ANNA: Specialized Architecture for Approximate Nearest Neighbor Search

Yejin Lee* Hyunji Choi* Sunhong Min* Hyunseung Lee* Sangwon Beak†
Dawoon Jeong* Jae W. Lee* Tae Jun Ham†

Seoul National University
*{yejinlee, hyunjichoi, sunhongmin, hs_lee, daun20211, jaewlee}@snu.ac.kr
†{bsw1907, ham.taejun}@gmail.com

***Abstract*—Similarity search or nearest neighbor search is a task of retrieving a set of vectors in the (vector) database that are most similar to the provided query vector. It has been a key kernel for many applications for a long time. However, it is becoming especially more important in recent days as modern neural networks and machine learning models represent the semantics of images, videos, and documents as high-dimensional vectors called *embeddings*. Finding a set of similar embeddings for the provided query embedding is now the critical operation for modern recommender systems and semantic search engines. Since exhaustively searching for the most similar vectors out of billion vectors is such a prohibitive task, approximate nearest neighbor search (ANNS) is often utilized in many real-world use cases. Unfortunately, we find that utilizing the server-class CPUs and GPUs for the ANNS task leads to suboptimal performance and energy efficiency. To address such limitations, we propose a specialized architecture named ANNA (Approximate Nearest Neighbor search Accelerator), which is compatible with state-of-the-art ANNS algorithms such as Google ScaNN and Facebook Faiss. By combining the benefits of a specialized dataflow pipeline and efficient data reuse, ANNA achieves multiple orders of magnitude higher energy efficiency, 2.3-61.6× higher throughput, and 24.0-620.8× lower latency than the conventional CPU or GPU for both million- and billion-scale datasets.**

I. Introduction

A naïve text search engine may find that the query *computer architecture* is more related to *Gothic architecture* than *hardware accelerator*, mainly because the term *Gothic architecture* shares a word with the query while *hardware accelerator* does not. Fortunately, this is not the case in modern search engines. The recent advancements in machine learning technologies enable the computer to semantically understand the query so that it can effectively retrieve the most relevant results.

A variety of modern machine learning and neural network algorithms represent an entity as a learned high-dimensional vector or *embedding*. For example, Word2Vec [30] represents words as dense vectors in a way that semantically similar vectors are located closer in the vector space. Similarly, it is possible to learn the vector representation of sentences [8], images [5, 38], videos [29], and many others. With such embeddings, the task of retrieving entities that are similar to the query becomes finding a set of vectors that are close to the query embedding vector in the vector space. This task is often called *similarity search* or *nearest neighbor search*.

The most popular application of NNS is recommender systems. For example, YouTube recommender systems first utilize NNS to identify a set of candidate videos for a specific user and then use a separate, heavy deep neural network to select top recommendations [11]. Similarly, NNS is often used to efficiently identify a set of candidates [33] for CTR (Click-Through-Rate) prediction models, such as Facebook DLRM [32], Google DCN [41] and Alibaba BST [9]. Moreover, similarity search is used for conventional search engines such as Microsoft Bing [28], and can also be used for multimedia search services where a user provides image or audio input to find similar ones.

The broad applicability of similarity search has motivated many researchers in academia and industry to explore various similarity search algorithms. For example, Facebook designed and maintains a library for similarity search named Faiss [24] and Google also recently released a similarity search library named ScaNN [18]. Microsoft [10], Yahoo Japan [44] also have their own similar search libraries. However, exhaustively inspecting every vector in the database and computing the similarity between the query vector and the candidate vector requires a large amount of computation as well as memory accesses. Therefore, most existing similarity search implementations utilize approximate nearest neighbor search (ANNS) algorithms which require much less computation as well as memory accesses at the expense of a slight reduction in search accuracy. There exist various ANNS algorithms such as hash-based ones [12, 39], graph-based ones [19, 26, 47], and compression-based ones [18, 22, 24]. Among these solutions, compression-based ones are the popular choices for billion-scale search scenarios. Graph-based ones and hash-based ones are very effective for million-scale searches (e.g., finding similar movies), but they are not well-suited for billion-scale searches where their memory requirement (i.e., whole dataset as well as their large index structures) often exceeds the main memory size. On the other hand, compression-based schemes can often fit necessary data in the main memory since they only need to hold the compressed version of the dataset and a lightweight index structure.

Unfortunately, we find running such compression-based ANNS schemes on conventional hardware (CPU or GPU) sub-optimal due to its specific computation pattern that heavily utilizes memoization. For CPUs, the sources of inefficiency are twofold: (1) application's lack of control on the on-chip memory usage makes it difficult to keep the memoization

results and frequently reused data on-chip, incurring further main memory accesses, and (2) CPU's ability to support dynamic control flow or dynamic extraction of instruction-level parallelism incurs extra energy overhead on ANNS scheme, which has relatively static control flow and easy-to-extract parallelism. On the other hand, on GPUs, (1) shared memory usage from the memoization table limits the GPU parallelism (i.e., the number of thread blocks scheduled for each streaming multiprocessor (SM)) and incurs resource underutilization, and (2) the low arithmetic intensity computation pattern results in gross underutilization of GPU computation capability.

To address such inefficiencies, we present ANNA (**A**pproximate **N**earest **N**eighbor search **A**ccelerator), which is specialized hardware for the efficient compression-based similarity search algorithms. ANNA is carefully designed to support various compression-based similarity search algorithms and thus is directly compatible with existing software libraries such as Facebook Faiss and Google ScaNN. With ANNA, finding vectors similar to a query vector out of billion vectors requires substantially less time and energy compared to the CPU and GPU implementations at a fraction of their chip area. Below are the key contributions of our work.

- We present ANNA, a specialized architecture for ANNS which can accelerate modern ANNS algorithms such as Facebook Faiss [24] and Google ScaNN [18] (Section III).
- We introduce a memory traffic optimization technique for ANNA, which reduces the memory traffic and improve performance through efficient data reuse (Section IV).
- We evaluate ANNA on multiple billion-scale workloads. ANNA improves the energy efficiency of ANNS by multiple orders of magnitude (97×+) while improving the search throughput and latency by 2.3-61.6× and 24.0-620.8× compared to the conventional CPU and GPU (Section V).

II. Background and Motivation

A. Similarity Search

Similarity search is a task of finding top-k vectors in a given set (i.e., database vectors) that are the most similar to a query vector. It is also called the nearest neighbor search (NNS) problem. Various similarity functions can be used to define the similarity between vectors. Among them, the most common similarity metrics are inner product similarity and L2 distance similarity shown in the equation below.

$$s_{ip}(q,x) = \sum_{i=0}^{D-1} q[i]x[i] = \|q\|\|x\|\cos\theta \text{ (Inner Product)}$$
$$s_{L2}(q,x) = -\sum_{i=0}^{D-1}(q[i]-x[i])^2 \text{ (L2 Distance)}$$

Similarity Metrics. Inner product is one of the most popular similarity metrics. Here, a similarity $s(q,x)$ between two D-dimensional vectors q and x is defined as the dot product of those two vectors. This metric is used for computing the similarity between embeddings, and the NNS using the inner product is called Maximum Inner Product Search (MIPS). L2 distance between vector q and x is computed as the sum of the squared value of each dimension's difference. Geometrically, this is the squared value of the length of the line connecting two points defined by each vector. Since L2 distance is technically a dissimilarity metric, the negative of squared L2 distance is utilized as a similarity metric, as shown in the above equation. A common use case of L2 distance search is image similarity search.

Approximate Similarity Search. The most naïve way to perform a similarity search is to compute the similarity between the query vector and all database vectors, then sort the similarities to obtain k vectors with the highest similarity. Assuming N database vectors having D dimensions, this requires ND multiply-and-add operations and $2ND$ bytes memory accesses, assuming 16-bit datatype for each vector element. The cost of this operation becomes prohibitive when N is large (e.g., a billion). To avoid this problem, many approximate similarity search algorithms (ANNS) have been proposed [13, 14, 18, 20, 21, 24, 26]. Such algorithms can significantly reduce the amount of computation as well as the memory accesses. However, the main problem with most of those solutions is that they still require all data as well as additional index structures to be resident in memory. The dataset itself is 256GB when N is a billion, and D is 128. This only fits in a relatively large single node machine. Moreover, many ANNS algorithms also require additional $O(N)$ memory capacity for index structure [27]. To avoid such huge capacity requirements, compression-based ANNS algorithms have been proposed. Such algorithms encode the original data in a compressed form and utilize the compressed form to compute the similarity between the query and the database vectors. Since these algorithms do not require the original, uncompressed vectors to be kept in memory, they require much less memory capacity and are better suited for large-scale datasets.

B. Product Quantization (PQ)

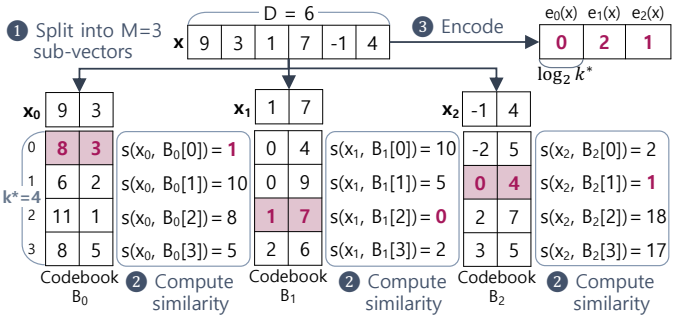

Fig. 1. Illustration of Product Quantization.

Encoding. A product quantization scheme encodes each database vector into a compressed form as shown in Figure 1. For this purpose, it first divides a vector into multiple sub-vectors. For example, a D-dimensional vector x is represented as a concatenation of M sub-vectors $x_0, x_1, x_2, ..x_{M-1}$ where each x_i is a D/M-dimensional vector. In the figure, ❶ we assume $D=6$ and $M=3$, so each sub-vector is a 2-dimensional vector. Then, the product quantization scheme encodes each sub-vector into an identifier and represents the vector as a concatenation of identifiers. To obtain a corresponding identifier for each sub-vector, the product quantization scheme uses a

set of vectors named codebook. A separate training/learning process obtains codebooks $B_0, ..., B_{M-1}$, one for each D/M dimensions, and each codebook consists of k^* codeword vectors, each having D/M dimension. The figure shows $M=3$ codebooks of $k^*=4$ codeword vectors each. There exist various algorithms to obtain codebooks [16, 18, 24, 25], and the quality of the codebook affects final search accuracy. Using a codebook B_i, a sub-vector x_i is encoded into an identifier. ❷ Specifically, the product quantization scheme first computes the similarity between a sub-vector x_i and each codeword vector in codebook B_i (i.e., $B_i[0], ...B_i[k^*-1]$). Eventually, the codeword with the highest similarity is selected.

Assuming $B_i[j]$ is selected, x_i is now represented as $e_i(x) = j$. ❸ Repeating this process for each sub-vector x_i, the vector x is now represented as a concatenation of $x' = e_0(x), ..., e_{M-1}(x)$ where $0 \leq e_i(x) < k^*$. Assuming that the original datatype is 2 bytes (float16) per each vector element, the original vector requires $2D$ bytes storage. With this encoding, each $e_i(x)$ is represented with $\log_2 k^*$ bits and the total number of bytes for the encoded vector x' is $M \log_2 k^*/8$ bytes. In the figure, original vector x requires 12 bytes storage, but with encoding it only requires less than 1 byte (e.g., 6/8 bytes). While the choice of smaller M and k^* significantly improves the compression rate, too much compression can negatively affect search accuracy.

Approximate Similarity Computation. To compute the approximate similarity between each encoded vector x' and a given query q, the PQ scheme utilizes the encoded identifier e_i to obtain the corresponding codeword from codebook B_i. Then, the decoded vector is represented as a concatenation of $B_0[e_0(x)], ...B_{M-1}[e_{M-1}(x)]$ where $B_i[e_i(x)]$ is a D/M-dimensional vector. The similarity between this D-dimensional decoded vector and the query vector ($q_0, ...q_{M-1}$ where q_i is a D/M-dimensional sub-vector) is computed as shown below. This process is repeated for all encoded vectors, and the top-k most similar vectors can be obtained.

$$s_{ip}(q, x') = \sum_{i=0}^{M-1} q_i B_i[e_i(x)] \text{ (Inner Product)}$$
$$s_{L2}(q, x') = -\sum_{i=0}^{M-1} \|q_i - B_i[e_i(x)]\|^2 \text{ (L2 Distance)}$$

Efficient Similarity Computation with Memoization. In above equations, an inner product similarity computation requires D multiplications and $D-1$ additions. Similarly, a L2 distance similarity computation requires D subtractions, D multiplications, and $D-1$ additions. These are repeated for all N database vectors. Carefully inspecting the above equations, we can see that it is more efficient to memoize $\{q_i B_i[0], \ ... \ , q_i B_i[k^*-1]\}$ (inner product) or $\{-\|q_i - B_i[0]\|^2, \ ... \ , -\|q_i - B_i[k^*-1]\|^2\}$ (L2 distance), and reuse them across different database vectors. Assuming those k^* values are memoized in a lookup table L_i (for all $0 \leq i < M$), the above similarity computation equation changes to the following for both the inner product similarity and the L2 distance similarity.

$$s(q, x') = \sum_{i=0}^{M-1} L_i[e_i(x)]$$

With this memoization, computing the similarity between q and x' only requires M lookup table references and $M-1$ additions. To construct the lookup table for inner product similarity, k^*D multiplications and $k^*(D-1)$ additions are necessary. Similarly, the L2 distance similarity computation requires k^*D subtractions, k^*D multiplications, and $k^*(D-1)$ additions. In both cases the required capacity is $2Dk^*$ bytes since there exist M lookup tables and each lookup table L_i has k^* entries where each entry stores D/M-dimensional sub-vector requiring $2D/M$ bytes storage. Note that the size of the lookup table or the amount of computation for construction of the lookup table is independent of N. With a large number of database vectors, the lookup table construction overhead can easily be amortized.

C. Two-level Product Quantization ANNS

The base product quantization scheme presented in the previous section i) reduces the memory capacity requirement and ii) reduces the amount of computation via memoization. Still, it does not reduce the number of database vectors that the similarity computation needs to be performed. As a result, the total search time is still proportional to N. To address this limitation, many popular similarity search implementations adopt a two-level product quantization scheme that substantially reduces the number of database vectors for which the distance computation needs to be performed.

Encoding. In this scheme, the database vectors are first grouped into multiple clusters using a clustering algorithm. Technically, any clustering algorithm can be utilized, but *kmeans* is the most popular choice. After clustering, a representative point from each cluster is decided. A popular way to obtain this representative point is simply using the centroid of all vectors within a cluster. For the cluster j, we call the centroid vectors as $c^{(j)} \in R^D$. Then, for each database vector x in cluster j, the residual $r(x) = x - c^{(j)}$ is computed. Finally, $r(x)$ is encoded using the product quantization scheme explained in Section II-B to obtain $r(x)'$ which is a concatenation of $e_0(r(x)), e_1(r(x)), ..., e_{M-1}(r(x))$. Then, these encoded vectors belonging to this specific cluster are stored together, along with the cluster centroid vector $c^{(j)}$.

$$\begin{aligned} s_{ip}(q, x') &= s_{ip}(q, c^{(j)} + r(x)) = \ q \cdot c^{(j)} + \sum_{i=0}^{M-1} q_i B_i[e_i(r(x))] \\ &= \ q \cdot c^{(j)} + \sum_{i=0}^{M-1} L_i[e_i(r(x))] \\ s_{L2}(q, x') &= \ s_{L2}(q, c^{(j)} + r(x)) = s_{L2}(q - c^{(j)}, r(x)) \\ &= \ -\sum_{i=0}^{M-1} \|(q_i - c_i^{(j)}) - B_i[e_i(r(x))]\|^2 \\ &= \ \sum_{i=0}^{M-1} L_i[e_i(r(x))] \end{aligned}$$

Similarity Computation. Such cluster-wise encoding scheme changes the way to compute the similarity, as shown in the equations above. L_i for inner product similarity stores $\{q_i B_i[0], ..., q_i B_i[k^*-1]\}$ and L_i for L2 distance similarity stores $\{-\|(q_i - c_i^{(j)}) - B_i[0]\|^2, ..., -\|(q_i - c_i^{(j)}) - B_i[k^*-1]\|^2\}$, where $c^{(j)}$ is one of C . For the inner product similarity case, the contents of the lookup table are invariant to the chosen clusters and the term $q \cdot \ c^{(j)}$ needs to be added at the end. On

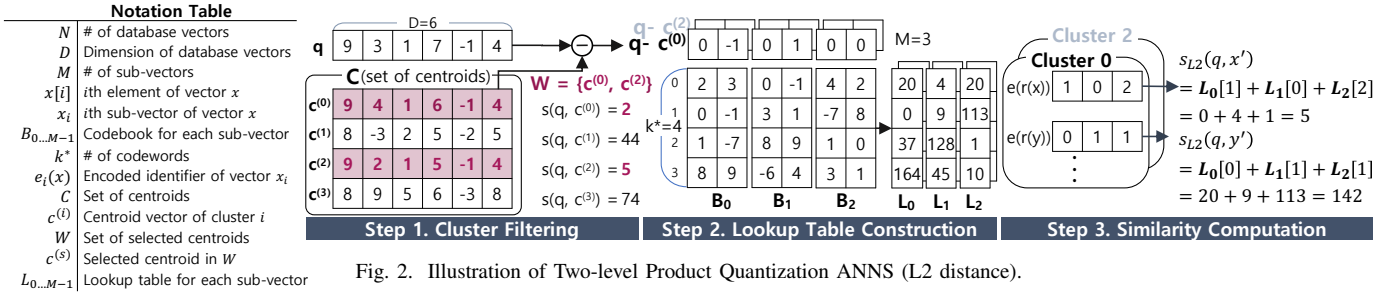

Fig. 2. Illustration of Two-level Product Quantization ANNS (L2 distance).

the other hand, for the L2 distance similarity case, the contents of the lookup table are variant to the chosen clusters.

Search Process Step 1 - Cluster Filtering. Figure 2 illustrates three steps of two-level product quantization. Given a query vector q, the first step to find similar vectors is cluster selection. During this step, it computes the similarity between vector q and all centroid vectors (i.e., $c^{(0)}, c^{(1)}, ..., c^{(|C|-1)}$), and then find the W most similar centroids as shown in the below equation. The set of selected centroids is denoted as W. In the figure, the most similar (in terms of L2 distance) centroids are $c^{(0)}$ and $c^{(2)}$ and thus $W = \{c^{(0)}, c^{(2)}\}$. Essentially, this step excludes vectors belonging to the cluster with centroids that are not similar to the query and hence effectively reduce the total number of candidates.

Search Process Step 2 - Lookup Table Construction. Once the nearby centroids are identified, the algorithm constructs the lookup table. This process slightly differs for the two metrics. First, for the inner product similarity, the contents of the lookup table L_i (see s_{ip} in the above equation) is independent of the selected centroid $c^{(s)}$, where $c^{(s)}$ is one of the selected clusters in W, and thus it is sufficient to construct the lookup table once and reuse the table for all clusters. On the other hand, the contents of the lookup table (see s_{L2} in the above equation) are dependent on the selected centroid $c^{(s)}$ and thus the table needs to be constructed for each cluster. The figure assumes $D=6$, $M=3$ and L2 distance case and illustrates creating a lookup table for cluster 0.

Search Process Step 3 - Similarity Computation. Once the lookup table is ready, the similarity computation between the query vector q and database vector in the selected clusters is performed. With the lookup table, each similarity computation is simply M additions (inner product) or $M-1$ additions (L2 distance). The figure shows the process of computing similarity of vector x. Assuming encoded vector $e(r(x))$ is $(1, 0, 2)$, it computes similarity by summing up $L_0[e_0(r(x))] + L_1[e_1(r(x))] + L_2[e_2(r(x))]$ which is 5. This process is repeated for all vectors in the selected clusters, and the top-k most similar vectors are selected and then returned. Overall, this PQ-based ANNS can i) substantially reduce the number of database vectors inspected by cluster-level filtering and ii) can efficiently compute the approximate similarity between the database vector and the query with memoization.

D. Analysis of PQ-based ANNS on Conventional Hardware

An algorithm suits best to the specific hardware if i) the hardware fully utilizes the available compute resources while executing the algorithm, ii) the hardware can maximize data reuse, and iii) the hardware fully utilizes the available memory bandwidth for data that needs to be loaded from memory. Unfortunately, we find that this is not the case when we run PQ-based ANNS on conventional hardware.

GPU Implementation Analysis. We profile Facebook Faiss [15, 24], one of the most popular implementations of the PQ-based ANNS, for GPU on NVIDIA V100 GPU. Overall, two kernels account for most (98%) of the query runtime. The first kernel simply performs the approximate similarity computation using memoization. Profiling the behavior of this kernel with Nvidia Nsight [34] tool and Nvidia Visual Profiler [35] reveals that this kernel fails to effectively utilize the available GPU memory bandwidth as well as its floating-point units. The kernel requires a relatively large amount of shared memory per block (32KB) to store the lookup table, and this requirement limits the number of thread blocks scheduled on SM to three since each SM has 96KB shared memory. The low number of resident thread blocks per SM limits the GPU's ability to hide the memory latency via parallelism, which eventually leads to a reduction in throughput. The second kernel selects top-1000 vectors having the largest similarity out of all vectors whose similarities are computed in the previous kernels. Despite many optimizations in [24], this kernel has limited parallelism (i.e., small grid size), which prevents it from fully utilizing GPU resources. Moreover, this operation only utilizes about 4% of the total FMA units since this kernel is mostly about selecting top-scoring vectors without performing much computation.

CPU Implementation Analysis. We analyze the performance characteristics of Google ScaNN [18] and Facebook Faiss [24] on Intel Skylake-X 8-core CPU. For both cases, the system spends most of its time on a loop that fetches encoded vectors from the main memory and utilizes those data to read the lookup tables, and performs sum reduction of the data read from the lookup tables. Although bottlenecks vary across configurations, we find two major sources for performance degradation. First, the system is often bounded by the memory bandwidth. Specifically, since an encoded vector is only utilized once per query with no reuse, such data do not benefit much from the CPU cache hierarchy and consumes a large amount of system bandwidth. On certain configurations where the memory bandwidth is not a bottleneck, the primary source of performance degradation is its inability to deal with sub-byte data types effectively. Specifically, when $k^*=16$, a single vector element is encoded as a 4-bit integer. Since the CPU does not have support for the 4-bit data type, it continuously utilizes

shift instructions (e.g., VPSRLW) to process 4-bit data. Such use of excess instructions ends up degrading the processor's effective throughput.

III. ANNA Hardware Accelerator

Product-quantization-based similarity search is an attractive idea with many potential benefits, but the existing hardware fails to fully harness its potentials. We identify this algorithm as a great target for specialized hardware due to its unique computation and data access patterns. With the goal of efficiently finding the most similar vectors out of multi-billion database vectors, we present ANNA (Approximate Nearest Neighbor search Accelerator), a specialized hardware accelerator which can substantially improve the throughput and energy efficiency of the approximate similarity search while flexibly supporting various search configurations.

A. Overview of Hardware Operations

ANNA is a stand-alone hardware accelerator that can be configured to perform various product-quantization-based similarity searches. ANNA supports both inner product and L2 distance search cases. Moreover, ANNA can accommodate various search configurations (e.g., metric, k^*, $|C|$, M). Before performing similarity search with ANNA accelerator, a host device first needs to i) configure ANNA by sending a search configuration and ii) place the set of necessary data structures in ANNA main memory (centroids C and encoded vectors) and ANNA's on-chip SRAM (codebook B). Then, the host sends a search command to ANNA with a query or a batch of queries as well as the number of similar vectors (top-k) to search for. ANNA performs a similarity search and finally returns the result to the host device. Figure 3 shows the overview of ANNA hardware accelerator. At a high level, ANNA consists of three main entities named Cluster/Codebook Processing Module (CPM), Encoded Vector Fetch Module (EFM), and Similarity Computation Module (SCM). Each module contains computation units, memory readers, and SRAM blocks. When provided a query, ANNA performs three steps of PQ-based ANNS outlined in Section II-C.

Search Process (Step 1). The very first step is cluster filtering, which is handled by the Cluster/Codebook Processing Module (CPM). ANNA utilizes a memory reader to read centroid vectors in a streaming manner, and these read centroid vectors and the query are passed to the compute units to compute the exact distance between two vectors. The resulting outcome is then passed to the top-k selection unit, which selects and holds top-$|W|$ most similar centroid vectors.

Search Process (Step 2 & 3)-Inner Product. The second step is lookup table construction. In the case of the inner product, the ith lookup table L_i needs to store $\{q_i B_i[0], ..., q_i B_i[k^* - 1]\}$. In turn, the inner products between each codeword in the codebook and the corresponding sub-vector of the query are computed using the compute units, and the resulting outcome is stored in the lookup table. Once all lookup tables are filled, the third step, similarity computation, is performed by the SCM. The EFM loads the encoded vectors within a cluster selected

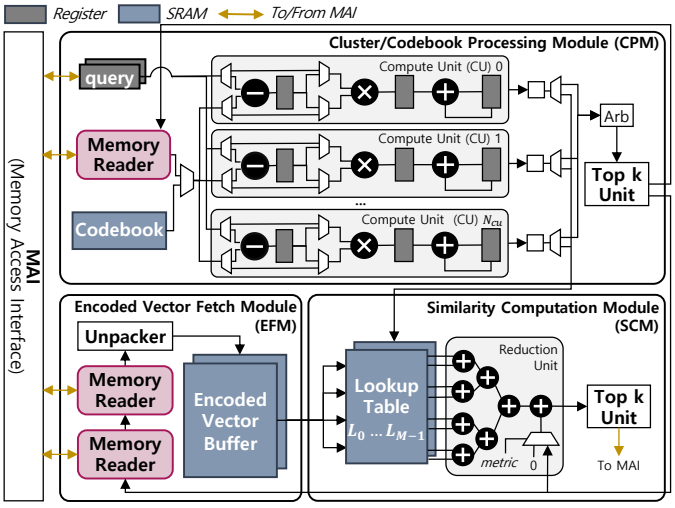

Fig. 3. Illustration of ANNA Hardware Overview.

in Step 1 from main memory, unpacks them with its unpacker hardware, and buffers them in the encoded vector buffer. The SCM reads encoded identifiers (i.e., $e_0(r(x)), ..., e_{M-1}(r(x))$) in the encoded vector buffer, and then use those identifiers as addresses to retrieve specific data in the lookup table. The loaded values are fed to the reduction unit, which computes the reduction outcome. Finally, the term $q \cdot c^{(s)}$ is added to the reduction outcome (see Section II-C) to obtain the final similarity between the query and an encoded vector. This similarity value, as well as the encoded vector's ID are passed to the top-k selection unit in the SCM. This process is repeated for all encoded vectors in the selected clusters, and the final top-k selected vectors in the SCM top-k selection unit are stored in memory as the search outcome.

Search Process (Step 2 & 3)-L2 Distance. For L2 distance search, the process is slightly different. Unlike in the inner product-based similarity search, the lookup table needs to be reconstructed for each selected cluster. The centroid reader first loads the selected centroid once again from memory and computes the difference between the query vector and this selected centroid vector (i.e., $q - c^{(s)}$) using the compute units. Then, the contents for the lookup table is computed, again using the compute units. At this point, the SCM is utilized to compute the similarity between the encoded vector in the currently selected cluster and the query using the lookup table. The lookup table construction and the similarity computation are repeated for selected clusters, and finally, the top-k selection unit stores the top-k selected vectors in the memory as in the inner product-based similarity search case.

Since L2 distance search requires a lookup table to be constructed for each selected cluster, such process can potentially account for a substantial amount of runtime. To improve the performance, ANNA overlaps lookup table construction on the CPM and similarity computation on the SCM through double buffering. Specifically, ANNA maintains two copies of lookup tables and lets the CPM fill the lookup table for the $(i+1)$th most similar cluster while the SCM computes the approximate similarity between the query and the database vectors for the

ith similar cluster. Once both are complete, the SCM now operates for the $(i+1)$th most similar cluster using the lookup table that the CPM just filled, and the CPM fills the lookup table that the SCM stopped using. This way, the lookup table construction time and similarity computation time can overlap and effectively reduce the total query processing time.

B. Design of Hardware Modules

(1) Cluster/Codebook Processing Module (CPM)

Figure 4 illustrates the hardware design of CPM. This module utilizes N_{cu} compute units to perform various computations. Overall, this module is utilized for three purposes: ❶ to compute the similarity between query q and each centroid in C during the cluster filtering step, ❷ to compute the residual vector $r(x) = q - c^{(s)}$ for the L2 distance search, and ❸ to compute the similarity between each codeword and the query (inner product) or residual (L2 distance).

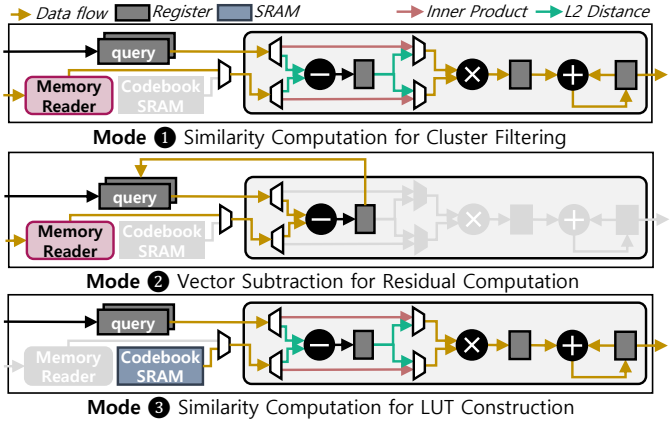

Fig. 4. Illustration of ANNA Cluster/Codebook Processing Module.

Mode ❶ (Similarity Computation for Cluster Filtering). For the first case, the goal is to compute $s(q, c)$ between query q and every c in C. Every cycle, an element of the query vector is broadcasted into N_{cu} compute units and used as an operand. At the same time, an element of N_{cu} different centroids is supplied to each compute unit every cycle. Then, depending on the metric, the partial similarity (i.e., $q[i]c[i]$ or $(q[i]-c[i])^2$) is accumulated at the last register of the compute units. Assuming D-dimensional query vector and centroids vector, this module requires D cycles to compute the similarity between the query vector and N_{cu} centroids. To complete the cluster filtering step, this module overall requires $D|C|/N_{cu}$ cycles.

Mode ❷ (Vector Subtraction for Residual Computation on L2 Distance Search). For the second case, this module performs a vector subtraction $(q - c^{(s)})$. Every cycle, different elements of each query as well as the selected centroid vector (i.e., $c^{(s)}$) are sent to different compute units. For example, $q[0]$ and $c^{(s)}[0]$ are sent to the first compute unit, and $q[N_{cu}-1]$ and $c^{(s)}[N_{cu}-1]$ are sent to the last compute unit. The resulting outcome is then stored in the residual register. Since each cycle processes N_{cu} elements of D-dimensional vector, this module processes residual computation in D/N_{cu} cycles.

Mode ❸ (Similarity Computation for LUT Construction). For the last case, the module is responsible for computing the values that will be stored in the lookup table. In this case, a single compute unit (CU) is responsible for computing all values that will be stored in one lookup table L_i. Specifically, in each cycle, an element of the different query sub-vectors (e.g., $q_0, ..., q_{M-1}$) is supplied to each compute unit. Also, a codeword from each codebook is supplied to different compute units. Each compute unit then utilizes these two values to compute the value that is stored in the lookup table. Over the first D/M cycles, the ith compute unit computes the value for $L_i[0]$ by utilizing sub-vector q_i and $B_i[0]$. This process is repeated for k^* times to fill up the lookup tables each having k^* entries. A total of $D/M \cdot k^*$ cycles are necessary to fill up N_{cu} lookup tables. Since a total of M lookup table needs to be filled up, the total number of cycles necessary to fill the whole lookup table is $D/M \cdot k^* \cdot M/N_{cu} = Dk^*/N_{cu}$ cycles.

(2) Encoded Vector Fetch Module (EFM)

This module is in charge of fetching encoded vectors of a selected cluster from the main memory and then buffering the fetched data in the on-chip buffer (named encoded vector buffer). Specifically, this module first receives the selected cluster IDs from the top-k unit. Then, its memory reader reads the cluster metadata (i.e., start address for the data within the cluster and the size of the cluster) from the main memory. The following memory reader utilizes the start address to fetch the encoded identifiers of the cluster from the main memory. The read data is passed to the unpacker hardware, which utilizes hardware shifters to unpack the packed data, and stores them in the encoded vector buffer. To overlap the data fetch and the later similarity computation, this module keeps two encoded vector buffers. When the SCM is utilizing one encoded vector buffer, this module fetches the data for the next cluster on the other data buffer. In some cases, a cluster's encoded vectors may be larger than the encoded vector buffer size. In that case, a contiguous portion of the cluster's data is first fetched, and the next contiguous portion of the cluster's data is fetched on the other buffer while the current buffer is utilized.

(3) Similarity Computation Module (SCM)

The main role of this module is to perform the approximate similarity computation, which is essentially a sum reduction of data retrieved from multiple lookup tables (See Step 3 in Figure 2). For this purpose, this module maintains lookup tables and an adder tree with $N_u - 1$ adders so that it can reduce N_u values every cycle. This module retrieves the set of N_u identifiers from the encoded vector buffer. Each of these identifiers is used as an address for a lookup table, and the total of N_u data are read from multiple lookup tables. These read data are passed to the reduction unit, which is a pipelined adder tree. In the case of the inner product similarity search, the reduction result is added with the $q \cdot c^{(s)}$ supplied from the top-k unit by an extra adder. This module can perform similarity computation with a single database vector per M/N_u cycles. For example, when $M = 128$ and $N_u = 64$, the module will take two cycles to process a single entry with pipelining.

(4) Top-k Selection Unit

This unit tracks k largest data ($k = 1000$ in our configuration) that this unit has taken as inputs during its operation. Essentially, this is a hardware priority queue. Every cycle, this unit takes a similarity score for the specific vector as an input. If the provided input is larger than the minimum of the currently tracked ones, the input is added to the structure, and the already tracked entry with the smallest score is discarded. Otherwise, the input is simply discarded and the structure remains intact. We implement P-heap hardware priority queue [7], which utilizes a binary-heap-like structure for high-throughput design. The unit consists of several SRAM buffers which can store k data and a set of comparators. This unit is designed to process a single input every cycle. It can also initialize its contents from the main memory or flush its contents to the main memory. This unit also maintains two copies of buffers so that a set of buffers can be utilized for top-k processing, while the other set of buffers can simultaneously flush/initialize its contents to/from the main memory.

(5) Memory Module

Memory Access Interface (MAI). MAI takes read requests from memory readers and issues memory read requests to the memory controller. When issuing a memory request, it reserves one of its 64B buffers and records the requested reader ID there. Then, MAI adds an entry to its associative table structure, which maintains the list of outstanding read addresses (as a key) and destination buffer ID (as a value). When the memory request returns from the main memory, it finds the matching address from the associative table and stores the read value to the destination buffer. Every cycle, the MAI utilizes an arbiter to forward one of the values in the MAI destination buffer to the memory reader that issued this memory request. For memory write requests, MAI buffers the write data until the write completes in the main memory. In general, this is quite similar to the MSHR in CPUs.

Memory Readers. Memory readers are in charge of issuing memory requests through the memory access interface (MAI) and buffering the received inputs. The reader is configured with the start address and the amount of data that it needs to fetch from the start address. When configured, this module prefetches the data from the start address as long as the MAI can accept its read requests. Once the MAI returns the data for issued read requests, the module buffers this 64B granularity read data and forwards the portion or all of the data to the next module at the requested granularity. There are three memory readers in ANNA. Memory readers in CPM are used to read centroid vectors, and memory readers in EFM are used to fetch cluster metadata and cluster's encoded vectors.

SRAM. ANNA has three SRAM structures. First, the Codebook SRAM is sized so that it can buffer the whole codebook which is $2k^*D$ bytes (e.g., 64KB in our evaluation). This SRAM is structured to read up to $2N_{cu}$ consecutive bytes (e.g., 64B) data every cycle. Next, the lookup table SRAMs have a total capacity of $2k^*M$ (e.g., 32KB in our evaluation) bytes for a single SCM. It can handle up to N_u (e.g., 64) lookups in parallel, where each lookup returns one of k^* entries. As explained above, we maintain two copies of the lookup tables to overlap the filling of the lookup table and the similarity computation. Finally, the encoded vector buffer is used to buffer the encoded vectors in the single cluster. As with the lookup tables, two copies of the encoded vector buffer are maintained to overlap data fetch and the similarity computation. The size of this SRAM structure is a design parameter (e.g., 1MB in our evaluation). The encoded vector buffer is structured to supply N_u data per cycle.

IV. ANNA Memory Traffic Optimization

By design, one can easily adjust ANNA's computation capability by adjusting design parameters such as N_{cu} (the number of compute units in CPM) or N_u (the number of entries which can be sum-reduced in a cycle). Alternatively, it is possible to utilize multiple copies of ANNA modules to improve ANNA's computation throughput. In such cases, the system's performance bottleneck eventually shifts to the memory bandwidth. At that point, the only way to further improve the throughput is to reduce the traffic between ANNA and the main memory. In this section, we present an optimization scheme that can significantly reduce the memory traffic consumption of ANNA on batched similarity search scenarios.

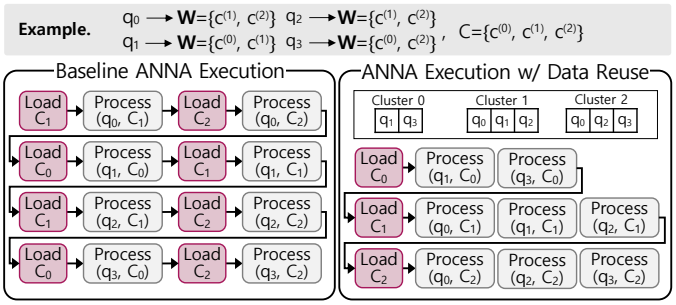

Fig. 5. Illustration of ANNA Memory Traffic Optimization.

The key idea is to reuse the encoded vectors for a specific cluster across multiple queries. Figure 5 illustrates how this optimization can substantially reduce the memory traffic. In figure, we define C_i as a set of encoded vectors in cluster i. The left of the figure shows conventional execution, where a single query is processed at a time. In such case, a query scans the encoded vectors in clusters belonging to W, whose centroids are closest to the query. This same process is repeated for the following queries. On the other hand, the right side of the figure shows the optimized execution. In this case, the set of $|W|$ relevant clusters for all queries are identified first. Based on that information, queries visiting a specific cluster are identified for all clusters. Then, each cluster is processed in series. Specifically, a specific cluster's encoded vectors are loaded and buffered on-chip, and queries visiting the cluster process the cluster using the buffered data. Once all queries visiting the cluster finish processing the cluster, the next cluster is processed for a different set of queries visiting the next cluster. Assuming each query visits $|W|$ clusters out of $|C|$ clusters,

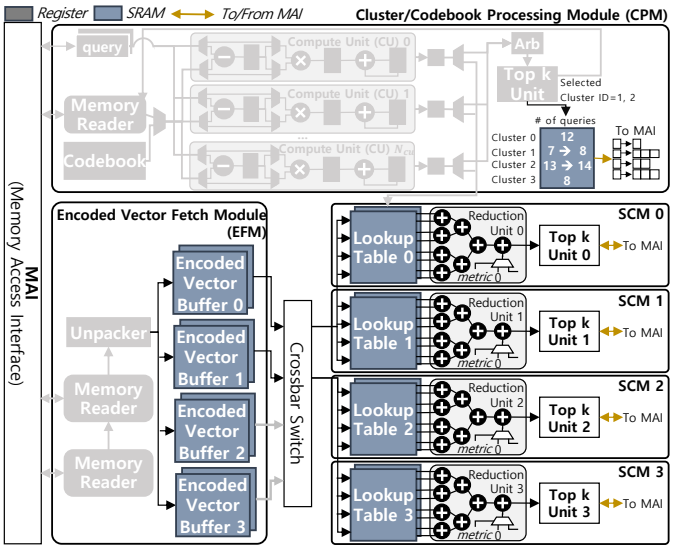

Fig. 6. Diagram of ANNA Hardware with Memory Traffic Optimization.

and the system processes B queries at a time, the original execution scheme requires loading $B \cdot |W|$ cluster's encoded vectors. On the other hand, the optimized execution scheme requires loading mere $|C|$ clusters' encoded vectors even in the worst case. When $B = 1000, |C| = 10000, |W| = 128$, this technique leads to a 12.8× traffic reduction.

A. Hardware Extensions

Recording Queries Visiting a Specific Cluster. In the baseline ANNA, which processes a query at a time, the top-k module in CPM keeps the cluster IDs in W and then passes the cluster ID of those selected clusters to EFM so that EFM can fetch the encoded vectors for the selected cluster. On the other hand, to implement the optimization represented in Figure 5, ANNA first performs cluster filtering step for all queries and stores the list of queries visiting each cluster in the memory. Specifically, for this purpose, ANNA maintains an array of arrays in the main memory, where ith array keeps the IDs of the queries visiting cluster i. Also, ANNA utilizes an on-chip SRAM whose ith row stores the base address (8B) for the ith array in the main memory and the number of queries visiting ith cluster (3B) as shown in Figure 6.

Once top-$|W|$ clusters for a query are identified, one of the selected cluster IDs is retrieved from the top-k module, and this ID is used to access the mentioned SRAM structure. The base address for the array associated with the selected cluster is obtained, and the number of queries in this cluster is used to compute the exact address where the query ID needs to be written. Next, the write request is passed to MAI, which performs a masked write to the main memory. This process is repeated for all $|W|$ selected clusters, and the CPM performs cluster filtering for the remaining queries.

Storing and Retrieving Intermediate Top-*k* Vectors. The baseline ANNA utilizes a top-k module in SCM to track the top-k most similar vectors to a particular query. This was possible since the baseline ANNA handles one query at a time. However, with the optimization, the same top-k module is now used by multiple queries processing the same cluster. As a result, once a query processes a cluster's data, its intermediate top-k results need to be stored in memory. In addition, before a query processes a cluster's data, the intermediate top-k results need to be loaded from the memory so that the top-k module correctly identifies whether the vectors in the currently processed cluster belong to top-k or not.

Improving Throughput with Multiple SCMs. Reduction in memory traffic means that more computations can happen without being bounded by the memory bandwidth. To improve the throughput, ANNA utilizes multiple SCMs (N_{SCM}) in parallel. Figure 6 shows the hardware extensions to support multiple SCMs. Encoded vector buffers in EFM are structured accordingly to supply data from EFM to SCMs at a higher rate. Also, a configurable crossbar switch is added to connect multiple encoded vector buffers with multiple SCMs. There are two different ways to utilize multiple SCMs at once. First, SCMs can be used to process multiple queries processing the same cluster in parallel (*inter*-query parallelism). In this case, the EFM simply broadcasts the same encoded vector to all SCMs so that each SCM having different, query-specific LUT content can process the data for each query. Alternatively, it is possible to allocate multiple SCMs to a single query (*intra*-query parallelism). For this purpose, each encoded vector buffer, storing a portion of the cluster's encoded vectors, passes data to different SCMs. In this case, each SCM then processes a subset of the cluster's encoded vectors with its own top-k modules. Once all clusters are processed, each SCM's top-k results are merged using top-k modules.

In ANNA, a user can specify the number of SCMs that a single query utilizes during the execution. In general, it is slightly better to utilize SCMs across multiple queries since a single query utilizing multiple SCMs tends to increase the traffic for saving/restoring intermediate top-k results to the main memory. On the other hand, when a single cluster is processed by a very small number of queries, it is better to utilize multiple SCMs for a single query so that ANNA hardware fully utilizes its available SCMs. One can easily compute the average number of queries for each cluster to find the required number of SCMs per query. For example, when $B = 1000$, $|C| = 10000$, and $|W| = 40$, 4 $(= B|W|/|C|)$ queries are expected for each cluster. Thus, for ANNA with 16 SCMs, we allocate four SCMs to a single query.

B. ANNA Execution with Optimization

Figure 7 visualizes the timeline for each compute module in ANNA (i.e., CPM and multiple SCMs) as well as its memory system in a steady-state (i.e., Step 2 and 3 of the search process outlined in Section III-A). When the SCMs perform similarity computation for the ith cluster (takes $|C_i|M/N_u$ cycles), multiple things happen in parallel. First, the CPM works on lookup table construction for the $(i+1)$th cluster in the case of the L2 distance similarity search, which requires the CPM to compute the lookup table for every cluster. Constructing a lookup table takes k^*D/N_{cu} cycles and when there exist N_{scm} SCM modules each running a different query, the CPM needs

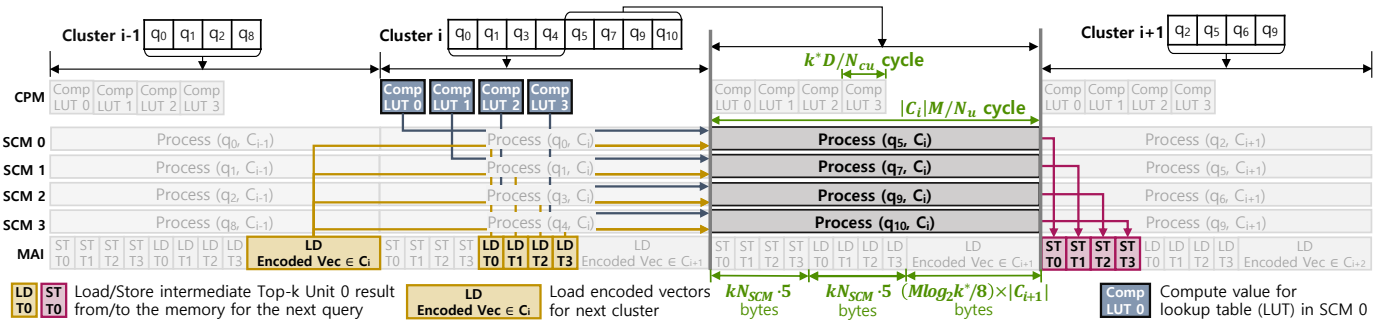

Fig. 7. Visualization of ANNA Execution Timeline with Optimization.

to construct N_{scm} separate lookup tables in $N_{scm}Dk^*/N_{cu}$ cycles. Meanwhile, on the memory side, each top-k unit in SCMs store the intermediate top-k results from its previous operations to the main memory and load the intermediate top-k results from the main memory for its upcoming operations. The memory traffic is $2kN_{SCM} \cdot 5B$, since each top-k unit in SCM loads/stores k entries which are 5B each (3B for vector ID, 2B for similarity score). Also, the EFM prefetches the encoded vectors for the cluster, which are $|C_{i+1}|$ encoded vectors each represented with M identifiers that are $\log_2 k^*$ bits each. Overall, the computation time for the specified time period in the figure is $max(N_{scm}k^*D/N_{cu}, |C_i|M/N_u)$ cycles, and $10kN_{SCM} + (M \log_2 k^*/8) \cdot |C_{i+1}|$ bytes needs to be fetched during this time. One should carefully set ANNA design parameters (e.g., N_u, N_{cu}, N_{scm}) so that the system is not heavily bottlenecked by computations or memory accesses.

V. Evaluation

A. Methodology

Dataset. We evaluate several representative similarity search datasets: SIFT1M [23] (N=1M, D=128, L2 Distance), Deep1M [4] (N=1M, D=96, L2 Distance), GloVe [37] (N=1M, D=100, Inner Product), SIFT1B [23] (N=1B, D=128, L2 Distance), Deep1B [4] (N=1B, D=96, L2 Distance), and TTI1B [45] (N=1B, D=128, Inner Product). In general, each vector within these datasets represents an embedding of an image feature or a word.

Software ANNS Implementations. We utilize two representative open-source implementations of PQ-based similarity search algorithms: Facebook Faiss [24] and Google ScaNN [18]. Both algorithms utilize different objective functions to train codebook and thus generates different codebooks as well as the encoding for each database vector. Faiss has both CPU and GPU implementations, and ScaNN only has CPU implementation. We train each dataset for each algorithm across varying configurations and obtain trained models where each is a set of i) a list of centroids, ii) codebooks, and iii) encoded vectors. We run Faiss or ScaNN similarity search with the trained models on 8-core Intel i7-7820X with 128GB memory (both Faiss and ScaNN) and NVIDIA V100 GPU [36] with 32GB memory (Faiss) to measure their performance, energy consumption (measured with Intel RAPL & `nvprof`), and model recall $X@Y$ (i.e., the portion of retrieved top X items among submitted Y candidates). Specifically, throughout the evaluation, we utilize the following search configurations: Faiss16 (CPU), Faiss256 (CPU), Faiss256 (GPU), and ScaNN16 (CPU). Here, the number after Faiss or ScaNN (i.e., 16 or 256) represents the k^* value that the configuration utilizes. ScaNN does not support $k^* = 256$ configuration, and FaissGPU does not support $k^* = 16$ configuration since their implementations are tightly coupled with the specific k^*. Throughout the experiment, we use $|C| = 10000$ and $|C| = 250$ for billion-scale and million-scale datasets, respectively, and M varies across experiments.

ANNA Evaluation Methodology. For performance comparison of ANNA with the software implementations, we implement a custom cycle-level simulator for ANNA. We evaluate four ANNA configurations, each utilizing the trained model from the corresponding software implementations. We compare the throughput/latency of queries at a given search recall, which is our quality metric. Each ANNA configuration is assumed to be paired with the memory system providing the 64GB/s bandwidth, which is identical to the evaluated CPU-based system's memory bandwidth. For the area and energy evaluation, we implement the ANNA accelerator with Chisel HDL [40], and perform functional verification with Synopsys VCS. Then, we synthesize RTL implementations of the ANNA with the TSMC 40nm GP standard cell library with 1GHz frequency to obtain its area and power consumption. Then, we post-process power consumption from each component to obtain the system energy consumption. ANNA design parameter is set as follows: $N_{cu} = 96$, $N_{SCM} = 16$, and $N_u = 64$. ANNA can support both $k^* = 16$ and $k^* = 256$.

B. Performance Evaluation

Throughout Improvements. Figure 8 shows the throughput comparison of ANNA and several software ANNS implementations across different configurations and workloads. Each configuration is represented as a solid or a dotted line because the model recall changes across user-specified search parameter W, the number of clusters inspected. The higher W means higher recall at the expense of lower throughput. The upper six plots in Figure 8 show the results for 4:1 compression ratio, meaning that the main memory needs to hold one-fourth of the original data size, which is $0.5N$ bytes. For $k^* = 256$ configurations which represent a single vector element as $log_2 k^* = 8$ bits, $M = D/2$ is utilized to compress the data by 4×. On the other hand, for $k^* = 16$ configurations, which represent a vector element as 4 bits, $M = D$ is utilized to

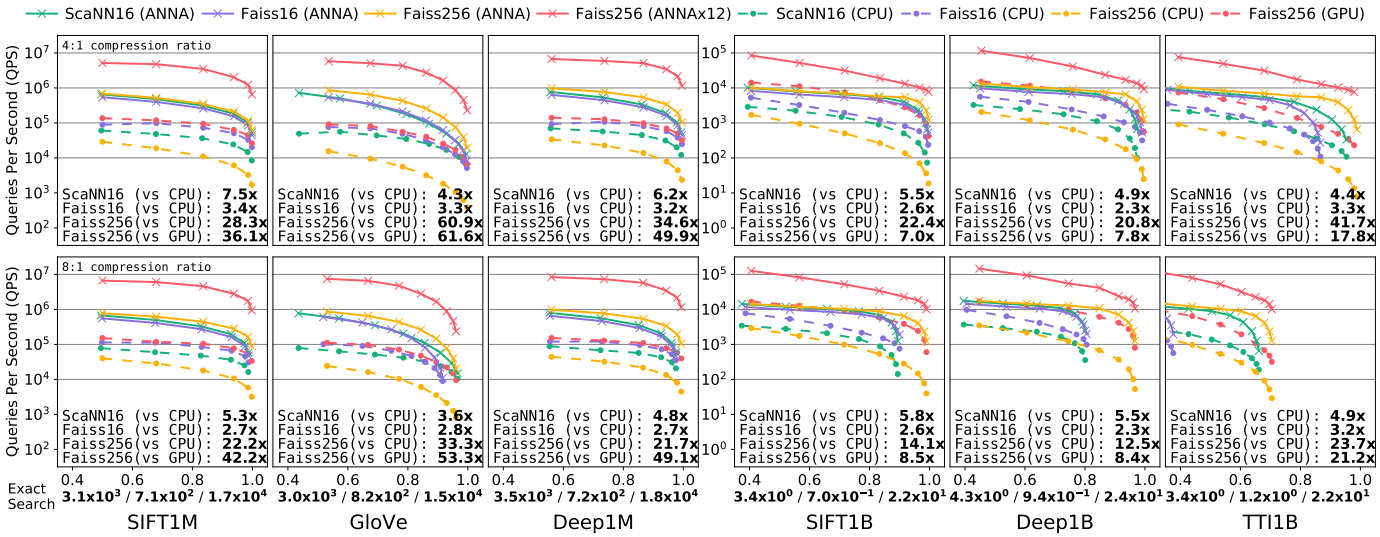

Fig. 8. Throughput Comparison of ANNA and Various Software ANNS Implementations on CPU/GPU. X-axis is recall 100@1000 (portion of true top 100 items included in 1000 candidates obtained by ANNS algorithms). Y-axis represents the queries processed per second in a **log scale**. At the bottom of each plot, we compare each ANNA configuration with its corresponding software implementation and report the geomean speedup. Three numbers below plots represent the QPS of exhaustive, exact nearest neighbor search on ScaNN (CPU), Faiss (CPU), and Faiss (GPU), respectively.

compress the data by 4×. Similarly, the lower six plots in Figure 8 show the results for the 8:1 compression ratio. For that case, the smaller M (e.g., $D/4$) is utilized to further compress the data. In general, a higher compression rate trades off model recall for less memory usage.

The figure shows that ANNA achieves substantial speedup over CPU or GPU implementations across varying recalls. Among various CPU implementations, we find that Faiss256 (CPU) achieves lower performance than other CPU implementations. This is because Faiss16 (CPU) and ScaNN16 (CPU) utilize low-level code optimizations to pin 16-entry lookup tables on vector registers. On the other hand, when $k^* = 256$, 256-entry lookup tables do not fit on vector registers, and thus the Faiss256 (CPU) achieves much lower speedup. Between Faiss16 (CPU) and ScaNN16 (CPU), Faiss16 (CPU) achieves better performance since Faiss16 (CPU) implementation processes queries in a way that is similar to ANNA memory traffic optimization represented in Figure 5. ANNA implementation of Faiss16 outperforms Faiss16 (CPU) while consuming a much smaller area and energy (presented in the following section). The major drawback of Faiss16 or ScaNN16 configuration is that the use of $k^* = 16$ sometimes fails to achieve high recall on challenging scenarios. For example, on Deep1B dataset (8:1 compression ratio), all $k^* = 16$ configurations cannot achieve recall beyond 0.9. Moreover, although not presented in the figure, those configurations fail to achieve 0.5 recall on 16:1 compression ratio scenarios for the same dataset. The same issue is observed in TTI1B dataset (8:1 compression ratio), but only with Faiss16 in this case. On the other hand, Faiss256 (CPU) can achieve substantially better maximum recall, but is much slower. Unlike those software implementations, Faiss256 (ANNA) can provide a very high recall and throughput at the same time. Finally, Faiss256 (GPU) shows very promising performance in some cases. However, this is because the V100 GPU has 900 GB/s memory bandwidth. For the fair comparison, we compare Faiss256 (GPU) with the Faiss 256 (ANNA ×12), which utilizes twelve ANNA instances, each paired with a 75GB/s memory system. It is clear that ANNA ×12 achieves a substantially larger throughput than the V100 GPU.

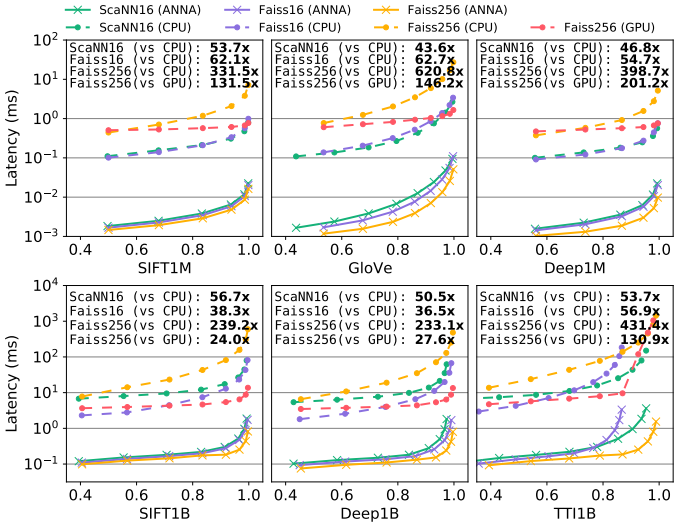

Fig. 9. Latency Comparison of ANNA and Various Software ANNS Implementations. Y-axis represents the latency for a single query represented in a **log scale**. Lower is better. (4:1 compression ratio).

Impact of ANNA Memory Traffic Optimization. We compare the throughput of ANNA without optimization and ANNA with memory traffic optimization (Section IV). On average (across multiple datasets), ANNA with the memory traffic optimization achieves 5.1×/5.0×/6.9× throughput compared to ANNA without the optimization for ScaNN16/Faiss16/Faiss256 configurations on 4:1 compression rate cases, respectively. Similarly, the extra speedup from optimization is 3.9×/3.9×/4.6× on cases with the 8:1 compression ratio. The speedup is greater on the 4:1 compression ratio cases since the performance in

TABLE I
AREA AND (PEAK) POWER OF ANNA.

Module Name	Area (mm^2)	Peak Pwr(W)
Codebook/Cluster Processing Module	1.17	0.391
Encoded Vector Fetch Module	2.87	1.065
Similarity Computation Module (16×)	13.30	3.795
Memory Access Interface (MAI)	0.17	0.147
ANNA Accelerator	17.51	5.398
ANNA Accelerators (12×)	210.12	64.776

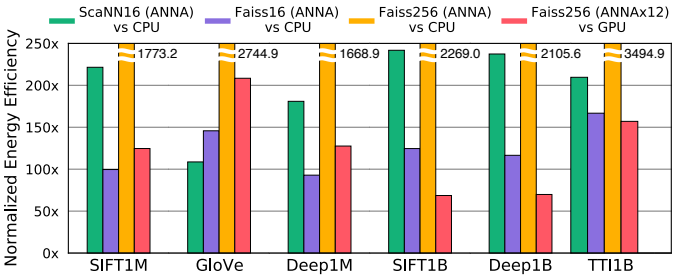

Fig. 10. Normalized Energy Efficiency of ANNA over Corresponding CPU/GPU Implementations. (4:1 compression ratio, $W = 32$ configuration).

those scenarios is more memory bandwidth-bound.

Latency Improvements. Figure 9 compares the average latency of processing a single query between ANNA and the other workloads on configurations whose compression ratio is 4:1. The figure shows that ANNA enables very low-latency processing of similarity search queries. For example, ANNA achieves high recall (0.9+) at sub-ms latency in billion-scale datasets. On the other hand, fastest CPU/GPU implementations achieve such recall around 11/5ms latency. Overall, ANNA achieves over 24× latency improvements across all configurations. This shows that ANNA utilizes parallelism within a single query more effectively than others.

C. Area/Energy Evaluation

Area. Table I reports the ANNA accelerator area usage. We find that a large portion of ANNA modules' area results from their SRAM structures. A single ANNA accelerator requires a 17.51mm^2 area at 40nm technology. In comparison, the evaluated CPU die size is 325.4mm^2 at 14nm technology [42] (effectively 151× larger), and the GPU die size is 815mm^2 at 12nm technology [36] (effectively 517× larger).

Power and Energy Consumption. Table I shows that a single ANNA accelerator consumes about 5.398W at its peak. In practice, not all modules are fully utilized at the same time, and thus the actual power usage (2-3W) is lower than the peak. We also measure the GPU and CPU power consumption. Their power vary across datasets and configurations, but on average, the CPU utilizes 116W/139W power (ScaNN/Faiss), and the GPU utilizes 151.8W power during their operations. Figure 10 presents the energy efficiency comparison between ANNA accelerator and corresponding CPU/GPU implementations on a specific configuration. Combining the substantially lower power consumption and runtime reduction, ANNA achieves orders of magnitudes energy efficiency improvements (97×+ across all configurations) over both CPU and GPU. The main sources of energy efficiency are i) the use of specialized computation modules and memory structures, ii) the use of dataflow pipeline, and iii) efficient data reuse.

VI. RELATED WORK

ANNS Accelerators. Abdelhadi et al. [1] design specialized FPGA implementation for PQ-based ANNS, exploiting the large on-chip memory on FPGA. This design achieves high throughput on a million-scale dataset whose compressed vectors fit in on-FPGA buffer, but is not really applicable for billion-scale datasets. Zhang et al. [46] also present FPGA-accelerated ANNS, which achieves 50K QPS for 0.94 recall (1@10) on SIFT1M dataset. For a similar recall on the same dataset, ours achieves around 256K QPS with a single ANNA, which uses multiple orders of magnitude lower area, energy, and on-chip SRAMs. Both FPGA implementations lack ANNA's data reuse optimization presented in Section IV and thus cannot efficiently utilize the limited off-chip memory bandwidth. Moreover, they use their own ANNS mechanism tailored for their hardware. Such specific mechanisms' accuracy is not as rigorously verified as widely known software implementations like ScaNN and Faiss. On the other hand, ANNA presents a hardware that is compatible with both ScaNN and Faiss while achieving much higher energy efficiency and improvement in speedup. Gemini APU [17] is proprietary hardware that utilizes LSH-based ANNS. The white paper states that it achieves 800 QPS for 0.92 recall (1@160) on Deep1B dataset. On the other hand, ANNA achieves over 4096 QPS for a similar recall on the same dataset. Tigris [43] is an academic work on accelerating KD-tree ANNS. The presented implementation is very specific to the point cloud registration task, which involves NNS for 3-dimensional data. Neither the presented hardware nor the KD-tree ANNS is effective for similarity search on billion-scale embeddings where each vector has over 100-dimensions.

ANNS Techniques. ANNS on software is a well-studied topic with a wide range of related works. Graph-based ANNS [13, 19, 21, 26] exploit nearest neighbor graph, a graph structure whose each node is a vector that is connected to its nearby nodes. Although those algorithms achieve high performance on million-scale datasets [6], they are impractical for billion-scale searches as they require a large graph to be resident in memory. There exist other ANNS techniques such as ones that utilize tree-structures [14, 31] or locality-sensitive-hashing [2, 12, 39], but these algorithms exhibit lower performance than alternatives [6]. There exist several variants of the product quantization-based ANNS algorithms which aim to improve the codebook quality. For that purpose, ScaNN [18] utilizes a novel object function, DPQ [25] utilizes deep-learning-based training, and OPQ [16] applies rotation to the original database. ANNA can support all these variations since their computation pattern for the search remains the same. ANNA can also be slightly extended to support other PQ variations such as AQ [3], which utilizes M identifiers each associated with D-dimensional codeword.

VII. Conclusion

Nearest neighbor search is the critical operation for recommender systems and semantic search, two of the most critical applications in Internet services today. Such a high demand, combined with the fact that this operation has a relatively static computation and data access pattern, makes this operation an ideal target for the specialized architecture. Our work presents an effective specialized architecture named ANNA and demonstrates that the deployment of specialized architecture can significantly improve the performance and energy efficiency of nearest neighbor search. We hope our work provides the ground for the promising research direction: designing the specialized architecture for ANNS.

REFERENCES

[1] A. M. Abdelhadi, C.-S. Bouganis, and G. A. Constantinides, "Accelerated approximate nearest neighbors search through hierarchical product quantization," in *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 90–98.

[2] "FALCONN - FAst Lookups of Cosine and Other Nearest Neighbors," https://github.com/FALCONN-LIB/FALCONN.

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

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

[5] B. Barz and J. Denzler, "Hierarchy-based image embeddings for semantic image retrieval," in *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2019, pp. 638–647.

[6] "Benchmarking nearest neighbors," https://github.com/erikbern/ann-benchmarks.

[7] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, vol. 2. IEEE, 2000, pp. 538–547.

[8] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Céspedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.

[9] Q. Chen, H. Zhao, W. Li, P. Huang, and W. Ou, "Behavior sequence transformer for e-commerce recommendation in alibaba," in *Proceedings of the International Workshop on Deep Learning Practice for High-Dimensional Sparse Data with KDD (DLP-KDD)*, 2019, pp. 1–4.

[10] "Sptag: A library for fast approximate nearest neighbor search," https://github.com/microsoft/SPTAG, 2018.

[11] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2016, pp. 191–198.

[12] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Symposium on Computational Geometry (SoCG)*, 2004, pp. 253–262.

[13] "KGraph: A Library for Approximate Nearest Neighbor Search," https://github.com/aaalgo/kgraph.

[14] "Annoy," https://github.com/spotify/annoy.

[15] "Faiss," https://github.com/facebookresearch/faiss, Facebook AI Research.

[16] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 36, no. 4, pp. 744–755, 2013.

[17] "Gemini APU: Enabling High Performance Billion-Scale Similarity Search," https://www.gsitechnology.com/sites/default/files/Whitepapers/GSIT-APU-Enabling-High-Performance-Billion-Scale-Similarity-Search-WP.pdf, GSI Technology.

[18] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2020, pp. 3887–3896.

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

[20] V. Hyvönen, T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos, "Fast k-nn search," *arXiv preprint arXiv:1509.06957*, 2015.

[21] M. Iwasaki, "Pruned bi-directed k-nearest neighbor graph for proximity search," in *Proceedings of the International Conference on Similarity Search and Applications (SISAP)*. Springer, 2016, pp. 20–33.

[22] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 33, no. 1, pp. 117–128, 2010.

[23] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2011, pp. 861–864.

[24] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data (TBD)*, 2019.

[25] B. Klein and L. Wolf, "End-to-end supervised product quantization for image search and retrieval," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 5041–5050.

[26] 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 (TPAMI)*, vol. 42, no. 4, pp. 824–836, 2018.

[27] Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh, "A survey of product quantization," *ITE Transactions on Media Technology and Applications*, vol. 6, no. 1, pp. 2–10, 2018.

[28] "Bing Vector Search," https://www.microsoft.com/en-us/ai/ai-lab-vector-search, Microsoft.

[29] A. Miech, J.-B. Alayrac, L. Smaira, I. Laptev, J. Sivic, and A. Zisserman, "End-to-end learning of visual representations from uncurated instructional videos," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 9879–9889.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[31] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 36, no. 11, pp. 2227–2240, 2014.

[32] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[33] "Nvidia Merlin," https://developer.nvidia.com/nvidia-merlin, NVIDIA.

[34] "NVIDIA Nsight Systems," https://developer.nvidia.com/nsight-systems, NVIDIA.

[35] "NVIDIA Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler, NVIDIA.

[36] "NVIDIA V100 Tensor Core GPU," https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf, NVIDIA, 2020.

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

[38] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2014, pp. 806–813.

[39] A. Shrivastava and P. Li, "Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips)," *arXiv preprint arXiv:1405.5869*, 2014.

[40] "Chisel," https://chisel.eecs.berkeley.edu, University of California, Berkeley.

[41] R. Wang, R. Shivanna, D. Cheng, S. Jain, D. Lin, L. Hong, and E. Chi, "Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems," in *Proceedings of the International World Wide Web Conference (WWW)*, 2021, pp. 1785–1797.

[42] "LCC SoC - Skylake (server) Microarchitectures Intel," https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#LCC_SoC, WikiChip.

[43] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 629–642.

[44] "Neighborhood Graph and Tree for Indexing High-dimensional Data," https://github.com/yahoojapan/NGT, Yahoo! Japan R&D.

[45] "Benchmarks for Billion-Scale Similarity Search," https://research.yandex.com/datasets/biganns, Yandex Research.

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

[47] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1033–1044.