Fast k Nearest Neighbor Search using GPU

Vincent Garcia Eric Debreuve
Michel Barlaud
Université de Nice-Sophia Antipolis/CNRS
Laboratoire I3S, 2000 route des Lucioles, 06903 Sophia Antipolis, France
garciav@i3s.unice.fr

Abstract

Statistical measures coming from information theory represent interesting bases for image and video processing tasks such as image retrieval and video object tracking. For example, let us mention the entropy and the Kullback-Leibler divergence. Accurate estimation of these measures requires to adapt to the local sample density, especially if the data are high-dimensional. The k nearest neighbor (kNN) framework has been used to define efficient variable-bandwidth kernel-based estimators with such a locally adaptive property. Unfortunately, these estimators are computationally intensive since they rely on searching neighbors among large sets of d-dimensional vectors. This computational burden can be reduced by pre-structuring the data, e.g. using binary trees as proposed by the Approximated Nearest Neighbor (ANN) library. Yet, the recent opening of Graphics Processing Units (GPU) to general-purpose computation by means of the NVIDIA CUDA API offers the image and video processing community a powerful platform with parallel calculation capabilities. In this paper, we propose a CUDA implementation of the "brute force" kNN search and we compare its performances to several CPU-based implementations including an equivalent brute force algorithm and ANN. We show a speed increase on synthetic and real data by up to one or two orders of magnitude depending on the data, with a quasi-linear behavior with respect to the data size in a given, practical range.

1. Introduction

Information theory provides robust[1] statistical measures for image and video processing tasks such as image or video restoration, segmentation, retrieval, and video object tracking. Such measures include entropies and divergences. In particular, the Kullback-Leibler divergence (KLD) has been successfully used as a (dis)similarity measure for region-of-interest tracking [3] or content-based image retrieval [1, 7, 9]. Let P and Q be two probability distributions with values in $\mathbb{R}^d$. The KLD is a measure of the difference between P and Q given by, in the continuous case,

$$D_{\mathrm{KL}}(P\|Q) = \int_{\mathbb{R}^d} f_P(x) \log \frac{f_P(x)}{f_Q(x)} \mathrm{d}x \tag{1}$$

where f_P and f_Q denote the density functions of P and Q, respectively. In practice, the estimation of (1) from point sets $\mathcal{P} = \{p_1, p_2, \cdots, p_{N_\mathcal{P}}\}$ and $\mathcal{Q} = \{q_1, q_2, \cdots, q_{N_\mathcal{Q}}\}$ drawn from P and Q, respectively, is a tough problem because of the often sparse and usually unevenly sampling of $\mathbb{R}^d$. As d gets larger, the estimation using classical techniques tends to get less accurate. This phenomenon is referred to as the curse of dimensionality. The k nearest neighbor (kNN) framework allows to adapt to the local point density, accounting for both the sparsity and the irregularity of the space sampling. The following kNN-based estimator of DKL has been proposed

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) = &\log \frac{N_\mathcal{Q}}{N_\mathcal{P} - 1} \\ &+ \frac{d}{N_\mathcal{P}} \sum_{i=1}^{N_P} \log(\rho_k(p_i, \mathcal{Q})) \\ &- \frac{d}{N_\mathcal{P}} \sum_{i=1}^{N_P} \log(\rho_k(p_i, \mathcal{P} \setminus \{p_i\})) \end{aligned} \tag{2}$$

where $\rho_k(p_i, \mathcal{S})$ denotes the distance between point p_i and its kth nearest neighbor in the point set $\mathcal{S}$. The distance is normally computed in the Euclidean sense. One can see that, behind its apparent simplicity, the actual computation of (2) is highly computationally demanding. Conceptually, for each point of $\mathcal{P}$, the distances to every other point of $\mathcal{P}$ and to every point of $\mathcal{Q}$ must be computed, then sorted (separately for $\mathcal{P}$ and $\mathcal{Q}$) to determine kth nearest neighbors. This represents a polynomial complexity in terms of

[1]In the usual sense: reduced sensitivity to outliers in the data.

point set size and will be referred to as the "brute force" approach. Several kNN algorithms [2] have been proposed in order to reduce the complexity of these kth nearest neighbor searches. They generally seek to reduce the number of distances that have to be computed using, for instance, a pre-arrangement of the data using a kd-tree structure.

Through the C-based API CUDA (Compute Unified Device Architecture), NVIDIA[2] recently brought the power of parallel computing on Graphics Processing Units (GPU) to general-purpose algorithmic [4, 5]. This opportunity represents a promising alternative to solve the kNN problem in reasonable time. In this paper, we propose a CUDA implementation for solving the brute force kNN search problem. We compared its performances to several CPU-based implementations. Besides being faster by up to two orders of magnitude, we noticed that the dimension of the sample points has only a small impact on the computation time with the proposed CUDA implementation, contrary to the C-based implementations.

2. Brute force kNN search

2.1. Principle

Let $\mathcal{R} = \{r_1, r_2, \cdots, r_m\}$ be a set of m reference points with values in $\mathbb{R}^d$, and let $\mathcal{Q} = \{q_1, q_2, \cdots, q_n\}$ be a set of n query points in the same space. The kNN search problem consists in searching the k nearest neighbors of each query point $q_i \in \mathcal{Q}$ in the reference set $\mathcal{R}$ given a specific distance. Commonly, the Euclidean or the Manhattan distance is used but any other distance can be used instead such as the Chebyshev norm or the Mahalanobis distance. Figure 1 illustrates the kNN problem with $k = 3$ and for a point set with values in $\mathbb{R}^2$.

One way to search the kNN is the "brute force" algorithm

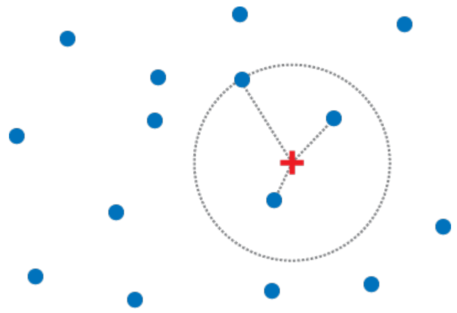
Figure 1. Illustration of the kNN search problem for $k = 3$. The blue points correspond to the reference points and the red cross corresponds to the query point. The circle gives the distance between the query point and the third closest reference point.

(noted BF) also called "exhaustive search". For each query point q_i, the BF algorithm is the following:

[2]http://www.nvidia.com/page/home.html
http://www.nvidia.com/object/cuda_home.html

1. Compute all the distances between q_i and r_j, $j \in [1, m]$.
2. Sort the computed distances.
3. Select the k reference points corresponding to the k smallest distances.
4. Repeat steps 1. to 3. for all query points.

The main issue of this algorithm is its huge complexity: $O(nmd)$ for the nm distances computed (approximately $2nmd$ additions/subtractions and nmd multiplications) and $O(nm \log m)$ for the n sorts performed (mean number of comparisons).
Several kNN algorithms have been proposed in order to reduce the computation time. Generally, the idea is to reduce the number of distances computed [6]. For instance, some algorithms [2] partition the point sets using a kd-tree structure, and only compute distances within nearby volumes. As will be seen in Section 3, according to our experiments, the use of such a method is generally faster than a CPU-based BF method up to a factor 10.

2.2. Sorting algorithm

The second step of the BF algorithm is the sort of the computed distances. In this section, we discuss the choice of the sorting algorithm.

The Quicksort being a popular algorithm, let us say a few words about it. In practice, it is one of the fastest algorithms. However, it is recursive and CUDA does not allow recursive functions. As a consequence, it cannot be used in our implementation.
The comb sort complexity is $O(n \log n)$ both in the worst and average cases. It is also among the fastest algorithms and simple to implement. Nevertheless, keeping in mind that we are only interested in the k smallest elements, k being usually very small compared to N_P and N_Q, we considered using an insertion sort variant which only outputs the k smallest elements. As will be illustrated in Section 3.2, this algorithm is faster than the comb sort for small values of parameter k.

2.3. CUDA implementation

The BF method is by nature highly-parallelizable. This property makes the BF method perfectly suitable for a GPU implementation. Let us remind that the BF method has two steps: the distance computation and the sorting. For simplicity, let us assume here that the reference and query sets both contain n points.
The computation of the n^2 distances can be fully parallelized since the distances between pairs of points are independent. Two kinds of memory are used: global memory and texture memory. The global memory has a huge

bandwith but the performances decrease if the memory accesses are non-coalesced. In such a case, the texture memory is a good option because there are less penalties for non-coalesced readings. As a consequence, we use global memory for storing the query set (coalesced readings), and texture memory for the reference set (non-coalesced readings). Therefore, we obtain better performances than when using global memory and shared memory[3] as proposed in the matrix multiplication example provided in the CUDA SDK.
The n sortings can also be parallelized while the operations performed during a given sorting of n values are clearly not independent of each other. Each thread sorts all the distances computed for a given query point. The sorting consists in comparing and exchanging many distances in a non-predictable order. Therefore, the memory accesses are not coalesced, indicating that the texture memory could be appropriate. However, it is a read-only memory. Only the global memory allows readings and writings. This penalizes the sorting performance.

3. Experiments

3.1. Setup

The computer used to do this comparison is a Pentium 4 3.4 GHz with 2GB of DDR2 memory PC2-5300 ($4 \times$512MB dual-channel memory). The graphic card used is a NVIDIA GeForce 8800 GTX with 768MB of DDR3 memory and 16 multiprocessors interfaced with a PCI-express 1.1 port.

3.2. Comb sort vs. insertion sort

Figure 2 shows the computation time of the kNN search as a function of the parameter k for the comb sort and the insertion sort, both implemented in CUDA. For this experiment, 4800 points (both reference and query sets) drawn uniformly in a 64 dimensional space were used. Using the comb sort, the computation time is constant whatever the value k because all the distances are sorted. On the contrary, using the insertion sort, the computation time linearly increases with k. We define k_0 as follow: the comb sort and the insertion sort are performed in the same computation time for $k = k_0$. k_0 is the abscissa value of the intersection of the two straight lines shown in Fig. 2. For $k < k_0$, the insertion sort is faster than comb sort. Beyond k_0, the comb sort is the fastest. Figure 3 shows the value of k_0 as a function of the size of sets. k_0 approximately increases linearly. According to our experiments, the affine function approximating this increase, computed by linear regression, is given by:

$$k_0(n) = 0.0247n + 1.3404 \tag{3}$$

[3]Memory shared by a set of threads with high bandwidth and no penalties for random memory accesses.

where n is the size of the reference and query sets. The judicious choice of the sorting algorithm used depends both on the size of sets and on the parameter k. In our experiments, we used the insertion sort because it provided the smallest computation time due to the value of k and the size of point sets used.

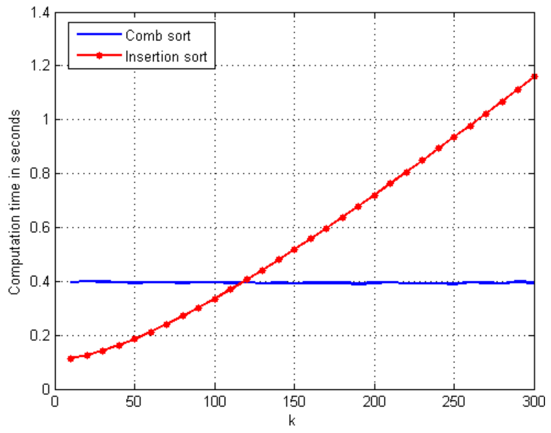

Figure 2. Evolution of the computation time for comb sort (blue line) and insertion sort (red line) algorithms as a function of parameter k. For this experiment, 4800 points (reference and query sets) are used in a 64 dimensional space. The computation time is constant for the comb sort and linearly increases for the insertion sort. The intersection represents the value of k where both algorithms provides a similar computation time. This k value is noted k_0.

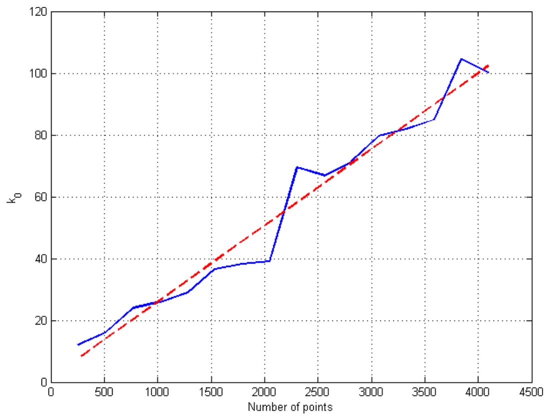

Figure 3. Evolution of k_0 as a function of the size of sets in a 64 dimensional space. The red dashed line is the linear approximation of the experimental curve (blue solid line) computed by linear regression. Bellow this line, the insertion sort is faster than the comb sort algorithm, and above this line, comb sort is the fastest algorithm.

3.3. Performances

The initial goal of our work was to speed up the kNN search process in a Matlab program. In order to speed up

computations, Matlab allows to use external C functions (Mex functions). Likewise, a recent Matlab plug-in allows to use external CUDA functions. In this section, we show, through a computation time comparison, that CUDA greatly accelerates the kNN search process. We compare three different implementations of the BF method and one method based on kd-tree. This kd-tree based kNN implementation is the ANN C++ library (Approximate Nearest Neighbor) [2, 8]. ANN supports both exact and approximate nearest neighbor searching in spaces of various dimensions. We used the exact search.
The methods compared are:

- BF method implemented in Matlab (noted BF-Matlab)
- BF method implemented in C (noted BF-C)
- BF method implemented in CUDA (noted BF-CUDA)
- ANN C++ library (noted ANN-C++)

The table 1 presents the computation time of the kNN search process for each method and implementation listed before. This time depends on the size of the point sets (reference and query sets), on the space dimension, and on the parameter k. In this paper, k was set to 20. The sort used for BF methods is the insertion sort because it provided smaller computation times than the comb sort and the quicksort [4].
In the Table 1, n corresponds to the number of reference and query points drawn uniformly, and d corresponds to the space dimension. The computation time, given in seconds, corresponds respectively to the methods BF-Matlab, BF-C, ANN-C++, and BF-CUDA. The chosen values for n and d are typical values that can be found in papers using the kNN search.

The main result of this paper is that CUDA allows to greatly reduce the time needed to resolve the kNN search problem. According to the table 1, BF-CUDA is up to 407 times faster than BF-Matlab, 295 times faster than BF-C, and 148 times faster than ANN-C++. For instance, with 38400 reference and query points in a 96 dimensional space, the computation time is 57 minutes for BF-Matlab, 44 minutes for BF-C, 22 minutes for the ANN-C++, and less than 10 seconds for the BF-CUDA. The considerable speed up we obtain comes from the highly-parallelizable property of the BF method.

Let us consider the case where $n = 4800$ (see Table 1). The computation time seems to increase linearly with the dimension of the points (see Fig. 4). The major difference between these methods is the slope of the increase. For sets of 4800 points, the slope is 0.54 for BF-Matlab method, 0.45 for BF-C method, 0.20 for ANN-C++ method, and quasi-null (actually 0.001) for BF-CUDA method. In other words, all the methods are sensitive to the space dimension in term of computation time. However, regarding to the tested methods, the impact of the dimension on the performances is quasi-negligible for the method BF-CUDA. This behavior is more important for sets of 38400 points. The slope is 31 for BF-Matlab, 29 for BF-C, 15 for ANN-C++, and 0.087 for BF-CUDA. This characteristic is particularly useful for applications like kNN-based content-based image retrieval [1, 9]: the descriptor size is generally limited to allow a fast retrieval process. With our CUDA implementation, this size can be much higher bringing more precision to the local description and consequently to the retrieval process.

[4]Quicksort was only tested for BF-Matlab and BF-C.

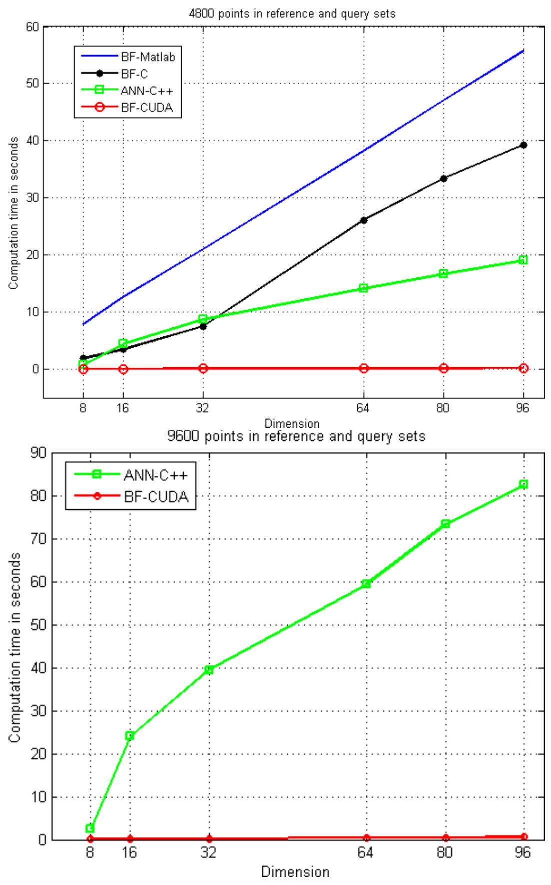

Figure 4. Evolution of the computation time as a function of the point dimension for methods BF-Matlab, BF-C, BF-CUDA, and ANN-C++ for a set of 4800 points (top figure). The bottom figure makes the same comparison for a set of 9600 points and only for methods BF-CUDA and ANN-C++ to compare CUDA with the fastest method tested. The computation time linearly increases with the dimension of the points whatever the method used. However, the increase is quasi-null with the BF-CUDA.

	Methods	n=1200	n=2400	n=4800	n=9600	n=19200	n=38400
d=8	BF-Matlab	0.51	1.69	7.84	35.08	148.01	629.90
	BF-C	0.13	0.49	1.90	7.53	29.21	127.16
	ANN-C++	0.13	0.33	0.81	2.43	6.82	18.38
	BF-CUDA	**0.01**	**0.02**	**0.04**	**0.13**	**0.43**	**1.89**
d=16	BF-Matlab	0.74	2.98	12.60	51.64	210.90	893.61
	BF-C	0.22	0.87	3.45	13.82	56.29	233.88
	ANN-C++	0.26	1.06	5.04	23.97	91.33	319.01
	BF-CUDA	**0.01**	**0.02**	**0.06**	**0.17**	**0.60**	**2.51**
d=32	BF-Matlab	1.03	5.00	21.00	84.33	323.47	1400.61
	BF-C	0.45	1.79	7.51	30.23	116.35	568.53
	ANN-C++	0.39	1.78	9.21	39.37	166.98	688.55
	BF-CUDA	**0.01**	**0.03**	**0.08**	**0.24**	**0.94**	**3.89**
d=64	BF-Matlab	2.24	9.37	38.16	149.76	606.71	2353.40
	BF-C	1.71	7.28	26.11	111.91	455.49	1680.37
	ANN-C++	0.78	3.56	14.66	59.28	242.98	1008.84
	BF-CUDA	**0.02**	**0.04**	**0.11**	**0.40**	**1.57**	**6.65**
d=80	BF-Matlab	2.35	11.53	47.11	188.10	729.52	2852.68
	BF-C	2.13	8.43	33.40	145.07	530.44	2127.08
	ANN-C++	0.98	4.29	17.22	73.22	302.44	1176.39
	BF-CUDA	**0.02**	**0.04**	**0.13**	**0.48**	**1.98**	**8.17**
d=96	BF-Matlab	3.30	13.89	55.77	231.69	901.38	3390.45
	BF-C	2.54	10.56	39.26	168.58	674.88	2649.24
	ANN-C++	1.20	4.96	19.68	82.45	339.81	1334.35
	BF-CUDA	**0.02**	**0.05**	**0.15**	**0.57**	**2.29**	**9.61**

Table 1. Comparison of the computation time, given in seconds, of the methods BF-Matlab, BF-C, ANN-C++, and BF-CUDA. BF-CUDA is up to 407 times faster than BF-Matlab, 295 times faster than BF-C, and 148 times faster than ANN-C++.

Fig. 5 shows the computation time as a function of the number of points n. The computation time increases polynomially with n except for the BF-CUDA which seems rather insensitive to n in the range of test.

Table 1 gives the computation time of the whole kNN search process. We studied how this total time decomposes between each step of the BF method as a function of parameters n, d, and k. This was done with the CUDA profiler[5] (see Tables 2, 3, and 4). Remember that n^2 distances are computed and n sortings are performed. The time proportion spent for distance computation increases with n and d. On the opposite, the time proportion spent for sorting distances increases with k, as seen in Section 3.2. In the case where $n = 4800$, $d = 32$, and $k = 20$, the total time decomposes in 66% for distance computation and 32% for sorting. The remaining time is mainly spent in memory transfer from host (CPU) to device (GPU) and conversely. As a comparison, table 5 presents the time decomposition for BF-C. Ninety-one percent of the time is dedicated to distance computation and 4% to sorting. In CUDA, the full parallelization of the n^2 distance computations makes the decomposition between distance computation and sorting more balanced while BF-C spends most of the time computing the distances, the sorting representing only a small overhead made of comparisons and swapping.

[5]The CUDA profiler is downloadable on the NVIDIA forum.

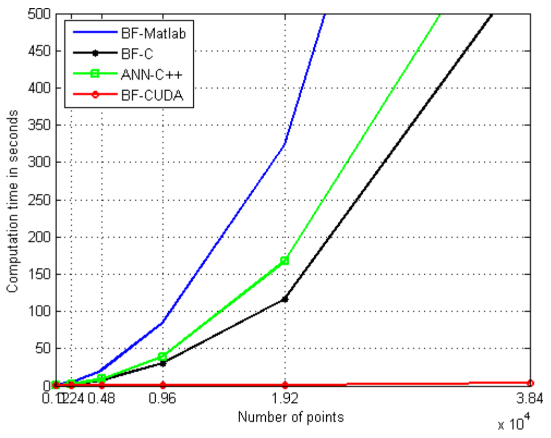

Figure 5. Evolution of the computation time as a function of the number of points for methods BF-Matlab, BF-C, BF-CUDA, and ANN-C++. In this table, D was set to 32 and k was set to 20. The computation time polynomially increases with the number of points whatever the method used. However, in comparison to other methods, the increase is quasi-null with the BF-CUDA.

n	2400	4800	9600
Distance	28%	51%	59%
Sort	70%	47%	40%
Memory copy	2%	2%	1%
Total time	0.023s	0.055s	0.169s

Table 2. Computation time decomposition for each step of the BF algorithm implemented in CUDA as a function of the size n of the point sets. In this table, $d = 16$ and $k = 20$.

d	8	16	32
Distance	37%	51%	66%
Sort	62%	47%	32%
Memory copy	1%	2%	2%
Total time	0.040s	0.055s	0.076s

Table 3. Computation time decomposition for each step of the BF algorithm implemented in CUDA as a function of the dimension d. In this table, $n = 4800$ and $k = 20$.

k	5	10	20
Distance	82%	71%	51%
Sort	15%	26%	47%
Memory copy	3%	3%	2%
Total time	0.033s	0.037s	0.055s

Table 4. Computation time decomposition for each step of the BF algorithm implemented in CUDA as a function of k. In this table, $n = 4800$ and $d = 16$.

	Distance	Sort	Total time
CPU	91%	4%	7.51s
GPU	66%	32%	0.076s

Table 5. Comparison of the computation time decomposition for BF-C (CPU) and BF-CUDA (GPU) implementation. In this table, $n = 4800$, $d = 32$, and $k = 20$. First, the CUDA implementation is 100 times faster than the C implementation. Second, the computation time decomposition between distance computation and sorting is more balanced with CUDA: the distance computation is more costly; however, CUDA allows its full parallelization.

Until now, the studies rely on synthetically generated sets of points. The CUDA implementation of the kNN search has also been used in a real-world image retrieval task [1, 9]. The authors propose a retrieval method based on a multiscale approach. A set of feature vectors is defined at each scale. The vector dimension is either 18 or 27, and the size of the vector sets varies between 4800 and 1200 depending on the scale. Kullback-Leibler divergences are then combined into a similarity measure to compare two images. The authors noticed that our CUDA implementation of the BF algorithm outperformed the “smarter” ANN implementation by a factor of 10 at least (0.2s to compare two images with CUDA instead of 2.2s with ANN).

4. Conclusion

In this paper, we proposed a fast, parallel k nearest neighbor (kNN) search implementation using a graphics processing units (GPU). We showed that the use of the NVIDIA CUDA API accelerates the kNN search by up to a factor of 400 compared to a brute force CPU-based implementation. In particular, this improvement can have a significant impact in content-based image retrieval applications which use kNN.

References

[1] S. Anthoine, E. Debreuve, P. Piro, and M. Barlaud. Using neighborhood distributions of wavelet coefficients for on-the-fly, multiscale-based image retrieval. In *Workshop on Image Analysis for Multimedia Interactive Services*, Klagenfurt, Austria, May 2008.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[3] S. Boltz, E. Debreuve, and M. Barlaud. High-dimensional statistical distance for region-of-interest tracking: Application to combining a soft geometric constraint with radiometry. In *IEEE International Conference on Computer Vision and Pattern Recognition*, Minneapolis, USA, 2007. CVPR’07.

[4] R. Dudek, C. Cuenca, and F. Quintana. Accelerating space variant gaussian filtering on graphics processing unit. In *Computer Aided Systems Theory EUROCAST 2007*, pages 984–991, 2007.

[5] S. Heymann, K. Muller, A. Smolic, B. Frohlich, and T. Wiegand. Sift implementation and optimization for general-purpose gpu. In *15-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG’07)*, 2007.

[6] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB ’07: Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.

[7] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions Pattern Analysis Machine Intelligence*, 27(10):1615–1630, 2005.

[8] D. M. Mount and S. Arya. Ann: A library for approximate nearest neighbor searching, http://www.cs.umd.edu/˜mount/ANN/.

[9] P. Piro, S. Anthoine, E. Debreuve, and M. Barlaud. Image retrieval via kullback-leibler divergence of patches of multiscale coefficients in the knn framework. In *IEEE International Workshop on Content-Based Multimedia Indexing*, London, UK, June 2008. IEEE Computer Society.