Accelerated Approximate Nearest Neighbors Search Through Hierarchical Product Quantization

Ameer M.S. Abdelhadi, Christos-Savvas Bouganis, and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2AZ, United Kingdom
{a.abdelhadi, christos-savvas.bouganis, g.constantinides}@imperial.ac.uk

***Abstract*—A fundamental recurring task in many machine learning applications is the search for the Nearest Neighbor in high dimensional metric spaces. Towards answering queries in large scale problems, state-of-the-art methods employ Approximate Nearest Neighbors (ANN) search, a search that returns the nearest neighbor with high probability, as well as techniques that compress the dataset. Product-Quantization (PQ) based ANN search methods have demonstrated state-of-the-art performance in several problems, including classification, regression and information retrieval. The dataset is encoded into a Cartesian product of multiple low-dimensional codebooks, enabling faster search and higher compression. Being intrinsically parallel, PQ-based ANN search approaches are amendable for hardware acceleration. This paper proposes a novel Hierarchical PQ (HPQ) based ANN search method as well as an FPGA-tailored architecture for its implementation that outperforms current state of the art systems. HPQ gradually refines the search space, reducing the number of data compares and enabling a pipelined search. The mapping of the architecture on a Stratix 10 FPGA device demonstrates over $\times$250 speedups over current state-of-the-art systems, opening the space for addressing larger datasets and/or improving the query times of current systems.**

***Index Terms*—Approximate search, similarity search, nearest neighbor search, online indexing, high-dimensional indexing, product quantization, vector quantization, artificial intelligence**

I. Introduction

Searching a high dimensional space for the Nearest Neighbors (NN) of a query point is a fundamental recurring task in many machine learning applications. Computer vision (specifically classification [1] and recognition [2]), information retrieval [3], robotics [4], and other machine learning tasks, all benefit from NN similarity search. With the rapid increase of data scale, exact search is becoming more cost-prohibitive in term of query time and memory space. To alleviate computation and memory complexity of this task, an Approximate Nearest Neighbors (ANN) search is used to find highly probable nearest neighbors.

Many CPU-based ANN search techniques have been proposed to accelerate the search process. The most popular are metric trees [5], *e.g.*, KD-trees [6], R-trees [7], K-D-B-tree [8], VP-trees [9] and their variants. A branch-and-bound is applied of these metric trees to perform space partitioning where the search space is gradually subdivided. However, these structures suffer from the *curse of dimensionality* and perform poorly for high-dimensional spaces in which their performance is not better than a brute-force exhaustive search [10].

Hashing [2], [11]–[14] and Product Quantization (PQ) [15] are recent approaches for ANN search which involve encoding the high-dimensional dataset into shorter codes. Encoding the data into compact codes dramatically reduces storage consumption and accelerates the search process since distances are preserved and embedded in the codes. In hashing, Hamming distance is used to approximate the similarity between two vectors, whereas PQ-based techniques partition the original space into a Cartesian product of several low-dimensional subspaces and quantizes each partition into clusters. The data is quantized by selecting the appropriate code from each low-dimensional codebook and a distance is approximated by summing the distances from the query vector to each subspace quantization code.

PQ-based methods demonstrate higher accuracy than hashing-based techniques [15], mainly because of their large effective codebook. Despite having small sub-codebooks, the effective codebook is actually the Cartesian product of these codebooks, thus the size of the effective codebook is the product of all sub-codebook sizes. Furthermore, PQ is computationally efficient since distances between the query vector and each subspace codes can be stored in small tables, and retrieved for total distance computation.

Product quantization and its extensions are leading approaches for high-dimensional large-scale Approximate Nearest Neighbors (ANN) search. Compared to other state-of-the-art methods, the advantages of PQ-based techniques are threefold. (1) The encoding methods and the data structure are simple, which make it a perfect candidate for parallel hardware acceleration and more promising for high-performance search. (2) The distances are computed and compared efficiently by performing codebook lookups. (3) The compression of the dataset into shorter codes enables higher storage efficiency.

The PQ quantization process incurs a distortion in the quantized data which introduces distance errors and impacts the accuracy of the ANN search. To reduce the quantization distortion, several improvements to the original PQ-based ANN search has been proposed. Optimized Product Quantization (OPQ) [16] and Cartesian K-Means (cK-Means) [17] reduce the quantization distortion by applying arbitrary rotation and dimension reordering to enhance the alignment to the data distribution. Locally Optimized Product Quantization (LOPQ) [18] also applies local optimization for the product quantizer by space decomposition to fit the underlying data distribution.

PQ-based methods can efficiently leverage FPGA devices to accelerate queries. FPGA devices contain several thousand distributed on-chip memory blocks, known as Block RAMs (BRAMs), and several thousand Digital Signal Processing (DSP) blocks. In modern devices, these DSP blocks also support Floating-Point (FP) arithmetic. While distributed memories can be utilized to store the codebooks and the coded dataset for fast and parallel retrieval, DSP blocks can be used to compute and compare distances in parallel. Since a query requires multiple access to codebooks and performs multiple distance computations, this combination of distributed BRAMs and DSPs is capable of alleviating the performance bottleneck.

In this paper, we introduce *Hierarchical PQ* (HPQ), a novel PQ-based ANN search method. HPQ benefits from the previously-mentioned PQ attributes and utilizes hierarchical search to refine the search space gradually, and thus avoids time-consuming exhaustive search. Simple Vector Quantization (VQ) is used for refinement of the PQ search space. The closest VQ centroid is found and its attached points (Voronoi cell) are solely used for the next level PQ search. To find the closest VQ centroid, PQ is employed recursively on VQ centroids. This technique allows a gradual partitioning of the search space. The suggested structure is also more suitable for hardware acceleration since the search process can be pipelined through the hierarchy. Our HPQ method is highly amendable for parallel hardware acceleration. We provide a custom accelerator for HPQ and implement it on Intel Stratix 10 and Arria 10 devices. The hardware implementation utilizes the on-chip distributed memory blocks and DSP blocks for massive parallel processing. In addition, we provide a mathematical analysis of memory and hardware consumption, performance, latency, and accuracy, and provide a guideline for tuning the design hyperparameters based on user requirements and hardware limitations.

Notation and abbreviations used for the rest of the paper are listed in Table I. The rest of this paper is organized as follows. Section II reviews related ANN techniques and quantization methods. Section III describes our Hierarchical Product Quantization (HPQ) ANN approach. Section V presents our experimental framework and results and Section VI concludes the paper with future suggestions.

TABLE I: List of Notations and Abbreviations

Architectural hyperparameter:

h	Number of PQ Search levels in the hierarchy.
N_i	Number of entries in level i. For top level $N_{h-1} = N$.
M_i	Number of subspaces in level i.
D	Dimension of search space.
$\tilde{D}_i$	Dimension of each subspace in level i ($\tilde{D}_i = D/M_i$).
$\tilde{k}_i$	PQ Codebook size in level i.
α_i	Tapering factor ($\alpha_i > 1; N_i = \alpha_i N_{i-1}$).
β_i	Voronoi cell centroid occupancy factor ($\beta_i > 1$).
w	Width of data format in bits (*e.g.*, 32 for single-precision FP).
$\tilde{p}, \hat{p}$	Parallelism of phase 1 (update) and phase 2 (compare), respectively.
$d(.,.)$	A distance metric between two argument vectors.

Variables, arrays, and matrices:

x	Query vector $x \in \mathbb{R}^D$.
$u_m(x)$	The m'th subspace of $x \in \mathbb{R}^D$. $u_m(x) = (x_{mD}, ..., x_{(m+1)D-1})$
$\mathcal{Y}^i$	Search space of level i; a dataset of N_i D-dim vectors $\mathcal{Y}^i = \{y^i_0, ..., y^i_{N_i-1}\} \subset \mathbb{R}^D$. For top level $\mathcal{Y}^{h-1} = \mathcal{Y} = \{y_0, ..., y_{N-1}\}$.
$\tilde{\mathcal{Y}}^i$	PQ-Encoded dataset. $\tilde{\mathcal{Y}}^i = (\tilde{y}^i_0, \cdots, \tilde{y}^i_{N-1}) \subset (0, \cdots, \tilde{k}_i - 1)^M$.
C^i_m	Codebook of subspace m in level i. $C^i_m = \{C^i_{m,0}, \cdots, C^i_{m,\tilde{k}_i-1}\}$
$d^i_{m,j}$	Meta-deta of codebook C^i_m. $d^i_{m,j} = d^2(u_m(x), C^i_{m,0})$.
$n^i_{m,j}$	Number of points attached to centroid $C^i_{m,j}$.
$\mathcal{V}^i_{j,k}$	VQ Voronoi cells. Point k of cell j from $\mathcal{Y}^i$ with indices to $\tilde{\mathcal{Y}}^{i+1}$.

Design attributes:

S_i	Storage consumption (in bits) of level i. (S is the total consumption)
$\tilde{F}_i, \hat{F}_i$	DSP consumption of phase 1 (distance update) and phase 2 (distance compare) of level i, respectively. (F is the total DSP consumption)
$\tilde{L}_i, \hat{L}_i$	Latency of phase 1 & 2 of level i, respectively. (L is the total latency)
T_i	Throughput of level i. (T is the overall throughput)
$R@r$	Recalls. Probability that an identified NN is among the actual r NNs.

II. Background and Preliminaries

In this section, we discuss different techniques for ANN search, in particular PQ-based approaches, the basis of our proposed method. In addition, a formal description of the Nearest Neighbor (NN) problem and different quantization methods is provided.

A. Related Work: Compact Codes for ANN Search

Hashing-based methods. To enable nearest neighbor search, hashing-based techniques create several hash functions that map points from the dataset into hash codes. The probability of two points to hash to the same code is proportional to the proximity of these points. In other words, two points have a high probability to be hashed into the same code if they are close. Conversely, it is unlikely that two far away points are hashed into the same code. A popular class of hashing-based methods is Locality Sensitive Hashing (LSH) [19] and its extensions. LSH methods use multiple random projections to hash database points. For querying, LSH maps the query vector into its corresponding code and finds the closest neighbors from this code only. LSH is data-independent; it does not guarantee that the data will be hashed evenly for every distribution. In some cases, the majority of data points may be hashed into the same code, which will adversely increase the search time. Conversely, data-dependent hashing learns to hash from data. Spectral Hashing [12] is an example of data-dependent hashing where similarity graphs of the inputs are encoded into hash functions. Other examples are Iterative Quantization (ITQ) [20] and Isotropic Hashing [21] (IsoH), both using a rotation matrix of the projected data.

Quantization-based methods. Quantization-based methods for ANN search quantize the dataset into smaller subsets of compact codes. Using these codes, the original data and the corresponding distances can be efficiently regenerated. Quantization-based methods revolve around minimizing the quantization distortion and thus reducing the search error. Other objectives are accelerating the search process and minimizing the storage consumption by quantizing to more compact codes. Product Quantization (PQ) [15] and its extensions decompose the search space into the Cartesian product of several smaller codebooks to compose a large effective codebook. To reduce the quantization distortion Optimized Product Quantization

(OPQ) [16] and Cartesian K-Means (cK-Means) [17] apply arbitrary rotation and dimension reordering for better alignment to the data distribution. Locally Optimized Product Quantization (LOPQ) [18] also applies space decomposition to match the underlying data distribution. Other quantization-based ANN methods include Additive Quantization (AQ) [22], Tree Quantization (TQ) [23], and Composite Quantization (CQ) [24]. These methods suggest an alternative quantization and aim to reduce quantization distortion.

B. Nearest Neighbor (NN) Search

Nearest Neighbor (NN) Search finds closest neighbors to a query point in a D-dimensional metric space $\mathbb{R}^D$. Given a finite dataset of N D-dimensional vectors $\mathcal{Y} = \{y_0, \cdots, y_{N-1}\} \subset \mathbb{R}^D$, a D-dimensional query vector $x \in \mathbb{R}^D$, and a distance metric $d(.,.)$, a Nearest Neighbor search of the query vector x, $\mathrm{NN}(x)$, finds the index i of a vector in the dataset $y_i \in \mathcal{Y}$ with the smallest distance to the query vector x. Namely,

$$\mathrm{NN}(x) = \underset{0 \le i < N}{\arg\min}\, d(x, y_i). \tag{1}$$

On the other hand, a k Nearest Neighbor (kNN) Search finds a set of k indices $\{i_{0k-1}\}$ of k vectors from the dataset with the smallest distance to the query

$$\mathrm{kNN}(x) = \mathrm{k}\underset{0 \le i < N}{\arg\min}\, d(x, y_i) \tag{2}$$

C. Vector Quantization (VQ)

Vector Quantization (VQ) subdivides the search space into several clusters. Each cluster has a centroid and database vectors are attached to their nearest centroid. A quantization function q maps a D-dimensional vector $y \in \mathbb{R}^D$ to the index $i \in \{0, 1, \cdots, k-1\}$ of its nearest centroid c_i from a k-centroid codebook $\mathcal{C} = \{c_0, \cdots, c_{k-1}\}$, namely,

$$q : \mathbb{R}^D \to \{0, \cdots, k-1\}; \quad y \mapsto q(y) = \underset{0 \le i < k}{\arg\min}\, d(y, c_i). \tag{3}$$

A Voronoi cell $\mathcal{V}_i$ is a set of vectors mapped to a centroid c_i,

$$\mathcal{V}_i = \{y \in \mathbb{R}^D : q(y) = i\}. \tag{4}$$

An example of a two-dimensional VQ, including Voronoi cells, is illustrated in Fig. 1 (left). 64 centroids are found using k-means clustering with $k = 64$.

D. Product Quantization (PQ)

Product quantization splits a high-dimensional vector into several orthogonal subspaces, where each subspace is quantized using a different vector quantizer. An input D-dimensional vector $y \in \mathbb{R}^D$ is split into M uniform sub-vectors $(u_0, \cdots, u_{M-1})$ of $\tilde{D} = D/M$ dimensions such that

$$u_j(y) = (y_{j\tilde{D}}, \cdots, y_{j\tilde{D}+\tilde{D}-1}), \tag{5}$$

that is,

$$y = (\overbrace{y_0, \cdots, y_{\tilde{D}-1}}^{u_0(y)}, \overbrace{y_{\tilde{D}}, \cdots, y_{2\tilde{D}-1}}^{u_1(y)}, \cdots, \overbrace{y_{(M-1)\tilde{D}}, \cdots, y_{M\tilde{D}-1}}^{u_{M-1}(y)}). \tag{6}$$

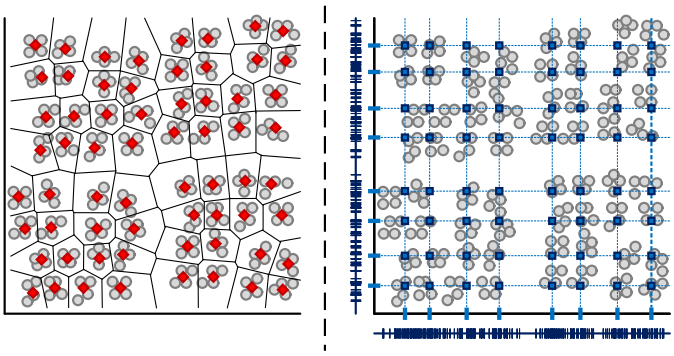

Fig. 1: Multi-dimensional quantization—a two-dimensional example. (left) Vector Quantization (VQ); k-means with $k = 64$. (right) Product Quantization (PQ); k-means for each of two single-dimensional subspaces with $k = 8$)

Two-dimensional PQ is illustrated in Fig. 1 (right). The two-dimensional space is split into two single-dimensional subspaces. k-means clustering is applied to each subspace individually with $k = 8$. The generated centroids are the Cartesian product of centroids of all subspaces. For only two codebooks with eight codes each, a total of $8^2 = 64$ centroids will be generated.

E. Quantization Quality

The quality of a quantizer is affected by the difference between the original vector, y, and the quantized vector, $q(y)$, using a quantizer q. To measure the quantization quality, it is common to use the mean square error of these values, namely,

$$\mathrm{MSE}(q) = E[d\,(y, q(y))^2] = \int p(y) d^2(y, q(y)) dy, \tag{7}$$

where E is expected value, and $p(y)$ is the probability density function.

For a uniformly distributed dataset of N elements, $\{y_i\}_{i=0}^{N-1}$,

$$\mathrm{MSE}(q) = E[d\,(y, q(y))^2] = \frac{1}{N} \sum_{i=0}^{N-1} d^2(y_i, q(y_i)). \tag{8}$$

The quality of a product quantizer is affected by each subspace quantizer. All subspaces of a product quantizer are orthogonal, thus the MSE of a product quantizer and the sum of all subspaces MSEs, thus

$$\mathrm{MSE}(q) = \sum_{j=0}^{M-1} \mathrm{MSE}(q_j), \tag{9}$$

where q_j is the subquantizer of subspace j.

III. Hierarchical Product Quantization (HPQ)

In this section, we present the proposed HPQ method. Subsection III-A motivates and explains the key idea for this work. Offline initialization of the HPQ database is described in Subsection III-B, whereas searching the HPQ database for the k-nearest neighbors is described in Subsection III-C.

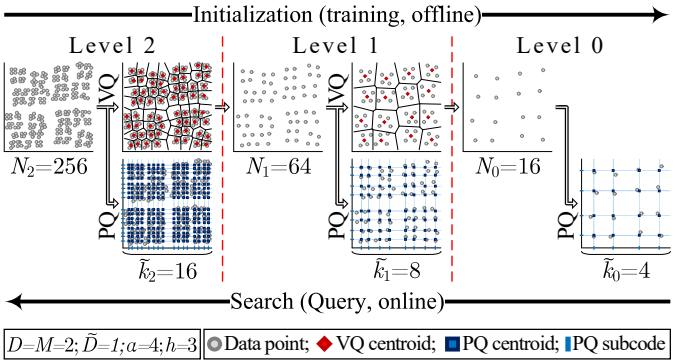

Fig. 2: A toy example of two-dimensional HPQ with three levels.

A. Motivation and Key Idea

In this paper, we propose a PQ-based ANN search via space-partitioning hierarchical search. The search space is efficiently and gradually refined by employing hierarchical search, preventing time-consuming exhaustive search. Vector Quantization (VQ) is utilized to subdivide the PQ search space gradually. For each level in the hierarchy, PQ is used to efficiently search within the refined subspace.

An example of two-dimensional ($D = 2$) HPQ with three levels ($h = 3$) is illustrated in Fig. 2. For each level, the search space is subdivided by four ($\alpha = 4$). PQ includes two subspaces ($M = 2$), whereas each subspace has a single dimension ($\tilde{D} = 1$). The initialization process starts from the higher level, creates PQ codebooks, and subdivides the search space down to the lower level. Querying the database starts from the lower level and searches a partial subspace that has been selected by the previous level.

Fig. 3 shows a recursive data structure for HPQ. The search space of the current level, $\mathcal{Y}^i$ is quantized using two methods. (1) PQ (Fig. 3, blue) to simplify the search process using multiple PQ codebooks, and (2) VQ (Fig. 3, red) to refine the search space and avoid exhaustive search. The PQ process clusters the D-dimensional search space into M_i subspaces, each of which is $\tilde{D}_i = D/M_i$ dimensional.

B. Training: Initializing the HPQ Data Structure

k-means clustering is the core of the initializing (training) process, and is defined as a function returning a 3-tuple of

$$(\mathcal{C}, \mathcal{V}, \tilde{\mathcal{Y}}) = k\text{-means}(\mathcal{Y}) \tag{10}$$

Where $\mathcal{Y}$ is a dataset of N D-dimensional vectors, k is the number of clusters required, $\mathcal{C}$ is an array of k D-dimensional centroids, $\mathcal{V}$ is an array of k Voronoi cells, and $\tilde{\mathcal{Y}}$ is the encoded dataset, an array of N indices to centroids from $\mathcal{C}$.

A Voronoi cell $\mathcal{V}_i$ is a set of vectors mapped to the nearest centroid c_i, satisfying Lloyd's first optimality condition [25], thus,

$$\mathcal{V} = \{\mathcal{V}_i\}_{i=0}^{k-1} \; ; \; \mathcal{V}_i = \{y \in \mathbb{R}^D : c_i = \arg\min_{c \in \mathcal{C}} d(y, c)\} \tag{11}$$

Each centroid c_i is the expectation of all vectors within its Voronoi cell $\mathcal{V}_i$, satisfying the second Lloyd's optimality condition [25], thus

$$\mathcal{C} = \{\mathcal{C}_i\}_{i=0}^{k-1} \; ; \; c_i = \mathrm{E}[y \in \mathcal{V}_i] = \int_{\mathcal{V}_i} p(y) y dy. \tag{12}$$

The encoded dataset matches each data point to its closest centroid, namely

$$\tilde{\mathcal{Y}} = \{\tilde{y}_i\}_{i=0}^{N-1} \; ; \; \tilde{y}_i = \arg\min_{0 \le i < N} d(y, c_i). \tag{13}$$

The HPQ data structure initialization process is described in Algorithm 1. For each level $0 \le i < h$, both PQ and VQ are performed. PQ splits the D-dimensional spaces into M_i equal subspaces. For each subspace $0 \le m < M_i$, a codebook is generated (Line 3). The m'th codebook is generated from the m'th subspace of the current level, that is $\mathcal{Y}^i[m\tilde{D}_i \cdots (m+1)\tilde{D}_i - 1]$. On the other hand, the VQ process reduces the the search space by α_i times. Thus the number of point in level $i-1$, $N-1$, will be α_i times less than its upper level i, namely, $N_{i-1} = N_i \alpha_i$. The VQ process finds those N_{i-1} centroids and their corresponding Voronoi cells by employing N_{i-1}-means on the dataset $\mathcal{Y}_i$ (Line 6). The initial training values of this data structure will be used to initialize the content of the relevant on-chip memories as will be described in Subsection IV-A.

C. Querying the k-nearest neighbors

Searching for the k nearest neighbors of a query vector x is performed in two phases. In the first phase, distances between the query vector and each centroid from all PQ codebooks are calculated. In the second phase, a hierarchical search is performed. In each level of the hierarchy, the codebooks are used to lookup the sub-distances. Each level will gradually refine the search space as shown in Fig. 4. Algorithm 2 describes both

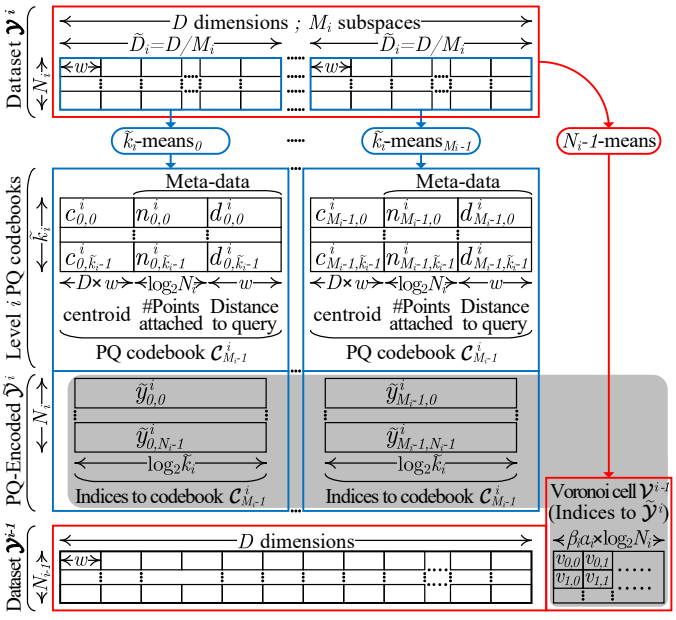

Fig. 3: A recursive definition of the HPQ data structure. Shaded tables will be combined and partitioned as shown in Subsection IV-A and Fig. 5 (middle).

Algorithm 1: Initialize HPQ data structure (train, offline)

1 **for** $i = h-1$ **to** 0 **do**
  // initialize PQ codebook for each subspace
2 **for** $m = 0$ **to** $M_i - 1$ **do**
3 $(\mathcal{C}^i_m, \tilde{\mathcal{V}}^i_m, \tilde{\mathcal{Y}}^i_m) \leftarrow \tilde{k}_i$-means$(\mathcal{Y}^i[m\tilde{D}_i, \cdots, (m+1)\tilde{D}_i - 1])$
4 $n^i_m \leftarrow |\tilde{\mathcal{V}}^i_m|$
  // Initialize VQ codebook
5 $N_{i-1} \leftarrow N_i/\alpha_i$
6 $(\mathcal{Y}^{i-1}, \hat{\mathcal{V}}^{i-1}, -) \leftarrow N_{i-1}$-means$(\mathcal{Y}^i)$

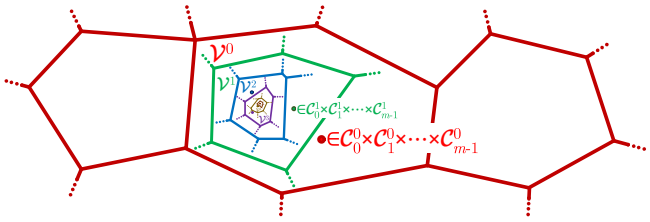

Fig. 4: HPQ search through levels of the hierarchy. Centroids are PQ-encoded

Algorithm 2: HPQ k-ANN search (query, online)

input : x: a D-dimensional query vector
output : kANN(x): k indices in $\mathcal{Y}$ for x's k ANN
// **Phase 1:** compute distances to query
1 **for** $i = 0$ **to** $h-1$ **do**
2 **for** $m = 0$ **to** $M_i - 1$ **do**
3 **for** $j = 0$ **to** $\tilde{k}_i - 1$ **do**
4 $d^i_{m,j} \leftarrow d^2(c^i_{m,j}, (x_{m\tilde{D}_i}, \cdots, x_{(m+1)\tilde{D}_i - 1}))$
// **Phase 2:** hierarchical search
// **2.a:** ANN PQ search for the lowest level
5 $\text{ANN}_0(x) \leftarrow \arg\min_{0 \le j < N_0} \sum_{m=0}^{M_0-1} d^0_{m,\tilde{y}^0_{m,j}}$
6 **for** $i = 1$ **to** $h-1$ **do**
7 $j(p) := v^i_{\text{ANN}_{i-1}(x),p}$
  // **2.b:** ANN HPQ search for intermediate levels
8 $\text{ANN}_i(x) \leftarrow \arg\min_{0 \le p < \beta_i \alpha_i} \sum_{m=0}^{M_i-1} d^i_{m,\tilde{y}^i_{m,j(p)}}$
9 **if** $i == h-1$ **then**
  // **2.c:** k-ANN HPQ search for the top level
10 **return** $k\text{ANN}(x) \leftarrow k \arg\min_{0 \le p < \beta_i \alpha_i} \sum_{m=0}^{M_i-1} d^i_{m,\tilde{y}^i_{m,j(p)}}$

of these phases in detail. The distance computation process is described in Line 4. The distance between a centroid j of subspace m in level i, namely $c^i_{m,j}$, and the m'th subspace of x, namely $u_m(x) = (x_{m\tilde{D}_i}, \cdots, x_{(m+1)\tilde{D}_i-1})$ is assigned to the corresponding distance $d^i_{m,j}$.

After computing the distances in the first phase, a hierarchical search is performed in the second phase. This phase consists of three steps, based on the location of each level in the hierarchy. In Phase 2.a the lowest level of the hierarchy is processed (Line 5). An exhaustive search PQ of the lowest level is performed. The number of points in this level is the lowest among all levels as it is refined α_i times by each higher level $1 \le i < h$, namely $N_0 = N/\prod_{i=0}^{h-1} \alpha_i$. The squared distance between each one of the $N_0 - 1$ points in this level (indxed with $0 \le j < N_0$) and the query vector x is computed and the index of the point with the minimum distance is returned as $\text{ANN}_0(x)$. The squared distance between each point and the query vector is the summation of all squared distances in each subspace $0 < m \le M_0$. These distances are computed and stored in PQ codebooks (as meta-data) in Phase 1, and are retrieved in Phase 2.

The distance between the m'th sub-vector of the query vector and a point, j, is retrieved from the distance stored in the codebook $\mathcal{C}^0_m$. The encoded dataset of level 0, $\tilde{y}^0$, stores for each point j in the subspace m its corresponding codebook index $\tilde{y}^0_{j,m}$. The search of the lowest level returns $\text{ANN}_0(x)$, the index of the Voronoi cell in the upper level with the nearest centroid to the query point. In the second step (Phase 2.b) the intermediate levels are searched (Line 8). First we find the Voronoi cell from the previous level with the nearest centroid to the query $v^i_{\text{ANN}_{i-1}(x),p}$ (points are indexed with p), then we search this Voronoi cell using the same method as in Phase 2.a. The search in the top level (Phase 2.c) is performed similarly to the intermediate levels, however, k nearest neighbors are found (Line 10).

IV. HPQ Hardware Acceleration

In this section we apply custom hardware acceleration to our HPQ method. The search process is pipelined through the hierarchy to increase the search throughput. To accelerate queries, FPGAs' massive parallelism is utilized to perform concurrent codebook lookups, distance computation, and comparison.

FPGAs contain several thousand distributed on-chip BRAMs and several thousand DSP blocks. These distributed memories are used to store the codebooks and the coded dataset for fast and parallel retrieval, whereas DSP blocks are utilized to compute and compare distances in parallel. Since a query requires multiple accesses to codebooks and performs multiple FP computations, this combination of distributed memory blocks and FP resources is capable of alleviating the performance bottleneck.

In Subsection IV-A, considerations for hardware acceleration, including memory partitioning are discussed. Memory and hardware consumption are estimated in Subsection IV-B and Subsection IV-C, respectively. Finally, the support of accessing an external memory is discussed in Subsection IV-D.

A. Hardware Considerations

The proposed HPQ accelerator is illustrated in Fig. 5. The search pipeline is shown in Fig. 5 (bottom). The implementation of each level in the hierarchy is shown in the rest of the figure. Fig. 5 (middle) shows memory partitioning for the encoded dataset, codebook lookup, and finding the minimal distance (Algorithm 2, Phase 2). The codebooks and distance computation (Algorithm 2, Phase 1) is shown in Fig. 5 (top).

Deep pipelining of the search hierarchy. The HPQ pipeline is shown in Fig. 5 (bottom). All intermediate levels of the hierarchy are identical. The first and the last levels minimally

differ, as will be shown later. Pipelining is possible since the search is hierarchical. Each level gradually refines the search and sends the index of the nearest centroid, $\texttt{minidx}_i$, forward. The next level will only search the Voronoi cell of this centroid. The whole pipeline is controlled by an FSM. Once a valid query `x` is received and `start` is asserted, codebooks in all levels will be concurrently updated with squared distances (Algorithm 2, Phase 1), then the search indices will propagate through the pipeline (Algorithm 2, Phase 2). To maximize throughput, all FP computations are internally pipelined.

Computation parallelism. To enhance querying performance, computation parallelism is required in both phases of the search process (Algorithm 2). In Phase 1, $\tilde{p}$ squared distances in each codebook are computed in parallel. As shown in Fig. 5 (top), $\tilde{p}$ centroids are retrieved from each codebook and distances to the corresponding query sub-vector are computed in parallel. Writing to the squared distances meta-data portion of the codebook is delayed by the same latency of the computing the squared distances, $L(\text{SQD})$. The squared distance between two n-dimensional vectors, x and y, in the Euclidean space is $d^2(x,y) = \sum_{j=0}^{n-1}(x_j - y_j)^2$. This sum is implemented as an addition-tree.

In Phase 2 of the search process (Fig. 5, middle) $\hat{p}$ different distances are computed and compared. The encoded dataset is organized in BRAMs. Indices to codebooks are retrieved from these BRAMs to lookup the codebooks for distances in each subspace. For each vector, an adder tree sums squared distances from all subspaces to find the total distance to the query vector. A min tree is then applied on all $\hat{p}$ to find the minimal distance. The top level of the hierarchy requires finding k vectors with minimal distances, thus the min-tree will be replaced with a k-selection mechanism as follows.

Memory partitioning. As depicted in Fig. 5, level i of the hierarchy receives from its previous level $i-1$ the index of the Voronoi cell that should be searched ($\texttt{minidx}_{i-1}$). Only points in $\mathcal{V}^i_{\texttt{minidx}_{i-1}}$ should be searched. To accelerate the search, $\hat{p}$ points are searched in parallel. Consequently, $\hat{p}$ points of $\mathcal{V}^i_{\texttt{minidx}_{i-1}}$ should be concurrently read from the BRAMs. To support reading $\hat{p}$ points from any Voronoi cell, we divide each Voronoi cell into $\hat{p}$ chunks, each with $\alpha_i\beta_i/\hat{p}$ points. (each Voronoi cell can accommodate up to $\alpha_i\beta_i$ points). Each chunk out of the $\hat{p}$ chunks is stored in a separate BRAM. To read $\hat{p}$ point from the same Voronoi cell $\texttt{minidx}_{i-1}$, all BRAMs will be addressed with the same address. The base address is $(\alpha_i\beta_i/\hat{p})\texttt{minidx}_{i-1}$. The address offset will iterate over all chunk content, thus will count up to $\alpha_i\beta_i/\hat{p} - 1$.

B. HPQ Memory Consumption

As depicted in Fig. 5, HPQ requires two memory structures, (1) the PQ codebooks (Fig. 5, top) , and (2) the encoded dataset (Fig. 5, middle). In total, each level, i, consumes

$$S_i = \overbrace{M_i\tilde{k}_i(\underbrace{\tilde{D}_i w}_{\substack{\text{sub-}\\\text{centroid}}} + \underbrace{\log_2 N_i}_{\substack{\#\text{points}\\\text{attached}}} + \underbrace{w}_{\substack{\text{dist-}\\\text{ance}}})}^{\text{Codebook of level } i} + \overbrace{\beta_i N_i M_i \log_2 \tilde{k}_i}^{\text{partitioned memory}}. \quad (14)$$

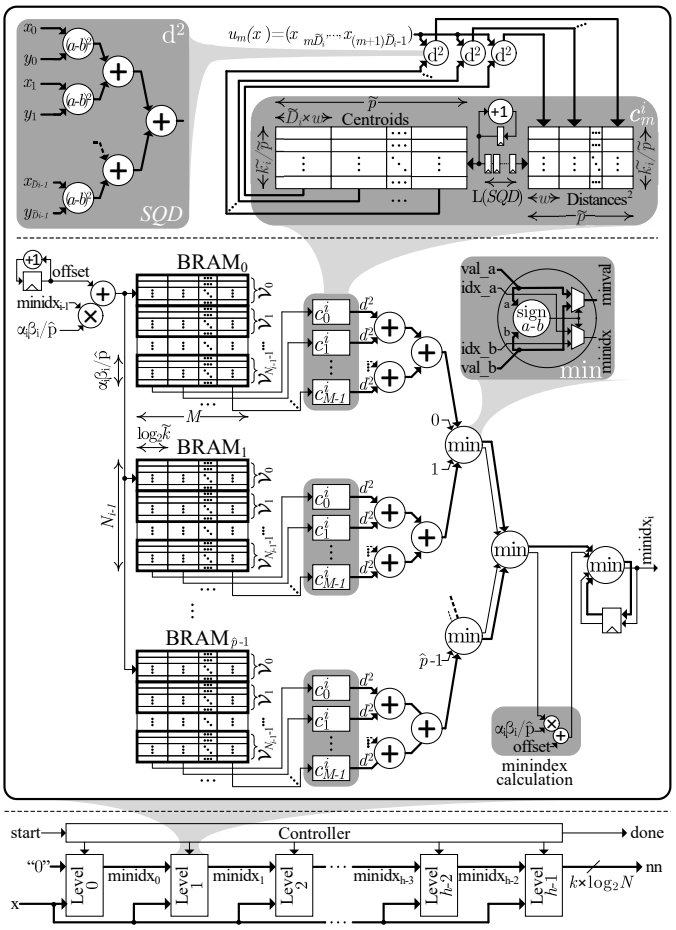

Fig. 5: Hardware architecture of a single level i. (top) Query Phase 1: updating codebook $\mathcal{C}^i_m$ with distances between query point and centroids. (middle) Query Phase 2: finding the point index with the minimum distance to the query. Batcher's odd–even mergesort sorting network [26] is used instead of min-tree for k-selection. (bottom) The complete pipeline.

Assuming that the dataset index is narrower than the FP representation, $\log_2 N_i < w$, and given that $\beta_i > 1$, the storage consumption of level i can be bounded by

$$\begin{aligned} S_i <& M_i\tilde{k}_i w(\tilde{D}_i + 2) + \beta_i N_i M_i \log_2 \tilde{k}_i \\ =& \tilde{k}_i w(D + 2M_i) + \beta_i N_i M_i \log_2 \tilde{k}_i. \end{aligned} \quad (15)$$

The total storage consumption is thus bounded by

$$\begin{aligned} S =& \sum_{i=0}^{h-1} S_i \\ <& wD\sum_{i=0}^{h-1}\tilde{k}_i + 2w\sum_{i=0}^{h-1}\tilde{k}_i M_i + \sum_{i=0}^{h-1}\beta_i N_i M_i \log_2 \tilde{k}_i \end{aligned} \quad (16)$$

Assuming that the depth of each codebook $\tilde{k}_i$, number of subspaces M_i, tapering factor α_i, and centroid occupancy factor β_i are the same across all levels, namely, $\tilde{k}_i = \tilde{k}$, $M_i = M$, $\tilde{D}_i = \tilde{D}$, $\alpha_i = \alpha$, and $\beta_i = \beta$, we obtain

$$S < hw\tilde{k}(D + 2M) + \beta M \log_2 \tilde{k} \sum_{i=0}^{h-1} N_i. \quad (17)$$

Using the tapering ratio, $N_i/N_{i-1} = \alpha_i$, and applying a constant tapering factor, α, the ratio between then number of points in the top level and any other level, i, is

$$N_i/N = \alpha^{i-h+1}. \tag{18}$$

A sepcial case of (18) is

$$\alpha^h = N \quad \Rightarrow \quad h = \log_\alpha N. \tag{19}$$

The total number of points in all levels is obtained from the sum of the geometric series in (18) as

$$\sum_{i=0}^{h-1} N_i = \frac{N-1}{1-1/\alpha}. \tag{20}$$

Using (19) and (20) in (17) provides

$$\begin{aligned} S <& hw\tilde{k}(D+2M) + \beta M \frac{N-1}{1-1/\alpha} \log_2 \tilde{k} \\ <& \underbrace{w\tilde{k}(D+2M)\log_\alpha N}_{\text{codebooks}} + \underbrace{\frac{\beta}{1-1/\alpha} MN \log_2 \tilde{k}}_{\text{partitioned memory}}. \end{aligned} \tag{21}$$

For large datasets, N, and large tapering factor, α, the codebooks size is negligible compared to the partitioned memory. The dominating term of the total memory consumption is therefore

$$S \approx \beta MN \log_2 \tilde{k} \tag{22}$$

C. Hardware Consumption and Performance Estimation

As depicted in Fig. 5, querying is performed in two phases. In the first phase (Fig. 5, right top) all PQ codebooks are updated with the distances between the query point and the PQ codebook centroids. $\tilde{p}$ distance computations are done in parallel, this requires

$$\widetilde{F}_i = \tilde{p}M(\tilde{D} + \tilde{D} - 1) \tag{23}$$

FP-DSPs for each stage. In the second phase (Fig. 5, bottom), $\hat{p}$ points are compared. M sub-distances are added using an adder tree, requiring $\hat{p}(M-1)$ FP adds for each stage. $\hat{p}$ distances are compared using a min-tree, requiring $\hat{p}-1$ FP compares for each stage, in total

$$\widehat{F}_i = \hat{p}(M-1) + \hat{p} - 1 \tag{24}$$

Since both search phases are performed serially, they can share FP hardware without performance degradation. The complete hierarchy thus requires

$$F = \max(\widetilde{F}_i, \widehat{F}_i)h \tag{25}$$

The latency of querying consists of the latency of both search phases. In the first phase (Fig. 5, right top), $\tilde{p}$ out of $\tilde{k}$ FP square computations are processed in parallel, whereas the pipeline depth is $L(\text{FPSQR})$, in addition to two memory accesses, thus the latency of the first phase is

$$\tilde{L} = \tilde{k}/\tilde{p} + L(\text{FPSQRSUB}) + L(\text{FPADD}) \log_2 \tilde{D} + 2. \tag{26}$$

in the second phase (Fig. 5, bottom), $\hat{p}$ out of $\alpha_i\beta_i$ points are processed in parallel, the depth of the add tree is $L(\text{FPADD}) \log_2 M$, whereas the depth of the min-tree is $L(\text{FPSUB}) \log_2 \hat{p}$. With the addition of two memory accesses, the latency of the second phase is therefore

$$\hat{L}_i = \alpha_i\beta_i/\hat{p} + L(\text{FPADD}) \log_2 M + L(\text{FPSUB}) \log_2 \hat{p} + 2. \tag{27}$$

The first phase is done in parallel, while the second phase is a pipelined hierarchical search; the latency of the whole hierarchy is therefore,

$$\begin{aligned} L =& \tilde{L} + \hat{L}_i h \\ =& \tilde{L} + \hat{L}_i \log_\alpha N. \end{aligned} \tag{28}$$

The proposed design is a pipelined hierarchical search, thus the throughput of each stage is equal to the total throughput, the reciprocal of the throughput is therefore,

$$T^{-1} = T_i^{-1} = \tilde{L} + \alpha_i\beta_i/\hat{p}. \tag{29}$$

D. External Memory Support

Modern high-end FPGA devices integrate several hard on-chip DRAM controllers together with their configurable fabric. Although the latency and bandwidth of external memories are inferior to on-chip SRAM memory blocks, they are capable of storing a large amount of data, not possible to store on chip. In case the on-chip memory is not sufficient to store the entire data structure, we propose to store a portion of the data structure in the external memory. To maximize the benefits of using external memory we propose to store the largest table in our design while keeping the bandwidth requirement minimal. The largest table of our data structure is the encoded dataset of the top level $\mathcal{Y}^{h-1}$. The size of this table is $NM \log_2 \tilde{k}$ bits.

If this table is stored in an external memory, the total on-chip memory consumption can be derived from (21) by substituting the number of points in the second upper-most level N/α:

$$S < \underbrace{w\tilde{k}(D+2M)\log_\alpha N}_{\text{codebooks}} + \underbrace{\frac{\beta}{\alpha-1} MN \log_2 \tilde{k}}_{\text{partitioned memory}}. \tag{30}$$

The external memory is now required to transfer $\hat{p}$ points, each with M indices of $\log_2 \tilde{k}$ bits every cycle. Thus $\hat{p}$ should be tuned to satisfy

$$T(\text{DRAM}) > \hat{p}M \log_2 \tilde{k} \quad \text{bits/cycle}, \tag{31}$$

where $T(\text{DRAM})$ is the read throughput (bandwidth) of the external memory.

Since reading from the external memory can be performed in parallel with computing the codebook distances, the total latency with external memory is derived from (28) where the codebook distance computation latency $\tilde{L}$ is substituted with $\max(L(\text{DRAM}), \tilde{L})$, namely

$$\begin{aligned} L =& \max(L(\text{DRAM}), \tilde{L}) + \hat{L}_i h \\ =& \max(L(\text{DRAM}), \tilde{L}) + \hat{L}_i \log_\alpha N, \end{aligned} \tag{32}$$

where L(DRAM) is the read latency of the external memory, measured by read transfer/cycle, and consists of the DRAM Column Access Strobe (CAS) latency, in addition to the DRAM controller latency.

V. Experimental Results

In this section, we present the experimental results of our proposed HPQ technique, show that these results match our previous analysis, evaluate our proposed method, and compare it to state-of-the-art techniques.

k-ANN benchmark. Our experiments are conducted on SIFT1M [27], a popular, state-of-the-art, and publicly available dataset for k-ANN evaluation. The SIFT1M is a dataset of 1M 128-dim SIFT vectors and 10K additional query vectors.

Accuracy metric. The search accuracy is measured in recalls, a popular metric for retrieval performance of a similarity search. The recall measure $R@r$ is the probability that the correctly identified nearest neighbor is among the actual r nearest neighbors. Namely, the probability that the nearest neighbor retrieved by the algorithm is ranked within the first r true nearest neighbors, computed over several queries.

Platform settings. To evaluate query time and resources consumption, all different HPQ instances were synthesized using Intel's Quartus Prime targeting a Stratix10 GX2800 `1SG280HH1F55I1VG`. This is the highest speed grade device with 933K ALMs, 229Mb BRAMs, and 5760 DSPs. Each DSP is capable of performing a single-precision FP operation.

Accuracy vs. resources trade-off. Table II shows different implementations of the SIFT1M dataset on Stratix10 GX2800 device. We are able to store all the data structure on-chip. This table also shows the trade-off between accuracy and memory requirements. This is in agreement with the original PQ-based k-ANN work [15]. The parameters M and $\tilde{k}$ are tuned to achieve a desirable accuracy within a memory storage limitation as shown previously in (21). On the other hand, the architectural parameters α, $\tilde{p}$, and $\hat{p}$ has no impact on the memory consumption. Instead, these parameter control the parallelism of the design, thus affect search throughput ((29)), latency ((28)), and DSP usage ((28)). This is in agreement with our analysis in Subsection IV-C.

Comparison. Query times of CPU-based methods [15], [18] and GPU based methods [28], [29] are reported on Xeon E5-1630v3 CPU and Nvidia GTX Titan Xp GPU, respectively. The FPGA-based OpenCL approch [30] is implemented on Intel's HARPv2 CPU+FPGA platform; an Arria 10 GX1150 FPGA together with a 14-core Broadwell Xeon CPU and a 17GB/sec QDI interface. To campare with the later work, we also implement our HPQ on Arria 10 GX1150, assuming the same memory bandwidth and using out estimates from Subsection IV-D. The QDA interface bandwidth limits our compare parallelism to $\hat{p} = 4$ as deduced from (31). Given a comparable accuracy, Table III shows that our query time (on Sttratix10 GX2800) is at least 5 magnitudes lower than CPU-based approaches [15], [18], $\times 255$ lower than GPU [28], [29] and FPGA-based OpenCL [30] approaches. For fair comparison with the later FPGA-based OpenCL approach [30], we implement our HPQ on the same FPGA device, Arria10 GX1150, assuming external memory overhead as estimated in Subsection IV-D. These settings shows that HPQ query time is $\times 61$ lower than the FPGA-based OpenCL approach [30].

TABLE II: Accuracy Trade-offs of the SIFT1M Dataset.

Accuracy			Parallelism				Resources		Performance[a]		
R@100	M	$\tilde{k}$	α	h	$\tilde{p}$	$\hat{p}$	BRAMs Mb	DSPs	F_{max} MHz	Latency	$\frac{1}{\text{Throughput}}$ us/query
0.973	16	64	128	3	8	64	204	5508	334	0.37	0.078
0.89	16	32	128	3	8	64	174	5514	357	0.34	0.062
0.752	8	64	128	3	8	128	110	5703	368	0.33	0.073
0.57	8	32	128	3	8	128	90	5698	412	0.29	0.056

[a] Measured on Stratix10 GX2800 FPGA using on-chip memory only.

TABLE III: Comparison of Performance and Accuracy on SIFT1M dataset.

Platform	Method	Latency $\frac{1}{\text{Throughput}}$ us/query		R@100
CPU[a]	LOPQ [18]		51.1k	0.97
	IVFPQ [15]		11.2k	0.93
GPU[b]	PQT [28]		20	0.86
	FAISS [29]		20	0.95
OpenCL FPGA[c]	LOPQ [30]		20	0.97
Custom FPGA[d]	HPQ_1	0.85	0.33	0.973
Custom FPGA[e]	HPQ_2	0.37	0.078	0.973

[a] Xeon E5-1630v3 CPU: quad core, 8 threads, 10MB cache, 3.7GHz.
[b] Nvidia GTX Titan Xp GPU: 3840 CUDA cores, 1.6GHz, 12 TFLOPs.
[c] Intel HARPv2: 14 core Broadwell Xeon CPU + Arria10 GX1150 FPGA.
[d] Arria10 GX1150 FPGA: 427K ALMs, 53Mb BRAMs, and 1518 DSPs. Optimal design parameters: $(M, \tilde{k}, \alpha, h, \tilde{p}, \hat{p}) = (16, 64, 16, 5, 1, 4)$
[e] Stratix10 GX2800 FPGA: 933K ALMs, 229Mb BRAMs, and 5760 DSPs. Optimal design parameters: $(M, \tilde{k}, \alpha, h, \tilde{p}, \hat{p}) = (16, 64, 128, 3, 8, 64)$

VI. Conclusions and Future Work

In this paper, a novel modular ANN search method is proposed. The proposed technique utilizes Product Quantization (PQ) to efficiently search for the closest neighbors of a query point in a high dimensional metric space. Our Hierarchical Product Quantization (HPQ) approach applies space partitioning through hierarchical search and gradually refines the search space; avoiding memory- and compute-intensive exhaustive search. Vector Quantization (VQ) performs a gradual subdivision of the PQ search space while PQ is used to search efficiently within the refined subspace. The HPQ approach is efficiently accelerated on custom hardware. While PQ successfully compresses memory by encoding the dataset into several codebooks, the hierarchical search allows for a deeply pipelined design and dramatically reduces the total amount of FP operations. The proposed method is implemented on Intel Stratix 10 FPGA devices. Experimental results show that the search performance of our technique significantly outperforms other state-of-the-art methods.

While the vast majority of ANN search techniques only support static databases, recent ANN search applications, such as sparse memory augmented neural networks [31], require fast online updates. To support these state-of-the-art applications, we plan to support online updates as a future work. Furthermore, the implementation of the codebooks would benefit greatly from using SRAM-based multi-ported memories. While the use of multi-ported memories in FPGAs has been cost-prohibitive, a recent work provides a near-optimal BRAM-based multi-ported memories [32].

REFERENCES

[1] O. Boiman, E. Shechtman, and M. Irani, "In defense of Nearest-Neighbor based image classification," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2008, pp. 1–8.

[2] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2008, pp. 1–8.

[3] J. Sivic and A. Zisserman, "Video Google: a text retrieval approach to object matching in videos," in *Proc. of the IEEE Int. Conf. on Comput. Vision (ICCV)*, Oct. 2003, pp. 1470–1477.

[4] A. Bewley and B. Upcroft, "Advantages of Exploiting Projection Structure for Segmenting Dense 3D Point Clouds," in *Proc. of the Australasian Conf. on Robot. and Autom.*, Dec. 2013, pp. 1–8.

[5] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[6] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. on Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sept. 1977.

[7] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *Proc. of the ACM SIGMOD Int. Conf. on Manag. of Data*, June 1984, pp. 47–57.

[8] J. T. Robinson, "The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes," in *Proc. of the ACM SIGMOD Int. Conf. on Manag. of Data*, Apr. 1981, pp. 10–18.

[9] P. N. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," in *Proc. of the Annu. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, Jan. 1993, pp. 311–321.

[10] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is "Nearest Neighbor" Meaningful?" in *Proc. of the Int. Conf. on Database Theory (ICDT)*, Jan. 1999, pp. 217–235.

[11] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," in *Proc. of the Annu. IEEE Symp. on Foundations of Comput. Sci. (FOCS)*, Oct. 2006, pp. 459–468.

[12] Y. Weiss, A. Torralba, and R. Fergus, "Spectral Hashing," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, Dec. 2008, pp. 1753–1760.

[13] J. Wang, S. Kumar, and S. Chang, "Semi-supervised hashing for scalable image retrieval," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2010, pp. 3424–3431.

[14] K. He, F. Wen, and J. Sun, "K-Means Hashing: An Affinity-Preserving Quantization Method for Learning Binary Compact Codes," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2013, pp. 2938–2945.

[15] H. Jegou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Trans. on Pattern Anal. and Mach. Intell.*, vol. 33, no. 1, pp. 117–128, Jan. 2011.

[16] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized Product Quantization for Approximate Nearest Neighbor Search," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2013, pp. 2946–2953.

[17] M. Norouzi and D. J. Fleet, "Cartesian K-Means," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2013, pp. 3017–3024.

[18] Y. Kalantidis and Y. Avrithis, "Locally Optimized Product Quantization for Approximate Nearest Neighbor Search," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2014, pp. 2329–2336.

[19] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Sept. 1999, pp. 518–529.

[20] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-Scale Image Retrieval," *IEEE Trans. on Pattern Anal. and Mach. Intell.*, vol. 35, no. 12, pp. 2916–2929, Dec. 2013.

[21] W. Kong and W.-J. Li, "Isotropic Hashing," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, Dec. 2012, pp. 1646–1654.

[22] A. Babenko and V. Lempitsky, "Additive Quantization for Extreme Vector Compression," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2014, pp. 931–938.

[23] ——, "Tree quantization for large-scale similarity search and classification," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2015, pp. 4240–4248.

[24] T. Zhang, C. Du, and J. Wang, "Composite Quantization for Approximate Nearest Neighbor Search," June 2014, pp. II–838–II–846.

[25] S. Lloyd, "Least Squares Quantization in PCM," *IEEE Trans. on Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982.

[26] K. E. Batcher, "Sorting Networks and Their Applications," in *Proc. of the Spring Joint Comput. Conf. (SJCC)*, Apr. 1968, pp. 307–314.

[27] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Process. (ICASSP)*, May 2011, pp. 861–864.

[28] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. P. A. Lensch, "Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2027–2035.

[29] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *arXiv preprint arXiv:1702.08734*, 2017.

[30] J. Zhang, S. Khoram, and J. Li, "Efficient Large-Scale Approximate Nearest Neighbor Search on OpenCL FPGA," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, June 2018, pp. 4924–4932.

[31] J. W. Rae *et al.*, "Scaling Memory-augmented Neural Networks with Sparse Reads and Writes," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, Dec. 2016, pp. 3628–3636.

[32] A. Abdelhadi and G. Lemieux, "A Multi-ported Memory Compiler Utilizing True Dual-Port BRAMs," in *Proc. of the IEEE Annu. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 140–147.