Quicker ADC : Unlocking the Hidden Potential of Product Quantization with SIMD

Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec

Abstract—
Efficient Nearest Neighbor (NN) search in high-dimensional spaces is a foundation of many multimedia retrieval systems. A common approach is to rely on Product Quantization, which allows the storage of large vector databases in memory and efficient distance computations. Yet, implementations of nearest neighbor search with Product Quantization have their performance limited by the many memory accesses they perform. Following this observation, André et al. proposed Quick ADC with up to $6\times$ faster implementations of PQ $m\times4$ product quantizers (PQ) leveraging specific SIMD instructions.
Quicker ADC is a generalization of Quick ADC not limited to PQ $m\times4$ codes and supporting AVX-512, the latest revision of SIMD instruction set. In doing so, Quicker ADC faces the challenge of using efficiently 5,6 and 7-bit shuffles that do not align to computer bytes or words. To this end, we introduce (i) *irregular product quantizers* combining sub-quantizers of different granularity and (ii) *split tables* allowing lookup tables larger than registers. We evaluate Quicker ADC with multiple indexes including Inverted Multi-Indexes and IVF HNSW and show that it outperforms the reference optimized implementations (i.e., FAISS and polysemous codes) for numerous configurations. Finally, we release an open-source fork of FAISS enhanced with Quicker ADC.

Index Terms—Image databases; Information Search and Retrieval; Nearest Neighbor Search; Product Quantization; SIMD

1 Introduction

The Nearest Neighbor (NN) search problem consists in finding the closest vector x to a query vector y in a database of N d-dimensional vectors. Efficient NN search in high-dimensional spaces is a core building-block in many multimedia retrieval applications, such as image similarity search, classification, or object recognition. These problems involve extracting high-dimensional feature vectors, or descriptors, and finding the NN of the extracted descriptors in a database of descriptors. For images, SIFT [26], GIST [31] or Deep-learning-based [8] descriptors are often used.

Although efficient solutions for exact NN search have been proposed for low-dimensional spaces, *exact* NN search remains challenging in high-dimensional spaces due to the notorious curse of dimensionality. Hence, much research work has been devoted to Approximate Nearest Neighbor (ANN) search. ANN search returns sufficiently close neighbors instead of the exact NN. Product Quantization (PQ) [19] is an ANN search used in numerous applications [24], [34]. PQ compresses high-dimensional vectors into short codes of a few bytes, enabling in-memory storage of large databases.

Fast answer is a key feature of PQ. It is enabled by Asymmetric Distance Computation (ADC), which efficiently computes distances between query vectors and compressed database vectors using in-memory lookup tables. Yet, despite being faster than regular distance computation, ADC remains bottlenecked by the many memory accesses it performs [1].

- *Fabien André and Nicolas Le Scouarnec were with Technicolor, France.*
- *Anne-Marie Kermarrec is with Inria, France, EPFL, Switzerland, and Mediego, France.*

IEEE Transactions on Pattern Analysis and Machine Intelligence
DOI: 10.1109/TPAMI.2019.2952606

To date, much of the related work has been devoted to the development of efficient inverted indexes [6], [33], which reduce the number of ADCs required to answer NN queries. Recently, there also has been a growing interest in increasing the performance of the ADC procedure itself with the introduction of PQ Fast Scan [1], Quick ADC [2], or polysemous codes [12]. Quick ADC leverages SIMD shuffle instructions to avoid memory accesses and implement very fast ADC; yet it is restricted to 4-bit sub-quantizers and has only been evaluated on simple inverted indexes, thus lacking performance results for more advanced indexes. Polysemous codes leverage the freedom to choose code indices to encode a binary code that is used to prune ADC computations using a low-cost hamming distance; yet, the use of an hamming distance affects the precision.

In this paper, we present Quicker ADC, that introduces two new features to improve performance and accuracy through the use of the latest revision of SIMD instructions, namely AVX512: (i) *irregular product quantizers* combining sub-quantizers of different sizes to allow using 5-bit or 6-bit sub-quantizers, (ii) *split tables* for lookup tables larger than registers thus allowing efficient implementation of 8-bit sub-quantizer from 6-bit or 7-bit shuffles. Quicker ADC is implemented into the reference library FAISS [18], [21] to allow comparison to reference optimized implementations. We released it at *https://github.com/nlescoua/faiss-quickeradc* in order to ease comparisons to our schemes, their adoption, and their evaluation in other settings.

We compare the performance of Quicker ADC to PQ codes [19] and polysemous codes [12] with multiple index types (i.e., simple inverted index [19], inverted multi indexes (IMI) [6], and inverted indexes based on HNSW [27]) for both the SIFT1000M dataset and the Deep1B dataset. Quicker ADC consistently outperforms polysemous codes [12], the state of the art solution for fast response time. For example, on SIFT1000M, for budget of 0.25ms per query and 128-bit codes, Quicker ADC (variant $24\times\{6,6,4\}$) achieves R@1 of

0.23 and R@100 of 0.60 with IMI ($K = 4096^2$) and R@1 of 0.24 and R@100 of 0.68 with IVF HNSW ($K = 2^{18}$), when polysemous codes achieve R@1 of 0.23 and R@100 of 0.47 with IMI and R@1 of 0.18 and R@100 of 0.36 with IVF HNSW.

2 Background

In this section, we first review the data-structures and algorithms used for nearest neighbor search with product quantization. We then discuss the impact of product-quantization parameters on search speed and recall. Finally, we introduce the capabilities of the latest processors supporting AVX-512 and their potential for supporting product quantization.

2.1 Nearest neighbor search with PQ

We describe the different steps for building an indexed database of vectors, and storing compressed representations of these vectors with product quantization. Such a database can then be used to search for the nearest neighbor of a *query* vector efficiently by identifying a subset of *database* vectors for which the distances to the *query* vector needs to be computed, and computing efficiently these distances on compressed representations using a procedure called ADC.

2.1.1 Index

Computing distances for all 1-billion vectors of a database is prohibitive. To tackle this issue in high-dimensional spaces, the common approach is to partition the database using a coarse indexing structure [6], [19]. The input vector space is partitionned into K Voronoi cells using a coarse-quantizer q_i. Vectors lying in each cell are stored in an inverted list. At query time, the inverted index is used to find the closest cells to the query vector, and distances for vectors in the inverted lists of these cells are computed. The two most popular approaches are inverted indexes (IVF) [19], [20] and multi-indexes [6]. The number of cells is limited in IVF (e.g., $K = 2^{16}$) due to the cost of training the IVF and computing distances to each cell. Multi-indexes [6] push this limit and allow a more fine-grained index (e.g., $K = 2^{24}$) at the price of imbalance in inverted list sizes. Recently, alternative approaches have leveraged nearest neighbor graphs to allow faster navigation in the index while avoiding the imbalance in inverted list sizes (e.g., HNSW) [10], [27] for improved performance. Our work is orthogonal to these and compatible with any type of index. In addition, we will discuss in the evaluation the fact that Quicker ADC fits well with the latest indexes (e.g., HNSW).

In order to fit the complete database into memory, short codes, which are much more compact, are stored instead of full vectors. To obtain a short code, the residual $\mathrm{r}(x) = x - \mathrm{q_i}(x)$ is encoded using a product quantizer described in the next section. Indexed databases therefore use two quantizers: a quantizer for the index ($\mathrm{q_i}$) and a product quantizer to encode residuals into short codes. The energy of residuals $\mathrm{r}(x)$ is smaller than the energy of input vectors x, thus there is a lower quantization error when encoding residuals rather than input vectors x into short codes. In the rest of the paper, we will note $y = \mathrm{r}(x)$. As a special case, when product quantization is used without any index, all vectors are stored in a single list. The short codes represent the input vectors rather than the residuals (i.e., $y = x$).

2.1.2 Short codes with product quantization

Vector quantizers. To encode residual vectors as short codes, PQ builds on vector quantizers. A vector quantizer q maps a vector $y \in \mathbb{R}^d$, to a vector $c \in \mathcal{C} \subset \mathbb{R}^d$. Vectors c are called *centroids*, and the set of centroids $\mathcal{C}$, of cardinality k, is the *codebook*. Generally, the quantizer is chosen so that it maps the vector y to its closest centroid c

$$\mathrm{q}(y) = \arg\min_{c \in \mathcal{C}} ||y - c||$$

The quantizer extends as an encoder e which encodes y into the index $i \in \{0 \dots k-1\}$ of the vector $c_i \in C$ it is mapped onto (i.e., $\mathrm{e}(y) = i$, such that $\mathrm{q}(y) = c_i$). The short code i only occupies $b = \lceil \log_2(k) \rceil$ bits, which is typically much smaller than the $d \cdot 32$ bits occupied by a vector $y \in \mathbb{R}^d$ stored as an array of d single-precision floats (32 bit each).

To maintain the quantization error sufficiently low for ANN search, a very large codebook e.g., $k = 2^{64}$ is required. However, building such codebooks is not tractable both in terms of processing and memory requirements.

Product quantizers. Product quantizers overcome this issue by dividing a vector $y \in \mathbb{R}^d$ into m sub-vectors, $y = (y^0, \dots, y^{m-1})$. Each sub-vector $y^j \in \mathbb{R}^{d/m}$, $j \in \{0, \dots, m-1\}$ is quantized independently using a sub-quantizer q^j. Each sub-quantizer q^j has a distinct codebook $\mathcal{C}^j = (c_i^j)_{i=0}^{k-1}$ of cardinality k. The cardinality of the product quantizer codebook $\mathcal{C} = \mathcal{C}^0 \times \dots \times \mathcal{C}^{m-1}$ is k^m. Thus, a product quantizer has many centroids k^m while only requiring storing and training m codebooks of cardinality k. A product quantizer encodes a vector y into a short code, by concatenating codes produced by sub-quantizers $\mathrm{e}(y) = (i_0 \dots i_{m-1})$, such that $\mathrm{q}(y) = (\mathrm{q}^0(y^0) \dots \mathrm{q}^{m-1}(y^{m-1})) = \left(c_{i_0}^0 \dots c_{i_{m-1}}^{m-1}\right)$. The code $(i_0, \dots, i_{m-1})$ requires $\lceil \log_2(k^m) \rceil$ bits of storage.

Interestingly, the order of vectors in the codebook C^j is not constrained. Thus one can choose how vectors of the codebook are mapped to indexes. This freedom allows storing additional information and has been used in [1] to nest a 4-bit product quantizer into the 8-bit product quantizer and in [12] to encode a binary code onto the index. In both case, this allows pruning the computation thanks to the approximation of distance provided by the nested code. With binary code, the resulting combination is called *polysemous codes* [12] and achieves state-of-the-art performance for approximate nearest neighbor search with product quantization (and inverted multi-indexes).

2.1.3 Search in the compressed domain

Search for the nearest neighbor computes the distance between the query vector z and a subset of database vectors.

As a first step, the a closest cells of the index quantizer q_i are determined (typically, $a = 8$ to 64 for IVFs). The inverted lists of these cells correspond to all the candidate vectors for which the distances must be computed. For efficiency reasons, the distance is computed directly on the compressed representation using a procedure called ADC.

ADC works the following way. First, for each cell, the residual $z' = \mathrm{r}(z)$ of the query vector is computed ($z' = z$ if no index is used). From this residual z', a set of m lookup tables are computed $\{D^j\}_{j=0}^m$, where m is the number of sub-quantizers of the product quantizer. The jth lookup table

TABLE 1
Instruction set capabilities

Instruction set	Instruction	Table size	Distance	Lookups per operation	Latency, Rec. Througput	Available since
SSE	`pshufb`	$16 = 2^4$	8 bit	16	1,1	2004 (Prescott)
AVX2	`pshufb`	$16 = 2^4$	8 bit	32	1,1	2013 (Haswell)
AVX512 BW	`pshufb`	$16 = 2^4$	8 bit	64	1,1	2017 (Skylake SP)
AVX512 BW	`vpermw`	$32 = 2^5$ *	16 bit	32	4,2	2017 (Skylake SP)
AVX512 BW	`vpermi2w`	$64 = 2^6$ **	16 bit	32	7,2	2017 (Skylake SP)
AVX512 VBMI	`vpermb`	$64 = 2^6$	8 bit	64	3,1	Exp. 2019 (Cannonlake)
AVX512 VMBI	`vpermi2b`	$128 = 2^7$ **	8 bit	64	5,2	Exp. 2019 (Cannonlake)
Neon	`vtbl`	$16 = 2^4$	8 bit	8	3,2 (Cortex A53, A72)	2009(ARMv7)
Neon	`vtbl`	$32 = 2^5$	8 bit	8	6,2 (Cortex A53, A72)	2009(ARMv7)
VMX/AltiVec	`vperm`	$16 = 2^4$	8 bit	16	Depend	2006 (PowerISA v2.03)

* Can be used for 16-values (2^4) table with 16-bit distances by zeroing the upper half of the table. ** Store the lookup table in two registers.

comprises the distance between the jth sub-vector of z' and all centroids of the jth sub-quantizer:

$$D^j = \left(\left\| z'^j - \mathcal{C}^j[0] \right\|^2, \ldots, \left\| z'^j - \mathcal{C}^j[k-1] \right\|^2 \right) \quad (1)$$

Second all candidates are scanned and the lookup tables are used to compute the distance between the query vector z and each short code c as follows:

$$\mathrm{adc}(z, c) = \sum_{j=0}^{m-1} D^j[c[j]] \quad (2)$$

Thus, ADC computes the distance between a query vector z and each code c by summing the distances between the sub-vectors of z' and centroids associated with code c in the m sub-spaces of the product quantizer. As the number of codes in inverted lists is large compared to k, the number of centroids of sub-quantizers, using lookup tables avoids computing $\|z'^j - \mathcal{C}^j[i]\|^2$ for the same i multiple times. Also, lookup tables provide a significant speedup by performing the computation directly in the compressed domain rather than reconstructing (i.e., decompressing) database vectors.

2.1.4 Impact of PQ parameters

m, the number of sub-quantizers and k, the number of centroids of each sub-quantizer impact: (1) the memory usage of codes, (2) the recall of ANN search and (3) search speed. The first tradeoff is between memory usage and accuracy. Both the memory usage of codes ($\lceil \log_2(k^m) \rceil$ bits = $m \cdot b$ bits, where $b = \lceil \log_2(k) \rceil$) and accuracy increase with the total number of centroids of the product quantizer (k^m). In practice, 64-bit or 128-bit codes are used in most cases.

The second tradeoff is between accuracy and search speed. For a constant memory budget of $m \cdot b$ bits per code, the respective values of m and b impact accuracy and speed. Decreasing m, which implies increasing b, increases accuracy [19]. A detailled analysis of the impact of m and b on performance is given in [1]. In a nutshell, b impacts the time for each lookup in the table: if b is too large, the lookup table does not fit into the fastest memory (i.e., the processor cache, which is limited in capacity) and lookup time will increase significantly. m impacts the number of lookups: thus as long as the table stays in the fastest memory, the lower the m, the better the performance. In addition, the cost of computing the lookup tables grows exponentially with b, thus smaller b also impact performance by reducing this cost; this is best seen on fine-grained indexes where the table computation cost becomes significant.

The standard notation for Product Quantization codes is $m \times b$ which specifies both parameters. The most common Product Quantization codes are $m \times 8$ (e.g., 8×8 or 16×8) as they ensure that tables fit in the processor cache and are not too costly to compute while being efficient to compute as they align well to computer bytes, and allow accessing the code without shifting nor masking.

2.2 ADC computation using SIMD

The common parameterizations of PQ (i.e., $m \times 8$) already exploit the fastest memory available on processors (i.e., the L1 cache), leaving no room for easy improvement. A common technique to improve performance is to use instructions that process a vector of values rather than a single value at each CPU cycle: this principle called SIMD *Single Instruction Multiple Data* allows significant performance boost for signal processing and matrix operations. Yet, for PQ, moving to SIMD improves performance for additions but the implementation remains bottlenecked by the memory accessed [1]. In [1], the sequential implementation of ADC is analyzed thoroughly and compared to various approaches using in-memory lookup tables but with SIMD additions. The improvement is limited as the bottleneck is the access to memory and specifically the in-memory lookup tables as the codes, which are sequentially read, are efficiently handled by the hardware prefetchers of the processor. Indeed SIMD does not allow an efficient implementation of in-memory table lookups, even using `gather` instructions introduced in recent processors [1], [15]. While SIMD can add up to 16 floating-point numbers (512 bits) at once, only 2 concurrent memory accesses can be performed per cycle in each CPU core. Those memory accesses are the bottleneck.

Thus, previous work [1], [2], [11], [32] moved lookup tables from memory to SIMD registers and leveraged in-register shuffles to implement lookups[1]. Yet, the width of SIMD registers (128-512 bits) challenges this approach. Indeed, for common PQ (i.e., $m \times 8$), each lookup table occupies 8192 bits ($k = 2^8 = 256$ floats). Previous work worked around this limit by (i) using 4-bit subquantizers and (ii) quantizing floats to 8-bit integers. The resulting lookup

1. Note that each core includes its own SIMD unit. We leave aside the use of multiple cores, as this is achieved easily by having multiple threads processing independent queries in parallel.

tables are thus small enough (128 bits) to fit SSE or AVX-2 registers. This has allowed Quick ADC [2] to achieve a significant performance improvement with a moderate loss of accuracy. This loss of accuracy comes mainly from the reduced precision of 4-bit subquantizers when compared to 8-bit subquantizers and to a smaller extent from the use of quantized distances. Yet, [2], [11], [32] remain limited to $m\times4$ as they follow the initial approach of [1] that use `pshufb`.

2.3 SIMD capabilities and alternatives

AVX512, introduced in 2017 with Xeon Scalable processors, is a significant redesign of Intel's SIMD instruction set. It provides additional shuffle instructions that support larger tables as described in Table 1. Available processors allow lookup tables of 32 or 64 16-bit values, thus 5-bit or 6-bit indexed lookup tables (i.e., $m\times5$ or $m\times6$ PQ codes). In addition, the wider registers (4× when compared to SSE and 2× when compared to AVX2) allow either an improved parallelism or more precise distances (16-bit instead of 8-bit). In the next section, we will explore how these new capabilities can be exploited for improving PQ efficiency.

Interestingly, shuffle instruction `pshufb` (used in, e.g., $m\times4$ PQ codes) can process between 128 (SSE) and 512 (AVX-512) bits of data per cycle while the `popcount` instruction (binary codes) can process only 64 bits of data per cycle. Hamming distance (used in, e.g., polysemous codes) is considered as much faster than product quantization's ADC (i.e., with in-memory lookup tables) thanks to these fast `popcount`. Yet, `pshufb` is even faster[2] allowing product quantization to rival binary codes regarding distance computation speed. As a side note, our paper focuses on ADC, yet hamming distance is conceptually closer to SDC [19]; SDC would trade accuracy for speed by avoiding costs associated to distance table computation. This could provide small product quantization codes competing directly with binary codes.

While AVX512 has improved capabilities (larger tables) and increased parallelism (i.e., number of lookups per cycle) thanks to wider registers, the gain obtained may be partially cancelled by the fact that processor cores running AVX2 or AVX512 code are down-clocked and thus run at a lower frequency than processor cores running sequential or SSE code [16]. Thus, an experimental evaluation is needed to assess the potential of these instructions for PQ-based applications. This is even more salient for non-exhaustive search as the overall query time is the result of the index search, the distance table computation if any, and the ADC-based scanning of all candidates whose performance change differently as the code or index is altered.

The work presented in this paper extends to ARM (Neon) and PowerPC (VMX) processors. Even if they have different instruction sets, they have similar limitations regarding the maximum number of concurrent memory accesses. They feature SIMD units with a 4-bit shuffle equivalent to `pshufb` and saturated arithmetics necessary to support Quicker ADC's $m\times4$ codes. Beyond CPUs, the tradeoff between lookup tables in memory and registers should be taken into account when designing programs for GPUs or FPGAs because of fundamental hardware design tradeoffs between memory size and access speed/parallelism.

3 Quicker ADC

In this section, we describe Quicker ADC, a generalization of Quick ADC [2] that aims at improving accuracy through the use of the latest AVX-512 SIMD instructions. Interestingly, AVX-512 provides shuffles indexed by 5 or 6 bits, thus allowing a more precise quantization of vectors. Yet, their use for ADC is not straightforward as practical constraints prevent the use of $m\times5$ or $m\times6$ PQ codes.

Product quantizers $m\times8$ or $m\times16$, that are commonly used, are composed of 8-bit or 16-bit sub-quantizers. This choice stems from the fact that manipulating byte-aligned or word-aligned values is both simpler and faster. Earlier work on using SIMD for PQ departs from these choices by relying on $m\times4$ PQ codes yet benefits from the fact that 4-bit values naturally align to bytes as $2\times4=8$ bits. Unfortunately, $m\times5$ or $m\times6$ PQ codes rely on 5-bit or 6-bit values, neither of which align well to computer words (16 bits) or bytes (8 bits). This prevents a computationally efficient implementation of such PQ codes which is the purpose of using SIMD.

A naive approach could be to add padding by storing 3×5 bits of PQ codes in each 16-bit word and leaving one unused bit in each word. Our initial experiments with this approach on an exhaustive search in the SIFT1M dataset show that a 16×4 PQ (R@1[3] of 0.159) outperforms a 12×5 PQ codes (R@1 of 0.153). Our hypothesis is that using only 60 bits out of 64 bits outweights the benefits of using 5-bit subquantizers in place of 4-bit subquantizers.

To this end, we explore two solutions to avoid padding. The first solution, relying on new Irregular PQ codes, is presented in Section 3.1. The second solution combines multiple large shuffles to implement each lookup for 8-bit subquantizers; it anticipates the availability of AVX512 VBMI 7-bit shuffle in a near future. Both solutions require an SIMD-compatible memory layout similar to the one of Quick ADC [2] and described in Section 3.3. Finally, Quicker ADC improves the distance quantization of Quick ADC [2] in order to allow using all 8 bits and not just 7 bits for improved accuracy as explained in Section 3.4.

3.1 Irregular PQ

As a first solution to alleviate the alignement issue, we propose Irregular Product Quantization which combines subquantizers of different sizes (4,5 and 6 bits) that can all be implemented in SIMD, such that their combination aligns well to words (16-bits). A first variant groups one 6-bit subquantizer with two 5-bit subquantizers for a total of 16-bits and a second variant groups two 6-bit subquantizers with one 4-bit subquantizers also for a total of 16 bits. Multiple such groups are combined to form the complete Product Quantizer. We will use the notation $m\times\{a,b,c\}$ for

2. Mula et al. [29] also noticed that `popcount` is relatively inefficient when compared to vectorized instructions and thus designed algorithms to replace `popcount` by a few vectorized instructions including one or two `pshufb` for long-enough bit vectors. Yet, this does not allow any accuracy improvement contrary to Quicker ADC.

3. R@1 designates the recall of the closest vector when considering only the top vector returned by the algorithm. R@10 or R@100 are relaxed versions where the closest vector is considered as recalled if found in the top 10 vectors or top 100 vectors returned by the algorithm.

TABLE 2
SIMD operations required to perform 8-bit lookups

Instruction set	Native Shuffle	Op. count Shuffle	Op. count Blend	Values per register	Ops per value
AVX2 / SSE4	4-bit	16	15	32 / 16	0.97 / 1.94
AVX512 BW	6-bit	4	3	32	0.21
AVX512 VBMI	7-bit	2	1	64	0.05

an Irregular Product Quantizer formed of m sub-quantizers grouped by three sub-quantizers of a bits, b bits and c bits. For example, the rest of the paper will often consider the 64-bit product quantizer $12\times\{6,6,4\}$ that has the following subquantizers $[6,6,4,\ \ 6,6,4,\ \ 6,6,4,\ \ 6,6,4]$. We note g the number of sub-quantizers grouped (e.g., $g=3$ for $m\times\{6,6,4\}$ and $g=2$ for $m\times\{4,4\}$); so that m/g is the number of groups.

With subquantizers of different precisions, the allocation of input dimensions to subquantizers cannot be uniform. With product quantizers [19], the input vector is split in sub-vectors of equal dimension which are then quantized independently. With irregular product quantizers, however, the input vector is not split in sub-vectors of equal dimensions so as to leverage the improved representation capabilities of finer subquantizers. We map input dimensions proportionally to the bit-width of the subquantizer. Let us consider a 128-dimension SIFT vector quantized by an irregular product quantizer $12\times\{6,6,4\}$. Each of the 4 groups of sub-quantizers is associated to 32 dimensions. Thus within each group, 6-bit sub-quantizers quantize sub-vectors of 12 dimensions and 4-bit subquantizers quantize sub-vectors of 8 dimensions. Note that as long as m/g is a multiple of 2, Irregular Product Quantizers remain compatible with multi-indexes which requires the two subquantizers of the index to be aligned with the subquantizers of the product quantizer.

If the number of input dimension cannot be divided exactly, the remaining dimensions are added one by one to the subquantizers. Yet, this alters performance and should be avoided: one should always ensure that allocation can be proportional. More specifically, when vectors are pre-processed by a PCA or a rotation (like in OPQ), the pre-processed vectors should be of a dimension which can be divided exactly. This implies that for a $m\times\{6,6,4\}$, the number of dimensions of the pre-processed vectors must be divisible by $(3+3+2)m/3=8m/3$, and for a $m\times\{6,5,5\}$ the number of dimension of the pre-processed vector must be divisible by $(6+5+5)m/3=16m/3$. For example, 128-dimension SIFT vectors can be encoded optimally by both $12\times\{6,6,4\}$ or $12\times\{6,5,5\}$ irregular PQ codes.

To validate this first solution, we compare the different 64-bit codes using an exhaustive search in the SIFT1M dataset. Both $12\times\{6,6,4\}$ (R@1 of 0.179) and $12\times\{6,5,5\}$ (R@1 of 0.174) outperform a 16×4 PQ code (R@1 of 0.158). Additional results are given in the evaluation in Table 3.

3.2 Split tables

AVX-512 brings 6-bit `vpermi2w` and 7-bit `vpermi2b` shuffles. These are only 2-bit away and 1-bit away from the very common 8-bit sub-quantizers. Rather than relying on highly-imbalanced irregular product quantizers, in particular a $m\times\{7,1\}$ that would perform poorly, it becomes interesting

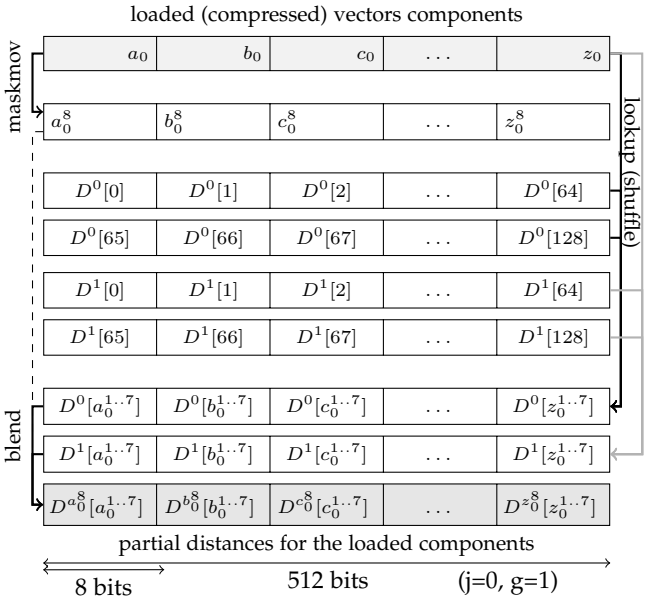


Fig. 1. Shuffle-blend lookup for $8\times\{8\}$ (AVX-512 VBMI)

to consider that each 8-bit lookup table can be split into 4 6-bit lookup tables or 2 7-bit lookup tables. As an example, we show an 8-bit lookup built from two 7-bit shuffles on Figure 1. The distance table D is split in two halves (D^0 and D^1, each of which occupies 2 registers). Two shuffles (each having 3 registers as inputs) are performed to get values indexed by the low 7-bits (e.g., $a_0^{1..7}$)). The final values are selected through a blend indexed by the 8-th bit (e.g., a_0^8).

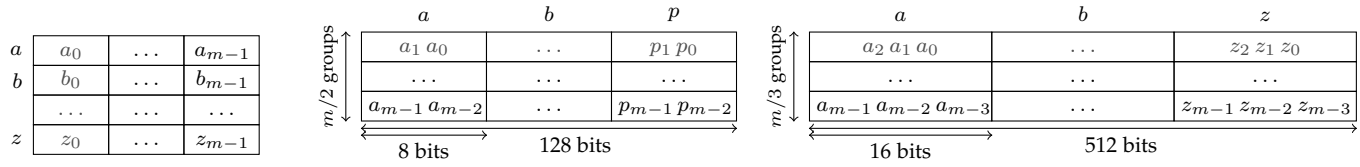
This approach cannot be built on 4-bit shuffle (`pshufb` from SSE4/AVX2) as it would require too many shuffles and blends. As shown in Table 2, performing an 8-bit lookup for 16 values using SSE4 requires 16 shuffles and 15 blends, an average of 1.94 operations/value. In comparison, when using AVX512 VBMI, only 0.05 operations/value are necessary. Due to the lack of available processors with AVX512 VBMI, we will evaluate performance only for AVX512 BW (0.21 operations/value); yet, it is clear that `vpermi2b` from AVX512 VBMI will provide significant gains[4].

Quicker ADC codes implemented with this approach require more instructions per lookup than irregular product quantizers, yet, they come with almost no compromises on accuracy when compared to PQ $m\times8$ PQ codes. In the evaluation, we'll see that in some context they outperform the alternative approach already, but they will become particularly interesting as AVX512 VBMI-capable processors become available in the near future.

3.3 Memory layout

Similarly to Quick ADC, Quicker ADC requires a transposed memory layout. Indeed, an SIMD in-register shuffle performs multiple lookups at once, but in a *single* lookup table e.g., D^0. Therefore, shuffles must operate on a single component of multiple codes (e.g., $a_0,\ldots,p_0$) at once, and not on multiple components of a single code (e.g., $a_0,\ldots,a_{15}$). Hence, to allow efficient loads from memory, all values of the SIMD

4. The software we release already includes an implementation supporting AVX512 VBMI even if it couldn't be evaluated yet.



(a) Standard layout ($m\times8$ or $m\times16$)

(b) Transposed layout ($m\times\{4,4\}$) QuickerADC with pshufb, 8-bit distances

(c) Transposed layout ($m\times\{6,6,4\}$) QuickerADC with vpermw/i2w, 16-bit distances

Fig. 2. Inverted list memory layouts. Each table cell represents a byte or a word.

register $a_0, \ldots, p_0$ must be *contiguous* in memory, which is not the case with the memory layout of inverted lists (Figure 2a) used in common PQ implementations [12], [19].

The size of the blocks transposed depends on the shuffle used. With 4-bit shuffles (`pshufb` in SSE4/AVX2/AVX512) operating on lanes of 128 bits composed of 16 values, we transpose blocks of 16 codes (a–p) as shown on Figure 2b. With 5-bit and 6-bit shuffles (`vpermw` and `vpermi2w` in AVX512) operating on a single lane of 512 bits composed of 32 values, we transpose blocks of 32 codes (a–z) as shown on Figure 2c. With 7-bit shuffles (`vpermi2b` from AVX512) operating on 64 values, we transpose blocks of 64 codes. A more straightforward approach would transpose to blocks of the size of a register (rather than the size of a lane), but this would hinder performance due to increased register pressure. Also for smaller sets of vectors encountered in inverted indexes, this would require additional padding resulting in lower performance.

In addition, values have a fixed width that depends on the shuffle instruction used: `pshufb` and `vpermb` operate on bytes (8 bits) while `vpermw` and `vpermi2w` operate on words (16 bits). Hence, multiple subcodes must be packed together to form bytes or words. We use the notation of irregular product quantizers to specify the packing applied (e.g., $m\times\{6,6,4\}$ packs 3 subcodes to form a 16-bit word ($6+6+4=16$) as shown on Figure 2c). This notation extends to regular product quantizers (e.g., Quick ADC [2]'s $m\times\{4,4\}$ packs 2 4-bit subcodes to form an 8-bit byte ($4+4=8$) as shown on Figure 2b). Split-table-based $m\times\{8,8\}$ packs two 8-bit subcodes to form a word for use with `vpermi2w`, and split-table-based $m\times\{8\}$ packs a single 8-bit subcode to form a byte for use with `vpermi2b`. Subcodes can be extracted from the packed values efficiently in SIMD through shift and mask. Note that the number of rows in the transposed layout on Figure 2 corresponds exactly to the number of groups in irregular PQ codes.

3.4 Quantization of distances

In standard ADC, lookup tables store partial distances as 32-bit floats. As subquantizer precision is key to accuracy, we seek to store partial distances as 8-bit or 16-bit integers so as to allow lookup tables of the same size, yet storing twice or four times as much values. The representation (8-bit or 16-bit integers) depends on whether the type of shuffle we use operates on bytes (e.g., `pshufb`, `vpermi2b`) or words (e.g, `vpermw`, `vpermi2w`).

As we are interested only in the top-k nearest neighbors, our distance quantization scheme must represent as precisely as possible the smallest distances, but can ignore (i.e., quantize to ∞) large distances. Thus, to perform distance computations (additions, ...), we rely on saturated integer arithmetics that handles ∞ through saturation. The approach is thus similar to that of Quick ADC but provides a tighter distance quantization as explained hereafter.

First, we use 8-bit and 16-bit *unsigned* integers whereas Quick ADC uses only the 7-bit positive range of *signed* integers: this doubles the precision. Note that AVX2 lacks straigtforward instructions for unsigned comparisons [23].

Second, we perform a tighter evaluation of the minimum and maximum values than in Quick ADC [2] in order to allow a more precise quantization. For each of the m lookup tables, we evaluate the smallest partial distance $p_{\text{min}}(i)$ to represent, which is the smallest partial distance in the i-th table. This also gives us the smallest distance to represent $d_{\text{min}} = \sum_{i=0}^{m} p_{\text{min}}(i)$. Then, we scan t vectors to find a candidate set of R nearest neighbor candidates, where R is the number of nearest neighbors requested by the user (e.g., R=100 when R@100 is evaluated) and t is larger than R but much smaller than the total number of vectors. We use the distance of the query vector to the R-th nearest neighbor candidate i.e., the farthest candidate, as the d_{max} bound. All subsequent candidates have to be closer to the query vector, thus d_{max} is the maximum distance we need to represent.

We determine the size of quantization bins $\Delta = \frac{d_{\text{max}} - d_{\text{min}}}{q_{\text{max}}}$. Partial distance p in the i-th table can thus be quantized as

$$q = \frac{p - p_{\text{min}}(i)}{\Delta}$$

To unquantize the sum Σq, one can use:

$$d = (\Sigma q)\Delta + d_{\text{min}}$$

Note that similarly to [1], [2], we learn our distance quantizer at query time on summed distances from the top-k rather than at training time on partial distances (i.e., values in the distance tables) [11]: the required evaluation of a few distances has a negligible impact on performance yet allows consistently increased accuracy. Also, contrary to [11], we limit distortion only for the shortest distances, whereas in [11], the distortion is limited for all partial distances, but all the shortest distances (i.e., below the first quantile) are quantized into a single bin resulting in higher distortion for these. Finally, as our $q_{\texttt{max}}$ bound is determined from summed distances, we can perform the accumulation on the same integer width as the lookups (i.e., no need to upcast 8-bit distances to 16-bit to avoid saturation) and preserve the semantic of $q_{\texttt{max}}$ (i.e., too large distance) during accumulation.

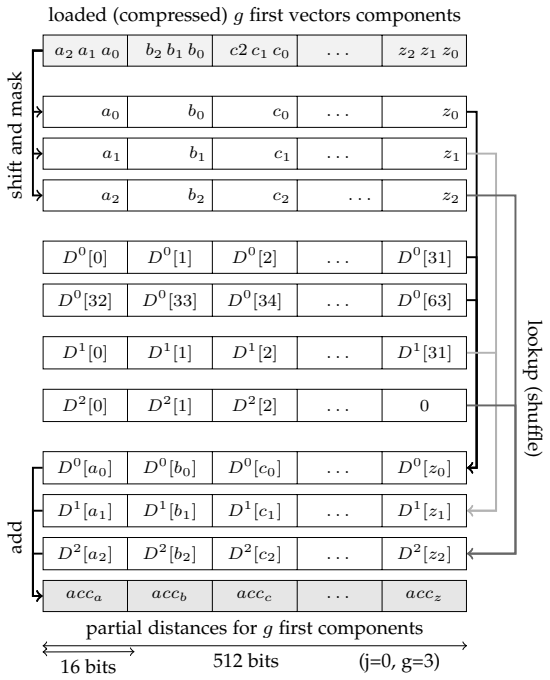


Fig. 3. SIMD Lookup-add in AVX-512 for $12\times\{6,5,4\}$

3.5 SIMD distance computation

Quicker ADC supports several combinations of sub-quantizers: (i) those operating on 128-bit lanes for 4-bit subquantizers with 8-bit distances (SSE, AVX2 and AVX512), (ii) those operating on 512 bits lanes for 4,5 and 6-bit subquantizers with 16-bit distances, and (iii) those operating on 512 bits lanes for 8-bit subquantizers with 8 or 16-bit distances implemented using multiple shuffles and blends as described in Section 3.2 and Figure 1. While implementation details vary, the overall principle is the same.

Once cells are selected and distance tables are computed, as explained in the background, each invert list is scanned block by block. The distances for vectors of the block are computed in the following way. We depict the processing applied to each group of components (i.e., each row of Figure 2b and 2c) in Figure 3. First, the subcodes are unpacked using shifts and masks. For each set of subcodes, the partial distances are looked up in the distance table using either a native shuffle, as explained in Section 3.1 or a lookup implemented through a combination of shuffle and blends, as explained in Section 3.2 and depicted in Figure 1. The distances are summed using saturated arithmetic. Note that distance tables may occupy half, one or multiple registers depending on the type of lookup. Figure 3 presents an hypothetical $12\times\{6,5,4\}$ code in which distance tables for 6-bit, 5-bit and 4-bit subquantizers fit in respectively 2, 1 and half a register. This process is repeated for all m/g groups that form the complete code and partial distances are summed to obtain the distances. The distances are compared to the worst candidate vector, and vectors for which distances are smaller are added to the binary heap of candidates.

The 8-bit shuffles (`pshufb`) in AVX2 or AVX-512 BW operate on two or four independent 128-bit lanes. Hence, they do not allow 32 or 64-element lookup tables. Yet, they can increase the throughput by processing more elements per cycle (see Table 1). We use this property in Quicker ADC ($m\times\{4,4\}$) by processing multiple groups (i.e., rows) per shuffle instruction (2 for AVX2, 4 for AVX512) rather than just one per call as in SSE. The number of iterations in the computation is thus reduced from m/g to $m/g/2$ (AVX2) or $m/g/4$ (AVX-512). This also reduces register pressure as fewer registers are needed to store distance tables.

4 Evaluation

We implemented Quicker ADC in C++ (4K lines of code) and release it as open-source[5]. The implementation contains numerous variants: $m\times\{4,4\}$, $m\times\{6,6,4\}$, $m\times\{6,5,5\}$, $m\times\{5,5,5\}$, $m\times\{8,8\}$, $m\times\{8\}$[6]. Note that, to allow further experimentation, the released code is highly generic thanks to templates so that adding a $m\times\{6,6,2\}$ operating on signed arithmetics requires a single line of code. Training, exhaustive search and non-exhaustive search (IVF, multi-indexes) rely on the implementation of FAISS.

We carry our experiments on Skylake-based servers, which are m5 AWS instances, built around Intel Xeon Platinum 8175 (2.5 GHz, supporting AVX512) with default settings from Amazon. We use g++ compiler version 7.3 with option `-O3` and enable SSE, AVX, AVX2 and AVX512. For BLAS, we use the Intel MKL 2018. We focus our evaluation on Intel's processors as they power almost all servers. Given that optimized implementations are key to relevant evaluations, we also release our source code, to ease evaluation of SIMD schemes on future processors and newer micro-architectures, and for improved quantization schemes.

Our evaluation relies on the publicly available datasets SIFT1M [19] and SIFT1000M [20] of 128-dimension SIFT vectors, Deep1M [7] of 256-dimension of Deep features, and Deep1B [8] of 96-dimension Deep features. We use the 1-million vectors datasets for the evaluation of exhaustive search, and the 1 billion vectors datasets for the evaluation of non-exhaustive search (i.e., with an index).

Our metrics are Recall@1 (R@1), which is the fraction of queries for which the true neighbor is the one returned during search, and Recall@100 (R@100), which is the fraction of queries for which the true neighbor is among the top-100 returned during search. R@100 reflects the performance for visual search applications where the user is presented a collection of images rather than a single image (e.g., Google Image). To evaluate the computational efficiency, we report the average time per query. Note that a query time of 0.5ms translates to a throughput of 2,000 queries/second (/core).

4.1 Exhaustive search

We first focus on evaluating the performance of Quicker ADC in isolation. We thus consider exhaustive search on the SIFT1M and Deep1M datasets. We do not use an inverted index and we encode the original vectors, not residuals, into

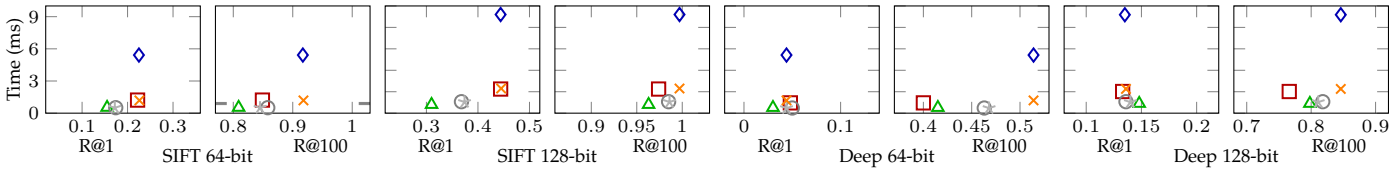
5. Our implementation integrated into FAISS is released under the Clear BSD license at https://github.com/nlescoua/faiss-quickeradc

6. In our evaluation, we execute this last implementation in a non-SIMD fallback mode as processors with AVX512 VBMI processors are not available yet. The released code, nevertheless, contains the optimized SIMD implementation. Our implementation, based on templates, allows adding additional/newer instruction sets easily as it automatically generates code regarding the specific memory layout.

TABLE 3
Exhaustive search (without index) on SIFT1M and Deep1M. Results are given as percentage R@1/R@100 *time(ms)*

Results as: R@1/R@100(%) *time(ms)*				***SIFT1M***						***Deep1M***					
Code	Instr. Set	Instr.	Dist.	64 bits		128 bits		256 bits		64 bits		128 bits		256 bits	
PQ $m\times 8$ **[18], [19])**	◇		float	22.5/91.7	*5.42*	44.4/99.7	*9.20*	62.7/100.0	*17.2*	4.40/51.4	*5.42*	13.5/84.6	*9.18*	31.2/100.	*17.2*
Polysemous [12], [18])*	□	popcnt	bin/f.	22.2/84.9	*1.22*	44.4/97.4	*2.25*	60.9/93.4	*4.15*	4.80/40.0	*0.96*	13.3/76.6	*2.02*	30.6/93.3	*3.60*
Hamming-only (τ=0)		popcnt	bin.	0.00/0.00	*0.87*	0.00/0.00	*1.30*	0.00/0.00	*3.01*	0.00/0.00	*0.85*	0.00/0.00	*1.41*	0.00/0.00	*2.97*
Quicker ADC															
$m\times\{4,4\}$	SSE	pshufb	int8	15.5/80.6	*0.68*	30.6/96.0	*1.25*	46.5/99.8	*4.26*	2.9/41.8	*0.67*	14.0/78.2	*1.32*	30.0/97.3	*4.28*
$m\times\{4,4\}$	SSE	pshufb	uint8	15.5/80.9	*0.69*	31.0/96.3	*1.21*	49.9/99.9	*4.40*	3.0/41.5	*0.68*	14.8/79.8	*1.31*	32.4/97.1	*4.45*
$m\times\{4,4\}$	AVX2	pshufb	int8	15.5/80.6	*0.51*	30.6/96.0	*0.83*	46.5/99.8	*2.48*	2.9/41.8	*0.51*	14.0/78.2	*0.92*	30.0/97.3	*2.45*
$m\times\{4,4\}$	AVX2	pshufb	uint8	15.5/80.9	*0.54*	31.0/96.3	*0.85*	49.9/99.9	*2.43*	3.0/41.5	*0.54*	14.8/79.8	*0.95*	32.4/97.1	*2.48*
$m\times\{4,4\}$	AVX512 BW	pshufb	int8	15.5/80.6	*0.49*	30.6/96.0	*0.78*	46.5/99.8	*2.26*	2.9/41.8	*0.49*	14.0/78.2	*0.86*	30.0/97.3	*2.17*
$m\times\{4,4\}$	AVX512 BW △	pshufb	uint8	15.5/80.9	*0.53*	31.0/96.3	*0.80*	49.9/99.9	*2.28*	3.0/41.5	*0.52*	14.8/79.8	*0.90*	32.4/97.1	*2.26*
$m\times\{4,4,4,4\}$	AVX512	vpermw	uint16	15.7/80.9	*0.84*	31.6/96.5	*1.63*	51.2/100.	*3.98*	3.1/42.1	*0.84*	14.6/79.9	*1.73*	33.6/97.5	*3.87*
Quick ADC [1]	AVX2	pshufb	int8	15.2/80.2	*0.51*	30.1/95.7	*0.82*	46.2/99.7	*2.42*	3.1/40.9	*0.5*	13.7/77.1	*0.91*	27.2/95.3	*2.36*
Bolt[16] [11]	AVX2	pshufb	uint8	12.8/77.0	*0.73*	29.7/95.6	*1.34*	50.7/100.	*3.45*	3.1/41.8	*0.73*	14.6/80.3	*1.41*	33.3/97.5	*3.42*
Bolt[8] [11]	AVX2	pshufb	uint8	12.5/66.7	*0.46*	15.0/32.2	*0.72*	8.5/12.2	*2.28*	0.1/0.5	*0.43*	0/0	*0.82*	0/0	*2.30*
$m\times\{4,4\}$			float	15.7/80.9	*25.6*	31.7/96.5	*54.0*	51.2/100.	*104*	3.0/42.1	*25.6*	14.6/79.8	*54.0*	33.6/97.5	*104*
$m\times\{6,6,4\}$	AVX512 BW	vpermi2w	int16	17.4/85.8	*0.51*	36.8/98.5	*0.98*	56.6/100.	*2.59*	4.8/46.5	*0.48*	13.6/81.8	*0.98*	34.4/98.5	*2.57*
$m\times\{6,6,4\}$	AVX512 BW ○	vpermi2w	uint16	17.4/85.8	*0.51*	36.8/98.5	*1.07*	56.7/100.	*2.57*	5.0/46.3	*0.48*	13.6/81.8	*1.07*	34.3/98.5	*2.58*
$m\times\{6,6,4\}$			float	17.4/85.8	*18.4*	36.8/98.6	*40.5*	56.5/100.	*77.9*	4.8/46.3	*18.4*	13.6/81.8	*40.4*	34.4/98.5	*77.9*
$m\times\{6,5,5\}$	AVX512 BW	vpermi2w	int16	17.1/84.5	*0.50*	37.5/98.6	*0.97*	56.8/100.	*2.54*	4.4/46.8	*0.48*	14.0/81.1	*0.97*	33.7/98.4	*2.5*
$m\times\{6,5,5\}$	AVX512 BW ☆	vpermi2w	uint16	17.2/84.5	*0.50*	37.4/98.6	*1.04*	56.8/100.	*2.55*	4.4/46.8	*0.47*	14.0/81.1	*1.03*	33.8/98.5	*2.47*
$m\times\{6,5,5\}$			float	17.1/84.5	*19.3*	37.4/98.6	*41.6*	56.8/100.	*79.5*	4.4/46.8	*19.2*	14.0/81.1	*41.6*	33.8/98.5	*79.6*
$m\times\{5,5,5\}$	AVX512 BW	vpermw	int16	14.9/80.5	*0.50*	34.4/97.6	*0.95*	53.2/99.9	*2.44*	2.8/39.3	*0.47*	12.0/77.0	*0.95*	29.7/98.5	*2.46*
$m\times\{5,5,5\}$	AVX512 BW	vpermw	uint16	14.9/80.5	*0.5*	34.4/97.6	*0.96*	53.2/99.9	*2.48*	2.9/39.3	*0.47*	12.0/77.0	*0.96*	29.7/98.5	*2.49*
$m\times\{5,5,5\}$			float	14.9/80.5	*19.3*	34.4/97.7	*41.6*	53.2/99.9	*79.5*	2.8/39.2	*19.1*	12.0/77.0	*41.6*	29.7/98.5	*79.6*
$m\times\{8,8\}$	AVX512 BW	vpermi2w	int16	22.5/91.8	*1.18*	44.5/99.7	*2.3*	62.8/100.	*4.89*	4.3/51.5	*1.19*	13.5/84.6	*2.27*	31.2/99.0	*4.9*
$m\times\{8,8\}$	AVX512 BW ×	vpermi2w	uint16	22.6/91.8	*1.19*	44.5/99.7	*2.35*	62.8/100.	*4.9*	4.4/51.4	*1.19*	13.6/84.6	*2.26*	31.2/99.0	*4.91*
$m\times\{8\}$	AVX512 VBMI	vpermi2b	int8	22.8/91.8	$<$ **	43.7/99.7	$<$ **	60.2/100.	$<$ **	4.3/51.6	$<$ **	13.7/84.6	$<$ **	30.6/99.0	$<$ **
$m\times\{8\}$	AVX512 VBMI	vpermi2b	uint8	22.3/91.6	$<$ **	42.5/99.5	$<$ **	55.6/100.	$<$ **	4.5/51.8	$<$ **	13.0/85.1	$<$ **	28.7/98.1	$<$ **
$m\times\{8\}$			float	22.5/91.8	*11.8*	44.4/99.7	*22.9*	62.7/100.	*47.0*	4.4/51.4	*11.8*	13.5/84.6	*22.8*	31.2/99.0	*47.0*

* For SIFT1M, $\tau = 21$ for 64-bit, $\tau = 51$ for 128-bit and $\tau = 99$ for 256-bit. For Deep1M, $\tau = 17$ for 64-bit , $\tau = 47$ for 128-bit and $\tau = 99$ for 256-bit.
** Timings for $m\times\{8\}$ cannot be obtained as current processors do not support AVX512 VBMI; recalls are obtained by simulating saturated arithmetic on quantized distances. As explained in Section 3.2, we expect AVX512 VBMI $m\times\{8\}$ to be significantly (up to 4 times) faster than AVX512 BW $m\times\{8,8\}$.



short codes. Table 3 gives results for the baseline product quantization implementation from FAISS [21], polysemous codes [12], Bolt [11], Quick ADC [1] and Quicker ADC. We also include a specific operating point where polysemous codes degenerate into binary codes ($\tau = 0$).

For Quick ADC [1] and Quicker ADC, we scan t=400 vectors to set the $q_{\texttt{max}}$ bound for the quantization of lookup tables (Section 3.4). We evaluate various SIMD-implementations and compare them to sequential implementations that use floating-point distances to evaluate the loss of recall due to distance quantization.

4.1.1 Impact of distance quantization

Distance quantization has no impact on recall with 16-bit integers and negligible impact with 8-bit integers (e.g., 0.155 instead of 0.157 for $m\times\{4,4\}$ for R@1 SIFT1M - 64 bit). Hence, scanning t=400 vectors is enough for estimating the distance-quantization bounds. The unsigned variants have a slight recall advantage over the signed ones. As query times for both are similar, we'll keep the unsigned variants.

Quicker ADC $m\times\{4,4\}$ with 8-bit distance is faster than $m\times\{4,4,4,4\}$ with 16-bit distances for similar recall. We thus prefer using 8-bit distances whenever an appropriate shuffle is available. Indeed, shuffling 16-bit values has a higher latency (see Table 2), converting 8-bit to 16-bit has a cost and the number of additions per cycle is divided by 2.

Regarding prior work, Quicker ADC $m\times\{4,4\}$ (AVX2) improves over Quick ADC [1], which also uses AVX2, with a better recall for the same query time thanks to the tighter $q_{\texttt{min}}$ bound. Quicker ADC outperforms Bolt[16] by providing better recall (thanks to the different distance quantization scheme) and by being faster (thanks to accumulating on 8-bit instead of upcasting partial distances to 16-bits). Note that Bolt[8] with an 8-bit accumulator has poor recalls as the quantization bounds are estimated on partial distances rather than summed distance, thus a lot of relevant distances saturate during accumulation.

As a side note, our sequential implementation tends to be slower than original the PQ ADC because it is not as specialized and it systematically uses shifting and masking for accessing subcodes. For $m\times\{4,4\}$, using SSE is slower than using AVX2 or AVX512 which give similar timings.

4.1.2 Comparing QuickerADC to PQ and Polysemous

Quicker ADC $m\times\{6,6,4\}$ is as fast as $m\times\{4,4\}$ yet improves recall for both 64-bit and 128-bit codes. Its recall R@1 (0.174 and 0.368 on SIFT1M 64-bit/128-bit) is lower than PQ and polysemous codes (0.225 and 0.444) but the recall R@100 of Quicker ADC is equivalent if not better than the one of polysemous codes. Yet, this slight recall decrease allows Quicker ADC $m\times\{6,6,4\}$ to be 10 times faster than PQ ADC and 2-3 times faster than polysemous codes. Interestingly, $m\times\{5,5,5\}$ has a lower recall than all other including $m\times\{4,4\}$ because it uses only 60 bits instead of all 64 bits. Hence, irregular product quantizers are preferable over simpler constructions not using all bits.

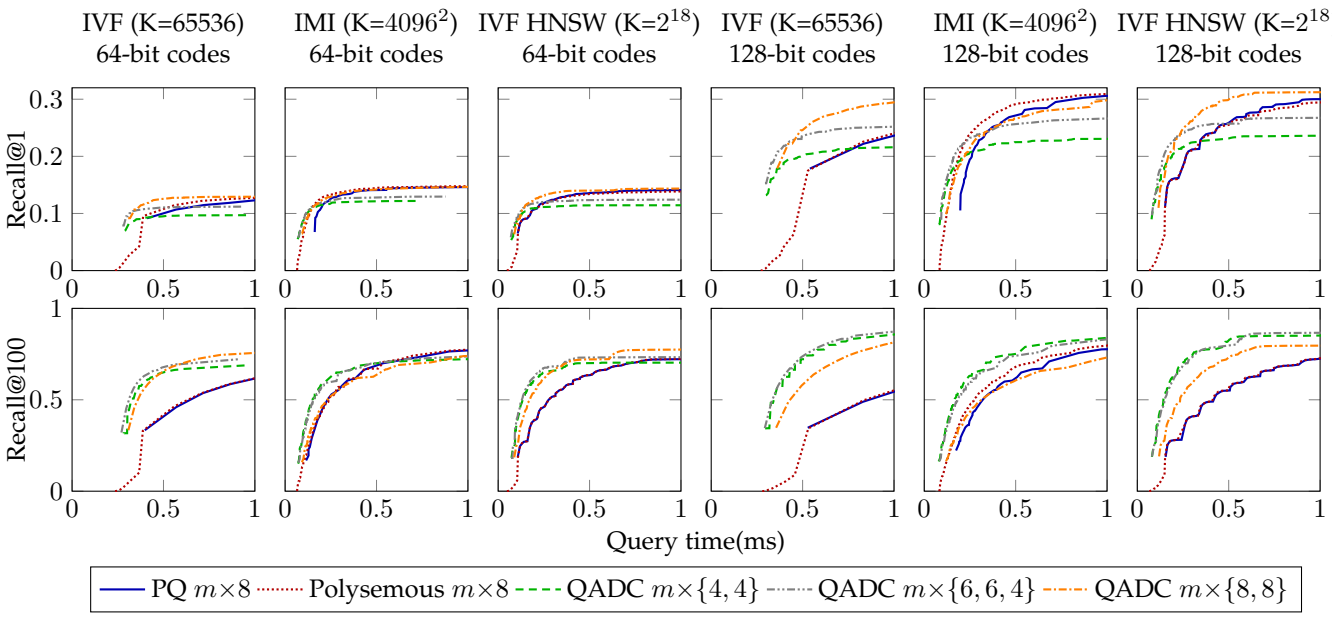


Fig. 4. Non-exhaustive search with different index types and different PQ or Irregular PQ codes. SIFT1000M

Quicker ADC $m\times\{8,8\}$ codes are 5× faster than regular PQ codes, and slightly faster than polysemous codes while achieving similar recall R@1 and improved recall R@100. Note that the timings reported here are using AVX512 BW and are likely to be up to 4× better with the availability of AVX512 VBMI without significant degradation of accuracy. Thus, $m\times\{8\}$ codes with Quicker ADC could prove extremely interesting due to their recall being very close to the original PQ with significantly improved performance.

Quicker ADC is faster than a binary code (polysemous with $\tau = 0$) as the shuffle instructions used in Quicker ADC are faster than `popcount`. This opens perspectives for using PQ with Symmetric Distance Computation with an SIMD implementation similar to that of Quicker ADC in order to build a very fast code that could be used in pruning applications (e.g., in place of the hamming code of polysemous codes). Bounding a PQ $m\times8$ with a PQ $m\times4$ code is one of the two mechanisms used in [1]; it could be used in isolation as explained in Derived Quantizers [3], in a way similar to the hamming codes of polysemous codes [12].

4.2 Non-exhaustive search

Each index and product quantizer combination can operate in numerous configurations by varying the hyper-parameters (i.e., How many cells to explore? How many distances to evaluate? What hamming threshold to use? How to estimate distance quantizers?). Hence, the performance of each combination is not a single point but a curve of the optimal tradeoffs between query time (in ms) and recall (R@1 or R@100). Thus, to compare the various combinations of indexes and product quantizers, we plot the best recall achieved for a given query time budget. We are particularly interested by small query times (less than 0.5ms). Among the parameters of importance is the parameter a (multiple assignement/probe) which allows faster codes to scan more vectors and more inverted lists for a given time budget.

We consider 3-types of indexes combined with either 64-bit or 128-bit codes. The first index considered is a relatively coarse index (IVF $K = 65536$) [19], the second is a very fine-grained index (inverted-multi-index, IMI $K = 4096^2$) [4], and the last one is a fine-grained index leveraging a neighborhood graph (IVF HNSW $K = 2^{18}$) [27]. The two latters are considered state-of-the-art and are widely used; they achieve similar performance and the more recent IVF HNSW produces fewer inverted lists, which are also more balanced. We report results for the SIFT1000M dataset [20] on Figure 4 and for the Deep1B dataset [8]) on Figure 5. Note that gaps in the curves (quite visible for 128-bit PQ and polysemous codes with IVF HNSW) are related to the fact that parametrizations are discrete (i.e., exploring 1,2,3,... inverted lists).

For all indexes, both codes (64-bit and 128-bit) and both datasets (SIFT1000M and Deep1B), the top performer is one of Quicker ADC's implementations, with a single exception (SIFT1000M, IMI, 128-bit codes for the metric R@1, with a budget for query time $>$ 0.2 ms). Indeed, as observed in Section 4.2, polysemous codes tend to perform better on R@1; they also avoid distance table computation which are numerous in IMI. The domination of Quicker ADC is particularly salient for low query time budget, where the recall gain of Quicker ADC over polysemous codes can be 50% (SIFT100M, IMI, 128-bit, 0.2 ms) or 100% (SIFT 1000M, IVF HSNW, 128-bit, 0.2ms). Quicker ADC is also particularly efficient for metric R@100, or for Deep 1B vectors. For example, on Deep1B with an IMI 2x12 with a query time budget of 0.25ms, Quicker ADC $32\times\{4,4\}$ allows a R@100 of 0.55 while polysemous allow a R@100 of 0.33 or would require a query time of 0.5ms to achieve the same R@100, and PQ 16×8 requires more than 1ms to achieve a recall R@100 of 0.55. Quicker ADC becomes particularly interesting when combined with IVF HNSW: for example, $32\times\{4,4\}$ achieves a R@100 of 0.55 in less than 0.16ms; faster than alternatives on IMI. Indeed, IMI is a worst case for Quicker ADC as these indexes tend to have numerous short lists, and

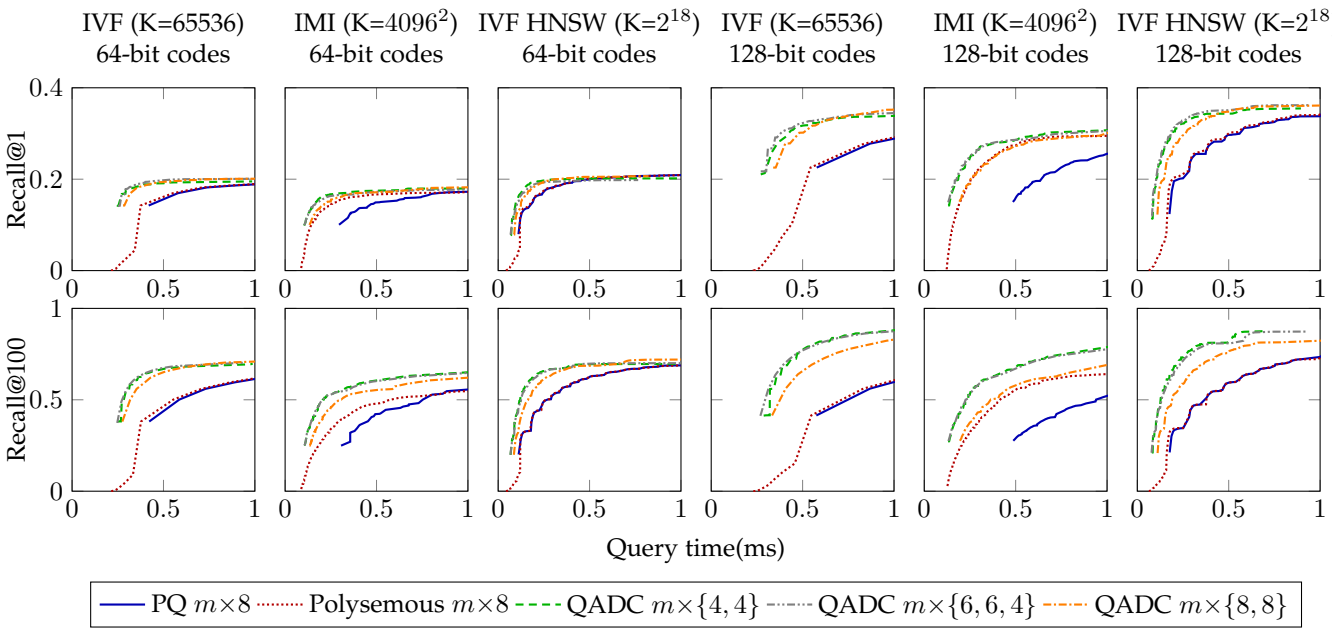


Fig. 5. Non-exhaustive search with different index types and different PQ or Irregular PQ codes. Deep1B

Quicker ADC requires (i) pre-computing distance tables, and (ii) processing batches of vectors. The benefits are higher for indexes that have longer inverted lists (IVF, IVF HNSW). The reduced accuracy of QADC $m\times\{4,4\}$ or $m\times\{6,6,4\}$, observed on exhaustive searched, is here offset by the ability to scan more inverted lists (larger hyper-parameter a) for a given time budget. This explains why QADC outperforms PQ and polysemous codes in term of accuracy.

Note that for index types other than IMI, polysemous codes allow additional operating points over PQ (lower query time by jeopardizing recall) but do not improve performance for the larger query times. It means that while the reduced cost per code of polysemous allows computing distances for more codes/more inverted lists, an equally effective alternative is to not use polysemous but scan fewer codes/less inverted lists. The fact that they are more benefical in the context of IMI can be explained by the fact that they allow avoiding distance table computations, that are more numerous for such fine-grained indexes. Interestingly, their benefits are limited on IVF HNSW.

4.3 Summary of experimental study

Our experiments show that Quicker ADC offers interesting operating points for exhaustive search and non-exhaustive search, even with fine-grained indexes. For exhaustive search, new variants relying on AVX512 are the fastest, with $m\times\{8,8\}$ offering good recall for both R@1 and R@100 with significantly reduced query times. For non-exhaustive search, Quicker ADC $m\times\{4,4\}$ or $m\times\{6,6,4\}$ outperforms other solutions, including polysemous when fast queries are desired. As the time budget for queries increases, or when R@1 is the metric of interest, Quicker ADC $m\times\{8,8\}$ is among the top performers. Hence, Quicker ADC $m\times\{8\}$ whose ADC procedure could be up to 4× faster (see Table 2) is likely to be well positioned when processors supporting the required instructions will become available.

5 Related Work

SIMD evaluation of distances. PQ Fast Scan [1] pioneered the use of SIMD for ADC distance evaluation. It inspired later work [2], [11], [32] that are more suitable for indexed databases. Quicker ADC is a generalization of Quick ADC [2] supporting additional shuffles and an improved distance quantization scheme. Bolt [11] is cotemporary of Quick ADC and both implement lookups using `pshufb`. Bolt however differs in the approach of distance quantization. In Bolt, the distance quantization scheme is decided once for all at training time and aims at minimizing distance distortion for all distances. In Quick ADC and Quicker ADC, the distance quantization scheme is decided dynamically at query-time based on the first vectors scanned and aims at minimizing distortion for only the k-closest vectors, quantizing distances for all other vectors to ∞. This also implies different choices for accumulation: Bolt quantizes distances to 8-bit and accumulates on 16-bit whereas Quick ADC and Quicker ADC quantize distances to 8-bit and accumulate them on 8-bit with saturated arithmetic to ensure semantic coherence between ∞ encoded onto 8-bit in tables and ∞ encoded onto 8-bit during accumulation — should we accumulate on 16-bit, 8-bit sub-distances quantized to ∞ wouldn't be interpreted as ∞ during accumulation. Quick ADC and Quicker ADC prove more effective for nearest neighbor search, while Bolt allows more general use in other applications (matrix multiplication...). Wu et al. [32] present an improved method for learning product quantizers and depart from the usual $m\times8$ to also evaluate $m\times4$ variants, which can be implemented in SIMD. They report execution speed 5× faster, and equaling `popcnt`-based hamming distances speed, similar to our observations. These propositions [1], [11], [32] have shown to be particularly efficient yet their evaluations remains limited to no or coarse indexes and thus lack results for fine grained indexes (e.g., Inverted Multi Index [6]).

By implementing Quicker ADC into FAISS, we have been able to evaluate Quicker ADC with both IMI and IVF HNSW. Also, as we release the implementation, it will be possible to evaluate Quicker ADC for future index designs. In addition, these works were limited to $m\times4$ codes, while we have proposed numerous other variants for a better usage of AVX512-compatible processors.

Indexes. Inverted Multi-Indexes [6] or graph-based inverted indexes [10], [27] provide a finer partition than vanilla inverted indexes. The design of indexes is rather independent from the design of the product quantizer, and thus Quicker ADC can be combined with any inverted index. We evaluate Quicker ADC with several types of indexes and shows that it works fine with IMI which are at the extreme of spectrum in term of cell sizes and works equally well, contrary to polysemous codes with more recent propositions such as IVF HNSW. We expect Quicker ADC to be relatively independent of index choices and to remain efficient with newer indexes.

Alternatively, PQTable [28] seeks to avoid the distance computation for numerous vectors by using a hash-table-like structure, using a PQ code as the key. While it's as fast as less optimized implementations of PQ with Inverted Multi Indexes [6], the evaluation reports 7 ms to achieve 0.06 R@1 and 0.57 R@100 which are much lower than speeds achieved by FAISS using polysemous [12] and Quicker ADC (< 0.3 ms in Figure 4). Yet, both polysemous using `popcount` and Quicker ADC are optimized according to hardware capabilities, and PQTable could also benefit from additional CPU-aware optimization, using batching and prefetching similarly to classical hash-tables [25].

Optimized and Compositional Quantization Models. Cartesian k-means (CKM) [30] and Optimized Product Quantizers (OPQ) [14] optimize the sub-space decomposition by performing an arbitrary rotation and permutation of vector components. This allows for improved accuracy with a moderate cost (i.e., a matrix multiplication to perform the rotation). These are compatible with Quicker ADC as the ADC procedure remains unchanged. The source code we release includes FAISS's implementation of OPQ. We focused our evaluation on the combination of Quicker ADC with regular PQ with various indexes. In addition, compositional vector quantization models inspired by PQ have been proposed. These models offer a lower quantization error than PQ or OPQ. Among these models are Additive Quantization (AQ) [5], Tree Quantization (TQ) [7] and Composite Quantization (CQ) [36]. These models also use cache-resident lookup tables to compute distances, therefore Quicker ADC may be combined with them. However, this may require additional work as some of these models use more lookup tables than the ADC procedure of PQ or OPQ.

Deep-Learning-based quantizers. Subic [17], DPQ [22] and [35] use deep neural networks to compute a compact vector representating images. Similarly to product quantization, the compact vector has a product structure which is exploited to compute distances by summing the contribution of sub-vectors. The distances to each sub-vectors are thus stored in lookup tables and an ADC-like procedure is used. Hence, Quicker ADC naturally extends to these quantizers, and can bring similar benefits. Quicker ADC could be particularly interesting if these quantizers can be adapted to accomodate well small (4-bits) or irregular quantizers.

Encodings based on neighborhood graphs. Some propositions [9], [13] leverage the nearest neighbor graph to have lower encoding error. This improves recall but tend to operate with a higher memory budget [13] and thus does not compare directly. Our work target operating points strictly identical to IVF or inverted multi-indexes combined with Product Quantization [6], [19] and Polysemous Codes [12]. Yet, as these propositions [9], [13] rely on lookup tables for distance computation, they could leverage some principles from Quicker ADC to speed up distance computation.

6 Conclusion

In this paper, we presented Quicker ADC, a novel distance computation method for product-quantization-based ANN search. Quicker ADC improves over previous proposition [2] by (i) supporting additional quantizers (e.g., $m\times\{6,6,4\}$, $m\times\{8,8\}$, $m\times\{8\}$) and (ii) having an improved implementation integrated into FAISS and compatible with various indexes (IMI, IVF HNSW). Through an extensive evaluation, we have shown that Quicker ADC outperforms schemes based on PQ or polysemous codes for both exhaustive and non-exhaustive (i.e., index-based) search, and that they combine well with the latest indexes such as HNSW-based IVF [27]. We release the implementation as open-source to allow a wider adoption and evaluation of this approach.

Techniques presented in this paper focus on the efficient evaluation of distances in the compressed domain. This problem is present in all quantization-based approaches, which rely on lookup tables to speed-up computation. Thus, the principles behind our work (i.e., replacing memory accesses by shuffles and quantizing distance tables) can be the basis for an improved implementation of other approaches [9], [13], [17]. This is of particular interest if those new approaches behave better with coarser (i.e., 4-bit) or irregular quantizers (i.e., combined 6-bit and 5-bit quantizers). Also, Quicker ADC brings product quantizer codes on par with binary codes regarding distance evaluation speed. Thus, Quicker ADC could inspire new designs where filtering with a binary code (e.g., polysemous codes) is replaced by filtering with a lower precision product quantizer (e.g., $m\times\{4,4\}$ with symmetric distance computation used to filter vectors before ADC computation on PQ $m\times8$ as in [1], [3]).

Finally, we would like to stress that upcoming processors will have improved SIMD capabilities allowing for a bright future for Quicker ADC. The Cannonlake processors expected in 2019 will have support for 7-bit shuffles thus quadrupling shuffle throughput as $m\times\{8\}$ codes replace $m\times\{8,8\}$, and Sunny Cove processors expected in 2020 will, in addition, double shuffle throughput by having two shuffle units per core instead of one. Hence, the performance of Quicker ADC will significantly improve in a near future just from hardware upgrade, without algorithm adaptation.

References

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache Locality is Not Enough: High-performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proceedings of the VLDB Endowment*, 9(4):288–299, December 2015.

[2] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Accelerated Nearest Neighbor Search with Quick ADC. In *Proceedings of the ACM International Conference on Multimedia Retrieval*, ICMR, pages 159–166, New York, NY, USA, 2017. ACM.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Derived Codebooks for High-Accuracy Nearest Neighbor Search. *arXiv:1905.06900*, 2019.
[4] Artem Babenko and Victor Lempitsky. The Inverted Multi-Index. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR' 12, pages 3069–3076, June 2012.
[5] Artem Babenko and Victor Lempitsky. Additive Quantization for Extreme Vector Compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 931–938, June 2014.
[6] Artem Babenko and Victor Lempitsky. The Inverted Multi-Index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, June 2015.
[7] Artem Babenko and Victor Lempitsky. Tree quantization for large-scale similarity search and classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 4240–4248, June 2015.
[8] Artem Babenko and Victor Lempitsky. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 2055–2063, June 2016.
[9] Artem Babenko and Victor Lempitsky. AnnArbor: Approximate Nearest Neighbors Using Arborescence Coding. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV, pages 4895–4903, Oct 2017.
[10] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Proceedings of the European Conference on Computer Vision*, ECCV, September 2018.
[11] Davis W. Blalock and John V. Guttag. Bolt: Accelerated Data Mining with Fast Vector Compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, pages 727–735, 2017.
[12] Matthijs Douze, Hervé Jégou, and Florent Perronnin. Polysemous Codes. In *Proceedings of the European Conference on Computer Vision*, ECCV, 2016.
[13] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. Link and Code: Fast Indexing With Graphs and Compact Regression Codes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR, pages 3646–3654, June 2018.
[14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, April 2014.
[15] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing*, WPMVP, pages 57–64, New York, NY, USA, 2014. ACM.
[16] Intel. Optimizing Performance with Intel AVX, 2014.
[17] Himalaya Jain, Joaquin Zepeda, Patrick Pérez, and Rémi Gribonval. SuBiC: A Supervised, Structured Binary Code for Image Search. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV, pages 833–842, Oct 2017.
[18] Hervé Jégou, Matthijs Douze, Jeff Johnson, and Lucas Hosseini. FAISS: A library for efficient similarity search and clustering of dense vectors, 2017. https://github.com/facebookresearch/faiss.
[19] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan 2011.
[20] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP, May 2011.
[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *arXiv:1702.08734*, 2017.
[22] Benjamin Klein and Lior Wolf. In Defense of Product Quantization. *arXiv preprint arXiv:1711.08589*, 2017.
[23] Alfred Klomp. A few missing SSE intrinsics, 2014.
[24] Josip Krapac, Florent Perronnin, Teddy Furon, and Hervé Jégou. Instance Classification with Prototype Selection. In *Proceedings of the International Conference on Multimedia Retrieval*, ICMR, pages 431:431–431:434, New York, NY, USA, 2014. ACM.
[25] Nicolas Le Scouarnec. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS, 2018.
[26] David G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV'99, pages 1150–1157 vol.2, Sep. 1999.
[27] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv:1603.09320*, 2016.
[28] Y. Matsui, T. Yamasaki, and K. Aizawa. PQTable: Nonexhaustive Fast Search for Product-Quantized Codes Using Hash Tables. *IEEE Transactions on Multimedia*, 20(7):1809–1822, July 2018.
[29] Wojciech Muła, Nathan Kurz, and Daniel Lemire. Faster Population Counts Using AVX2 Instructions. *The Computer Journal*, 2018.
[30] Mohammad Norouzi and David J. Fleet. Cartesian K-Means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 3017–3024, June 2013.
[31] Aude Oliva and Antonio Torralba. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *International Journal of Computer Vision*, 42(3):145–175, May 2001.
[32] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. Multiscale Quantization for Fast Similarity Search. In *Advances in Neural Information Processing Systems*, NIPS, 2017.
[33] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. Joint Inverted Indexing. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV, pages 3416–3423, Dec 2013.
[34] Lingxi Xie, Richang Hong, Bo Zhang, and Qi Tian. Image Classification and Retrieval are ONE. In *Proceedings of the ACM International Conference on Multimedia Retrieval*, ICMR, pages 3–10, New York, NY, USA, 2015. ACM.
[35] Tan Yu, Junsong Yuan, Chen Fang, and Hailin Jin. Product Quantization Network for Fast Image Retrieval. In *Proceedings of The European Conference on Computer Vision*, ECCV, 2018.
[36] Ting Zhang, Chao Du, and Jingdong Wang. Composite Quantization for Approximate Nearest Neighbor Search. In *Proceedings of the International Conference on International Conference on Machine Learning*, ICML, pages II–838–II–846. JMLR.org, 2014.

BIOGRAPHIES

Fabien André is a software engineer. He worked at Technicolor from 2013 to 2018 on multimedia databases and high-performance networking systems. He earned an engineering degree from INSA de Rennes, France in 2013, and a PhD from INSA de Rennes, France in 2016.

Anne-Marie Kermarrec is Full Professor at EPFL, Swizerland. She founded the Mediego startup in April 2015. Mediego provides online predictive marketing services that directly leverage her recent research. Before that, after her PhD thesis at University of Rennes in 1996, she has been with Vrije Universiteit, NL and Microsoft Research Cambridge, UK. She was Research Director at Inria in Rennes from 2004 to 2019. She is an ACM fellow since 2016.

Nicolas Le Scouarnec is a research engineer at Broadpeak. Before that he was a senior scientist in distributed systems and high performance computing at Technicolor. His current research focuses on reliable and high performance networked systems, as well efficient implementation of machine learning algorithms. He earned a M.Sc. degree from INSA de Rennes in 2007 and a Ph.D. in computer science from INSA de Rennes, France in 2010.