K-NEAREST NEIGHBOR SEARCH: FAST GPU-BASED IMPLEMENTATIONS AND APPLICATION TO HIGH-DIMENSIONAL FEATURE MATCHING

Vincent Garcia[1], *Éric Debreuve*[2], *Frank Nielsen*[1,3], *Michel Barlaud*[2]

[1]École Polytechnique, Laboratoire d'informatique LIX, 91128 Palaiseau Cedex, France
[2]Laboratoire I3S, 2000 route des lucioles, BP 121, 06903 Sophia Antipolis Cedex, France
[3]Sony CSL 3-14-13 Higashi Gotanda, Shinagawa-Ku, 141-0022 Tokyo, Japan

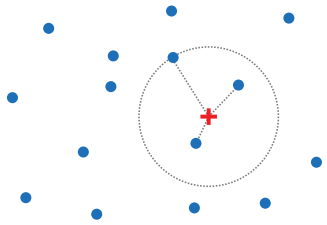

Fig. 1. Illustration of the kNN search problem in $\mathbb{R}^2$ with $k = 3$ using the Euclidean distance.

ABSTRACT

The k-nearest neighbor (kNN) search problem is widely used in domains and applications such as classification, statistics, and biology. In this paper, we propose two fast GPU-based implementations of the brute-force kNN search algorithm using the CUDA and CUBLAS APIs. We show that our CUDA and CUBLAS implementations are up to, respectively, 64X and 189X faster on synthetic data than the highly optimized ANN C++ library, and up to, respectively, 25X and 62X faster on high-dimensional SIFT matching.

Index Terms— k-nearest neighbors, GPU, CUDA/CUBLAS, SIFT

1. INTRODUCTION

The k-nearest neighbor (kNN) search is a problem found in many research and industrial domains such as 3-dimensional object rendering, content-based image retrieval [1], statistics (estimation of entropies and divergences [2]), biology (gene classification [3])...

Let us consider a set $\mathcal{R}$ of m reference points $\mathcal{R} = \{r_1, r_2, \cdots, r_m\}$ defined in a d-dimensional space, and let q be a query point defined in the same space. The kNN search problem consists in determining the k points closest to q among $\mathcal{R}$. The distance considered between two points is not restricted to the Euclidean distance. For some applications, other distances are indeed more adapted to the nature of points (*e.g.*, a mutual information-based metric in the case of histograms). Figure 1 illustrates an example of the kNN search problem in a 2-dimensional space for $k = 3$ using the Euclidean distance. The blue dots are the reference points and the red cross is the query point.

The exhaustive search, also called brute-force algorithm, is a basic kNN search method consisting in computing the distances between the query point and each of the reference points. Then, the k-nearest neighbors are trivially determined using a sorting algorithm. Behind its apparent simplicity, this algorithm is highly demanding in terms of computation time. In the last decades, several approaches [4, 5] have been proposed with one common goal: to reduce the computation time. These methods generally seek to reduce the number of distances that have to be computed using, for instance, a pre-arrangement of the data. The direct consequence is a speed-up of the searching process. However, in spite of this improvement, the computation time required by the kNN search still remains the bottleneck of methods based on kNN.

General-purpose computing on graphics processing units (GPGPU) is the technique of using a Graphics processing unit (GPU) to perform the computations usually handled by the CPU. The key idea is to use the parallel computing power of the GPU to achieve significant speed-ups. Numerous recent publications use the GPU programming to speed-up their methods [6, 7]. In a previous work [8], we showed that the implementation of the brute-force method using GPU programming (through NVIDIA CUDA) enables to greatly reduce the computation time in comparison to a similar C implementation and to the highly optimized ANN C++ library [4]. In this paper, we modify our approach in order to use the CUBLAS library (CUDA implementation of the BLAS library). We show that the new method/implementation is up to 189 times faster than ANN and up to 4 times faster than our previous approach. Experiments on high dimensional feature matching (SIFT [9]) are reported.

2. BRUTE-FORCE KNN SEARCH AND GPU IMPLEMENTATION

2.1. Algorithm

Let us consider a set $\mathcal{R}$ of m reference points $\mathcal{R} = \{r_1, r_2, \cdots, r_m\}$ in a d-dimensional space and a set $\mathcal{Q}$ of n query points $\mathcal{Q} = \{q1, q_2, \cdots, q_n\}$ in the same space. Given a query point $q \in \mathcal{Q}$, the brute-force algorithm (denoted by BF) is composed of the following steps:

1. Compute the distance between q and the m reference points of $\mathcal{R}$;
2. Sort the m distances;
3. The k-nearest neighbors of q are the k points of $\mathcal{R}$ corresponding to the k lowest distances. The output of the algorithm can be the ordered set of these k distances, the set of the k neighbors (actually their indices in $\mathcal{R}$) ordered by increasing distance, or both.

If we apply this algorithm for the n query points and if we consider the typical case of large sets (both references and queries), the complexity of this algorithm is overwhelming: $O(nmd)$ multiplications for the $n \times m$ distances computed and $O(nm \log m)$ for the

n sorting processes. However, the BF method is by nature highly-parallelizable and, as a consequence, is perfectly suitable for a GPU implementation.

2.2. CUDA implementation

In a recent publication [8], we proposed a GPU implementation of the BF method. This implementation was written using the API NVIDIA CUDA and was composed of two kernels (CUDA functions):

1. The first kernel computed the *distance matrix* of size $m \times n$ containing the distances between the n query points and the m reference points. The computation of this matrix was fully parallelized since the distances between pairs of points are independent: each thread computed the distance between a given query point q_i and a given reference point r_j;
2. The second kernel sorted the distance matrix. The n sorting processes (one for each query point) were parallelized since they are independent: each thread sorted all the distances computed for a given query point.

The sorting algorithm used was a modified version of the insertion sort. Let us assume that the first k element of the array D are already sorted, the proposed version insert an element (let's say the l-th element) into the correct position only if $D[l] < D[k]$. For small values of k, the proposed sorting algorithm appears to be faster than the efficient comb sort algorithm.

Besides the distances, in case the indices of the k-nearest neighbors were also needed, an index matrix of size $m \times n$ and containing on each column the indices of the reference points per query was defined, each column being initialized with the vector $(1, 2, \dots m)^\top$, where $M^\top$ denotes the transpose of M. The element insertion performed in the distance matrix as part of the sorting processes were simultaneously applied to the index matrix so that, in the end, its uppermost $k \times n$-submatrix corresponded to the queries k-nearest neighbor indices ordered by increasing distance.
Working with an initial $m \times n$-index matrix represents a waste of memory. A simple "trick" allows us to work with a $k \times n$-index matrix from the beginning, thus avoiding the $(m - k) \times n$ memory overhead. The sorting process deals with each array column *monotonically* from the first to the last element. Consequently, at each iteration, the index of the considered reference point is known. While sorting, if the l-th element needs to be inserted into the distance matrix, the index value l is also inserted into the $k \times n$-index matrix at the exact same position, iteratively filling in the whole matrix.

The main result of [8] is that the proposed CUDA implementation was up to 300X faster than a similar C implementation and up to 150X faster than the highly optimized ANN C++ library [4].

2.3. CUBLAS implementation

BLAS is the celebrated, highly optimized linear algebra library specialized in vector/matrix operations. CUBLAS is the CUDA implementation of BLAS and improves the performance of the classical BLAS functions. The CUDA implementation of the kNN search [8] was very efficient in terms of computation time. The distance matrix computation represented the main part of the computation time. In this paper, we show that we can greatly improve the global performances of the kNN search by reformulating the way to compute the distance matrix and by using the CUBLAS library.
Let us consider two points x and y in a d-dimensional space

$$x = (x_1, x_2, \cdots x_d)^\top, \quad y = (y_1, y_2, \cdots y_d)^\top, \tag{1}$$

where $M^\top$ denotes the transpose of M. The classical way to compute the Euclidean distance, denoted by ρ, between x and y is

$$\rho(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}. \tag{2}$$

However, this distance computation can be rewritten to involve matrix additions and multiplications

$$\rho^2(x, y) = (x - y)^\top (x - y) = \|x\|^2 + \|y\|^2 - 2x^\top y \tag{3}$$

where $\|.\|$ is the Euclidean norm. The square root is then computed at the end. This approach can be extended to handle sets of points. Let R and Q be two matrices of size $d \times m$ and $d \times n$, resp., containing the m reference points and the n query points, respectively. The $m \times n$-matrix $\rho^2(R, Q)$ containing all the pairwise squared distances between query points and reference points is given by

$$\rho^2(R, Q) = N_R + N_Q - 2R^\top Q. \tag{4}$$

The elements of the i^{th} row of N_R are all equal to $\|r_i\|^2$. The elements of the j^{th} column of N_Q are all equal to $\|q_j\|^2$. The way $\rho^2(R, Q)$ is expressed in Eq. (4) (*i.e.*, through matrix additions and multiplications) is perfectly adapted to a CUBLAS implementation (only N_R and N_Q have to be computed separately using for instance a CUDA kernel). However, this method is highly demanding in terms of memory usage and need to be optimized. Indeed, N_R, N_Q, and $R^\top Q$ are stored in three matrices of size $m \times n$. Nevertheless, the matrices N_R and N_Q have a specific form. To optimize the memory usage, we took advantage of this: we stored N_R and N_Q as vectors of dimension $1 \times m$ and $1 \times n$, respectively. The i^{th} element of N_R is equal to $\|r_i\|^2$, and similarly for N_Q. Then, the addition and subtraction in Eq. (4) were handled by classical CUDA kernels. The proposed kNN search implementation is then based on CUDA and CUBLAS and is composed of the following kernels:

1. Compute the vector N_R using CUDA (coalesced read/write);
2. Compute the vector N_Q using CUDA (coalesced read/write);
3. Compute the $m \times n$-matrix $A = -2R^\top Q$ using CUBLAS;
4. Add the i^{th} element of N_R to every element of the i^{th} row of the matrix A using CUDA (grid of $m \times n$ threads, non coalesced read/write: use of the shared memory); The resulting matrix is denoted by B;
5. Sort in parallel each column of B (with n threads) using the modified insertion sort proposed in [8]; The resulting matrix is denoted by C;
6. Add the j^{th} value of N_Q to the first k elements of the j^{th} column of the matrix C using CUDA (coalesced read/write); The resulting matrix is denoted by D;
7. Compute the square root of the first k elements of D to obtain the k smallest distances (coalesced read/write); The resulting matrix is denoted by E;
8. Extract the uppermost $k \times n$-submatrix of E; The resulting matrix is the desired distance matrix for the k-nearest neighbors of each query.

Note that the matrix names were given for algorithmic clarity only. Actually, once A is computed, all the remaining computations are done "in place", meaning that the matrices from A to E are in fact a single matrix occupying a unique area of memory.

The main computation task (*i.e.*, the computation of A in kernel 3) is performed by CUBLAS. The addition of N_Q to C and the

computation of the square root can be done after the sorting process since these steps do not influence the distance order. By applying these two kernels (6 and 7) at the end of the procedure, the computation time is reduced since only the first k elements of each column are processed.

If the matrix A does not fit into the GPU memory, the query points are splitted, processed separately, and the distances to the k-nearest neighbors are then merged together on the CPU/classical memory side.

As explained in Section 2.2, if the indices of the k-nearest neighbors are also required, an index matrix I of size $k \times n$ is defined. In the kernel 5, the distance ordering is replicated in this index matrix.

2.4. Source code

The source code corresponding to this paper, as well as the code for the paper [8], are available at:
`http://www.i3s.unice.fr/~creative/KNN` under a Creative Common License. They are proposed in two versions: one computing only the distances to the k-th nearest neighbors, the second computing both the distances and the corresponding neighbor indices. In the experiments, we used the second (thus *slowest*) version.

3. EXPERIMENTS

We compared our CUDA and CUBLAS implementations to one of the fastest kNN search method: the C++ library ANN [4]. CUDA will refer to the fully CUDA-based algorithm proposed in [8], CUBLAS will refer to the mixed CUBLAS/CUDA algorithm proposed in this paper, and ANN will refer to the C++ ANN library. The three algorithms were compared using Matlab (The MathWorks, Inc.) Mex files.

The computer used for this comparison was a Dell Precision M6400 laptop (Intel Core 2 duo/2.53GHz, 4Go DDR2 memory, NVIDIA Quadro FX 3700M) with Microsoft Windows XP 32 bits, NVIDIA CUDA 2.2, and Matlab 2007b.

3.1. kNN search on synthetic datasets

We compared the computation times of the three algorithms for synthetic datasets. The points (references and queries) were randomly drawn from a normal distribution $\mathcal{N}(0, 1)$. In this experiment, $n = m$ denotes the number of points (identical for references and queries) and k (the number of neighbors to consider) was set equal to 20 (as a reminder, d is the dimension of the points).

Table 1 shows the evolution of the computation time for different values of n and d for each algorithm. Figure 2 shows the evolution of the log-computation time as a function of d and n, respectively. The computation time increases with n and d for all three algorithms. For ANN, this was expectable. When checking Fig. 2, one can note that the parallelization (CUDA and CUBLAS) is "better achieved" in terms of the dimension than in terms of the number of points. Moreover, regarding the dimension, CUBLAS is closer to the full parallel performances (flat curve) than CUDA.

ANN is consistently faster than CUDA and CUBLAS only for small dimensions ($d \leq 4$) or small number of points ($n \leq 256$). Indeed, in such cases, CUDA and CUBLAS are penalized by the time needed to transfer data from host memory (CPU) to device memory (GPU) and back. For high dimensions and large number of points (usually the case in practice), this penalty becomes negligible compared to the gain in computation time achieved by parallelization. In such conditions, CUDA and CUBLAS were up to 64X and 189X faster than ANN, respectively.

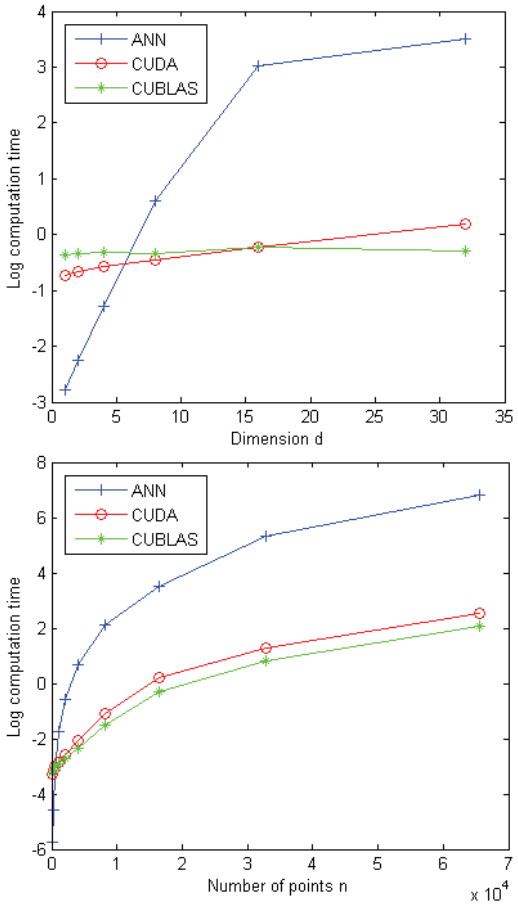

Fig. 2. Log-computation time as a function `(Top)` of d for $k = 20$ and $n = 16384$, and `(Bottom)` of n for $k = 20$ and $d = 32$.

CUBLAS appears faster than CUDA (up to 4X faster) for high dimensional spaces and large datasets. For small values of d and n, CUDA is faster than CUBLAS due to the smaller number of kernels called (2 instead of 7), each kernel costing some time to be called: the time needed to compute the distances for CUDA (1 kernel) is too small to be efficiently replaced by the 6 CUBLAS kernels. Again, in practice, spaces are of high dimension and datasets are large.

3.2. kNN search applied to high-dimensional SIFT matching

We compared the computation times of the three algorithms in the context of high-dimensional SIFT [9] feature matching. This kind of matching can be found in applications such as content-based image retrieval: SIFT features are extracted from a set of reference images and stored into a database. Then, given a query image I, the retrieval process extracts SIFT features from I and, for each of them, finds the k closest features in the database. Finally, a voting algorithm enables to determine the images most similar to I in the image database.

For this experiment, we considered a set $\mathcal{Q}$ of 1024 query points (SIFT features), which is approximately the usual number of features extracted from a single image. The reference set $\mathcal{R}$ contained from $2^7 = 128$ to $2^{16} = 65536$ points. These two sets of SIFT features correspond to a subset of the features extracted from the INRIA Holidays dataset [10]. The dimension of a SIFT feature is

	Method	n=256	n=512	n=1024	n=2048	n=4096	n=8192	n=16384	n=32768	n=65536
d=1	ANN	**0.001**	**0.002**	**0.004**	**0.007**	**0.015**	**0.031**	**0.063**	**0.132**	**0.277**
	CUDA	0.042	0.046	0.054	0.063	0.081	0.154	0.480	2.489	8.302
	CUBLAS	0.042	0.047	0.055	0.066	0.091	0.213	0.698	1.615	5.694
d=4	ANN	**0.003**	**0.005**	**0.012**	**0.027**	**0.059**	**0.125**	**0.275**	**0.591**	**1.364**
	CUDA	0.042	0.048	0.055	0.066	0.086	0.176	0.561	2.591	8.425
	CUBLAS	0.042	0.048	0.057	0.068	0.093	0.220	0.733	1.812	6.076
d=16	ANN	**0.007**	**0.028**	0.109	0.385	1.421	5.468	20.289	84.503	378.496
	CUDA	0.043	0.049	**0.056**	0.070	0.105	0.247	0.805	2.542	8.900
	CUBLAS	0.044	0.049	**0.056**	**0.067**	**0.092**	**0.203**	**0.791**	**2.123**	**7.225**
d=64	ANN	**0.017**	0.073	0.299	0.949	3.279	13.365	74.183	313.527	1296.367
	CUDA	0.044	0.050	0.062	0.087	0.176	0.528	1.950	5.518	20.441
	CUBLAS	0.044	**0.049**	**0.057**	**0.069**	**0.102**	**0.242**	**0.904**	**3.104**	**10.887**
d=256	ANN	0.051	0.194	0.742	2.933	14.579	76.454	334.509	1053.819	3559.731
	CUDA	0.045	0.055	0.081	0.159	0.459	1.641	6.910	19.381	75.718
	CUBLAS	**0.044**	**0.050**	**0.060**	**0.079**	**0.146**	**0.405**	**2.394**	**7.157**	**29.480**

Table 1. Computation times (in seconds) for the kNN search. CUDA and CUBLAS are up to 64X and 189X faster than ANN, respectively.

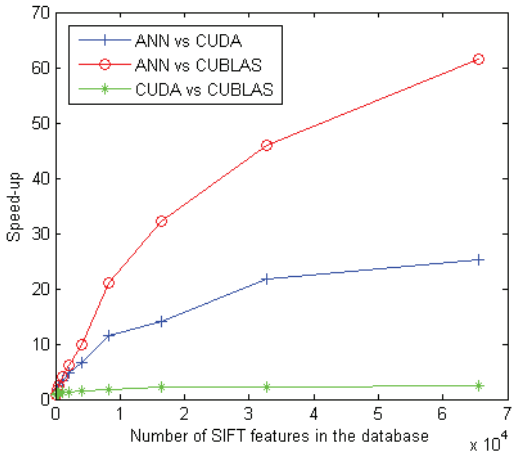

Fig. 3. Speed-up between every pair of algorithms as a function of the number of reference SIFT features ($d = 128$, $k = 20$).

128 and k was set equal to 20. Figure 3 shows the evolution of the speed-up between every pair of algorithms among the three as a function of the number of reference SIFT features. The curve "Alg1 vs. Alg2" must be interpreted as the computation time of Alg1 divided by the computation time of Alg2. Therefore, when the curve is higher than one, it means that Alg2 is faster than Alg1 by a factor equal to the curve level. The speed-up achieved by CUDA or CUBLAS in comparison with ANN increased significantly with the number of features. The speed-up achieved by CUBLAS in comparison with CUDA increased much less. Indeed, both algorithms have recourse to parallelization. However, CUBLAS computes and sorts the distances more efficiently. In this experiment, CUDA was up to 25X faster than ANN, CUBLAS was up to 62X faster than ANN, and CUBLAS is up to 2.5X faster than CUDA.

4. CONCLUSION

We proposed two fast GPU-based implementations of the naive brute-force k-nearest neighbor (kNN) search algorithm based on the APIs CUDA and CUBLAS. In our experiments, CUDA and CUBLAS implementations were up to 64X and 189X faster, respectively, on synthetic data than the highly optimized ANN C++ library, and up to 25X and 62X faster, respectively, on high-dimensional SIFT matching.

5. REFERENCES

[1] H. Zhang, A. C. Berg, M. Maire, and J. Malik, "SVM-KNN: Discriminative nearest neighbor classification for visual category recognition," in *International Conference on Computer Vision and Pattern Recognition*, New York (NY), USA, 2006.

[2] M. N. Goria, N. N. Leonenko, V. V. Mergel, and P. L. Novi Inverardi, "A new class of random vector entropy estimators and its applications in testing statistical hypotheses," *J. Nonparametr. Stat.*, vol. 17, pp. 277–297, 2005.

[3] F. Pan, B. Wang, X. Hu, and W. Perrizo, "Comprehensive vertical sample-based knn/lsvm classification for gene expression analysis," *J. Biomed. Inform.*, vol. 37, pp. 240–248, 2004.

[4] 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*, vol. 45, pp. 891–923, 1998.

[5] H. Jégou, M. Douze, and C. Schmid, "Searching with quantization: approximate nearest neighbor search using short codes and distance estimators," Tech. Rep. RR-7020, INRIA, 2009.

[6] D. Qiu, S. May, and A. Nüchter, "GPU-accelerated nearest neighbor search for 3D registration," in *International Conference on Computer Vision Systems*, Liège, Belgium, 2009.

[7] Y. Zhuge, Y. Cao, and R.W. Miller, "GPU accelerated fuzzy connected image segmentation by using CUDA," in *Engineering in Medicine and Biology Conference*, Minneapolis (MN), USA, 2009.

[8] V. Garcia, É. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *CVPR Workshop on Computer Vision on GPU*, Anchorage (AK), USA, 2008.

[9] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, pp. 91–110, 2004.

[10] H. Jégou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *European Conference on Computer Vision*, Marseille, France, 2008.