The Inverted Multi-Index

Artem Babenko [1,2] and Victor Lempitsky [1]
[1] Yandex, Moscow [2] Moscow Institute of Physics and Technology

Abstract

A new data structure for efficient similarity search in very large datasets of high-dimensional vectors is introduced. This structure called the inverted multi-index generalizes the inverted index idea by replacing the standard quantization within inverted indices with product quantization. For very similar retrieval complexity and preprocessing time, inverted multi-indices achieve a much denser subdivision of the search space compared to inverted indices, while retaining their memory efficiency. Our experiments with large datasets of SIFT and GIST vectors demonstrate that because of the denser subdivision, inverted multi-indices are able to return much shorter candidate lists with higher recall. Augmented with a suitable reranking procedure, multi-indices were able to improve the speed of approximate nearest neighbor search on the dataset of 1 billion SIFT vectors by an order of magnitude compared to the best previously published systems, while achieving better recall and incurring only few percent of memory overhead.

1. Introduction

In computer vision, *inverted indices* (inverted files) [23] are widely used for retrieval and similarity search. For a large dataset of visual descriptors, a typical inverted index is built around a *codebook* containing a set of *codewords*, i.e. a representative set of vectors that may be constructed by performing clustering on the initial dataset. An inverted index then stores the list of vectors that lie in the proximity of each codeword (belong to its *Voronoi cell*). The purpose of an inverted index is then to efficiently generate a list of dataset vectors that lie close to any query vector. Given a query, either the closest codeword or a set of few closest codewords are identified. The lists corresponding to those codewords are then concatenated to produce the answer to the query.

Querying the inverted index avoids evaluating distances between the query and every point in the dataset and, thus, provides a substantial speed-up over the exhaustive search. Furthermore, as the index does not need to contain the original dataset vectors to perform the search, the memory footprint of each data point can be reduced significantly, and only useful metadata (e.g. image IDs or heavily compressed original vectors) can be stored in the list entries. Because of these efficiency benefits, inverted indices are widely used within computer vision systems such as image and video search [23] or location identification [18]. More generally, they can be used within any computer vision task that involves fast near(est) neighbor retrieval or kernel density estimation (i.e. image classification [3, 5], understanding [14], image editing [6], etc.).

The efficiency of inverted indices has however certain limitations that begin to show up for very large datasets of vectors (hundreds of million to billions), which computer vision researchers and practitioners are starting to tackle [1, 12, 24]. In this scenario, a very fine partition of the search space is desirable to avoid returning excessively large lists in response to the queries or, put differently, to return vectors that are better localized around the query point. Unfortunately, increasing the number of codewords in order to achieve finer partition also increases the query time and the index construction time. While approximate nearest neighbor approaches (e.g. tree codebooks [16] or kd-trees [2]) may be invoked to make this deceleration graceful, these techniques often reduce the accuracy (recall and precision) of the returned candidate lists considerably.

The goal of this paper is to introduce and evaluate a new data structure called the *inverted multi-index* that is in many respects similar to the inverted index and can therefore be used within computer vision systems in a similar way. The advantage of multi-indices is in their ability to produce much finer subdivisions of the search space without increasing the query time and the preprocessing time compared to inverted indices with moderately-sized codebooks (importantly, the relative increase of memory usage for large datasets is also small). Consequently, multi-indices result in faster and more accurate retrieval and approximate nearest neighbor search, especially when dealing with very large scale datasets, while retaining the memory efficiency of standard inverted indices.

In a nutshell, inverted multi-indices are obtained by replacing the vector quantization inside inverted indices with the *product quantization (PQ)* [9]. PQ proceeds by splitting high-dimensional vectors into dimension groups. PQ then effectively approximates each vector as a concatenation of several codewords of smaller dimensionality, coming from several codebooks pretrained for each group of dimensions separately. Following the PQ idea, an inverted multi-index is constructed as a multi-dimensional table. The entries of this table correspond to all possible tuples of codewords from the codebooks corresponding to different dimension groups. This multi-dimensional table replaces a "flat" table containing entries corresponding to codewords of the standard inverted index.

Similarly to a standard inverted index, each entry of a

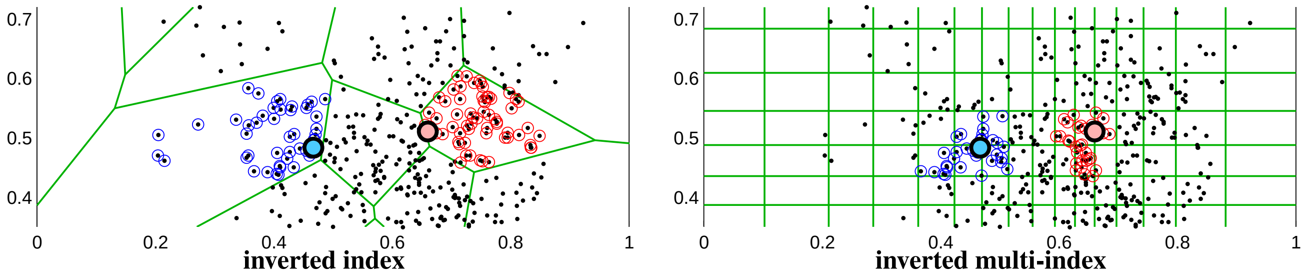


Figure 1. Indexing the set of 600 points (small black) distributed non-uniformly within the unit 2D square. **Left** – the inverted index based on standard quantization (the codebook has 16 2D codewords; boundaries are in green). **Right** – the inverted multi-index based on product quantization (each of the two codebooks has 16 1D codewords). The number of operations needed to match a query to codebooks is the same for both structures. Two example queries are issued (light-blue and light-red circles). The lists returned by the inverted index (left) contain 45 and 62 words respectively (circled). Note that when a query lies near a space partition boundary (as happens most often in high dimensions) the resulting list is heavily "skewed" and may not contain many of the nearest neighbors. Note also that the inverted index is not able to return lists of a pre-specified small length (e.g. 30 points). For the same queries, the candidate lists of at least 30 vectors are requested from the inverted multi-index (right) and the lists containing 31 and 32 words are returned (circled). As even such short lists require visiting several nearest cells in the partition (which can be done efficiently via the *multi-sequence algorithm*), the resulting vector sets span the neighborhoods that are much less "skewed" (i.e., the neighborhoods are approximately centered at the queries). In high dimensions, the capability to visit many cells that surround the query from different directions translates into considerably higher accuracy of retrieval and nearest neighbor search.

multi-index table corresponds to a part of the original vector space and contains a list of points that fall within that part. Importantly, we propose a simple and efficient algorithm that produces a sequence of multi-index entries ordered by the increasing distance between the given query vector and the centroid of the corresponding entry. Similarly to standard inverted indices, concatenating the vector lists for a certain number of entries that are closest to the query vector then produces the candidate list.

Crucially, given comparable time budgets for querying the dataset as well as for the initial index construction, inverted multi-indices subdivide the vector space orders of magnitude more densely compared to standard inverted indices (Figure 1). Our experiments demonstrate the advantages resulting from this property, in particular in the context of very large scale approximate nearest neighbor search. We evaluate the inverted multi-index on the BIGANN dataset of 1 billion SIFT vectors recently introduced by Jegou et al. [11] as well as on the "Tiny Images" dataset of 80 million GIST vectors introduced by [24]. We show that as a result of the "extra-fine" granularity, the candidate lists produced by querying multi-indices are more accurate (have shorter lengths and higher probability of containing true nearest neighbors) compared to standard inverted indices. We also demonstrate that in combination with a suitable reranking procedure, multi-indices substantially improve the state-of-the-art approximate nearest neighbor retrieval performance on the BIGANN dataset.

2. Related Work

The use of inverted indices has a long history in information retrieval [15]. Their use in computer vision was pioneered by Sivic and Zisserman [23]. Since then, a large number of improvements that transfer further ideas from text retrieval (e.g. [4]), improve the quantization process (e.g. [20]), and integrate the query process with geometric verification (e.g. [27]) have been proposed. Many of these improvements can be used in conjunction with inverted multi-indices in the same way as with regular inverted indices.

Approximate near(est) neighbor (ANN) search is a core operation in AI. ANN-systems based on tree-based indices (e.g. [2]) as well as on random projections (e.g. [7]) are often employed. However, the large memory footprint of these methods limits their use to smaller datasets (up to millions of vectors). Recently, lossy compression schemes that admit both compact storage and efficient distance evaluations and are therefore more suitable for large-scale datasets have been developed. Towards this end, binary encoding schemes (e.g. [22, 25, 21]) as well as product quantization [9] have brought down both memory consumption and distance evaluation time by order(s) of magnitude compared to manipulating uncompressed vectors, to the point where exhaustive search can be used to query rather large datasets (up to many millions of vectors).

The idea of fast distance computation via product quantization introduced by Jegou et al. [9] has served as a primary inspiration for this work. Our contribution, however, is complementary to that of [9]. In fact, the systems presented by Jegou et al. in [9, 11, 10] use standard inverted indices and, consequently, have to rerank rather long candidate lists when querying very large datasets in order to achieve high recall. Unlike [9, 11, 10], we focus on the use of PQ for indexing and candidate list generation. We also note that while we combine multi-indices with the PQ-based reranking [9] in some of our experiments, one can also employ binary embedding [8] or any other compression/fast distance computation scheme to rerank lists returned by a multi-index (or, depending on the application, omit the reranking altogether) .

3. The Inverted Multi-Index

The structure of the inverted multi-index. We now explain how an inverted multi-index is organized. Along the way, we will compare the analogous parts between inverted multi-indices and standard inverted indices.

We assume that a large collection $\mathcal{D}$ of N M-dimensional vectors $\mathcal{D} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N\}$, $\mathbf{p}_i \in \mathcal{R}^M$ is given. The construction of a standard inverted index then starts with learning a codebook $\mathcal{W}$ of K M-dimensional vectors $\mathcal{W} = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K\}$ via a k-means algorithm. The initial dataset is then split into K *lists* $W_1, W_2, \ldots, W_K$, where each list W_i contains all vectors that fall in its Voronoi cell in $\mathcal{R}^M$, i.e. $W_i = \{\mathbf{p} \in \mathcal{D} | i = \arg\min_j d(\mathbf{p}, \mathbf{w}_j)\}$. Here, d is a distance measure in $\mathcal{R}^M$. In practice, each list W_i can be represented in memory as a contiguous array, where each entry may contain the compressed version of the initial vector (which is useful for reranking) and typically some metadata associated with the vector (e.g. the class label or the ID of the image that the visual descriptor p was sampled from).

Following the product quantization idea [9], the inverted multi-index is organized around splitting the M input dimensions into several dimension blocks. The number of blocks affects the accuracy of retrieval and its speed. In previous works where PQ was used for compression and fast distance evaluation, the best trade-off was achieved for 8 or so blocks [9, 11, 10]. In the multi-index case, however, it is optimal to split dimensions in just two blocks, at least for the characteristic scales considered in our evaluation and assuming that the accuracy and low query time are more important than low index construction time. We comment more on the choice of the number of blocks below. For the time being, to simplify the explanation we discuss how a multi-index can be built for the case of splitting vectors into two halves. Where required, we refer to this case as the *second-order* inverted multi-index. It will be evident how to generalize the proposed algorithms to *higher-order* inverted multi-indices (which split vectors into more than two dimension groups).

Let $\mathbf{p}_i = [\mathbf{p}_i^1 \; \mathbf{p}_i^2]$ be the decomposition of a vector $\mathbf{p}_i \in \mathcal{R}^M$ from the dataset into two halves, where $\mathbf{p}_i^1 \in \mathcal{R}^{\frac{M}{2}}, \mathbf{p}_i^2 \in \mathcal{R}^{\frac{M}{2}}$. As in the case of other PQ-based systems, inverted multi-indices perform better when the correlation between $\mathcal{D}^1 = \{\mathbf{p}_i^1\}$ and $\mathcal{D}^2 = \{\mathbf{p}_i^2\}$ is lower and the amount of variance within $\mathcal{D}^1$ and $\mathcal{D}^2$ are closer to each other. For SIFT-vectors, splitting them directly into halves seems to be a near-optimal strategy, while in other cases one can regroup the dimensions to reduce the correlation or multiply all vectors by a random orthogonal matrix to balance the variances between the halves [10, 9].

The PQ codebooks for the inverted multi-index are obtained via independent k-means clustering of the sets $\mathcal{D}^1$ and $\mathcal{D}^2$ independently, producing the codebooks $\mathcal{U} = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_K\}$ for the first half and $\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_K\}$ for the second half of dimensions[1]. We then perform the product quantization of the dataset vectors, so that the K^2 lists corresponding to all possible pairs of codewords $(\mathbf{u}_i, \mathbf{v}_j), i = 1..K, j = 1..K$ are created. We denote each of the K^2 lists as W_{ij}. Each point $\mathbf{p} = [\mathbf{p}^1 \; \mathbf{p}^2]$ is assigned to the closest point $[\mathbf{u}_i \; \mathbf{v}_j]$, so that:

$$W_{ij} = \{\mathbf{p} = [\mathbf{p}^1 \; \mathbf{p}^2] \in \mathcal{D} \,| \qquad (1)$$
$$i = \arg\min_k d_1(\mathbf{p}^1, \mathbf{u}_k) \wedge j = \arg\min_k d_2(\mathbf{p}^2, \mathbf{v}_k)\} \, .$$

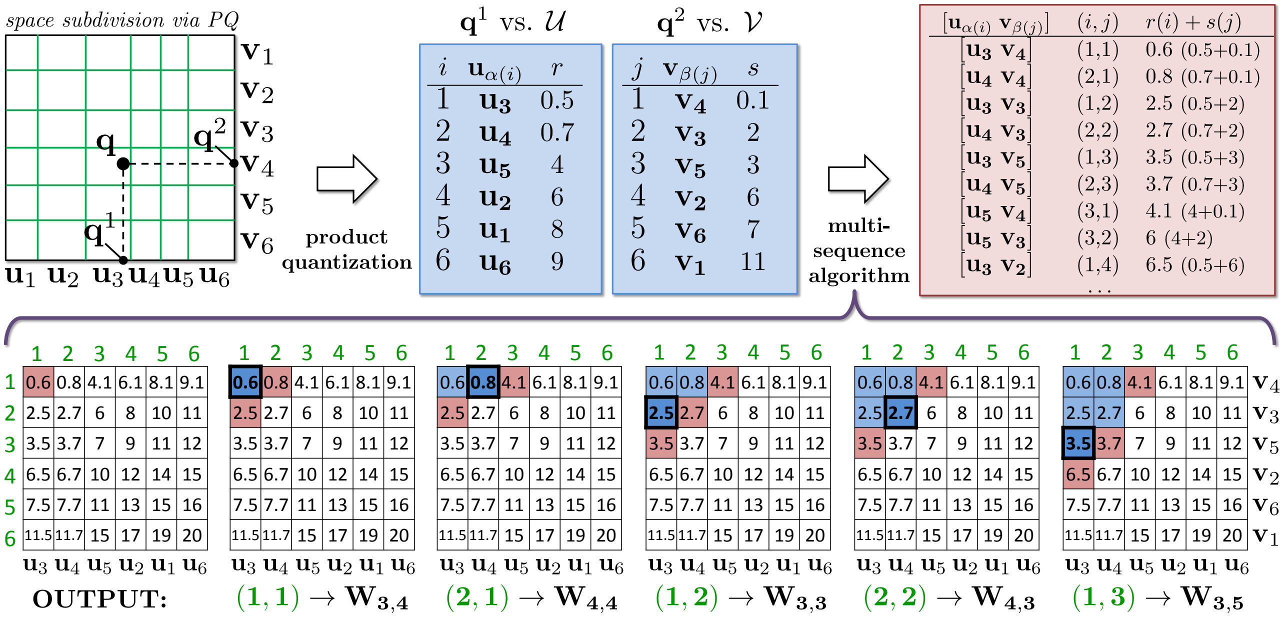
Note that the "catchment area" of each list W_{ij} is now a Cartesian product of the two Voronoi cells in $\mathcal{R}^{\frac{M}{2}}$ spaces. In (1), the distance measures d_1 and d_2 in $\mathcal{R}^{\frac{M}{2}}$ are induced by d, so that $\forall \mathbf{a}, \mathbf{b} : \; d(\mathbf{a}, \mathbf{b}) = d_1(\mathbf{a}^1, \mathbf{b}^1) + d_2(\mathbf{a}^2, \mathbf{b}^2)$. The simplest and most important case is setting d, d_1, and d_2 to be squared Euclidean norms in respective spaces, so that the resulting multi-index can be used to retrieve points with low Euclidean distance from the query. We briefly discuss alternative distances in Section 5.

Querying the multi-index. Given a query $\mathbf{q} = [\mathbf{q}^1 \; \mathbf{q}^2] \in \mathcal{R}^M$ and a desired candidate list length $T{<<}N$, an inverted multi-index allows to generate a list of T (or slightly more) points from $\mathcal{D}$ that tend to be close to $\mathbf{q}$ with respect to the distance d. This is achieved via identifying a sufficient number of codeword pairs $[\mathbf{u}_i \; \mathbf{v}_j]$ that are closest to $\mathbf{q}$ in $\mathcal{R}^M$ and concatenating their lists W_{ij}. Finding those $[\mathbf{u}_i \; \mathbf{v}_j]$ is performed in two stages (Figure 2-top).

On the first stage, $\mathbf{q}^1$ and $\mathbf{q}^2$ are independently matched to corresponding codebooks. Thus, for $\mathbf{q}^1$ and $\mathbf{q}^2$ the L nearest neighbors among $\mathcal{U}$ and $\mathcal{V}$ respectively are identified (where $L{<}K$ depends on the specified T). As the size of $\mathcal{U}$ and $\mathcal{V}$ is typically not large (thousands of vectors), exhaustive search can be used. Denote with $\alpha(k)$ the index of the kth closest neighbor to $\mathbf{q}^1$ in $\mathcal{U}$ (i.e. $u_{\alpha(1)}$ is the nearest neighbor to $\mathbf{q}^1$ in $\mathcal{U}$, $u_{\alpha(2)}$ is the second closest, etc.). Similarly, denote with $\beta(k)$, the index of the kth closest neighbor to $\mathbf{q}^2$ in $\mathcal{V}$. Also, denote with $r(k)$ and $s(k)$ the distances from $\mathbf{q}^1$ and $\mathbf{q}^2$ to $\mathbf{u}_{\alpha(k)}$ and $\mathbf{v}_{\beta(K)}$ respectively, i.e. $r(k) = d_1\left(\mathbf{q}^1, \mathbf{u}_{\alpha(k)}\right)$ and $s(k) = d_2\left(\mathbf{q}^2, \mathbf{v}_{\beta(k)}\right)$.

On the second stage, given the two monotonically increasing sequences $r(1), r(2), \ldots, r(L)$ and $s(1), s(2), \ldots, s(L)$, we traverse the set of pairs

[1] We have deliberately chosen different letters $\mathbf{u}$ and $\mathbf{v}$ in the notation of the two sub-codebooks, to emphasize that they are learned separately and that $\mathbf{w}_i \neq [\mathbf{u}_i \; \mathbf{v}_i]$.



space subdivision via PQ
$\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ $\mathbf{v}_4$ $\mathbf{v}_5$ $\mathbf{v}_6$
$\mathbf{u}_1$ $\mathbf{u}_2$ $\mathbf{u}_3$ $\mathbf{u}_4$ $\mathbf{u}_5$ $\mathbf{u}_6$
$\mathbf{q}$
$\mathbf{q}^1$
$\mathbf{q}^2$
product quantization
$\mathbf{q}^1$ vs. $\mathcal{U}$
$\mathbf{q}^2$ vs. $\mathcal{V}$
multi-sequence algorithm
OUTPUT:
$(1,1) \rightarrow \mathbf{W}_{3,4}$
$(2,1) \rightarrow \mathbf{W}_{4,4}$
$(1,2) \rightarrow \mathbf{W}_{3,3}$
$(2,2) \rightarrow \mathbf{W}_{4,3}$
$(1,3) \rightarrow \mathbf{W}_{3,5}$

i	$\mathbf{u}_{\alpha(i)}$	r
1	$\mathbf{u_3}$	0.5
2	$\mathbf{u_4}$	0.7
3	$\mathbf{u_5}$	4
4	$\mathbf{u_2}$	6
5	$\mathbf{u_1}$	8
6	$\mathbf{u_6}$	9

j	$\mathbf{v}_{\beta(j)}$	s
1	$\mathbf{v_4}$	0.1
2	$\mathbf{v_3}$	2
3	$\mathbf{v_5}$	3
4	$\mathbf{v_2}$	6
5	$\mathbf{v_6}$	7
6	$\mathbf{v_1}$	11

$[\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]$	(i,j)	$r(i)+s(j)$
$[\mathbf{u_3}\ \mathbf{v_4}]$	(1,1)	0.6 (0.5+0.1)
$[\mathbf{u_4}\ \mathbf{v_4}]$	(2,1)	0.8 (0.7+0.1)
$[\mathbf{u_3}\ \mathbf{v_3}]$	(1,2)	2.5 (0.5+2)
$[\mathbf{u_4}\ \mathbf{v_3}]$	(2,2)	2.7 (0.7+2)
$[\mathbf{u_3}\ \mathbf{v_5}]$	(1,3)	3.5 (0.5+3)
$[\mathbf{u_4}\ \mathbf{v_5}]$	(2,3)	3.7 (0.7+3)
$[\mathbf{u_5}\ \mathbf{v_4}]$	(3,1)	4.1 (4+0.1)
$[\mathbf{u_5}\ \mathbf{v_3}]$	(3,2)	6 (4+2)
$[\mathbf{u_3}\ \mathbf{v_2}]$	(1,4)	6.5 (0.5+6)
...		

	1	2	3	4	5	6	
1	0.6	0.8	4.1	6.1	8.1	9.1	$\mathbf{v}_4$
2	2.5	2.7	6	8	10	11	$\mathbf{v}_3$
3	3.5	3.7	7	9	11	12	$\mathbf{v}_5$
4	6.5	6.7	10	12	14	15	$\mathbf{v}_2$
5	7.5	7.7	11	13	15	16	$\mathbf{v}_6$
6	11.5	11.7	15	17	19	20	$\mathbf{v}_1$
	$\mathbf{u}_3$	$\mathbf{u}_4$	$\mathbf{u}_5$	$\mathbf{u}_2$	$\mathbf{u}_1$	$\mathbf{u}_6$	

Figure 2. **Top** – The overview of the query process within the inverted multi-index. First, the two halves of the query $\mathbf{q}^1$ and $\mathbf{q}^2$ are matched w.r.t. sub-codebooks $\mathcal{U}$ and $\mathcal{V}$ to produce the two sequences of codewords ordered by the distance (denoted r and s) from the respective query half. Then, those sequences are traversed with the multi-sequence algorithm that outputs the pairs of codewords ordered by the distance from the query. The lists associated with those pairs are concatenated to produce the answer to the query. **Bottom** – The first iterations of the multi-sequence algorithm in this example. Red denotes pairs in the priority queue, blue indicates traversed pairs (the pair traversed at the current iteration is emphasized). Green numbers correspond to pair indices (i and j), while black symbols give original codewords ($\mathbf{u}_{\alpha(i)}$ and $\mathbf{v}_{\beta(j)}$). The numbers in entries are the distances $r(i)+s(j) = d\left(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]\right)$.

$\{(r(i), s(j))\,|\,i = 1 \ldots L, j = 1 \ldots L\}$ in the order of the increasing sum $r(i) + s(j)$ (which equals $d(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}])$). In this way, the centroids $[\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]$ are visited in the order of increasing distance from $\mathbf{q}$. The traversal starts from the pair $(1,1)$ naturally corresponding to the cell around the centroid $[\mathbf{u}_{\alpha(1)}\ \mathbf{v}_{\beta(1)}]$, which the query falls into. During the traversal, the lists $W_{\alpha(i)\,\beta(j)}$ are concatenated, until the length of the answer exceeds the predefined length T, at which point the traversal stops.

We propose an algorithm to perform such a traversal (Figure 2-bottom). This *multi-sequence* algorithm is based around a priority queue of index pairs (i,j), where the priority of each pair is defined as $-(r(i)+s(j)) = -d\left(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]\right)$. The queue is initialized with a single pair $(1,1)$. At each subsequent step t, the pair (i_t, j_t) with top priority (lowest distance from $\mathbf{q}$) is popped from the queue and considered traversed (the associated list $W_{\alpha(i)\,\beta(j)}$ is added to the output list). The pairs (i_t+1, j_t) and (i_t, j_t+1) are then considered for the insertion into the priority queue. The pair $(i_t{+}1, j_t)$ is inserted into the queue if its other preceding pair (i_t+1, j_t-1) has also been traversed (or if $j_t{=}1$). Similarly, the pair $(i_t, j_t{+}1)$ is inserted into the queue if its other preceding pair (i_t-1, j_t+1) has also been traversed (or if $i_t{=}1$). The idea is that each pair is inserted only once when both of its preceding pairs are traversed.

The multi-sequence algorithm produces a sequence of pairs (i,j), whose lists $W_{i,j}$ are accumulated into the query response. One can prove the correctness of the algorithm:

Corollary 1 (correctness): the multi-sequence algorithm produces the sequence of pairs in the order of increasing $r(i) + s(i)$ and will eventually visit every pair in $\{1 \ldots L\} \otimes \{1 \ldots L\}$.

Regarding the efficiency of the algorithm, one can prove that the queue within the algorithm grows slow enough:

Corollary 2: at the tth step of the algorithm, when t pairs have been output, the priority queue is no longer than $0.5 + \sqrt{2t + 0.25}$.

The proof of both corollaries and the pseudocode of the multi-sequence algorithm are given in the supplementary material.

Inverted index vs. inverted multi-index. Let us now discuss the relative efficiency of the two indexing structures, given the same codebook size K. In this situation, the induced subdivision of the space is very different for the standard inverted index and for the inverted multi-index (Figure 1). In particular, the standard index maintains K lists that correspond to the space subdivision into K cells, while the multi-index maintains K^2 lists corresponding to a much finer subdivision of the space. While the lengths of the cell lists within the inverted index tend to be balanced (due to the nature of the k-means algorithm), the distribution of list lengths within the multi-index is highly non-uniform. In particular, there are lots of empty lists that correspond to $\mathbf{u}_i$

and $\mathbf{v}_j$ that never co-occur together (e.g. cells in the bottom-right corner in Figure 1-right). Still, as will be revealed in the experiments, despite a highly non-uniform distribution of list lengths, inverted multi-indices enjoy a large boost in retrieval accuracy due to higher sampling density.

Furthermore, despite the increase in the subdivision density, matching a query with codebooks for both structures requires the same number of operations. Thus, in the inverted multi-index case one has to compute the K distances between M-dimensional vectors, while in the multi-index case $2K$ distances between $M/2$-dimensional vectors are computed (while the number of the scalar operations is the same, vector instructions on modern CPUs can make the matching moderately faster in the inverted index case). Querying the multi-index also incurs an overhead in computational cost due to the use of the multi-sequence algorithm. In our experiments, we however observed (Section 4) that the overhead was small compared to the quantization cost even for rather long list lengths T.

The use of the inverted multi-index also incurs a memory overhead, as it has to maintain K^2 rather than K lists. However, the joint length of all lists remains the same (as the number of entries equals the total number of vectors i.e. N). Therefore, given that all lists are stored contiguously as a large array, maintaining each list W_{ij} effectively requires one integer (that contains the starting location of the list within the large array). Within our experimental setting of $N = 10^9$ and $K = 2^{14}$, this overhead amounts to one byte per dataset vector (4 bytes*K/N). Such overhead is small compared to several bytes of meta-data and/or compressed vector that are typically stored for each instance.

Coming back to higher-order multi-indices, which split vectors into more than two dimension groups, our experiments (partially presented in Section 4) suggest that while they result in much smaller quantization times (for the same subdivision densities), their memory overheads grow quite rapidly with K and so does non-uniformity of list lengths and the numbers of empty cells in the index. This memory inefficiency limits the usage of such "higher-order" multi-indices to small values of K, where the accuracy of retrieval is limited. Overall, in our experiments, second-order multi-indices proved to be a sweet spot between inverted indices (low memory overhead, large quantization times for sufficient subdivision density) and higher-order multi-indices (high memory overhead, low quantization time). The use of the latter, however, is justified when small pre-processing times are required, as the time required to product quantize all dataset vectors during higher-order multi-index construction is much smaller (due to lower K).

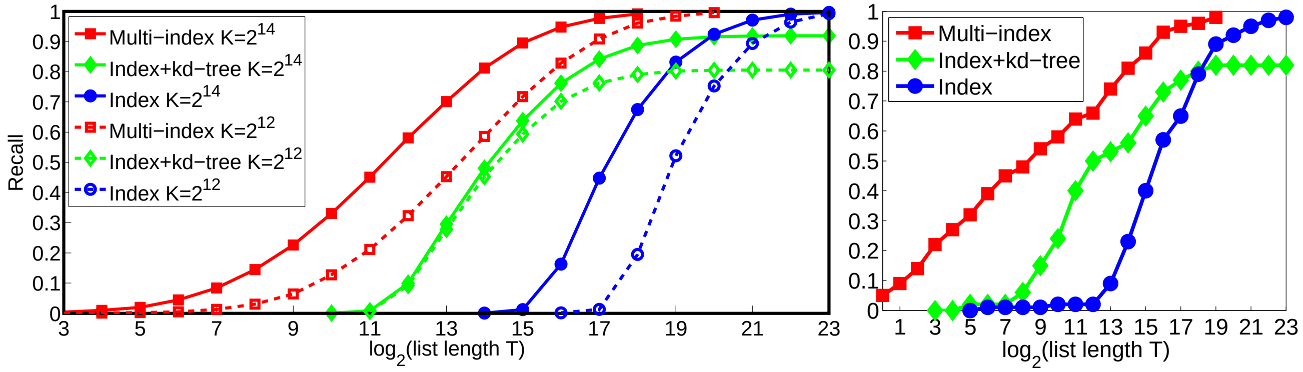
4. Experiments

We now present the results of the experimental evaluation of inverted multi-indices on two large-scale datasets of visual descriptors. The majority of experiments were performed on the *BIGANN* dataset of 1 billion 128-dimensional SIFT descriptors [13] extracted from natural images that was introduced by Jegou et al. [11]. The ground truth (true Euclidean nearest neighbors) for a hold-out set of 10000 queries is provided with the dataset. We also perform experiments with the 384-dimensional GIST [17] descriptors corresponding to 80 million *Tiny Images* [24]. For this set, we picked a subset of 100 vectors and computed their Euclidean nearest neighbors within the rest of the dataset through exhaustive search thus obtaining the query set (which was excluded from the original dataset).

Below we report different measurements related to list lengths, timings, and the *recall*, which is defined as the probability of finding the *first* nearest neighbor of a query in a list returned by a certain system. This probability is always evaluated by averaging the rate of success (true nearest neighbor is on the list) over the available query set. In practice, the performance of retrieving other nearest neighbors (beyond the first one) is often important, however, this performance is highly correlated with the ability to retrieve the first nearest neighbor, and is therefore omitted from this evaluation. All timings were obtained on a single core of Intel Xeon 2.40 GHz CPU (using BLAS instructions in the single-thread mode).

How successful are inverted multi-indices at returning nearest neighbors in candidate lists? This is, arguably, the most important question. We address it by comparing the recall of a second-order inverted multi-index and an inverted index for the same codebook size K. We perform this comparison for $K = 2^{14}$ for the BIGANN and 80 million dataset and, additionally, for a smaller $K = 2^{12}$ for the BIGANN dataset. For a set of predefined list lengths T (powers of two) and for each query, we traverse both data structures concatenating the lists stored in the entries. The traversal stops one step before the concatenated list length exceeds the predefined length T. Figure 3 plots the recall of such lists (to which we refer as *recall@*T) versus the length T. In general, for a fixed K, the advantage of multi-indices over indices is very significant for the whole range of list lengths.

We then evaluate a more challenging baseline. As kd-trees [2] have emerged as a popular tool for working with very large codebooks, we took such a codebook (2^{18} codewords) and used a kd-tree (`vl_feat` [26] implementation) to match the queries and the dataset vectors to this codebook (thus replacing the exhaustive search within the inverted index quantization with the fast approximate search). For a fair comparison, we limited the number of vector distance evaluations within the kd-tree to the respective K (either 2^{14} or 2^{12}). As can be seen in Figure 3, the new baseline is more competitive in the low recall area, although it performs worse than the first baseline when high recall



1 billion SIFTs, $K = 2^{14}$ (solid), $K = 2^{12}$ (dashed)

80 million GISTs, $K = 2^{14}$

Figure 3. Recall as a function of the candidate list length. For the same codebook size K, we compare three systems with similar retrieval and construction complexities: an inverted index with K codewords, an inverted index with larger codebook (2^{18} codewords) sped up by a kd-tree search with a maximum of K comparisons, an inverted multi-index with codebooks having K codewords. In all three experiments, multi-indices returned shorter lists with higher recall.

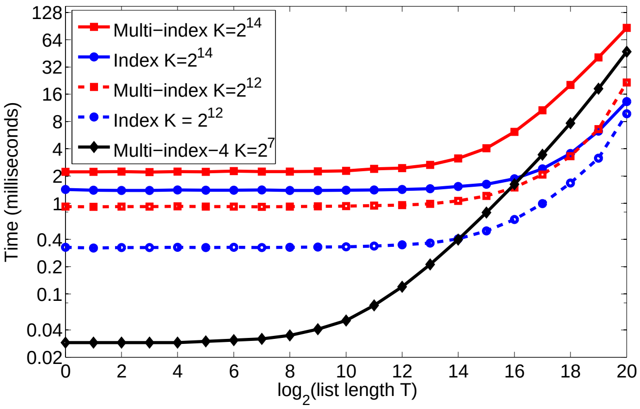


Figure 4. Time (in milliseconds) required to retrieve a list of a particular length from the inverted multi-index and index on the BIGANN dataset.

is needed. Overall, the recall@T of both baselines was uniformly worse than the recall@T of the inverted multi-indices in our experiments. Both, kd-trees and multi-indices incur some computational overhead over inverted indices (tree search and multi-sequence algorithm, respectively) and we now address the question how big this overhead is for the inverted multi-indices.

How fast is querying an inverted multi-index? To answer this question, we give the timings for the inverted multi-indices ($K = 2^{12}, K = 2^{14}$) on the BIGANN dataset as a function of the requested list length in Figure 4. The multi-index retrieval time essentially remains flat until the list length grows into many thousands, which means that the computational cost of the multi-sequence algorithm remains small compared to the quantization. We also give the timing curves for inverted indices with $K = 2^{12}, 2^{14}$. Their approximately two-fold speed advantage over the second-order indices (for the same K) stems most likely from the particular efficiency of vector instructions (BLAS library) on our CPU. This efficiency makes matching against codebooks faster in the inverted index case despite the same number of scalar operations.

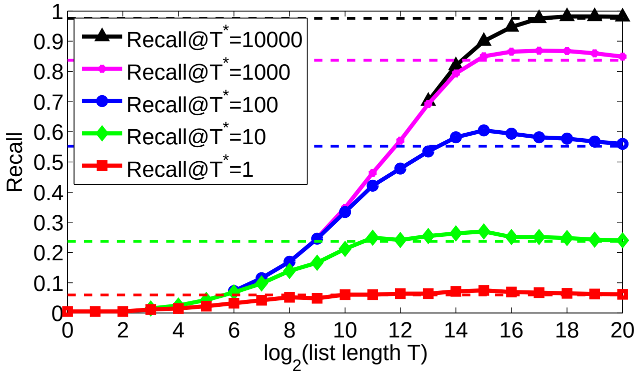


Figure 5. Recall@T^* ($T^* = 1$ to 10000) of the Multi-ADC system (storing $m = 8$ extra bytes per vector for reranking) for the BIGANN dataset. The curves correspond to the Multi-ADC system that reranks a candidate list of a certain length T (x-axis) returned by the second-order multi-index ($K = 2^{14}$), while the flat dashed lines corresponds to the system that reranks the entire dataset. After reranking a tiny part of the billion-size dataset, Multi-ADC is able to match the performance of the exhaustive search-based system.

Put together, Figure 3 and Figure 4 demonstrate the advantage of the second-order inverted multi-index over the standard inverted index. Thus, the multi-index with $K = 2^{12}$ provides much higher recall and is faster to query than the inverted index with $K = 2^{14}$. In Figure 4, we also provide timings for the fourth-order index and small K. Here, querying for short list lengths is much faster, however the overhead from the multi-sequence algorithm kicks in at shorter lengths (hundreds) exhibiting the main weakness of higher-order inverted multi-indices.

Nearest neighbor search with reranking. The goal of

System	List len.T	R@1	R@10	R@100	Time
BIGANN, 1 billion SIFTs, 8 bytes per vector					
IVFADC[11]	8 million	0.112	0.343	0.728	155
		(0.088)	(0.372)	(0.733)	(74*)
Multi-D-ADC	10000	0.158	0.472	0.706	6
Multi-D-ADC	30000	0.164	0.506	0.813	13
Multi-D-ADC	100000	0.165	0.517	0.860	37
BIGANN, 1 billion SIFTs, 16 bytes per vector					
IVFADC+R[11]	8 million	(0.262)	(0.701)	(0.962)	(116*)
Multi-D-ADC	10000	0.304	0.665	0.740	7
Multi-D-ADC	30000	0.328	0.757	0.885	16
Multi-D-ADC	100000	0.334	0.793	0.959	49
Tiny Images, 80 million GISTs, 8 bytes per vector					
Multi-D-ADC	10000	0.06	0.40	0.59	19
Multi-D-ADC	30000	0.06	0.41	0.63	41
Multi-D-ADC	100000	0.06	0.41	0.66	119
Tiny Images, 80 million GISTs, 16 bytes per vector					
Multi-D-ADC	10000	0.06	0.49	0.64	19
Multi-D-ADC	30000	0.06	0.56	0.76	46
Multi-D-ADC	100000	0.06	0.56	0.85	139

Table 1. The performance (recall for the top-1, top-10, and top-100 matches after reranking + time in milliseconds) of the Multi-D-ADC system (based on the second-order multi-index with K=2^{14}) for different datasets, different compression levels. We also give the performance of the IVFADC and IVFADC+R (our reimplementation for IVFADC as well as numbers reproduced from [11] in brackets – the timings are not directly comparable in the latter case).

the remainder of the section is to evaluate the performance of the inverted multi-index within the systems for the approximate nearest neighbor search. The systems we consider combine querying an inverted multi-indices with the subsequent reranking of candidate lists. To enable reranking, for each entry of the lists in the index, we store few extra bytes corresponding to the product quantization (PQ) of the respective point. At test time, for each candidate point on the list, those extra bytes are used to reconstruct its position in R^M [9]. We then evaluate the distance between the query $\mathbf{q}$ and the reconstructed candidate point and rerank all candidates according to the increasing distance (i.e. we use the *asymmetric distance computation (ADC)* idea of [9]). We then consider the top T^* points with the lowest reconstructed distance and report the recall@T^* (T^*<T).

We consider two variants of the query+reranking system: *Multi-ADC* and *Multi-D-ADC* (analogous to systems called 'ADC' and 'IVFADC', respectively, in [9]). To obtain the extra bytes for reranking, the Multi-ADC system applies product quantization to the original dataset vectors, while Multi-D-ADC applies product quantization to the residual displacement between each point p and the closest centroid $[\mathbf{u}_i\mathbf{v}_j]$ (at test time this residual displacement is reconstructed and added to the centroid). In general, for the same number of extra bytes, Multi-D-ADC leads to higher recall@T^* than Multi-ADC, because residual displacements have smaller magnitudes than the original points and hence allow less lossy PQ compression. At the same time, Multi-ADC is faster since it allows efficient precomputation of a single look-up table for the ADC computation [9].

In the first experiment, we evaluate the Multi-ADC system with $m = 8$ extra bytes per vector (each vector is split into 8 dimension chunks and the PQ vocabularies of size 256 are used). Figure 5 then gives recall@T^* for $T^* = 1, 10, 100, 1000, 10000$ (different curves) on the BIGANN dataset as a function of the original candidate list length T returned by the inverted multi-index. As a baseline, we give the performance of the ADC system of [9] that essentially reranks the entire dataset ($T = 1$ billion), which takes several seconds per query. Figure 5 shows that, depending on T^*, it is sufficient to query only few hundred to few tens of thousand (i.e. a tiny fraction of the entire billion-size dataset) to match the performance of a system that reranks the entire dataset. At this point, the shortcomings of lossy compression within ADC seem to supersede (on average) whatever retrieval errors are made within the inverted multi-index[2].

In the final set of experiments, we compare the performance (recall@T^* and timings) of the Multi-D-ADC system for $T^* = 1, 10, 100$, $T = 10000, 30000, 100000$, and the number of extra bytes $m = 8, 16$. This performance is summarized in Table 4. For the *Tiny Images* dataset, we visualize few qualitative results of retrieval with Multi-D-ADC in Figure 6. For the BIGANN dataset, we give the recall and timings for our own re-implementation of the IVFADC system closely following the description in [9, 11]. We also reproduce the perfromance for the IVFADC system (state-of-the-art for $m = 8$ extra bytes) and for IVFADC+R system (state-of-the-art for $m = 16$ extra bytes) from [11] (the timings are thus computed on a different CPU).

Overall, it can be observed that for the same level of compression, the use of the inverted multi-indices gives Multi-D-ADC a very substantial speed advantage (about an order of magnitude for comparable recall accuracy) over IVFADC(+R). This is achieved because Multi-D-ADC has to rerank much shorter candidate lists (tens of thousands vs. millions) to achieve similar or better recall values compared to IVFADC(+R). The memory overhead of Multi-D-ADC compared to IVFADC(+R) is about 8% ($\sim$13GB vs. $\sim$12GB) for $m = 8$ and about 5% ($\sim$21GB vs. $\sim$20GB) for

[2] Curiously, the curves for Multi-ADC actually rise above the performance of full reranking before converging to it. We believe that this effect can have the following explanation. Because the PQ encoding is lossy, some "nasty" vectors are considered to be closer than the true nearest neighbor (NN) *after* reranking. In some cases, as T grows, the true NN first enters the top T^* short list but then "sinks" out of it, as more and more of such "nasty" vectors enter the list of T candidate points.

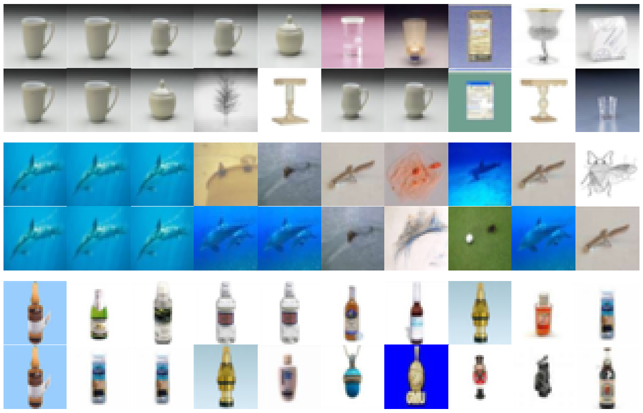


Figure 6. Retrieval examples on the Tiny Images dataset (the images associated with GIST vectors are shown). In each of the three row pairs, the left-most images correspond to the query, the top row corresponds to Euclidean nearest neighbors found by exhaustive search, the bottom row are the top matches returned by the Multi-D-ADC system ($K = 2^{14}$, $m = 16$ extra bytes). Empirically, for most examples, we observed that the top matches returned by a Multi-D-ADC are similar in terms of semantic similarity to the exhaustive search on uncompressed vectors (top two rows) with few exceptions (bottom row).

$m = 16$ (all numbers include 4GB that are required to store point IDs).

5. Summary and outlook

We have introduced the *inverted multi-index*, which is a new data structure for the large-scale retrieval in the datasets of high-dimensional vectors. In our evaluation, multi-indices significantly outperformed standard inverted indices in terms of the accuracy of the returned candidate lists. In combination with a suitable reranking procedure, inverted multi-indices improved considerably previously reported speed and accuracy of approximate nearest neighbor search on the BIGANN dataset of 1 billion SIFT vectors.

The idea of multi-index can have a wider applicability then just approximate nearest neighbor search. Thus, multi-indices can be used within retrieval systems that combine the candidate lists returned for multiple descriptors extracted from the same query image [23]. There, replacing candidate lists corresponding to a single codeword with something closer to nearest neighbor search has been shown to improve the accuracy significantly albeit at a considerable computational cost (cf. e.g. [19]). Furthermore, it is straightforward to replace the (square of the) Euclidean distance within the multi-index with any other additive distance measure or kernel; it will thus be interesting to evaluate inverted multi-indices within large-scale machine learning systems.

Acknowledgement. We would like to thank two CVPR reviewers for their exceptionally useful reviews.

References

[1] Google Goggles. `http://www.google.com/mobile/goggles`.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9), 1975.

[3] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. *CVPR*, 2008.

[4] O. Chum, J. Philbin, J. Sivic, M. Isard, and A. Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval. *ICCV*, 2007.

[5] J. Deng, A. C. Berg, K. Li, and F.-F. Li. What does classifying more than 10, 000 image categories tell us? *ECCV*, 2010.

[6] J. Hays and A. A. Efros. Scene completion using millions of photographs. *ACM Trans. Graph.*, 26(3), 2007.

[7] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *STOC*, 1998.

[8] H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. *ECCV*, 2008.

[9] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1), 2011.

[10] H. Jegou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. *CVPR*, 2010.

[11] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. *ICASSP*, 2011.

[12] H. Lejsek, B. T. Jónsson, and L. Amsaleg. NV-Tree: nearest neighbors at the billion scale. *ICMR*, 2011.

[13] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2), 2004.

[14] T. Malisiewicz and A. A. Efros. Beyond categories: The visual memex model for reasoning about object relationships. *NIPS*, 2009.

[15] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. 2008.

[16] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. *CVPR*, 2006.

[17] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42(3), 2001.

[18] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. *CVPR*, 2007.

[19] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. *CVPR*, 2008.

[20] J. Philbin, M. Isard, J. Sivic, and A. Zisserman. Descriptor learning for efficient retrieval. *ECCV*, 2010.

[21] M. Raginsky and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. *NIPS*, 2009.

[22] R. Salakhutdinov and G. E. Hinton. Semantic hashing. *Int. J. Approx. Reasoning*, 50(7), 2009.

[23] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. *ICCV*, 2003.

[24] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *TPAMI*, 30(11), 2008.

[25] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. *CVPR*, 2008.

[26] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. `http://www.vlfeat.org/`.

[27] Y. Zhang, Z. Jia, and T. Chen. Image retrieval with geometry-preserving visual phrases. *CVPR*, 2011.