

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

Iacopo Erpichini and Federico Magnolfi

University of Florence

Outline

① Introduction

② Implementation

Sequential

Parallel

OpenMp

CUDA

③ Results

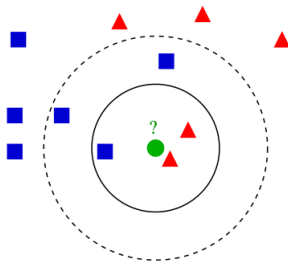
④ Conclusion

k -Nearest Neighbors problem

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:

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

Example of k-NN problem



● : query

▲ ■ : dataset points

In this example, the neighbors have information (shape/color) that can be used for other tasks, such as classification/regression

Objective

Compare a sequential and two parallel versions of k-NN measuring:

- compute time
- speedup

while varying:

- number of threads
- dataset size

Dataset

We use the k-NN algorithm on a SIFT dataset

Dataset

Why k-NN it's important on SIFT data

- SIFT is a feature detection algorithm used in computer vision to detect and describe local features in images
- SIFT descriptors are 128-dimensional floating point vectors, used for image classification and object detection
- **k-NN** is used to find training SIFTs “**near/similar**” to the SIFTs in the test image

Sequential Implementation

Algorithm 1: k-Nearest Neighbors - Sequential version

Input : dataset, query, k

Output: nearestNeighbors

```
1 // init
2 indexes = new int[dataset.size]
3 distances = new float[dataset.size]
4 // calculate distances
5 for p from 0 to dataset.size-1 do
6     indexes[p] = p
7     distances[p] = euclideanDistance(query, dataset[p])
8 end
9 // sort by increasing distance
10 sortByKey(indexes, keys=distances)
11 // slice
12 nearestNeighbors = indexes[:k]
```

Open MP Implementation (1)

Algorithm 2: k-Nearest Neighbors - OpenMP version

Input : dataset, query, k, numThreads

Output: nearestNeighbors

```
1 // init
2 indexes = new int[dataset.size]
3 distances = new float[dataset.size]
4 // calculate distances

6 for p from 0 to dataset.size-1 do
7     indexes[p] = p
8     distances[p] = euclideanDistance(query, dataset[p])
9 end
```

Open MP Implementation (1)

Algorithm 3: k-Nearest Neighbors - OpenMP version

Input : dataset, query, k, numThreads

Output: nearestNeighbors

```
1 // init
2 indexes = new int[dataset.size]
3 distances = new float[dataset.size]
4 // calculate distances
5 # pragma omp parallel for
6 for p from 0 to dataset.size-1 do
7     indexes[p] = p
8     distances[p] = euclideanDistance(query, dataset[p])
9 end
```

Open MP Implementation (2)

```
10 chunkSize = dataset.size / numThreads
11 // sort chunks by increasing distance
12 # pragma omp parallel for
13 for j from 0 to numThreads-1 do
14     | s = j * chunkSize
15     | e = s + chunkSize
16     | sortByKey(indexes[s:e], keys=distances[s:e])
17 end
```

Open MP Implementation (2)

```
10 chunkSize = dataset.size / numThreads
11 // sort chunks by increasing distance
12 # pragma omp parallel for
13 for j from 0 to numThreads-1 do
14     s = j * chunkSize
15     e = s + chunkSize
16     sortByKey(indexes[s:e], keys=distances[s:e])
17 end
18 // merge sorted chunks
19 for j from 1 to numThreads-1 do
20     i, d = indexes[:k], distances[:k]
21     s = j * chunkSize
22     e = s + k
23     indexes[:k], distances[:k] = merge(i,d,indexes[s:e],distances[s:e])
24 end
25 // slice
26 nearestNeighbors = indexes[:k]
```

Scheduling Open MP

OpenMP creates the threads and distributes the iterations between them depending on the **scheduling**, which can be:

Scheduling Open MP

OpenMP creates the threads and distributes the iterations between them depending on the **scheduling**, which can be:

- static: *equal-sized chunks, circular order*

```
****          ****
      ****          ****
          ****          ****
```

Scheduling Open MP

OpenMP creates the threads and distributes the iterations between them depending on the **scheduling**, which can be:

- static: *equal-sized chunks, circular order*

```
****          ****
      ****          ****
          ****          ****
```

- dynamic: *equal-sized chunks, available thread*

```
*  ** **  * * * *  *  *
   *      *    * *    * *
*      *  *      **   *
```

Scheduling Open MP

OpenMP creates the threads and distributes the iterations between them depending on the **scheduling**, which can be:

- static: *equal-sized chunks, circular order*

```
****          ****
      ****          ****
          ****          ****
```

- dynamic: *equal-sized chunks, available thread*

```
*  ** **  * * * *  *  *
   *      *  * *  *  *
*      *  *      **  *
```

- guided: *decreasing-size chunks, available thread*

```
*****          ***          **
      *****          **          **
          *****          **
```

Scheduling Open MP

- static: *equal-sized chunks, circular order*

```
*****          *****
      *****          *****
            *****          *****
```


CUDA Implementation

Algorithm 4: k-Nearest Neighbors - CUDA version

Input : dataset, query as q, k, bSize

Output: nearestNeighbors

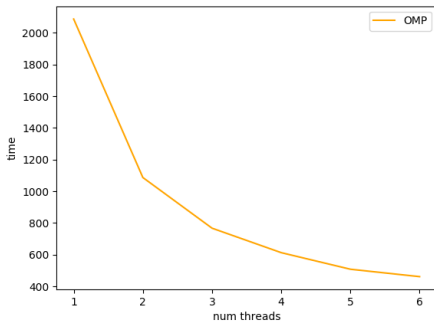
```
1 // init
2 indexes = new int[dataset.size]
3 distances = new float[dataset.size]
4 // move query to GPU memory
5 q = toGPU(q)
6 // determine number of blocks
7 blocks = (datasetSize + bSize - 1) / bSize
8 // calculate the distances
9 idxs, dists = cudaDistances<<<blocks,bSize>>>(dataset,q)
10 // sort by increasing distance on GPU
11 sortByKeyOnGPU(idxs, keys=dists)
12 // slice and move results to CPU memory
13 nearestNeighbors = toCPU(idxs[:k])
```

Test configuration

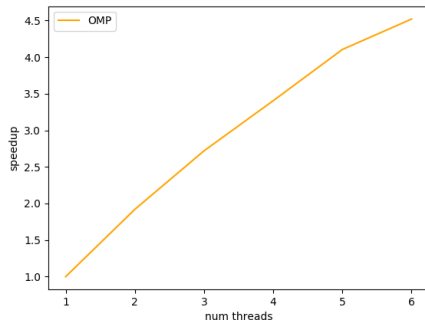
- CPU: Intel Core i7-8750H, 6 cores / 12 threads
- GPU: NVIDIA GeForce GTX 1050 Ti (Notebook), 768 CUDA cores, 4 GiB video RAM, 6 SM
- RAM: 16 GiB
- CUDA: 10.0
- OpenMP: 4.5

OpenMP

Evaluation time



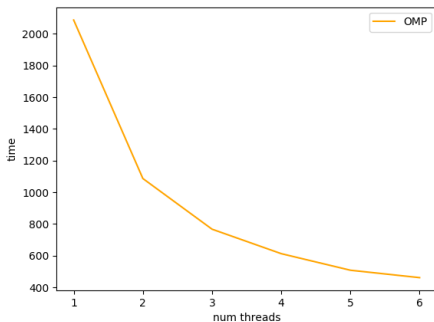
Speedup



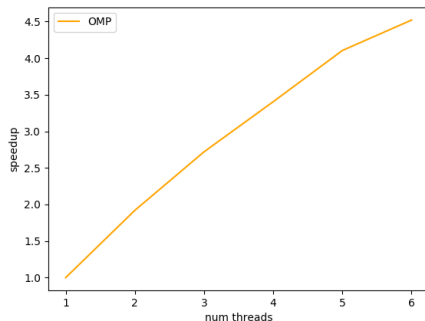
Plots in function of the number of threads

OpenMP

Evaluation time



Speedup

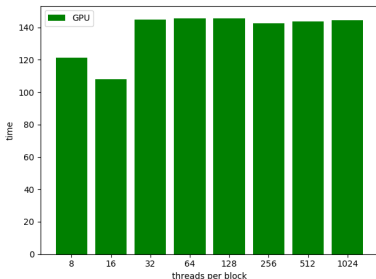


Plots in function of the number of threads

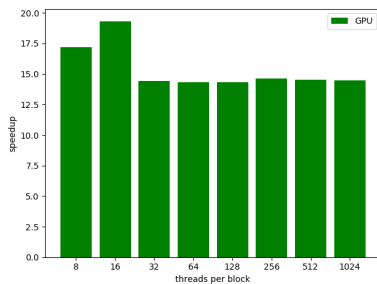
The obtained SpeedUp is sub-linear

CUDA

Evaluation time



Speedup



Plots in function of the number of threads per block

Observation on CUDA results

Best block size is 16

The block size in k-NN is not much relevant for the evaluation time.

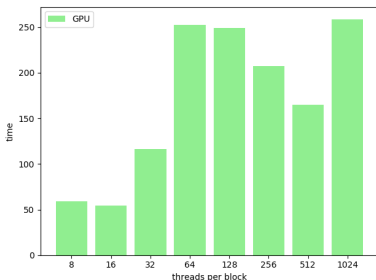
- GPU used in our tests has 6 *Streaming Multiprocessors* only
- Even with a big block size, it's easy to occupy every **SM**
- $\text{sort time} > \text{distances kernel} + \text{data transfer} + \text{sync time}^*$

*: verified using Valgrind tools: <https://valgrind.org>

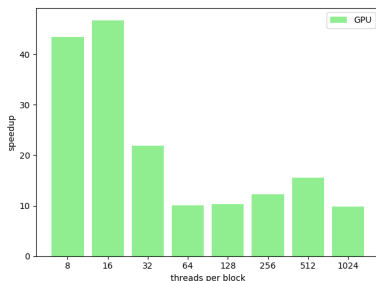
Other experiment on CUDA

To check if with a more powerful GPU the block size becomes important, we run the experiments on a NVIDIA GTX 980 GPU (which has 16 SM)

Evaluation time

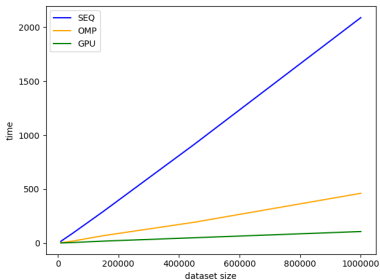


Speedup

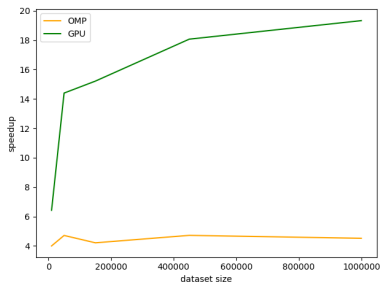


Comparison

Evaluation time



Speedup



Plots in function of the dataset size

OpenMP

- *Implementation difficulty*: low
- *Speedup*: sub-linear
- Makes parallel computing very accessible

OpenMP

- *Implementation difficulty*: low
- *Speedup*: sub-linear
- Makes parallel computing very accessible

CUDA

- *Implementation difficulty*: high
- *Speedup*: huge (~ 10 -100)
- Highest level of single-machine parallelism

End

Thanks for the attention