# k-Nearest Neighbors algorithm: comparison between sequential, OpenMP and CUDA implementations

Federico Magnolfi
*Università degli Studi di Firenze*
Florence, Italy
federico.magnolfi2@stud.unifi.it

Iacopo Erpichini
*Università degli Studi di Firenze*
Florence, Italy
iacopo.erpichini@stud.unifi.it

***Abstract*—Retrieving the *k-Nearest Neighbors* in a given dataset is a very common problem. Due to the inherent highly parallelizable structure of the problem, it's possible to speed up the search by using a multi-core CPU or a GPU. In this study, we try to investigate how much we can speed up the search task when implementing the algorithm in OpenMP/CUDA, compared to the C++ sequential version. We discuss the results for different dataset sizes and different threads number.**

***Index Terms*—k-Nearest Neighbors, parallel computing, OpenMP, CUDA**

## I. Introduction

Given a dataset $\mathcal{D}$ of $N$ points $x_i \in \mathbf{R}^n$, a test point $q \in \mathbf{R}^n$ and a distance measure $d$, the $k$-Nearest Neighbors problem (k-NN) is to find the $k$ points closest to $q$ in $\mathcal{D}$, i.e. find an ordered subset $\mathcal{S}$ of $\mathcal{D}$ such that:

- $\mathcal{S} = \{x_{i_1}, x_{i_2}, ..., x_{i_k}\}$
- $j < h \implies d(q, x_{i_j}) \leq d(q, x_{i_h})$
- $d(q, x_{i_j}) \leq d(q, x_m), \forall x_{i_j} \in \mathcal{S}, \forall x_m \in \mathcal{D} \setminus \mathcal{S}$

Such a problem is very common in pattern recognition: an algorithm that solves k-NN can be used as a subroutine inside a method that addresses a *classification* or a *regression* task.

In other words, in the case where each dataset point $x_i$ is associated with a value $y_i$, the correspondences obtained with k-NN can be used to learn how to approximate the unknown function $f : x \to y$, for any $x$.

### *Approaches*

There are many possible approaches to solve the k-NN problem: the simplest one (naive approach) consists in calculate distances of all dataset points from the query, and select only the smallest distances after sorting; other approaches like [1] [2] or something that try construct data structures (such as k-d trees) that allow reducing the number of points from which it is necessary to calculate the distance [3]. In this study, we use the naive version because we are not interested in obtaining the state of the art performance, but only in seeing the potential of implementing highly parallelizable algorithms on GPU.

### *Variants*

The k-NN problem formulation can be slightly relaxed to reduce even more the execution times of the searches. One example is *Approximate Nearest Neighbors (a-NN)*: in this variant, we allow the finding of k-Near Neighbors, which don't have to be the *nearest*. The algorithms that solve *ANN* take advantage of techniques and data structures (such as *locality-sensitive hashing*) that make the search stochastic. For these type of algorithms is also important the *quality* of the search results, measured ad example with the *recall* metric.

We do not investigate these types of algorithms in this paper.

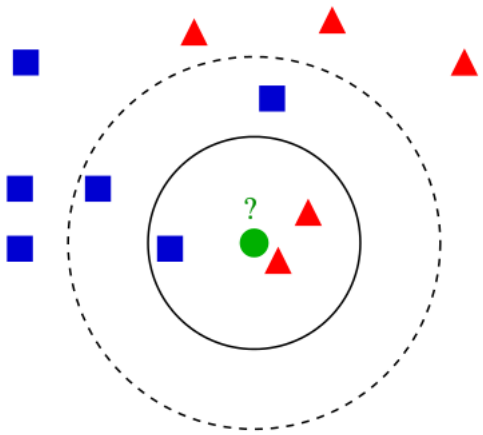


Fig. 1. Example of a k-NN problem: the circle represents the query of which find nearest neighbors. All the others are the dataset points.
In this case, every dataset point is associated with information (shape/color) that can be used for other tasks, after finding the nearest neighbors.

## II. Implementations

### A. Sequential Version

For the sequential version of the k-NN problem we use the simple (naive) implementation: given a query, we calculate the euclidean distance from each point in the dataset. After that, we sort the array of points by increasing distance: finally, the first $k$ elements in the ordered array are the nearest neighbors of the query.

As sorting algorithm we use *qsort*, the implementation of quicksort included in *stdlib*.

The pseudocode is reported in Algorithm 1.

### B. Parallel OpenMP Version

The OpenMP [6] version is similar to the sequential one. The differences are the parallelization of the calculation of the distances and the sorting. The parallelization of the sorting implies a subsequent merge procedure.

**Algorithm 1:** k-Nearest Neighbors - Sequential version

```
Input  : dataset, query, k
Output: nearestNeighbors
// init
indexes = new int[dataset.size]
distances = new float[dataset.size]
// calculate distances
for p from 0 to dataset.size-1 do
    indexes[p] = p
    distances[p] = euclideanDistance(query, dataset[p])
end
// sort by increasing distance
sortByKey(indexes, keys=distances)
// slice
nearestNeighbors = indexes[:k]
```

The pseudo-code is reported in Algorithm 2: here, details about border effects (due to data split among threads) are omitted; furthermore, it is intended that all assignments of arrays are deep copies.

**Algorithm 2:** k-Nearest Neighbors - OpenMP version

```
Input  : dataset, query, k, numThreads
Output: nearestNeighbors
// init
indexes = new int[dataset.size]
distances = new float[dataset.size]
chunkSize = dataset.size / numThreads
// calculate distances
# pragma omp parallel for
for p from 0 to dataset.size-1 do
    indexes[p] = p
    distances[p] = euclideanDistance(query, dataset[p])
end
// sort chunks by increasing distance
# pragma omp parallel for
for j from 0 to numThreads-1 do
    s = j * chunkSize
    e = s + chunkSize
    sortByKey(indexes[s:e], keys=distances[s:e])
end
// merge sorted chunks
for j from 1 to numThreads-1 do
    i, d = indexes[:k], distances[:k]
    s = j * chunkSize
    e = s + k
    indexes[:k], distances[:k]
      =merge(i,d,indexes[s:e],distances[s:e])
end
// slice
nearestNeighbors = indexes[:k]
```

### C. Parallel CUDA Version

The CUDA version differs from the previous mainly because we need to take care of memory transfers between CPU and GPU. Moreover, since GPU threads are very efficient, we launch the kernel that calculates the distances with as many threads as many elements in the dataset: each thread is responsible to calculate the distance from the query to one single dataset element.

As sorting algorithm we use *sort_by_key*, a sorting algorithm included in *thrust* that allows ordering on GPU.

The pseudo-code is reported in Algorithm 3.

**Algorithm 3:** k-Nearest Neighbors - CUDA version

```
Input  : dataset, query as q, k, bSize
Output: nearestNeighbors
// init
indexes = new int[dataset.size]
distances = new float[dataset.size]
// move query to GPU memory
q = toGPU(q)
// determine the number of blocks
blocks = (datasetSize + bSize - 1) / bSize
// calculate the distances
idxs,dists=cudaDistances<<<blocks,bSize>>>(dataset,q)
// sort by increasing distance on GPU
sortByKeyOnGPU(idxs, keys=dists)
// slice and move results to CPU memory
nearestNeighbors = toCPU(idxs[:k])
```

## III. Setup

All the experiments are done in C++14, the output is a csv file. The graphs are obtained with Python3 using the matplotlib library.

### A. Test configuration

We test our code in a laptop with this configuration:

- CPU: Intel Core i7-8750H, 6 cores / 12 threads
- GPU: NVIDIA GeForce GTX 1050 Ti (Notebook), 768 CUDA cores, 4 GiB video RAM, 6 SM
- RAM: 16 GiB
- CUDA: 10.0
- OpenMP: 4.5

### B. Compilation

A full compilation requires the use of the *nvcc* compiler and the *OpenMP* library. However, if one or both of them aren't available, our *CMakeLists.txt* automatically allows a partial compilation.

For the experiments, we compile the code using all available compiler optimizations (-O3) to achieve better performances, taking advantage of techniques like loop vectorization when possible.

## IV. Experiment

We set $k$ (number of *Nearest Neighbors*) to 100 for all tests. When calculating the wall clock time of the algorithms, we distinguish between *initialization time* (required only once) and the time needed to answer all queries (*evaluation time*).

As comparison metrics, we calculate the **wall clock time** for all three algorithms and we report the **speedup** obtained by the parallel versions compared to the sequential version. The *speedup* $S$ is defined as the ratio between the evaluation time of the sequential version $t_S$ and the evaluation time of the parallel version $t_P$, i.e. $S = t_S/t_P$.

We calculate these metrics in many different configurations: distinct configurations have a different *number of dataset examples* and/or a different *number of threads* for the parallel versions. In particular, the number of threads goes from 1 to 6 for the *OpenMP* version, while for *CUDA* we try different setting (32, 64, 128, 256, 512, 1024) of the number of threads per block (a.k.a. *block size*).

When varying the number of threads, we use all dataset examples. When varying the number of dataset examples, we use the maximum number of threads for the *OpenMP* version, and the best block size setting for the *CUDA* version.

The output of the experiment is a *.csv* file that contains a row for each different configuration. Then, this file is read by a Python script that generates the plots reported in the *Results* section.

### A. Dataset

The experiments are made on two SIFT datasets [4], which are sets of 128-dimensional vectors. Each of these vectors is a feature vector that describes local properties in an image.

This dataset is usually used in pattern recognition: keypoints of objects are first extracted from a set of reference images; for each keypoint, the SIFT descriptor is calculated and stored in a database. Objects in a new image are recognized by individually comparing each feature from the new image to this database and finding candidate matching features based on Euclidean distance of their feature vectors: here is where k-NN comes into play.

Each of the two SIFT datasets used in our tests comprises 3 main components:

- *query*: the vectors of which to find nearest neighbors
- *base*: the vectors in which the search is performed
- *groundtruth*: indexes of true nearest neighbors for queries

The first dataset (*siftsmall*) has 10000 base vectors and 100 query vectors each of them with its 100 nearest neighbors. The second dataset (*sift*) has 1000000 base vectors and 10000 query vectors each of them with its 10000 nearest neighbors.

We use the small dataset to verify the correctness of our algorithms. Then we switched to the bigger dataset to run all the tests.

## V. Results

Table I reports the results of the experiment: for each different setting, are reported the processing unit, the number of threads (intended per block for the GPU), the dataset size (number of vectors in the search space) and elapsed times to answer all the queries (divided into init_time and eval_time). From these data, we generate the subsequent plots to better visualize the results. To create the plots, we consider only the evaluation time, whereas the initialization time it's not used.

TABLE I
Results of the experiment (times in seconds)

| hw | threads | dataset_size | init_time | eval_time | total_time |
|---|---|---|---|---|---|
| cpu | 1 | 10000 | 0.001 | 17.387 | 17.389 |
| cpu | 6 | 10000 | 0.001 | 4.345 | 4.346 |
| gpu | 8 | 10000 | 0.284 | 2.855 | 3.139 |
| gpu | 16 | 10000 | 0.035 | 2.705 | 2.740 |
| gpu | 32 | 10000 | 0.035 | 3.128 | 3.163 |
| gpu | 64 | 10000 | 0.034 | 3.056 | 3.089 |
| gpu | 128 | 10000 | 0.035 | 3.072 | 3.107 |
| gpu | 256 | 10000 | 0.035 | 3.126 | 3.161 |
| gpu | 512 | 10000 | 0.034 | 3.248 | 3.282 |
| gpu | 1024 | 10000 | 0.035 | 3.238 | 3.273 |
| cpu | 1 | 50000 | 0.007 | 93.854 | 93.861 |
| cpu | 6 | 50000 | 0.006 | 19.909 | 19.915 |
| gpu | 8 | 50000 | 0.168 | 7.133 | 7.301 |
| gpu | 16 | 50000 | 0.167 | 6.518 | 6.685 |
| gpu | 32 | 50000 | 0.162 | 8.335 | 8.496 |
| gpu | 64 | 50000 | 0.175 | 8.430 | 8.604 |
| gpu | 128 | 50000 | 0.164 | 8.457 | 8.622 |
| gpu | 256 | 50000 | 0.173 | 8.484 | 8.657 |
| gpu | 512 | 50000 | 0.170 | 8.592 | 8.762 |
| gpu | 1024 | 50000 | 0.173 | 8.658 | 8.832 |
| cpu | 1 | 150000 | 0.021 | 293.606 | 293.627 |
| cpu | 6 | 150000 | 0.021 | 69.687 | 69.708 |
| gpu | 8 | 150000 | 0.479 | 21.161 | 21.640 |
| gpu | 16 | 150000 | 0.503 | 19.300 | 19.803 |
| gpu | 32 | 150000 | 0.507 | 24.717 | 25.224 |
| gpu | 64 | 150000 | 0.496 | 24.983 | 25.480 |
| gpu | 128 | 150000 | 0.523 | 25.057 | 25.580 |
| gpu | 256 | 150000 | 0.519 | 24.940 | 25.460 |
| gpu | 512 | 150000 | 0.520 | 24.896 | 25.416 |
| gpu | 1024 | 150000 | 0.499 | 24.936 | 25.435 |
| cpu | 1 | 450000 | 0.064 | 914.098 | 914.163 |
| cpu | 6 | 450000 | 0.063 | 193.692 | 193.755 |
| gpu | 8 | 450000 | 1.485 | 56.703 | 58.187 |
| gpu | 16 | 450000 | 1.511 | 50.597 | 52.108 |
| gpu | 32 | 450000 | 1.566 | 67.106 | 68.673 |
| gpu | 64 | 450000 | 1.561 | 67.870 | 69.431 |
| gpu | 128 | 450000 | 1.510 | 67.848 | 69.358 |
| gpu | 256 | 450000 | 1.565 | 67.319 | 68.884 |
| gpu | 512 | 450000 | 1.536 | 66.805 | 68.341 |
| gpu | 1024 | 450000 | 1.557 | 67.001 | 68.558 |
| cpu | 1 | 1000000 | 0.144 | 2 087.270 | 2 087.410 |
| cpu | 2 | 1000000 | 0.139 | 1 087.240 | 1 087.380 |
| cpu | 3 | 1000000 | 0.140 | 767.498 | 767.638 |
| cpu | 4 | 1000000 | 0.139 | 613.098 | 613.237 |
| cpu | 5 | 1000000 | 0.141 | 508.506 | 508.647 |
| cpu | 6 | 1000000 | 0.138 | 461.747 | 461.885 |
| gpu | 8 | 1000000 | 3.440 | 121.412 | 124.851 |
| gpu | 16 | 1000000 | 3.352 | 108.001 | 111.354 |
| gpu | 32 | 1000000 | 3.336 | 144.574 | 147.910 |
| gpu | 64 | 1000000 | 3.293 | 145.634 | 148.927 |
| gpu | 128 | 1000000 | 3.409 | 145.493 | 148.902 |
| gpu | 256 | 1000000 | 3.452 | 142.660 | 146.112 |
| gpu | 512 | 1000000 | 3.449 | 143.456 | 146.905 |
| gpu | 1024 | 1000000 | 3.448 | 144.207 | 147.655 |

### A. OpenMP

In Figure 2 we show the evaluation time for the OpenMP version of k-NN when the number of threads is varying. We observe that the evaluation time for the parallel OpenMP

version decreases when the number of threads increases, as expected.

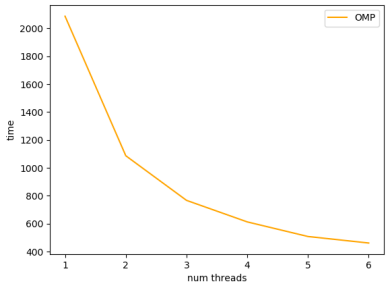


Fig. 2. Evaluation time for the OpenMP version of k-NN, in function of the number of threads.

In Figure 3 we show the speedup of the OpenMP version of k-NN (w.r.t. the sequential version) when the number of threads is varying. The speedup of the parallel OpenMP version increases when the number of threads increases, as expected. The speedup we got is sub-linear.

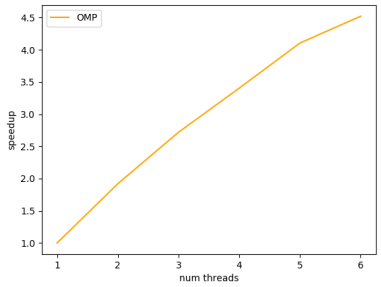


Fig. 3. Speedup of the OpenMP version of k-NN, in function of the number of threads.

### B. CUDA

In Figure 4 we show the evaluation time for the CUDA version of k-NN when the number of threads per block is varying.

In Figure 5 we show the speedup of the CUDA version of k-NN (w.r.t. the sequential version) when the number of threads per block is varying.

The sorting phase requires a bigger time compared to distances kernel + data transfer + synchronization between CPU and GPU. We verified this using Valgrind tools (in particular *Callgrind* and *KCachegrind*) [5], that show the elapsed time in every *statement* of the code.

The best block size in the CUDA version seems to be 16. We can observe that the block size is not much relevant for the evaluation time in these experiment. In our opinion, the reason why the time does not change very much is that the GPU used in our tests has 6 Streaming Multiprocessors only: even with a big block size (and so a not very big number of blocks), it's easy to occupy all the Streaming Multiprocessors.

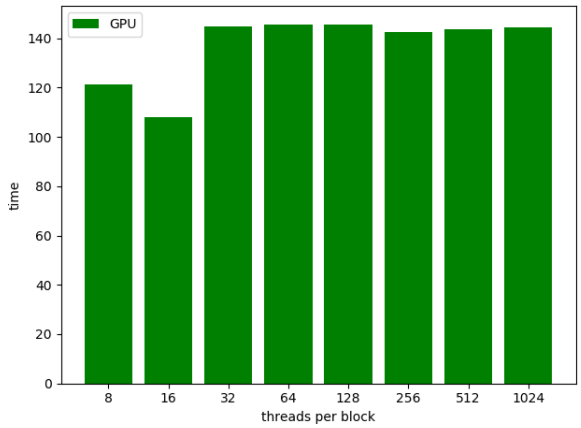


Fig. 4. Evaluation time for the CUDA version of k-NN, in function of the number of threads per block.

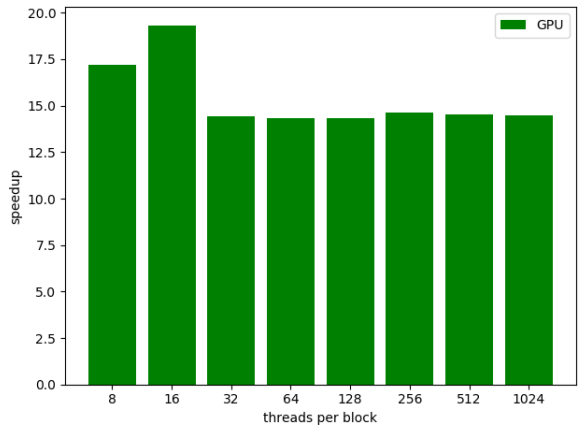


Fig. 5. Speedup of the CUDA version of k-NN, in function of the number of threads per block.

To validate these intuitions, we replicate the experiments on a different machine with this configuration:

- CPU: Intel Core i7-860, 4 cores / 8 threads
- GPU: NVIDIA GeForce GTX 980, 2048 CUDA cores, 4 GiB video RAM, 16 Streaming Multiprocessors
- RAM: 16 GiB
- CUDA: 10.2

With this configuration, the block size greatly affects the calculation time. The reason why the block size becomes relevant with this configuration could be interpreted as if with more complex architectures it's difficult to harness the full power of the GPU.

Table II in the appendix shows the numeric results of the experiment with this different setting.

### C. Comparison

In this comparison, we set the maximum number of threads (6) for the OpenMP version and the best *blockSize* for the CUDA version. In Figure 8 we show the comparison between

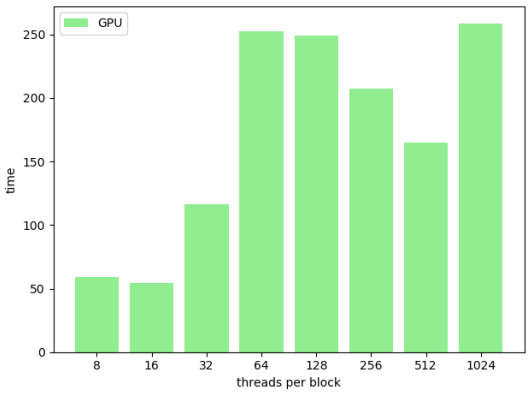


Fig. 6. Evaluation time for the CUDA version of k-NN, in function of the number of threads per block with GPU NVIDIA GTX 980.

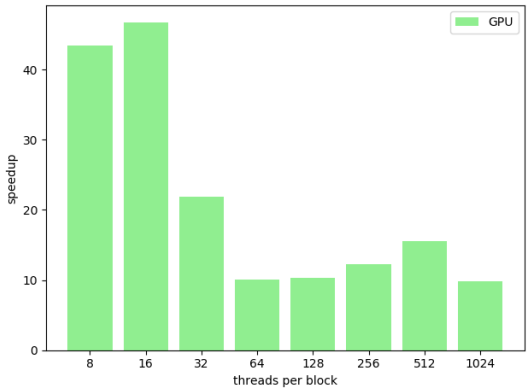


Fig. 7. Speedup of the CUDA version of k-NN, in function of the number of threads per block, with GPU NVIDIA GTX 980.

the evaluation time of the three versions (sequential, OpenMP, CUDA) of k-NN when the dataset size is varying.

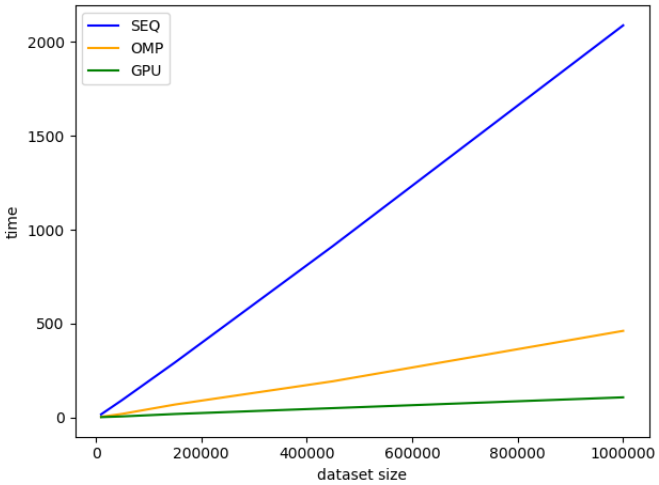


Fig. 8. Comparison between the evaluation time of the three versions (sequential, OpenMP, CUDA) of k-NN, in function of the dataset size.

As expected, we can observe that the *Evaluation time* increases almost linearly in function of the dataset size. The asymptotic computational complexity of the sequential version is $\Theta(nlog(n))$: $\Theta(n)$ for the calculation of Euclidean distances from every query node to all the dataset nodes, and $\Theta(nlog(n))$ for the subsequent sorting.

In Figure 9 we show the comparison between the speedup of the two parallel versions (OpenMP, CUDA) of k-NN (w.r.t. the sequential version) when the dataset size is varying.

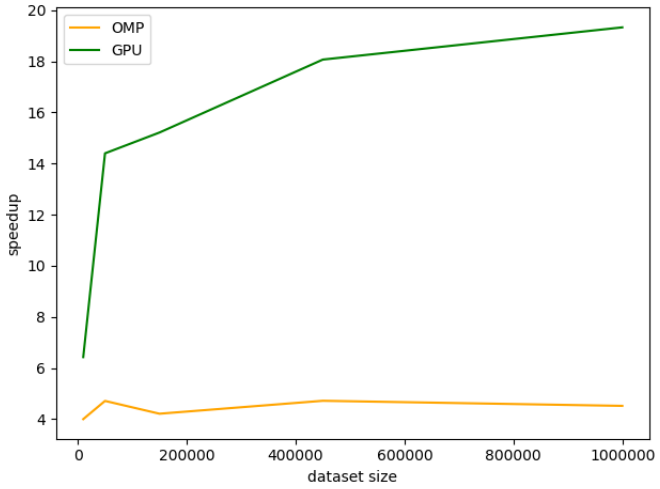


Fig. 9. Comparison between the speedup of the two parallel versions (OpenMP, CUDA) of k-NN, in function of the dataset size.

In particular, the CUDA version performs better than the OpenMP version even when the dataset is very small, and when varying the dataset size the difference in performance is even more pronounced.

## VI. Conclusion

In this study, we evaluate the performance of different implementations of the *k-Nearest Neighbors* algorithm to better understand the potentiality of parallel computing. We observed that substantial performance gain can be obtained with both the OpenMP and CUDA parallel version, w.r.t. the simple sequential algorithm.

We obtained a sub-linear speedup with OpenMP: the time of the sequential version turned out to be 4.5 times bigger than that of the OpenMP version with 6 threads (equal to the number of cores of the CPU used).

With CUDA, the speedup obtained was about 13.5 times, therefore much higher than that obtained with OpenMP.

Parallelizing with OpenMP is a relatively simple task, and it doesn't have much impact on the general structure of the code, so we believe there is no reason not to parallelize the code whenever possible. Parallelizing with CUDA requires greater and specific technical knowledge but allows to obtain potentially higher speedups: this is especially useful in contexts where performance is a crucial requirement of the software, for example in the analysis of large quantities of data in realtime.

Parallelism is gaining broader interest due to the need for processing an increasing quantity/dimension of data. Substantially improve single-core performance is difficult and therefore we believe that in the future more and more applications will take advantage of parallelism techniques to exploit the full performance of the computational resources.

## REFERENCES


[1] S. A. Dudani, "The Distance-Weighted k-Nearest-Neighbor Rule," in IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-6, no. 4, pp. 325-327, April 1976. doi: 10.1109/TSMC.1976.5408784

[2] J. M. Keller, M. R. Gray and J. A. Givens, "A fuzzy K-nearest neighbor algorithm," in IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-15, no. 4, pp. 580-585, July-Aug. 1985. doi: 10.1109/TSMC.1985.631342

[3] K. He and J. Sun, "Computing nearest-neighbor fields via Propagation-Assisted KD-Trees," 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, 2012, pp. 111-118. doi: 10.1109/CVPR.2012.6247665

[4] Hervé Jégou , Matthijs Douze and Cordelia Schmid, "Product Quantization for Nearest Neighbor Search", Inria Grenoble - Rhône-Alpes, LJK - Laboratoire Jean Kuntzmann, INPG - Institut National Polytechnique de Grenoble, IRISA - Institut de Recherche en Informatique et Systèmes Aléatoires, Inria Rennes – Bretagne Atlantique

[5] Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools. Url: `https://valgrind.org`

[6] OpenMP `https://www.openmp.org`

[7] CUDA `https://developer.nvidia.com/cuda-zone`

## VII. Appendix

TABLE II
RESULTS OF THE EXPERIMENT ON CPU I7 860 (4 CORE) AND GPU NVIDIA GTX 980

| hw | threads | dataset_size | init_time | eval_time | total_time |
|---|---|---|---|---|---|
| cpu | 1 | 1000000 | 0.222 | 2 553.550 | 2 553.770 |
| gpu | 8 | 1000000 | 9.369 | 58.855 | 68.224 |
| gpu | 16 | 1000000 | 7.151 | 54.646 | 61.797 |
| gpu | 32 | 1000000 | 7.154 | 116.588 | 123.742 |
| gpu | 64 | 1000000 | 7.112 | 252.817 | 259.928 |
| gpu | 128 | 1000000 | 7.122 | 248.962 | 256.084 |
| gpu | 256 | 1000000 | 7.152 | 207.338 | 214.490 |
| gpu | 512 | 1000000 | 7.097 | 164.903 | 172.000 |
| gpu | 1024 | 1000000 | 7.162 | 258.748 | 265.911 |