Efficient Large-scale Approximate Nearest Neighbor Search on OpenCL FPGA

Jialiang Zhang
UW-Madison
jialiang.zhang@ece.wisc.edu

Soroosh Khoram
UW-Madison
khoram@wisc.edu

Jing Li
UW-Madison
jli@ece.wisc.edu

Abstract

We present a new method for Product Quantization (PQ) based approximated nearest neighbor search (ANN) in high dimensional spaces. Specifically, we first propose a quantization scheme for the codebook of coarse quantizer, product quantizer, and rotation matrix, to reduce the cost of accessing these codebooks. Our approach also combines a highly parallel k-selection method, which can be fused with the distance calculation to reduce the memory overhead. We implement the proposed method on Intel HARPv2 platform using OpenCL-FPGA. The proposed method significantly outperforms state-of-the-art methods on CPU and GPU for high dimensional nearest neighbor queries on billion-scale datasets in terms of query time and accuracy regardless of the batch size. To our best knowledge, this is the first work to demonstrate FPGA performance superior to CPU and GPU on high-dimensional, large-scale ANN datasets.

1. Introduction

Approximate nearest neighbor (ANN) search in high-dimensional spaces is important to many computer vision tasks, such as image retrieval [3], recognition[11] and classification [5]. This algorithm searches a high-dimensional dataset for data points that are close to a query point. Such searches, however, are slow as a consequence of the *curse of dimensionality*. In the past years, there has been increasing interest [4, 9] in compressing high-dimensional data into compact codes. Compact encoding can compress the size of index to only tens of bits per dataset entry. It not only reduces the size of database, but also improves the efficiency of nearest neighbor search on large-scale datasets.

To generate the compact encoding, a large body of existing works rely on hashing [4], which approximates the similarity between two dataset entries using the Hamming distance between the hashed codes. *Product quantization* (PQ) is an alternative approach for compact encoding, which has been proven to be more effective on large-scale datasets than hashing-based approaches, especially when combined with *inverted files* (IVF)[19].

PQ uses the Cartesian product of multiple small sub-codebooks to achieve a large effective codebook size. Due to its lower quantization distortions, PQ has been shown to be more accurate than hashing-based methods. Moreover, PQ is amenable to be implemented on modern parallel processors (e.g., CPU, GPU), which makes it more promising for large-scale ANN applications. Further, the distances between codewords can be pre-computed and stored in look-up-tables (LUTs). Thus, the query computation involves merely reading the LUTs. To deal with large datasets, PQ is typically combined with the IVF technique, which consists of a coarse, exhaustive step to find a set of probing centroids near the query vector. The number of probing centroids is typically much smaller than the dataset size. Thus, it can significantly reduce the the number of PQ-based distance calculations.

Quantizing dataset elements into a small codebook introduces a quantization error to the ANN results. To reduce the quantization error, existing works [6, 17] try to make the quantization better fit to the underlying distribution of database entries. *Optimized product quantization* (OPQ) and its equivalent Ck-means[17] apply an optimal global rotation to all product quantizers. Another method is *locally optimized product quantization* (LOPQ) [14], which locally optimizes product quantizer over rotation and space decomposition.

Although the PQ-based methods can effectively reduce the database size, they cannot leverage the parallel processing capability of modern processors (e.g. multi-core CPU, GPU, and FPGA) well to reduce the query time. One of the reasons is that codebooks are too large to fit into their small but fast on-chip memories (cache, shared memory and block RAM) and need to be moved to the larger but slower external memories. Furthermore, for each query, these codebooks need to be accessed *multiple* times, which amplifies the storage delay problem creating a key performance bottlengeck. To make matters worse, approaches like OPQ and LOPQ, introduce one or more large rotation matrices, also too large to fit into the on-chip memory, that need to be accessed *multiple* times for every single query. As a result, the query speed in these approaches are bounded by the external memory delay, preventing full utilization of the available computation capabilities. Therefore, existing works on GPU-based ANN acceleration focus on reducing the number of computations of PQ-based method. The method in [22] introduces a Product-quantization tree, which can reduce number of distance calculations, using an additional codebook. Faiss [13] provides several detailed GPU-specific optimizations of the IVFPQ method and reports the state-of-the-art throughput on billion-scale similarity search [22, 13]. Noticeably, Faiss uses a very large batch size (10000) to achieve superior throughput at the cost of the query latency.

In addition to the codebook size issue, another bottleneck in the PQ-based methods is the k-selection, which sorts the distances from query to all database points and chooses the k-smallest ones. Existing approaches, such as Faiss [13], require to store all distances in the external memory, thus, their performance is limited by the external memory bandwidth.

In this paper, we propose a new method to achieve better trade-offs between accuracy and query time on billion-scale dataset. The main contributions of our work are: **1)** a quantization method to compress the codebook and the rotation matrix, which can make a better use of the fast on-chip memory;**2)** a k-selection algorithm, which can be fused with the distance calculation to eliminate the slow external memory access;**3)** a highly optimized OpenCL-FPGA implementation. To the best of our knowledge, this work is the first to implement PQ-based ANN search on HARPv2 [8] FPGA platform.

We evaluate our method using three common benchmarks,YFCC100M, BigANN [12] and Deep1B[1].YFCC100M consists of 100M of feature vectors from CNN model. BigANN consists of 1 billion 128-dimensional SIFT-vectors and 10000 query vectors and Deep1B consists of 1 billion 96-dimensional CNN-descriptor. These two datasets are considered challenging due to the ultra-large scale, which makes the ANN search extremely difficult.

2. Background

In this section, we formally describe the ANN problem and the previous approaches to quantization and their implementations. In particular, we will discuss the IVFPQ technique, which is the basis of our approach. We will then discuss the GPU-based implementation. We will finally present a brief introduction of Open-CL based FPGA development framework, which our approach targets.

2.1. ANN Search

Nearest Neighbor (NN) problem searches a dataset for points near a query point. Let $\mathcal{Y} = \{\boldsymbol{y_1},...,\boldsymbol{y_n}\} \subset \mathbb{R}^D$ represent this dataset and $\boldsymbol{x} \in R^D$ the query. NN finds the k closest data points in $\mathcal{Y}$ to $\boldsymbol{x}$, when the distance between $\boldsymbol{x}$ and $\boldsymbol{y} \in \mathcal{Y}$ is defined as $d(\boldsymbol{x},\boldsymbol{y})$. The result is the set $N_{\boldsymbol{x}} \subset \mathcal{Y}$ such that **1)** $|N_{\boldsymbol{x}}| = k$ and **2)** $\forall \boldsymbol{y} \in \mathcal{Y} - N_{\boldsymbol{x}}, \boldsymbol{y_x} \in N_{\boldsymbol{x}} : d(\boldsymbol{x},\boldsymbol{y_x}) \leq d(\boldsymbol{x},\boldsymbol{y})$.

$$N_{\boldsymbol{x}} = \text{k-}\underset{\boldsymbol{y} \in \mathcal{Y}}{\text{argmin}}\, d(\boldsymbol{x},\boldsymbol{y}) \tag{1}$$

In general, the NN searching has two step: **(i)** Calculating distance between query and database vectors; **(ii)** k-selection, which finds k-smallest distances. As D and the size of dataset $|\mathcal{Y}|$ is often large in computer vision tasks, the computation load of an exhaustive search is extremely high. Because of this, approximate nearest-neighbor (ANN) search is proposed to reduce the query time. This method eases the second condition on $N_{\boldsymbol{x}}$ and requires it to hold with *high probability*. In the following, we discuss one of the ANN search techniques for large-scale dataset.

2.2. PQ-based ANN search

Vector Quantization (VQ) encodes each $\boldsymbol{y} \in \mathbb{R}^D$ by a codebook $\mathcal{C} = \{\boldsymbol{c_1},...,\boldsymbol{c_k}\}$ to a vector $q(\boldsymbol{y}) \in \mathcal{C}$, where $\mathcal{C}$ is a finite subset of $\mathbb{R}^D$. The codebook $\mathcal{C}$ can be built using classical Lloyd iterations [15]. The vector quantizer maps database vector $\boldsymbol{y}$ to its nearest centroids in the codebook:

$$\boldsymbol{y} \mapsto q(\boldsymbol{y}) = \underset{\boldsymbol{c} \in \mathcal{C}}{\text{argmin}}\, d(\boldsymbol{y},\boldsymbol{c}) \tag{2}$$

where the distance $d(.,.)$ is defined as the Euclidean distance between its argument vectors:

$$d(\boldsymbol{y},\boldsymbol{c}) = \|\boldsymbol{y} - \boldsymbol{c}\|_2 \tag{3}$$

The vector quantization method needs to store $D \times k$ values for its codebook.

Product quantization (PQ) compresses high-dimensional vectors to shorter codewords [10]. PQ takes high-dimensional vectors $\boldsymbol{y} \in \mathbb{R}^D$ as the input and splits them into m subvectors $\boldsymbol{y} = [\boldsymbol{y}^{1^T},...,\boldsymbol{y}^{m^T}]^T$ of dimension $D' = D/m$, where D is a multiple of m. Then, these subvectors are quantized by m distinct subquantizer to generate the sub-codewords $q(\boldsymbol{y}) = [q^1(\boldsymbol{y}^1)^T,...,q^m(\boldsymbol{y}^m)^T]^T$. Each sub-quantizer in this method has its own codebook, denoted by $\mathcal{C}^1,...,\mathcal{C}^m \subset \mathbb{R}^{D'}$, each of which has k entries:

$$\forall i \in [1,m] : q^i(\boldsymbol{y}^i) = \underset{\boldsymbol{c} \in \mathcal{C}^i}{\text{argmin}}\, \|\boldsymbol{y}^i - \boldsymbol{c}\|_2 \tag{4}$$

Thus, the codebook of the product quantizer is the Cartesian product of all sub-codebooks:

$$\mathcal{C} = \mathcal{C}^1 \times \cdots \times \mathcal{C}^m, \tag{5}$$

Although this method creats k^m bins, it only requires $k \cdot m$ centroids. However, the compressed codebook size is still considered large when processing large-scale datasets (e.g. Deep1B, YFCC100M, and BigANN). For instance, to achieve R@1=0.45 on Deep1B dataset, Faiss [13] uses $m = 64$ and $k = 256$, which needs to store 64×256 values (64kB). In section 2.3, we will discuss that such a codebook nearly utilizes all the on-chip shared memory (96kB) of the latest GPU devices [18] and further increasing codebook size will result in significantly increased query processing delay. Therefore, due to the constraints of on-chip memory size, it is not feasible to keep increasing the m or k to obtain larger number of bins with this method. Our compression method, as discussed in section 3, can efficiently reduce the size of the codebook, so that we can keep increasing the values of m or k to achieve higher accuracy.

The product quantization calculates the distance from query vector, similarly decomposed into m subvectors

$\boldsymbol{x} = [\boldsymbol{x}^{1^T}, ..., \boldsymbol{x}^{m^T}]^T$ to quantized database vector $q(\boldsymbol{y})$ by:

$$||\boldsymbol{x} - \boldsymbol{q}(\boldsymbol{y})||_2^2 = \sum_{i=1}^{m} ||\boldsymbol{x}^{\boldsymbol{i}} - q^i(\boldsymbol{y}^i)||_2^2 \tag{6}$$

Since each sub-quantizer $q^1, ..., q^m$ can produce a limited number of values, for a given query, distances in each sub-space can be pre-calculated and stored in look-up tables (LUTs)[13]. While processing a query, PQ first build LUTs for all sub-quantizers, which requires $k \times D$ floating-point multiplication. Then, it computes the distance between query vector and all the n dataset vectors using LUTs only, which requires $n \times m$ look-up operations (which are faster than floating point multiplications). Compared to explicit computation, which requires $n \times D$ floating-point multiplications, using LUTs can effectively reduce the computational load if number of database vector n is larger than the number of sub-quantizer level k.

Optimized Product Quantization (OPQ) is proposed to refine PQ using the following mapping:

$$\boldsymbol{y} \mapsto q(\boldsymbol{y}) = \operatorname*{argmin}_{\boldsymbol{c} \in \mathcal{C}} ||R\boldsymbol{x} - \boldsymbol{c}||_2^2, \tag{7}$$

where $R \in \mathbb{R}^{D \times D}$ is an orthogonal matrix which enables arbitrary rotations and permutations of vector components. Optimizing the matrix R and the $\mathcal{C}^1, \cdots, \mathcal{C}^m$ can be either joint as in Ck-means [17] and in the non-parametric solution of OPQNP [6], or decoupled, as in the parametric solution OPQP of [6].

Inverted Index (IVF) [19] is a widely used index structure in the field of information retrieval and approximate nearest neighbor search. IVF first performs clustering on the initial dataset to divide the space into *Voronoi cells* by a codebook $\mathcal{C}_{coarse}$. Then, it stores the list of dataset vectors, which belong to each *Voronoi cell*. While processing a query, IVF enables non-exhaustive searches, by finding either the closest or several of the closest *Voronoi cells*. As a result, IVF effectively reduces the search space, and achieves a substantial speed-up over the exhaustive search. However, IVF requires to store $|\mathcal{C}_{coarse}|$ D-dimensional vectors, creating a large storage requirement. For instance, to achieve R@1=0.45 on Deep1B dataset, Faiss [13] use $|\mathcal{C}_{coarse}| = 262144$, which needs to store 262144×128 floating point values (128MB). All these coarse codebooks need to be accessed from the slow external memory. These accesses dominate the query time.

2.3. ANN Implementation

Table 1. Memory capacity and bandwidth of CPU[7], GPU[18] and FPGA[23]

	CPU [7]	GPU [18]	FPGA [23]
On-chip Mem. Capacity	4608 KB	6144KB	56.875MB
On-chip Mem. Bandwidth	2.5 TB/s	27.6TB/s	36 TB/s
External Mem. Bandwidth	68 GB/s	320 GB/s	68 GB/s

Table 2. Computation Capability of modern processors

	CPU	GPU	FPGA
Peak FP Arithmetic (FLOPS)	600 G	11T	10.5T
Peak look-up (Loop-up per sec)	625 G	6.9 T	9 T

Hardware platform model: Previously, we mentioned that large codebooks in PQ-based approaches can severely increase the query processing time. That is because most hardware platforms feature two main types of storage where these codebooks can be stored: on-chip memory and external memory. Figure 2 shows these two memory types and their relations to the different computation cores. On-chip memory is often very small but is close to computation cores. Thus, the data stored in it can be accesses quickly. In contrast, external memory is large but provides slow accesses. In table 1 we compare the on-chip memory and the external memory for latest CPU, GPU and FPGA platforms[8]. As this table shows, these platforms all have very small on-chip memories which means large codebooks would have to be stored in the external memory. However, the external memories has orders of magnitude smaller bandwidth and, consequently, much lower speeds.

In table 2, we also compare the computation capabilities in terms of peak floating-point operations per second (FLOPS) and peak look-ups per second of these platforms. As this table shows, these platforms offer look-up performances comparable to floating-point operations. Therefore, exact distance calculation, which uses floating-point operations exclusively, does not leverage the available computation capacity effectively. On the other hand, approximate methods, such as PQ, perform both look-ups and floating-point operations. Thus, they can fully exploit the available computation capacity and reduce query time significantly. In section 3 and section 4, we will discuss how to utilize the computation and memory resources effectively by optimizing the algorithm.

GPU-based implementation: Though there are many implementation of similarity search on GPUs, most of them focus on the hashing, small datasets, or exhaustive search. To the best of knowledge, only works in [22, 13] have implemented ANN search on GPU. In [22], authors introduces a product quantization tree (PQT), that can reduces the number of distance calculations. In [13], authors optimized the GPU implementation of the PQ-based distance look-up and the k-selection implementation. We note that both of these two works are focusing on only optimizing the computation kernel, but ignore the increasing codebook size. In our work, we focus on compressing the codebook to reduce the cost of codebook access and to achieve a better trade-off between query speed and accuracy.

OpenCL-FPGA development framework: Field programming gate array (FPGA) is an excellent acceleration platform for ANN search, because of its inherent parallelism, large on-chip distributed memory, and reconfigurability based

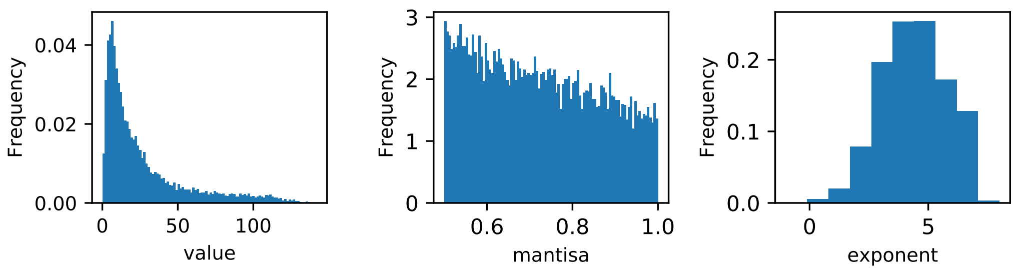

Figure 1. Value distributions of coarse centroids: (a) floating point value; (b) mantissa; (c) exponent

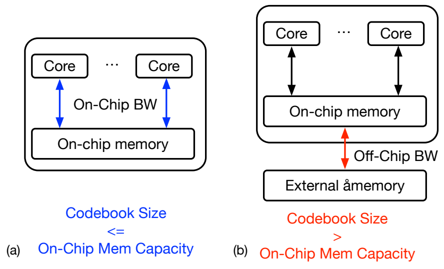

Figure 2. Hardware platform model: (a) bottleneck is the on-chip memory bandwidth if codebook can fit into on-chip memory; (b) bottleneck is off-chi memory is codebook cannot fit into on-chip memory

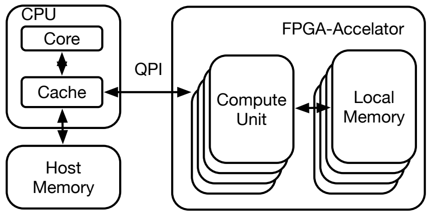

Figure 3. Overview of OpenCL FPGA accelerator

on the needs of the applications. Recently, major cloud service vendors (e.g. Microsoft and Amazon) have deployed FPGAs in their data centers, effectively creating an AI super computer [2].

To allow software developers with little or no hardware background to use FPGA, FPGA vendors start to offer high-level programming interface, such as C/C++ and OpenCL. In figure 3, we show the system architecture of the OpenCL-FPGA framework. It allows programmers to reuse their code on other platforms (CPUs and GPUs), thus significantly enhancing programmability and portability.

3. Codebook compression

High accuracy on large datasets in the PQ-based approach may entail cost-prohibitive storage requirements. As the size of the dataset increases, larger codebooks need to be allocated. In billion-scale similarity search, this overhead becomes non-negligible. For example, Faiss[13] achieves $R@10 = 0.35$ on the BIGANN dataset by introducing 256, 100-dimensional centroids in the product quantizer. These many centroids occupy all the on-chip memory on a GPU. We alleviate this problem by compressing the codebook, to fit in the fast on-chip memory. In this section, we first describe the proposed compression method and then discuss how queries are processed in the compressed codebook scheme.

3.1. Compression Method

To compress the codebook, we propose to apply the scalar quantization to each value in it. Specifically, we construct a second, scalar codebook similar to the finite set $\Gamma \subset \mathbb{R}$ and then sample elements of each member of $\mathcal{C}$ from Γ:

$$\forall i \in [1,k], j \in [1,D] : [c_i]_j \rightarrow \underset{\gamma \in \Gamma}{\operatorname{argmin}} \| [c_i]_j - \gamma \|_1 \tag{8}$$

The resulting quantized codebook, $\mathcal{C}_q \subset \Gamma^D$, then can be efficiently stored in a small-size memory. Our approach to constructing Γ is guided by a comprehensive analysis of the value distributions of codebooks. We present this analysis for a sample dataset here. Then, we discuss a naive approach which we later compare to our proposed quantization scheme.

We apply IVFPQ method to the DEEP1B dataset. For a recall rate R@10 = 0.35 on this dataset, the coarse quantizer ($\mathcal{C}$) needs to contain 262144 centroids, and the product quantizer needs 64 sub-quantizers, each of which would have 256 reproduction full-precision values. In Figure 1 (a), we plot the histogram of these values. As shown in this figure, the distribution is highly biased, with most values populating a small region near zero.

A naive approach to quantizing the codebook elements would be to use k-means to find representative values in this biased distribution. Specifically, after a codebook is trained, we apply Lloyd's algorithm [15] to the set of all elements of codebook entries, $\{[c_i]_j \,|\, \forall i \in [1,k], j \in [1,D]\}$, to find a predetermined number of clusters, say $k_\Gamma = |\Gamma|$. Then, each quantized codebook entry is replaced by a vector of integer indexes to elements of Γ. These vectors can be further compressed using the Huffman encoding, which is a lossless optimal coding technique that generates variable-length code words. By combining scalar quantization and Huffman encoding, we can fit very large codebooks into small-size memory.

This naive method has two shortcomings. First, in this approach, the elements of Γ need to be explicitly stored alongside $\mathcal{C}_q$, occupying our scarce memory and creating additional storage overhead. Second, the overhead of Huffman decoder is high. The storage complexity of Huffman decoder is proportional to the number of symbols squared (k_Γ^2 in this case) [21]. Furthermore, in order to achieve a small enough quantization error on large dataset, we need to use a relatively large k_Γ which exacerbates the problem.

In this work, we propose a new compression technique that reduces the storage overhead and the decoder complexity of the compressed codebook. This technique has been designed based on the observation that the distributions of the codebook elements are different from the distributions of their *mantissa* and *exponents*, and thus can be compressed separately, using simpler techniques [1]. In Figure 1 (b) and Figure1 (c), we plot the histograms for mantissa and exponents of all elements in the codebook, respectively. By adapting to their different distributions, we can reduce the overhead of the codebook compression. In our proposed compression scheme, instead of constructing Γ directly, we construct the sets M and E for the mantissa and exponents and obtain Γ through their product ($\Gamma = M \times E$).

As Figure 1 (b) shows, the mantissa have an approximately uniform distribution. Therefore, we could use a linear quantization method, which divides $[0.5,1]$ into k_M equally-sized subsets ($M = \{\frac{i}{2(k_M-1)} + 0.5 \,|\, i \in [0, k_M - 1]\}$). Different from the k-means clustering, the linear quantization does not need to explicitly store a codebook, as we can obtain the codebook entries from their indexes (the n-th element in the codebook M is $\frac{n-1}{2(k_M-1)} + 0.5$). Under the uniform distribution, the linear quantization gives the clustering results similar to k-means. As a result, we can remove the overhead of storing the extra codebook in the k-means clustering and retain a similar quantization error. Moreover, there is no need to further compress the cluster index using Huffman coding, as the optimal coding scheme is achieved using equal-length coding if the data distribution is uniform.

Similar to the distributions of the elements (Figure 1 (a)), the exponents have a biased distribution, but over a much smaller range (Figure 1 (c)). Thus, applying the naive compression to exponents does not entail the same disadvantages. We construct E by first sorting the exponents based on their values and then encoding each value using Huffman coding. As a result of the sorting, the Huffman encoding assigns smaller codes to more frequent symbols. Furthermore, due to their small dimension, the exponents require fewer symbols, imposing negligible overhead in the Huffman decoder.

The proposed, decoupled quantization scheme constructs large codebooks Γ easily since $|\Gamma| = |M| \times |E|$. To check this scheme, we compare different compression methods in terms of the accuracy, codebook sizes and number of symbols for Huffman coding in table 3. The proposed technique can achieve the similar codewords compression ratio and the accuracy compared to the naive, k-means based clustering, but without the need of storing a large additional codebook and with minimal Huffman decoding overhead.

[1]Each full-precision value, $[c_i]_j$ in the memory is stored as a mantissa, exponent pair $(\mu_{i,j}, \epsilon_{i,j})$, such that $[c_i]_j = \mu_{i,j} \times 2^{\epsilon_{i,j}}$.

3.2. Approximate Nearest Neighbor Search

In the following, we discuss how to apply codebook quantization to conduct ANN search towards a large-scale image-retrieval task. Particularly, we quantize all codebooks in the IVFPQ method and the rotation matrix in the OPQ matrix, if needed. Then, we encode all vectors in the dataset using the quantized codebook. While processing a query, we compute the distance using the asymmetric distance calculation (ADC), which is formulated as

$$\begin{aligned} d^2(\boldsymbol{x},\boldsymbol{y}) &= \sum_{i=1}^{m} ||\boldsymbol{x} - q(\boldsymbol{y})^i||^2 \\ &= ||\boldsymbol{x}||^2 + ||q(\boldsymbol{y})||^2 - 2\sum_{i=1}^{m} \boldsymbol{x} \cdot q(\boldsymbol{y}) \\ &= ||\boldsymbol{x}||^2 + ||q(\boldsymbol{y})_{man} \cdot 2^{q(\boldsymbol{y})_{exp}}||^2 \\ &\quad - 2\sum_{i=1}^{m} \boldsymbol{x} \cdot q^2(\boldsymbol{y})^i_{man} \cdot 2^{q(\boldsymbol{y})_{exp}} \\ &= ||\boldsymbol{x}||^2 + \sum_{i=1}^{m} q^2 q(\boldsymbol{y})^i_{man} \cdot 2^{2 \cdot (\boldsymbol{y})_{exp}} \\ &\quad - 2\sum_{i=1}^{m} \boldsymbol{x} \cdot q(\boldsymbol{y})^i_{man} \cdot 2^{q(\boldsymbol{y})_{exp}}, \end{aligned} \tag{9}$$

Here, $q(\boldsymbol{y})_{man}$ and $q(\boldsymbol{y})_{exp}$ are vectors of the mantissa and the exponent of the quantized codebook element $\boldsymbol{y}$, respectively. It is worth noting that the $q(y)_{man}$ has only a few reproduction values (e.g. 32). To reduce the computational load of the ADC calculation, we can pre-compute all the possible outcome values of $q^2(\boldsymbol{y})$ offline after codebook quantization, and all the possible outcome values of $\boldsymbol{x} \cdot q(\boldsymbol{y})$ online, before calculating the distance. During the calculation of ADC, we can replace a large portion of multiplications by look-ups. As discussed in section 2.3, this technique increases the computation capability, resulting in lower query times.

Table 3. Comparison of the accuracy, the codebook size and the number of symbol for Huffman coding k_Γ (lower is better)

Dataset	R@10	Codebook Size	k_Γ
Uncompressed [9]	0.353	100.6MB	-
K-means	0.364	25MB	-
K-means + Huffman	0.364	12.7MB	256
Proposed	0.366	14.5 MB	8

4. Fast K-selection

Apart from the distance calculation, another costly step in ANN search is the k-selection. In this section, we first identify that the bottleneck is the external memory bandwidth. Then,

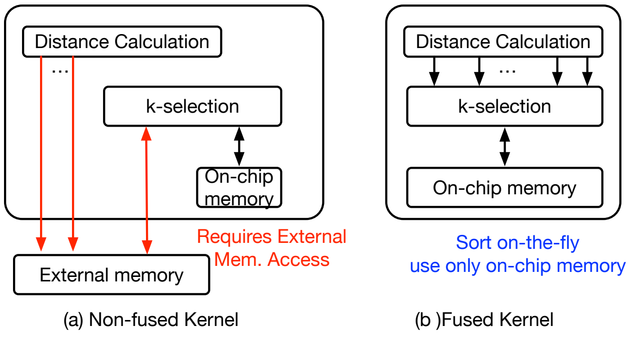

Figure 4. Comparison of non-fused and fused k-selection

Algorithm 1: Inserting p distance to the parallel priority queue

```
input  : $p$ unsorted distance $\mathcal{D}[]$, $p$ Index $I[]$
input  : top-k distance $\mathcal{D}_{min}[]$, top-k index $I_{min}[]$
output : top-k distance $\mathcal{D}_{min}[]$, top-k index $I_{min}[]$
1  $\mathcal{D}'[] \leftarrow \{\mathcal{D}_{min}[],\mathcal{D}[]\}$;
2  $I'[] \leftarrow \{I_{min}[],I[]\}$;
3  for $i \leftarrow 1$ to $p$ do in parallel
4      idx[i] = 0;
5      for $j \leftarrow 1$ to $k$ do in parallel
6          cmp[i] = i;
7      end
8      for $j \leftarrow 1$ to $p$ do in parallel
9          if $\mathcal{D}[i] \geq \mathcal{D}[j]$ then $cmp[i]++$;
10     end
11     for $j \leftarrow 1$ to $k$ do in parallel
12         if $\mathcal{D}[i] \geq d[j]$ then $cmp[i]++$;
13         else $cmp[j+p]++$;
14     end
15 end
16 for $i \leftarrow 1$ to $k$ do in parallel
17     $\mathcal{D}_{min}[cmp[i]] \leftarrow \mathcal{D}'[i]$;
18     $I_{min}[cmp[i]] \leftarrow I'[i]$;
19 end
```

we propose a new k-selection method, which eliminates the external memory access.

K-selection sorts the distances of the query from all dataset points and selects the k smallest ones. Johnson et al. [13] performed this step in a non-“fused” manner and decouple it from distance calculation. As shown in Figure 4 (a), all distances in the non-fused approach need to be stored in and later read back from the slow, external memory, creating a bottleneck on query time. Fusing these two steps can potentially solve this issue by storing only the k smallest distances at any moment during the distance calculation. Thus, in the fused manner all necessary data can be stored in the fast on-chip memory (as shown in figure 4 (b). However, fusing the distance calculation step and the k-selection step in a parallel manner is a challenge, as distances are computed in parallel, which requires k-selection to accept several distances at a time. Otherwise, the distance calculation needs to stall to wait for the k-selection step, which will increase the query time. Next we will present the detail of the fused k-selection.

The proposed fused k-selection kernel maintains a priority queue of the k smallest distances in parallel to the distance calculation step. As the distance between the query point and a new dataset entry is calculated, it is compared against the elements of the queue and, if it is smaller than any of them, is inserted into the queue. Therefore, at any moment during the distance calculation step, the queue will hold the k smallest data entries up to that point. Since the value of k is usually small, this priority queue can be easily stored in the on-chip memory.

Algorithm 1 shows how to insert p newly calculated distances into a priority queue of length k. In this algorithm, $\mathcal{D}$ represents a seqence of distances from a query for p dataset points and I represents the corresponding indexes of these points in $\mathcal{Y}$. Furthermore, $\mathcal{D}_{min}$ and I_{min} represent the k smallest distances and their corresponding indexes befor the p new distances are inserted. This algorithm finds the k smallest distances in the intersection of $\mathcal{D}$ and $\mathcal{D}_{min}$ and stores their values and corresponding indexes back into $\mathcal{D}_{min}$ and I_{min}. All throughout this process, the algorithm guarantees that $\mathcal{D}_{min}$ is maintained sorted. This is done by using a 2-D comparator that for each element in $\mathcal{D}$ the position in $\mathcal{D}_{min}$ where it needs to be inserted. Thus, at the end of the run of the algorithm, all elements in $\mathcal{D}_{min}$ are sorted.

We show a toy example in Figure 5, which inserts 3 distances ($p=3$) to a priority queue for $k=5$. The number of comparisons required is $(p+k)\cdot k$. We compare each distance in $\mathcal{D}$ with other distances in $\mathcal{D}$ as well as with all distances in the priority queue $\mathcal{D}_{min}$. The number of the ”larger or equal” in each row is the index in the $\mathcal{D}_{min}$ of the distances in $\mathcal{D}$. The sum of column index and the number of the ”smaller” in each column determines the new index in the $\mathcal{D}_{min}$. We note that all these comparison can be performed concurrently in parallel, without interfering each other. In other words, the proposed priority queue has no data dependency between the p insertions. The resulting parallelism allows us to perform these inserts at the same rate at which new distances are calculated. Thus, we can fuse the distance calculation and the k-selection steps together.

5. Experiments

5.1. Experimental setup

Table 4. Datasets for evaluation

Datasets	D	#train	#test
SIFT1M	128	100k	1,000,000
SIFT1B	128	1M	1,000,000,000
YFCC100M	4096	2M	96,419,740
Deep1B	96	1M	1,000,000,000

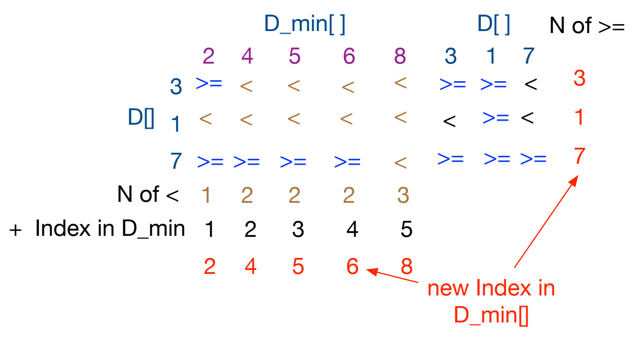

Figure 5. Example of parallel priority queue insertion

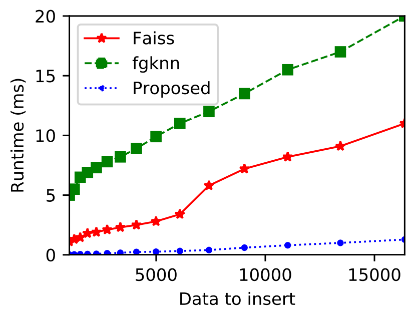

Figure 6. Runtime of different k-selection methods (k=100), as a function of the number of data in the array.

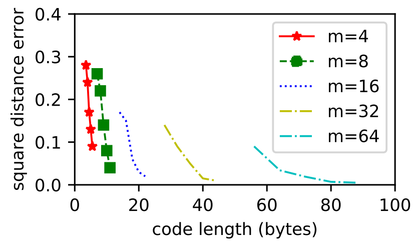

Figure 7. BigANN: quantization error associated with the parameters m and k.

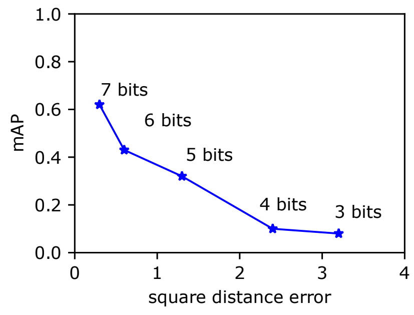

Figure 8. mAP vs. Square Distortion under different mantissa quantization level

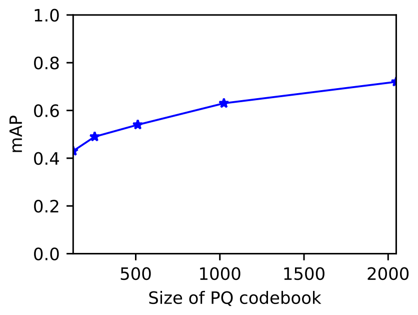

Figure 9. mAP vs. size of product quantization codebook

Table 5. Comparison on SIFT1M dataset

Approaches	Query time (ms /query)	Recall @1	Recall @10	Recall @ 100
LOPQ[14] (CPU)	51.1	0.51	0.93	0.97
IVFPQ [9](CPU)	11.2	0.28	0.70	0.93
FLANN [16](CPU)	5.32	0.97	-	-
PQT[22](GPU)	0.02	0.51	0.83	0.86
FAISS[13](GPU)	0.02	0.8	0.88	0.95
Proposed	0.02	0.88	0.94	0.97

In this section, we present the results of the proposed technique on several publicly available datasets.

Datasets: We conduct experiments on four publicly available datasets: SIFT1M, BIGANN[12], YFCC100M[13] and DEEP1B[1], which are popular in state-of-the-art ANN evaluations. SIFT1M dataset contains 1 million 128-dimensional SIFT vectors and 10K query vectors; BIGANN (SIFT1B) contains 1 billion SIFT vectors and 10K queries; YFCC100M dataset contains 95 million CNN descriptors; DEEP1B contains 1 billion 100-dimension CNN representations for images and 10K query vectors.

Experimental platform: All query time reported on CPU is measured from a single-thread C++ implementation on a Xeon E5-1630v3 CPU. All query time of GPU-based methods is obtained using a Nvidia GTX Titan Xp GPU[18]. All query times of FPGA-based approach are measured on Intel HARPv2 platform[2]. The Intel HARPv2 combines a 14 core Broadwell Xeon CPU and an Arria 10 GX1150 FPGA. The FPGA accelerator and CPU is connected via a QPI interface, which can provide 17GB/s bandwidth and sub-micro second level latency.

Settings: We discuss the experimental settings in the following. Figure 8 shows mAP with respect to the quantization square distortion under different quantization levels of the mantissa. Clearly, the square distortion decreases consistently when the quantization level increases and at the same time mAP increases. Since the memory consumption grows with the mantissa quantization level, we set the mantissa quantization level to 5 bits in the following experiments as a good trade-off between efficiency and accuracy. In section 5.3, we will see that this quantization level is sufficient to outperform the state-of-the-art methods.

Figure 9 shows mAP with respect to the number of centroids in the PQ with the same code length of PQ. We can see that

[2]Results in this publication were generated using pre-production hardware or software, and may not reflect the performance of production of future system

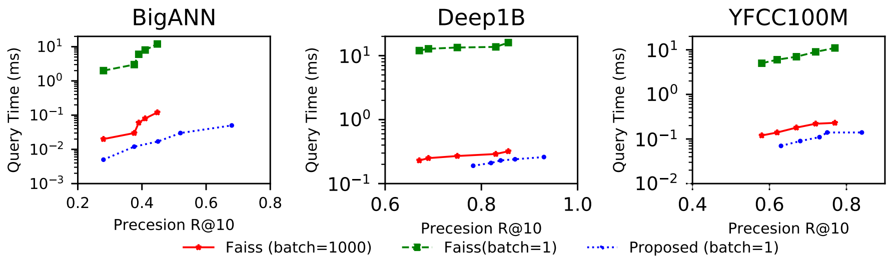

Figure 10. Query time versus Precision on different datasets

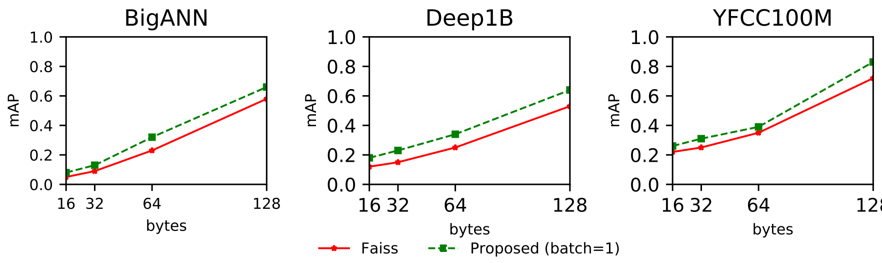

Figure 11. Query time versus database size on different datasets

increasing the number of centroids leads to better precisions compared to increasing the number of sub quantizers. As the proposed design can compress the code, it is possible for us to use a larger size of PQ code book for higher accuracy while achieving the same memory usage. In the following experiments, we use a PQ codebook with 1024 entries and will show in Section 5.2 that our approach leads to a better trade-off between query time and accuracy.

5.2. Results

K-selection: We compare the proposed k-selection methods with two state-of-the-art GPU implementations of fgknn [20] and faiss [13]. We evaluate the k-selection for k=100 from a row-major matrix of $Size_{batch} \times Size_{array}$ random single-precision floating point values. We fix the $Size_{batch}$ to 10000 and vary the $Size_{array}$ from 1024 to 16384. Figure 6 shows our relative performance compared to fgknn and Faiss. The proposed K-selection algorithm is **8.59x** faster than Faiss and **15.65x** faster than fgknn.

Query time and accuracy: Table 5 shows the query time and accuracy on the SIFT1M dataset. We compare the proposed method with three CPU-based approaches and two GPU-based approaches. Compared to other CPU-based approaches, the proposed method shows a significant speed-up. To compare with two GPU-based methods, we tune the hyperparameters of our implementation to achieve the same query time. The accuracy of our approach is 0.08 higher than the state-of-the-art GPU-based implementation.

In Figure 10, we show the search time per query with respect to the accuracy and compare it with FAISS, which provides the state-of-the-art performance. The proposed implementation always uses a batch size of 1, and FAISS uses two different batch sizes (10000 and 1). For all three large-scale datasets, the proposed method has a slightly faster search speed, if the targeting recall rate is relatively low, as compared with FAISS in batch mode. However, it is difficult for FAISS to achieve a higher accuracy due to its large codebook size, which has already used all the on-chip memory resource.

Precision and database size: We first show the benefit of increasing the number of sub-quantizers m compared to increasing the codebook size k. As shown in Figure 7, to achieve the same square distortion on the BigANN dataset, choosing a larger k instead of m leads to shorter code length. In Figure 11, we further show the relationship between accuracy with respect to the database size. Compared to Faiss, the proposed implementation can achieve a better accuracy while using the same codeword length. Benefiting from the compressed codebook, the proposed method can store a product quantization codebook with larger k.

6. Conclusion

In conclusion, we present a new method for performing efficient billion-scale similarity search on high-dimensional vectors. We propose to compress codebooks as well as the rotation matrices. We further combined this technique with a novel k-selection method, which can be fused with the distance calculation. Overall our techniques provide significant reductions in the memory access overheads.

Our prototype implementation on Intel HARPv2 platform demonstrates a better trade-off between accuracy and the query time over the state-of-the-art ANN search on CPU and GPU. We demonstrated that the performance of proposed approach outperforms the state-of-the-art methods on both small benchmark (SIFT1M) and billion-scale large datasets (BigANN, Depp1B and YFCC100M). The proposed approach can be easily implemented on a single-machine OpenCL-FPGA platform and ported to cloud FPGA architectures.

References

[1] 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*, pages 2055–2063, 2016. 2, 7

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016. 4

[3] O. Chum, J. Philbin, J. Sivic, M. Isard, and A. Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007. 1

[4] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004. 1

[5] C. Doersch, S. Singh, A. Gupta, J. Sivic, and A. Efros. What makes paris look like paris? *ACM Transactions on Graphics*, 31(4), 2012. 1

[6] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013. 1, 3

[7] Intel. Intel xeon processor e5-2699 v3. 3

[8] Intel. Xeon+fpga platform for the data center. 2, 3

[9] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011. 1, 5, 7

[10] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011. 2

[11] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3304–3311. IEEE, 2010. 1

[12] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 861–864. IEEE, 2011. 2, 7

[13] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017. 1, 2, 3, 4, 6, 7, 8

[14] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2321–2328, 2014. 1, 7

[15] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982. 2, 4

[16] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009. 7

[17] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3017–3024, 2013. 1, 3

[18] Nvidia. Nvidia tesla p40 inferencing accelerator. 2, 3, 7

[19] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *null*, page 1470. IEEE, 2003. 1, 3

[20] X. Tang, Z. Huang, D. Eyers, S. Mills, and M. Guo. Efficient selection algorithm for fast k-nn search on gpus. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 397–406. IEEE, 2015. 8

[21] J. Van Leeuwen. On the construction of huffman trees. In *ICALP*, pages 382–410, 1976. 5

[22] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. Lensch. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2035, 2016. 1, 2, 3, 7

[23] Xilinx. Ultrascale plus fpga product selection guide. 3