable with that of the (local-DRAM only) oracle system. To this end, CXL-ANNS considers inter-node relationship and performs ANNS-aware prefetches. It also calcualate distances in its EP while scheduling the ANNS subtasks to utilize all the resources in the CXL network. Our empirical results show that CXL-ANNS exhibits 111.1× better performance compared to the state-of-the-art billion-scale ANNS methods and 3.8× better performance than oracle system, respectively.

References

[1] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, et al. Approximate nearest neighbor search under neural similarity metric for large-scale recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*, 2022.

[2] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[3] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, et al. Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2022.

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

[5] Minjia Zhang and Yuxiong He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, 2019.

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

[7] Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. RIANN: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 USENIX Conference on Operational Machine Learning (OpML 20)*, 2020.

[8] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. Autosys: The design and operation of learning-augmented systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 323–336, 2020.

[9] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2008.

[10] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[11] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[12] Pandu Nayak. Understanding searches better than ever before. https://blog.google/products/search/search-language-understanding-bert/, 2019.

[13] Charlie Waldburger. As search needs evolve, microsoft makes ai tools for better search available to researchers and developers. https://news.microsoft.com/source/features/ai/bing-vector-search/, 2019.

[14] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[15] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998.

[16] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6), 1998.

[17] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. An investigation of practical approximate nearest neighbor algorithms. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems (NIPS)*, 2004.

[18] Harsha Simhadri. Research talk: Approximate nearest neighbor search systems at scale. https://www.youtube.com/watch?v=BnYNdSIKibQ&list=PLD7HFcN7LXReJTWFKYqwMcCc1nZKIXBo9&index=9, 2021.

[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 2019.

[20] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE*

Transactions on Pattern Analysis and Machine Intelligence, 2010.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning (ICML)*, 2020.

[22] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[23] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[24] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

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

[26] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[27] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023 (WWW 23)*, 2023.

[28] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighbor search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[29] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[30] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 2022.

[31] Sunil Arya and David M Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, 1993.

[32] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[33] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 36(2):388–403, 2013.

[34] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 2015.

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

[36] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International conference on similarity search and applications*. Springer, 2017.

[37] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the lernaean hydra: experimental evaluation of data series approximate similarity search. *Proceedings of the VLDB Endowment*, 2019.

[38] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

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

[40] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[41] CXL Consortium. Compute express link 3.0 white paper. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[42] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[43] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[44] Intel. Optane ssd 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html, 2021.

[45] Michael Anderson, Benny Chen, Stephen Chen, Summer Deng, Jordan Fix, Michael Gschwind, Aravind Kalaiah, Changkyu Kim, Jaewon Lee, Jason Liang, et al. First-generation inference accelerator deployment at facebook. *arXiv preprint arXiv:2107.04140*, 2021.

[46] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[47] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[48] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[49] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[50] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[51] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. Efficient single-source shortest path and distance queries on large graphs. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013*. ACM, 2013.

[52] Linaro. The devicetree specification. .https://www.devicetree.org/.

[53] Inc UEFI Forum. Advanced configuration and power interface (acpi) specification version 6.4. https://uefi.org/specs/ACPI/6.4/, 2021.

[54] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 1965.

[55] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[56] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and Alexandr Guzhva. Faiss. https://github.com/facebookresearch/faiss, 2018.

[57] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

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

[59] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[60] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/software co-programmable

framework for computational SSDs to accelerate deep learning service on large-scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[61] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[62] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Failure tolerant training with persistent memory disaggregation over cxl. *IEEE Micro*, 2023.

[63] Jun Heo, Seung Yul Lee, Sunhong Min, Yeonhong Park, Sung Jun Jung, Tae Jun Ham, and Jae W Lee. Boss: Bandwidth-optimized search accelerator for storage-class memory. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[64] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[65] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, et al. Genstore: a high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[66] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. A case study of processing-in-memory in off-the-shelf systems. In *USENIX Annual Technical Conference (ATC)*, 2021.