K-Nearest Neighbors: comparison CUDA and CPU algorithm

Giuseppe Palazzolo

palazzolo1995@gmail.com

Abstract

K-nearest neighbors is a non-parametric algorithm used for classification and regression. It calculates the distance between the query observation and each data point in the train dataset and finds the K closest observations. In this article is presented its execution over CPU and over GPU, using CUDA programming language, to show if and how many its performance can improve.

1. Introduction

Nearest neighbor search (NNS) is the optimization problem of finding the point in a given set that is closest (or most similar) to a given point. In 1973 Donald Knuth called it the post-office problem, referring to an application of assigning to a residence the nearest post office. K-NN search is its direct generalization where we need to find the k closest points. Nowadays it is used from pattern recognition to chemical similarity. [2] With today's hardware, especially with GPUs, it is possible to parallel the algorithm in order to increase performance. In fact, KNN is an embarrassingly parallel algorithm that can be implemented on multiple threads. In this article it will presents the algorithm in its most generic form and its operation in the **Section 1.**, after which it will be explained as it has been implemented in the various versions (**Section 2 .**) and finally the experimental results will be shown (**Section 3.**) and the conclusions presented (**Section 4.**).

1.1. Definition

K-Nearest Neighbors its very simple to understand in fact it's often used to introduce people to machine learning and classification in general. The idea is to calculate the distance from each pair dataset element - query and return the k indices of the elements with less distance.

1.2. Algorithm

The pseudocode of the algorithm that will be implemented is presented below:

Algorithm 1 K-Nearest Neighbour Pseudo Code

1: **Inputs:**
   $Dataset, Query, k$
2: **Initialize:**
   $DS \leftarrow Dataset$
   $Q \leftarrow Query$
   $D \leftarrow new\ vector(|DS|)$
   $K \leftarrow k$
   $KMin \leftarrow new\ vector(|K|)$
3: **for** $X_i \in DS$ **do**
4: $D[d_i] \leftarrow Distance(X_i, Q)$
5: **end for**
6: $KMin \leftarrow ExtractFirstKMinArgument(D, K)$
7: **return** $KMin$

You can already notice that the algorithm is embarrassingly parallel because the calculation of the distance do not require any additional data and even if the queries are used by more than one thread they are only read.

2. Implementation

The implementation of the algorithm has several versions, one for the CPU and four for CUDA, in which the memories used and the way

in which they are used vary, in order to maximize performance and to test and observe the results. As far as CUDA is concerned, the various versions refer to the implementation of the distance calculation, while the algorithm used to calculate the k indices of the minimum distances is a modified insertion sort that is the same for each version.

2.1. CPU

The algorithm implemented in C ++ does not differ much from the one above, being very simple and clear. The algorithm core is distances computation and sorting; it is called for each couple dataset element - query a simple euclidean distance function, the result is saved in a `pair(distance, index)`. After calculating all distances with all queries they are sorted by key with sort algorithm of `<algorithm>` library and returned an array of `k × query_size` size with the minimum distances.

2.2. GPU

In CUDA the function to be executed on GPU is called *kernel*, that takes in input, in addition to the arguments of the function, two values: *gridDim* and *blockDim*, that represent the number of blocks that must be instantiated and the number of threads of each block [1]. Each kernel can have different configurations and all the threads of a kernel run in the same code each with a unique id. Regarding the number of threads per block, various tests have been carried out reported to the **Section 3.**

2.2.1 GPU Naive version

The first and simplest GPU version **Algorithm 2** is an algorithm similar to that used in C ++. For each element of the dataset one thread is assigned so will be start as many threads as number on elements in dataset: `dimGrid = ceil(dataset_size/block_size)` and `dimBlock = block_size`. There are already many improvements compared to the CPU algorithm but it can still be improved.

Algorithm 2 Naive CUDA K-Nearest Neighbour

Initialize:
  $DS \leftarrow Dataset$
  $Q_{DS} \leftarrow QueryDataset$
  $D \leftarrow new\ vector(|DS| * |Q_{DS}|)$
  $I \leftarrow new\ vector(|DS| * |Q_{DS}|)$
$th \leftarrow blockIdx.x * blockDim.x + threadIdx.x$;
if $th < |DS|$ **then**
  for $Q_i \in Q_{DS}$ **do**
    for $j < DataSize$ **do**
      $d+ = (x_j - q_j)^2$
    end for
    $D[th * i] \leftarrow d$
  end for
end if
$cudaInsertionSort(D, I)$

2.2.2 GPU Coalescing

The naive algorithm does not take into account important aspects concerning the overall memory architecture of the GPU and the management of threads. The data in global memory are saved in contiguous and not randomly. In this version of the algorithm the optimization is given by the fact that each thread of each warp can access in a single transaction the element of the assigned vector. This prevents the loss of performance caused by multiple global memory accesses. So the dataset is saved as AoS instead of SoA **Figure 1.** i.e. at `i` row's dataset there are all `i`-th elements of all vector. Coalescing is an best practice when we write a cuda kernel, AoS dataset is been used for the other two implementation.

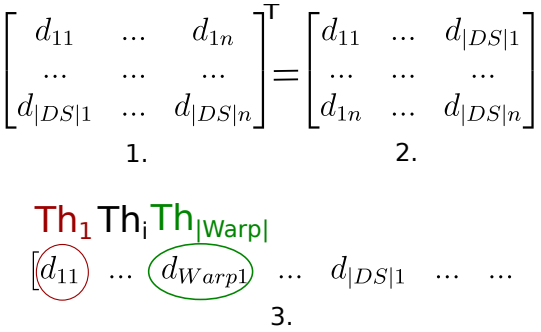

Figure 1. 1.: SoA: Structure of arrays; 2.: AoS: Arrays of structure; 3. Coalescing; (d_{ij}: i dataset's element; j vector's element)

2.2.3 GPU Constant memory

An interesting experiment was to use constant memory to save queries. Constant memory in GPU is a 64 KB memory used to, as name indicate, save data that will no change during execution, in fact its is loaded before calling the kernel. For all threads of a half warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. Accesses to different addresses by threads within a half warp are serialized, so cost scales linearly with the number of different addresses read by all threads within a half warp. [3] This is what happens when reading long queries from constant memory. Results aren't bad as tests show but algorithm is more performed without using constant memory. When more than 100 queries were used it was necessary to call the kernel more than once, loading the new queries into constant memory each time.

2.2.4 GPU Shared memory

A significant optimization was given by the use of shared memory. The second for speed in the hierarchy of memories, after the registers. It is a block-level cache memory; each thread accesses the same shared memory of every other thread in a block. It is used here to save the queries. Shared memory available per block is 48 KB so the queries are loaded in several phases, in particular it will be loaded `TILE` queries at time, with tile size set to 32, so it will be `query_size/TILE` phases. At any phase the first `TILE` threads load query in shared memory array:

```
__shared__ float
sh_query[TILE][DATA_SIZE];
```

After that each thread has calculated the distance between its element and all queries in shared memory, remaining ones will be loaded in the same way until all the distances have been calculated. Thus fewer accesses to global memory are made, reducing latency.

3. Experimental results

Several tests have been conducted in order to show both how much the use of the GPU's computing power can give advantages and to point out how the methods of use of memory can affect the results. Results can be shown based on execution time and *speed-up*:

$$Speed\ Up = \frac{time_{CPU}}{time_{GPU}}$$

All execution have been do on Intel i7 8700k CPU and NVIDIA 1080Ti 11GB GPU. Two datasets were used `ANN_SIFT10K` and `ANN_SIFT1M` respectively of 10000 and 1000000 dataset's element and 100 and 10000 query of 128 float; result's veracity has been guaranteed by ground truth dataset.

3.1. Illustrations, graphs, and photographs

In **Table 1** it is shown results about `ANN_SIFT10K` dataset, the few data allow the cpu to be quite performing, the highest speed up obtained here is of 17x with 100 queries. Increasing dataset size, using `ANN_SIFT1M`, the results are more evident as shown in **Table 2**. The algorithm that best lent itself to a large number of queries is these that uses shared memory to saved the queries, it is been tested with a maximun of 600 queries and the Spedd Up graphics it shown in **Figure 2**.

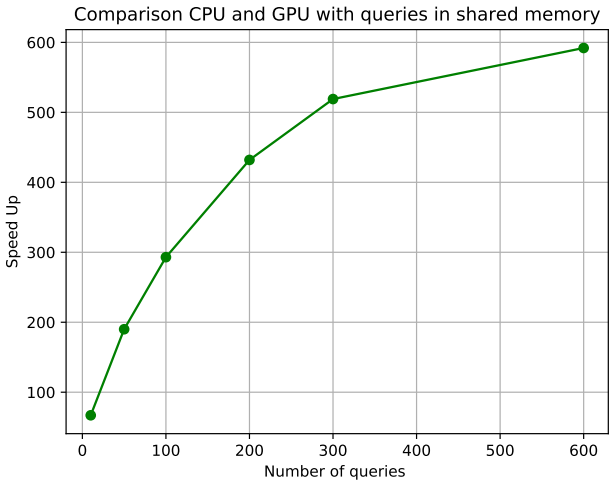

Figure 2. Speed Up with ANN_SIFT1M dataset

4. Conclusion

It has been shown how the use of GPU computing power has improved the performance of this algorithm. In the sequential version, the times increase linearly as the size of the dataset and the number of queries. Using CUDA language, so GPU, a performance improvement of a factor of 600x could be achieved.

References

[1] Cuda c programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] K-nearest neighbors. https://it.wikipedia.org/wiki/K-nearest_neighbors.

[3] What is constant memory in cuda. http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html.

Query size	CPU Time	GPU Naive	GPU Coal	GPU Sh	GPU Const
1	0.0286	**0.0165**	**0.0165**	0.0074	0.0069
10	0.2773	0.0727	0.0725	0.0183	**0.0137**
50	1.3587	0.1520	0.1385	0.1236	**0.1048**
100	2.7150	0.2000	0.1853	**0.1695**	0.17093

Table 1. Dataset: ANN10K. Times (sec.) obtained by varying number of queries.

Query Dim	CPU Time	GPU Naive	GPU Coal	GPU Sh	GPU Const
10	37.9093	0.9059	0.7333	**0.5732**	0.6134
50	191.7177	2.2710	1.1035	1.0114	**0.8463**
100	384.3876	3.6871	1.3681	**1.3133**	1.9052

Table 2. Dataset: ANN1M. Times (sec.) obtained by varying number of queries.