Comparison between a sequential and multi-thread version of the k-Nearest Neighbors problem

Iacopo Erpichini and Federico Magnolfi

University of Florence

Outline

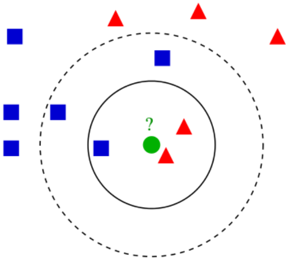
1. Introduction
2. Implementation
   - Sequential
   - Parallel
     - OpenMp
     - CUDA
3. Results
4. Conclusion

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:

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

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

Problem

In this project we compare a sequential version and two parallel version of the k-NN algorithm and we:

- Measure the Evaluation Time at the variation of cores
- Measure the Evaluation Time at the variation of dataset size
- Measure the Speed Up at the variation of cores
- Measure the Speed Up at the variation of dataset size

Dataset

successivamente nella presentaz segue pseudo codici, e risultati che sono stati ottenuti su SIFT + citazione

Sequential Implementation

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]
```

Open MP Implementation (1)

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
```

Open MP Implementation (2)

```
// 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]
```

CUDA Implementation

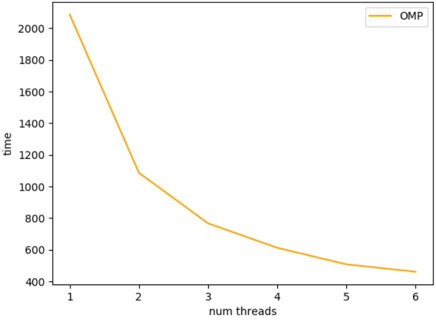
Algorithm 3: k-Nearest Neighbors - CUDA version

Input : dataset, query as q, k, bSize
Output: nearestNeighbors

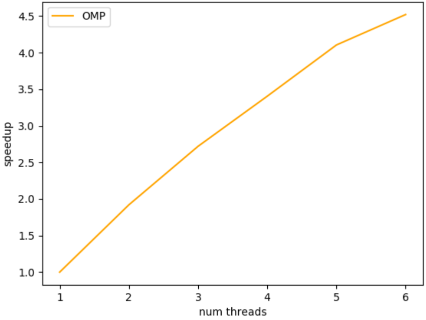
```
// init
indexes = new int[dataset.size]
distances = new float[dataset.size]
// move query to GPU memory
q = toGPU(q)
// determine number of blocks
blocks = (datasetSize + bSize - 1) / bSize
// calculate the distances
idxs, dists = cudaDistances<<<blocks,bSize>>>(dataset,q)
// sort by increasing distance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

OpenMP

Evaluation time



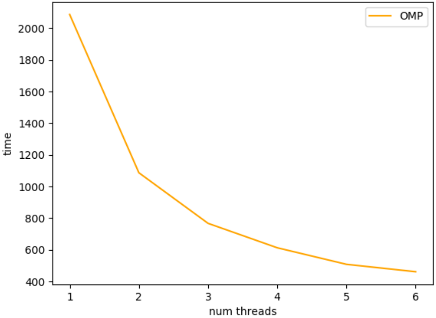
Speedup



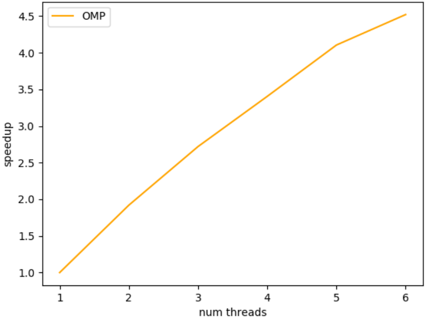
Plots in function of the number of threads

OpenMP

Evaluation time



Speedup

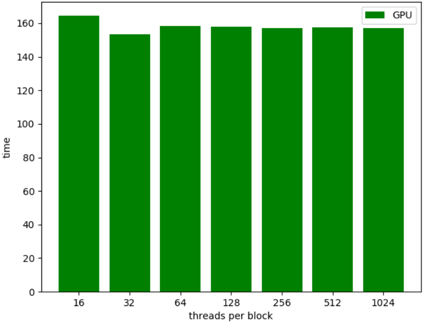


Plots in function of the number of threads

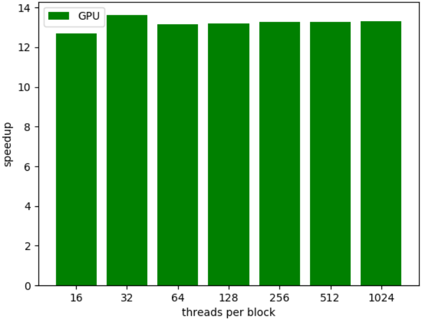
The obtained SpeedUp is sub-linear

CUDA

Evaluation time



Speedup



Plots in function of the number of threads per block

Observation on CUDA results

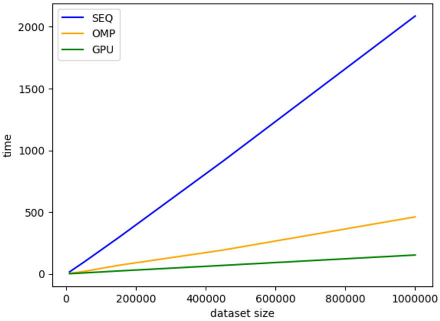
The block size in k-NN is not much relevant for the evaluation time.

- GPU used in our tests has 6 *Streaming Multiprocessors* only
- Even with a big block size, it's easy to occupy all **SM**
- Distances calcuation time $>$ data transfer $+$ sync time *

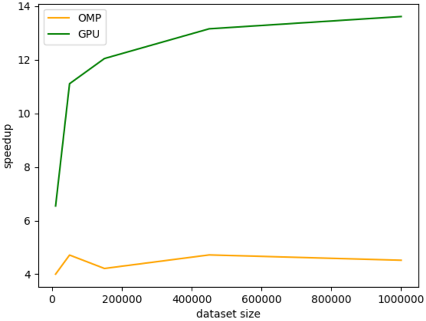
*: verified using Valgrind tools: `https://valgrind.org`

Comparison

Evaluation time



Speedup



Plots in function of the dataset size

Conclusion

- We can observe that the *Evaluation time* increases sub-linearly in function of the dataset size.
- The asymptotic computational complexity of the sequential version is $\Theta(nlog(n))$

Conclusion

- We can observe that the *Evaluation time* increases sub-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
  - $\Theta(nlog(n))$ for the subsequent sorting

Conclusion

- We can observe that the *Evaluation time* increases sub-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
  - $\Theta(nlog(n))$ for the subsequent sorting

The sequential time is 4.5 times bigger than that of the OpenMP version with 6 threads.
CUDA speedup obtained was about 13.5 times, much higher than OpenMP.

End

Thanks for attention

Extra

?