ARM 4-BIT PQ: SIMD-BASED ACCELERATION FOR APPROXIMATE NEAREST NEIGHBOR SEARCH ON ARM

Yusuke Matsui[⋆] *Yoshiki Imaizumi*[†] *Naoya Miyamoto*[†] *Naoki Yoshifuji*[†]

[⋆] The University of Tokyo [†] Fixstars Corporation

ABSTRACT

We accelerate the 4-bit product quantization (PQ) on the ARM architecture. Notably, the drastic performance of the conventional 4-bit PQ strongly relies on x64-specific SIMD register, such as AVX2; hence, we cannot yet achieve such good performance on ARM. To fill this gap, we first bundle two 128-bit registers as one 256-bit component. We then apply shuffle operations for each using the ARM-specific NEON instruction. By making this simple but critical modification, we achieve a dramatic speedup for the 4-bit PQ on an ARM architecture. Experiments show that the proposed method consistently achieves a 10x improvement over the naive PQ with the same accuracy.

***Index Terms*—** ARM, nearest neighbor search, product quantization, SIMD.

1. INTRODUCTION

The approximate nearest neighbor search is a fundamental technique in many fields of computer science, such as signal processing, computer vision, and natural language processing. In this paper, we focus on the product quantization (PQ) family of nearest neighbor methods [1, 2, 3, 4, 5, 6, 7]. When data are too large to be stored directly in memory, compressing them using PQ is the de facto standard for data handling. One can then perform the nearest neighbor search using a table lookup on the compressed data. One of the fastest methods is 4-bit PQ [8, 9], which can search 10^6 vectors in less than 1 ms. Notably, previous approaches were usually evaluated on the x86 architecture.

Meanwhile, searching using architectures other than x86 has gained significant attention recently. For example, ARM architecture has gained rapid popularity, from its applications to advanced consumer computers (e.g., Apple M1) to high-performance cloud resources (e.g., the Amazon Web Services (AWS) Graviton2). PQ-based methods often rely on SIMD instructions specified for x86, such as AVX2 and AVX512. Therefore, there is a great demand for the fast realization of PQ-based methods for ARM.

The technical problem is that the size of the SIMD registers for ARM is usually smaller than that of x86, implying that we cannot make full use of SIMD-based acceleration. 256-bit SIMD registers (i.e., AVX2) are typically available for recent x86 architectures, whereas ARM typically use registers with 128-bit or less (i.e., the NEON family).

We propose a simple but drastically efficient approach for fast 4-bit PQ searching on ARM. The main idea is to handle two 128-bit SIMD registers as if they were a single 256-bit register. In doing so, we can apply a shuffle operation of NEON instructions instead of the x86-specific AVX2. Our contributions are as follows:

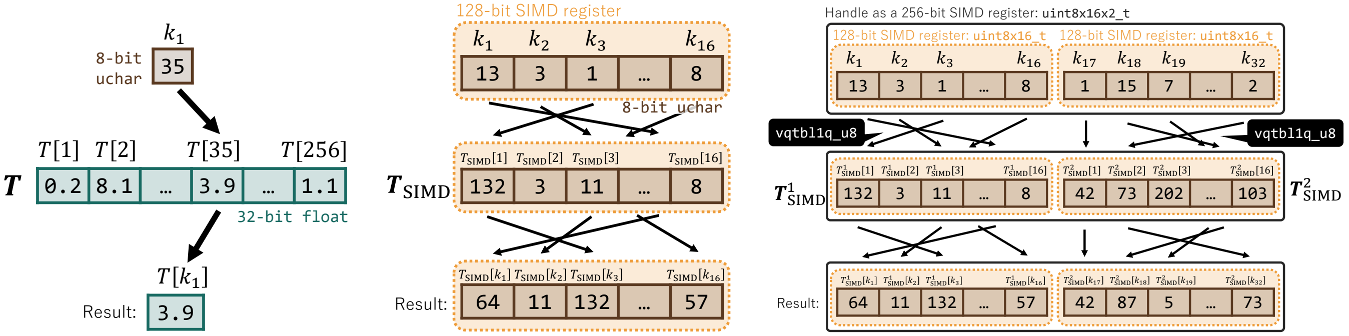
- The proposed 4-bit PQ for ARM is 10x faster than the naive ARM implementation when applied to the SIFT1M and Deep1M datasets.
- The proposed approach, combined with inverted indexing and hierarchical navigable small-world (HNSW) graphs, achieves an efficient search on Deep1B data (less than 1 ms/query). We believe this result is the current record for the fastest nearest neighbor search on the ARM architecture.
- The proposed method has already been implemented in the Faiss library and is available freely [10].

2. PRELIMINARILY

We first describe the problem setting and introduce the most relevant approach. Let $\{\mathbf{x}_n\}_{n=1}^N$ be N D-dimensional database vectors, where $\mathbf{x}_n \in \mathbb{R}^D$. Given a query vector, $\mathbf{q} \in \mathbb{R}^D$, our task is to find the database vector that minimizes the (squared) Euclidean distance: $\|\mathbf{q} - \mathbf{x}_n\|_2^2$. We can find this vector using a simple linear scan, but doing so presents two problems. First, we need at least $32ND$ bits of memory space to store the database vectors (using 32-bit floats). This cost is prohibitive if the database is large, such as when $N = 10^9$ and $D = 100$. Second, the computational complexity is $O(ND)$, which is practically slow.

Product Quantization [1]: To solve the memory issue, the de facto approach is to compress each database vector by PQ [1], where a vector is split into sub-vectors, each quantized by traditional vector quantization (VQ) [11]. As PQ is a straightforward extension of VQ, we discuss VQ in this section for simplicity. We then extend the discussion to PQ at the end of Sec. 3.

978-1-6654-0540-9/22/$31.00 ©2022 IEEE


k_1
8-bit uchar
35
$T[1]$ $T[2]$ $T[35]$ $T[256]$
T
0.2 8.1 … 3.9 … 1.1
32-bit float
$T[k_1]$
Result: 3.9
128-bit SIMD register
k_1 k_2 k_3 k_{16}
13 3 1 … 8
8-bit uchar
$T_{\mathrm{SIMD}}[1]$ $T_{\mathrm{SIMD}}[2]$ $T_{\mathrm{SIMD}}[3]$ $T_{\mathrm{SIMD}}[16]$
T_{SIMD}
132 3 11 … 8
$T_{\mathrm{SIMD}}[k_1]$ $T_{\mathrm{SIMD}}[k_2]$ $T_{\mathrm{SIMD}}[k_3]$ $T_{\mathrm{SIMD}}[k_{16}]$
Result: 64 11 132 … 57
Handle as a 256-bit SIMD register: uint8x16x2_t
128-bit SIMD register: uint8x16_t
128-bit SIMD register: uint8x16_t
k_1 k_2 k_3 k_{16} k_{17} k_{18} k_{19} k_{32}
13 3 1 … 8 1 15 7 … 2
vqtbl1q_u8
vqtbl1q_u8
$T^1_{\mathrm{SIMD}}[1]$ $T^1_{\mathrm{SIMD}}[2]$ $T^1_{\mathrm{SIMD}}[3]$ $T^1_{\mathrm{SIMD}}[16]$ $T^2_{\mathrm{SIMD}}[1]$ $T^2_{\mathrm{SIMD}}[2]$ $T^2_{\mathrm{SIMD}}[3]$ $T^2_{\mathrm{SIMD}}[16]$
T^1_{SIMD}
132 3 11 … 8 42 73 202 … 103
T^2_{SIMD}
$T^1_{\mathrm{SIMD}}[k_1]$ $T^1_{\mathrm{SIMD}}[k_2]$ $T^1_{\mathrm{SIMD}}[k_3]$ $T^1_{\mathrm{SIMD}}[k_{16}]$ $T^2_{\mathrm{SIMD}}[k_{17}]$ $T^2_{\mathrm{SIMD}}[k_{18}]$ $T^2_{\mathrm{SIMD}}[k_{19}]$ $T^2_{\mathrm{SIMD}}[k_{32}]$
Result: 64 11 132 … 57 42 87 5 … 73

(a) ADC ($K = 256$) (b) ADC with 128-bit SIMD register (c) Proposed: ADC with 128x2-bit SIMD registers on ARM

Fig. 1: Comparison of asymmetric distance computation (ADC) lookup operations

Given an input vector, $\mathbf{x} \in \mathbb{R}^D$, let us define a vector quantizer, $Q : \mathbb{R}^D \to \{1, \dots, K\}$, as follows:

$$Q(\mathbf{x}) = \underset{k \in \{1,\dots,K\}}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{c}_k\|_2^2. \tag{1}$$

Here, $\{\mathbf{c}_k\}_{k=1}^K \subset \mathbb{R}^D$ provides codewords, which are typically created by running k-means clustering over the training dataset. Note that K is the number of codewords. With Q, a D-dimensional vector is quantized into a short code (integer) k, which can be represented by only $\log_2 K$ bits. Here, K is usually set to 256, so that each code takes $\log_2 256 = 8$ bits = 1 B (unsigned char). Given k, we can lossy-reconstruct the original vector, $\mathbf{x}$, by fetching the codeword, $\mathbf{c}_k$.

We apply Q for each database vector, $\mathbf{x}_n$, which results in the quantized codes, $\{k_n\}_{n=1}^N$, where $k_n = Q(\mathbf{x}_n) \in \{1, 2, \dots, K\}$. Here, we discard the original database vectors, $\{\mathbf{x}_n\}_{n=1}^N$, and maintain only $\{k_n\}_{n=1}^N$. Compared with the original vectors, which require $32ND$ bits, the quantized codes require only $N \log_2 K$ bits.

Given $\mathbf{q}$, our task is to find a similar item from the quantized codes, $\{k_n\}_{n=1}^N$. A straightforward approach is to reconstruct all vectors, $\mathbf{c}_{k_1}, \mathbf{c}_{k_2}, \dots, \mathbf{c}_{k_N}$, and run a linear scan, but this requires considerable time and memory. Alternatively, we can obtain the same result via an asymmetric distance computation (ADC) [1]. Let us define a K-dimensional vector, $\mathbf{T} \in \mathbb{R}^K$, whose k-th element is defined as follows:

$$T[k] = \|\mathbf{q} - \mathbf{c}_k\|_2^2. \tag{2}$$

We create $\mathbf{T}$ only once when given $\mathbf{q}$. To approximate the distance between $\mathbf{q}$ and $\mathbf{x}_n$, we look up $\mathbf{T}$ by k_n as the key:

$$\|\mathbf{q} - \mathbf{x}_n\|_2^2 \sim \|\mathbf{q} - \mathbf{c}_{k_n}\|_2^2 = T[k_n]. \tag{3}$$

This is a fast $O(1)$ operation. Fig. 1a visualizes the operation.

4-bit PQ [8, 9]: Although the table lookup discussed above is fast, it is not "extremely" fast because (1) we must use the main memory for the lookup, and (2) the entire operation lacks concurrency. A faster approach is to look up the element from the SIMD register [8, 9], which is a special hardware that can efficiently perform the same arithmetic operation on multiple elements. The problem is that the SIMD register is small (typically 128 or 256 bits). Thus, we must modify the algorithm to fully employ it. The 4-bit PQ achieves an approximate but fast PQ via the SIMD register.

We next describe the 4-bit PQ. Recall that $\mathbf{T}$ is a K-dimensional vector. Thus, $\mathbf{T}$ is represented by a K-length array with floating points ($32K$ bits total). Under the typical setting, $K = 256$, the total memory cost of $\mathbf{T}$ is 8,192 bits; hence, we cannot load it using the SIMD register. Hence, we apply a drastic approximation:

- Set $K = 16$ so that each code takes only $\log_2 16 = 4$ bits. This explains the "4-bit" aspect of PQ.
- Apply a scalar quantization for each element in $\mathbf{T}$, so that each element is represented by an 8-bit unsigned char.

By this, we obtain a new look-up table, $\mathbf{T}_{\mathrm{SIMD}} \in \{1, \dots, 256\}^{16}$. Using this table, we can replace the Eq. (3) as follows:

$$\|\mathbf{q} - \mathbf{x}_n\|_2^2 \sim \|\mathbf{q} - \mathbf{c}_{k_n}\|_2^2 \sim f(T_{\mathrm{SIMD}}[k_n]), \tag{4}$$

where f is the reconstruction of an unsigned char to float, which is trivial. This table is represented by an array of 16 unsigned chars, which is 128 bits total and can be loaded onto the SIMD register. Because each code, k_n, is represented by only 4 bits, we can load 16 elements (e.g., k_1 to k_{16}) at once. Using the shuffle operation, we can run the table lookup inside the SIMD register in parallel, as shown in Fig. 1b. Then, we store the result in another SIMD register. This approach is fast because the SIMD-register lookup is much faster than a memory lookup, and we can process 16 elements at once.

Similar SIMD-based formulations have been proposed (Bolt [12], ScaNN [13], and MADNESS [14]). The Faiss library [15] also supports the SIMD-based approach[1]. For the latest x86 architectures, the AVX512-based method has also been proposed [16].

[1] In Faiss, the 4-bit PQ algorithm is referred to by the class name `PQFastScan`, from the original paper's title[8, 9].

3. PROPOSED APPROACH

We next describe our approach for achieving a 4-bit PQ for the ARM architecture. We select the Faiss implementation for our baseline as it is the de facto library for approximate nearest neighbor (ANN) searching. The 256-bit SIMD register (the `__m256i` data type for AVX2) is used for 4-bit PQ Faiss searching on x86 architectures. Using a 256-bit register, we can look up Eq. (3) twice with a single SIMD operation (i.e., the 256-bit shuffle operation; `_mm256_shuffle_epi8`). Thus, we can operate 32 elements at once. This choice is natural for the x86 architecture because all applicable modern computers have 256-bit SIMD registers.

However, the problem is that the ARM architecture usually does not have 256-bit registers. For example, only 64- and 128-bit SIMD registers are available for ARMv7 and ARMv8, respectively. Moreover, ARM does not offer a 256-bit lookup operation.

Our approach is simple; we concatenate two 128-bit SIMD registers and use them as if it is a single 256-bit register (`uint8x16x2_t`). We run a 128-bit lookup operation (`vqtbl1q_u8`) twice: one for the first 128 bits of the register and the other for the last 128. These operations achieve the same result as the `_mm256_shuffle_epi8` of AVX2.

We visualize our approach in Fig. 1c. Let us define the two tables as $\mathbf{T}^1_{\mathrm{SIMD}}, \mathbf{T}^2_{\mathrm{SIMD}} \in \{1, \dots, 256\}^{16}$. Each table takes 128 bits, as discussed. We stack these two registers and handle the whole as a 256-bit component. To perform a lookup operation, the system receives 32 8-bit unsigned chars as `uint8x16x2_t` (e.g., $k_1, \dots, k_{32}$). We run the 128-bit shuffle operation, `vqtbl1q_u8`, twice as follows:

- For the first 128 bits ($k_1, \dots, k_{16}$), we run the shuffle with $\mathbf{T}^1_{\mathrm{SIMD}}$.
- For the remaining 128 bits ($k_{17}, \dots, k_{32}$), we run the shuffle with $\mathbf{T}^2_{\mathrm{SIMD}}$.

The result is also stored in the `uint8x16x2_t` registers. With the proposed simple approach, we emulate a 256-bit 4-bit PQ operation using two 128-bit SIMD registers.

From VQ to PQ: The above discussion is regarding a VQ that directly quantizes an input vector. PQ, however, splits the input vector into M D/M-dimensional sub-vectors. VQ is then applied for each sub-vector. With this adaptation, the memory cost for each vector becomes $M \log_2 K$ bits. Thus, for a typical $K = 256$ setting, the cost is $8M$ bits. For a 4-bit PQ with $K = 16$, the cost is $4M$ bits. Note that we must carefully maintain the code layout [8, 9]. Optimizing the layout is also a central topic of 4-bit PQ, but it is outside the scope of this paper.

Implementation detail: From a data-structure perspective, the provision of an interface that is equivalent for both x86 and ARM is essential. In this research, we designed a transparent interface for ARM by appropriately reinterpreting the x86-optimized base class for 256-bit registers in Faiss. We also implemented auxiliary instructions that are only present in AVX2 but not in ARM, including `_mm256_movemask_epi8`. Please refer to our source code for more details [10].

4. CONNECTION TO EXISTING APPROACHES

Next, we discuss the connection between 4-bit PQ and other ANN methods. Recall that 4-bit PQ is used on compressed data. Hence, 4-bit PQ is a building block of an ANN system.

Inverted index: To handle large-scale datasets, inverted indexing is widely used [1]. The idea is to split the entire dataset ($\mathcal{X} = \{\mathbf{x}_n\}_{n=1}^N$) into n_{list} smaller disjoint subsets: $\mathcal{X}_1, \dots, \mathcal{X}_{n_{\mathrm{list}}}$, where $\mathcal{X}_i \cap \mathcal{X}_j = \emptyset$ for all i, j. Note that $\bigcup \mathcal{X}_i = \mathcal{X}$. Let $\boldsymbol{\mu}_i \in \mathbb{R}^D$ be a representative vector for $\mathcal{X}_i$. We then run the search as follows [2]:

1. For *coarse quantization*, run the search between the query, $\mathbf{q}$, and the representative vectors, $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_{n_{\mathrm{list}}}$.
2. Select the n_{probe} nearest representative vectors (i.e., the n_{probe} nearest subsets).
3. For *distance estimation*, run the second (fine) search for the selected subsets.

This inverted index provides a general framework that we can use with any NN algorithm for coarse quantization.

We can also compress the data via PQ to handle large data in memory. For this case, we can achieve distance estimation via the ADC of the PQ. Because 4-bit PQ is a special case of the usual PQ, we can use 4-bit PQ for inverted indexing.

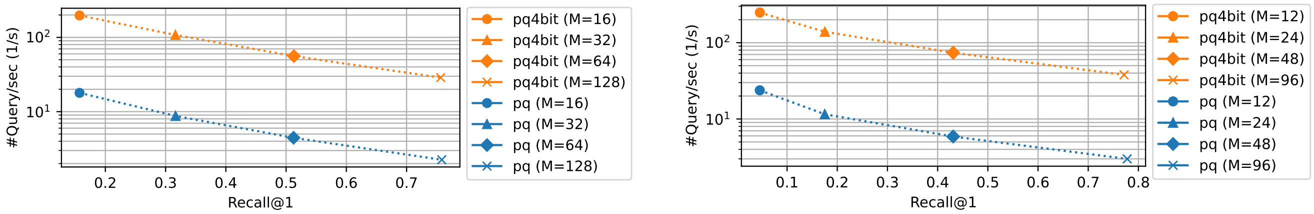
Graph-based approaches The current best algorithms for million-scale datasets are graph-based methods [17, 18, 19, 20, 21, 22]. HNSW is an example [18] that is surprisingly fast and accurate, but it requires vast memory resources.

Our 4-bit PQ and HNSW are complementary. The purpose of a 4-bit PQ is to reduce the memory cost while providing an efficient distance estimation; hence, we can combine the HNSW and 4-bit PQ in the inverted index framework: (1) using HNSW for *coarse quantization*, and (2) using 4-bit PQ for *distance estimation*. This combination of inverted index + HNSW + PQ is a common approach to handling large-scale data when they cannot be directly loaded into memory [23, 24]. We evaluate this hybrid approach in Sec. 5.2.

5. EVALUATION

We performed all experiments on the AWS platform using a high-performance ARM computer: a `c6g.4xlarge` instance with a Graviton2 processor (16 vCPU and 32-GB RAM). We implemented our method on the faiss codebase. We evaluated all methods with a single thread for fair comparison. All results were the average of five trials.

We used the following datasets for our evaluation:



(a) Results on the SIFT1M dataset

(b) Results on the Deep1M dataset

Fig. 2: Comparison between the proposed 4-bit product quantization (PQ) and standard PQ. Note that $K = 16$ for all cases (i.e., each vector takes $M \log_2 K = 4M$ bits).

- The Deep1B dataset [25] consists of 96D deep features with 10^9 base, 10^4 query, and 10^6 training vectors; we used the top 10^6 vectors for the original training set.
- The Deep1M dataset is a subset of the Deep1B dataset (the top 10^6 and the top 10^5 vectors for the base and the training set, respectively).
- The SIFT1M dataset [26] consists of 128D features with 10^6 base, 10^4 query, and 10^5 training vectors.

5.1. Comparison with the original PQ

Fig. 2 shows the comparison between the original PQ and our 4-bit PQ for the SIFT1M and Deep1M datasets. In these figures, the farther the point to the right , the more accurate the method; furthermore, the higher the point, the faster.

For both datasets, the two approaches achieved the same accuracy for the same M, whereas **our 4-bit PQ always achieved 10x faster searches than the original PQ**. This dramatic speedup was made possible by the proposed SIMD operation with the NEON instruction.

5.2. Large-scale evaluation

In this subsection, we show that our approach works well for the large-scale Deep1B dataset. For billion-scale datasets, we can run neither linear scans nor HNSW, which handles the data directly without compression. As discussed in Sec. 4, we used the inverted index framework with HNSW and the 4-bit PQ. The parameters of the space-division were set to $n_{\text{list}} = 30,000$ by following a well-known heuristic: $\sqrt{N}$. We compressed each vector using 4-bit PQ with $K = 16$ and $M = 16$, resulting in 64 bits per code. For the first coarse search, we ran HNSW for $30,000$ representative vectors $(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_{30,000})$. The distance estimation step was achieved using the proposed SIMD-based search.

Table 1 summarizes the results. Although the accuracy was low, our approach achieved a search in less than 1 ms/query. To our knowledge, this is the fastest result for the ARM computer with a billion-scale dataset reported in the literature. Note that our approach provides a highly memory-efficient setting. The best accuracy is given using a carefully designed approach with data compression and graph-search (e.g., Link&Code [27]; Recall@1 = 0.668, runtime = 3.5 ms/query, and memory cost = 864 bit/vector). Compared with Link&Code, our approach was far less accurate (0.072 vs. 0.668); however, it was fast (0.51 vs. 3.5 ms) and memory-efficient (64 vs. 864 bits/code).

n_{list}	n_{probe}	M	K	Accuracy (Recall@1)	Runtime (ms/query)
30,000	1	16	16	0.072	0.51
30,000	2	16	16	0.082	0.83
30,000	4	16	16	0.086	1.3

Table 1: Large-scale evaluation using the Deep1B dataset.

6. CONCLUSION

We developed a fast SIMD-based acceleration scheme for a 4-bit PQ algorithm using the ARM architecture. The concept here is to bundle two 128-bit SIMD registers and handle them as a single 256-bit register. Evaluations showed that our approach accelerates the original PQ by 10x. We believe that our result is one of the fastest ANN evaluations for ARM to date. We have already implemented our approach in the Faiss library, and it is freely available for anyone to use.

Future work: The comparison between x86 and ARM machines was not thoroughly performed because it is difficult to compare different architectures in the same experimental setting. Our preliminary benchmark with same-level AWS instances (`c6i.4xlarge` and `c6g.4xlarge`) suggested that x86 seems to still be cost-efficient as of 2021. Further evaluation remains as future work.

Acknowledgment: This work was supported by JST, PRESTO Grant Number JPMJPR1936, Japan.

7. REFERENCES

[1] Hervé Jégou, Matthijis Douze, and Cordelia Schmid, "Product quantization for nearest neighbor search," *IEEE TPAMI*, vol. 33, no. 1, pp. 117–128, 2011.

[2] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh, "A survey of product quantization," *ITE Transactions on Media Technology and Applications*, vol. 6, no. 1, pp. 2–10, 2018.

[3] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun, "Optimized product quantization," *IEEE TPAMI*, vol. 36, no. 4, pp. 744–755, 2014.

[4] Ting Zhang, Chao Du, and Jingdong Wang, "Composite quantization for approximate nearest neighbor search," in *Proc. ICML*, 2014.

[5] Artem Babenko and Victor Lempitsky, "Additive quantization for extreme vector compression," in *Proc. IEEE CVPR*, 2014.

[6] Julieta Martinez, Joris Clement, Holger H. Hoos, and James J. Little, "Revisiting additive quantization," in *Proc. ECCV*, 2016.

[7] Julieta Martinez, Shobhit Zakhmi, Holger H. Hoos, and James J. Little, "Lsq++: Lower running time and higher recall in multi-codebook quantization," in *Proc. ECCV*, 2018.

[8] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec, "Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan," in *Proc. VLDB*, 2015.

[9] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec, "Accelerated nearest neighbor search with quick adc," in *Proc. ICMR*, 2017.

[10] "`simdlib_neon.h` in the faiss library," `https://github.com/facebookresearch/faiss/blob/main/faiss/utils/simdlib_neon.h`.

[11] Robert M. Gray, "Vector quantization," *IEEE ASSP Magazine*, vol. 1, no. 2, pp. 4–29, 1984.

[12] Davis W. Blalock and John V. Guttag, "Bolt: Accelerated data mining with fast vector compression," in *Proc. ACM KDD*, 2017.

[13] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *Proc. ICML*, 2020.

[14] Davis Blalock and John Guttag, "Multiplying matrices without multiplying," in *Proc. ICML*, 2021.

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou, "Billion-scale similarity search with gpus," *IEEE TBD*, vol. 7, no. 3, pp. 535–547, 2021.

[16] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec, "Quicker adc : Unlocking the hidden potential of product quantization with simd," *IEEE TPAMI*, vol. 43, no. 5, pp. 1666–1677, 2021.

[17] Dmitry Baranchuk, Dmitry Persiyanov, Anton Sinitsin, and Artem Babenko, "Learning to route in similarity graphs," in *Proc. ICML*, 2019.

[18] Yury A. Malkov and Dmitry A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE TPAMI*, vol. 42, no. 4, pp. 824–836, 2020.

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

[20] Liudmila Prokhorenkova and Aleksandr Shekhovtsov, "Graph-based nearest neighbor search: From practice to theory," in *Proc. ICML*, 2020.

[21] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Inf. Syst.*, vol. 87, pp. 101374, 2020.

[22] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin, "Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement," *IEEE TKDE*, vol. 32, no. 8, pp. 1475–1488, 2020.

[23] Kimihiro Tanaka, Yusuke Matsui, and Shin'ichi Satoh, "Efficient nearest neighbor search by removing anti-hub," in *Proc. ICMR*, 2021.

[24] Dmitry Baranchuk, Artem Babenko, and Yury Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *Proc. ECCV*, 2018.

[25] Artem Babenko and Victor Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proc. IEEE CVPR*, 2016.

[26] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *Proc. IEEE ICASSP*, 2011.

[27] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou, "Link and code: Fast indexing with graphs and compact regression codes," in *Proc. IEEE CVPR*, 2018.