k-NN Report

朱倖民 X1034051

August 6, 2015

1 Introduction

This project solves the k-nearest neighbor or k-NN problem on GPU with NVIDIA CUDA API. We use the parallelized brute-force method to solve the program. We also use three tricks to optimize the program to reach better performances.

2 Problem Description

2.1 Background

The k-NN problem is a method for classifying unknown objects based on the closest training samples in feature space. It is a type of supervised learning and is among the simplest of all machine-learning algorithms [1]. The objects are assumed to be a list of vectors $x \in T$, where T is the *training data set* and x stands for each data. Vector x is usually an n-dimensional vector, with each dimensional showing a feature of x. It is reasonable to see that when two data are nearer, they are more likely to belong to the same class. Thus, when given a set of unknown objects Q, which is also called *query data set*, for each object $y \in Q$, we then try to find the k nearest data in T. Please note that k is a user defined parameter. Usually, programmers with rich experience can select the best parameter to make the best accuracy.

According to which classes these data belong to, we can guess which class is suitable for y.

For example, as **Figure 1** shown, the query object (green circle) should be classified either to the first class of blue squares or to the second class of red triangles. If $k = 3$ (solid line circle) it is assigned to the second class because there are 2 triangles and only 1 square inside the inner circle. If $k = 5$ (dashed line circle) it is assigned to the first class (3 squares vs. 2 triangles inside the outer circle).

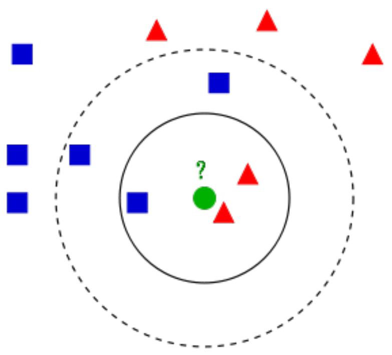

Figure 1: An example of k-NN based classification

2.2 Restatement of the Problem

In this project, we only focus on how to find the k nearest neighbors. The k-NN problem is usually defined as:

Let $T = \{t_1, t_2, \ldots, t_m\}$ be a set of m training points in a n-dimensional space, and let $Q = \{q_1, q_2, \ldots, q_l\}$ be a set of n query points in the same space. The k-NN problem consists in searching the k nearest neighbors of each query point $q_i \in Q$ in the training set T given a specific distance. Commonly, the Euclidean or the Manhattan distance is used.

Please note that in this project, we use Euclidean distance. And specially, our query set is exactly the same as the training set, i.e, $Q = T$. However, the algorithm and implementations discussed later can apply to more genernal conditions where $Q \neq T$.

3 The Brute-force Algorithm

The Brute-force Algorithm (BF) is a simple and natural way to solve k-NN problem. For each query point q_i, the BF algorithm is the following:

1. Compute all the distances between points q_i and t_j where $j = 1, 2, \ldots, m$.
2. Sort the computed distances.
3. Select the k training points providing to the smallest distances.

Once all query points are done, the algorithm stops.

The main issue of this algorithm is its huge complexity $O(mln)$ for the ml distances computed and $O(lm \log m)$ for the l sorts performed.

However, The BF method is by nature highly-parallelizable. Indeed, all the distances can be computed in parallel. Likewise, the l sorts can be done in parallel. This property makes the BF method perfectly suitable for a GPU implementation. In this project, since query set and training set are equal, we only need to compute less than a half of all distances.

4 Implementation of BF Algorithm

The BF algorithm is a two-pass algorithm: compute the distances and sort. Both passes can be parallelized.

4.1 Compute Distances

Here, we need to compute the distances matrix d, where d_{ij} donates the distances between t_i, t_j. By using Euclidean distance,

$$d_{ij} = \sqrt{\sum_{x=0}^{k} (t_{ix} - t_{jx})^2}$$

where t_{ix} notates the x-th dimension of point t_i. Since we only want to know the relative distances between these points, it is a waste of time to compute the square root. Reversely, computing d_{ij}^2 is enough.

We use two-dimension threads to compute. For thread indexed (i, j), it computes d_{ij}^2 when $i < j$. When $i = j$, we set d_{ij}^2 as ∞. In our code, we use 1073741824 as the very huge number. When $i > j$, no computation is done because $d_{ij}^2 = d_{ji}^2$ and d_{ji}^2 is already computed.

Table 1 shows the performances compared with sequential version. Note that in this report *speed up* always refer to the ratio of

$$\frac{\text{time of parallel program}}{\text{time of sequential program}}.$$

and the time costs are measured in ms.

Table 1: Time cost of first pass: compute distances

sample	sequential time	parallel time	speed up
small	70.0000	26.1445	2.68
middle	2870.0000	1667.4698	1.72
large	261060.0000	108000.8828	2.42

4.2 Sort

Quick sort and *merge sort* are two widely used sorting algorithms on CPU, and they both have a low time complexity of $O(s \log s)$, where s notates the size of array to be sorted. For k-NN problem, it is not necessary to sort the whole array-we only need to find out the k smallest. Since $k \ll s$, we consider using another straight-forward but not very complex way:

1. Traverse the array to find the minimal element, which makes the 1st nearest.

2. Then we modify that element with ∞, traverse the array again and find the minimal element, which makes the 2nd nearest.

3. Continue the strategy until we find all k elements.

It is obviously that the algorithm we introduce is $O(ks)$, when $k < \log s$, this straight-forward algorithm is even better than quick sort and merge sort.

To parallelize, we create m threads, each compute the k nearest neighbors of one point. From **Table 2** we see that the performance is improved.

Table 2: Time cost of second pass: sort

sample	sequential time	parallel time	speed up
small	20.0000	3.6265	5.51
middle	380.0000	96.7870	3.93
large	12530.0000	3085.1785	4.06

4.3 Conclusion

Table 3 shows the overall speed up after we use parallel programs.

Table 3: Time cost of both passes after parallel

sample	sequential time	parallel time	speed up
small	90.0000	29.7710	3.02
middle	3250.0000	1764.2568	1.84
large	273590.0000	111086.0613	2.46

5 Tricks for Optimising

5.1 First Trick: Shared Memory

We find that the first pass of the BF algorithm, distances computing, costs a lot of time. And since the computation is rather similar to matrix multiplication, we can use the exactly same shared memory trick: we divide the distance matrix into several blocks, we load necessary data from device memory into shared memory, which has the exact block size, then we do actual computations in shared memory. Again, two-dimension threads are enough for this trick.

By using 16 as block size, we see the optimising is obvious from **Table 4**.

Table 4: Time cost of distance computing using trick 1

sample	original time	optimised time	original speed up	optimised speed up
small	26.1445	6.0994	2.68	11.48
middle	1667.4698	386.6234	1.72	7.42
large	108000.8828	24760.4727	2.42	10.54

We then test different block size, the results are shown in **Table 5**. We see that 16 is the best size.

Table 5: Performance on different block sizes

block size	time of distances computing
4	109694.3672
8	27687.4863
16	27449.7461
32	34528.1953

5.2 Second Trick: Three-dimension Threads

By using shared memory, we save time compared with global memory. By using three-dimension threads, we can save even more time due to more threads work together. This is proved in **Table 6**.

Table 6: Time cost of distance computing using trick 2

sample	original time	optimised time	original speed up	optimised speed up
small	6.0994	4.4384	11.48	15.77
middle	386.6234	284.2299	7.42	10.10
large	24760.4727	18620.1328	10.54	14.02

In the previous trick, we still use two dimensions where x-dimension and y-dimension stand for the two points we want to compute their distance. Now, we introduce z-dimension to simply divide the matrix into four equal parts. Thus, z can be $0, 1, 2$ or 3, pointing to the four parts. Therefore, the indices (i, j) a thread computes now becomes

```
int block = ceil(m / (double)BLOCK_SZ);
int tx = threadIdx.x;
int ty = threadIdx.y;
int i = BLOCK_SZ * (blockIdx.x + (blockIdx.z / 2) * block) + tx;
int j = BLOCK_SZ * (blockIdx.y + (blockIdx.z % 2) * block) + ty;
```

5.3 Third Trick: Selection Sort

Now let's focus on optimising the second pass, sorting. The method we used previously may work well when k is small. However, when k is a little bit large and the array is long, we have to spend huge time on traversing elements again and again. How can we make it faster?

First of all, let's find out the k minimal elements in the array. We can do it like this:

1. Pick up the first k elements as if they are already the smallest.
2. Find the maximum element in the first k elements as max.
3. For each of the remaining $s - k$ elements, say e, if $e < max$, replace max with e and refind the maximum element in the first k elements

as *max*; otherwise, do nothing for this element.

After traversing all elements in the array, we now find the k minimal elements. The time complexity of the algorithm above is $O(ks)$. In fact, the times we need to refind *max* is less than s.

Next, we want to sort the k elements we find. Please note that if we do not care the order of the k elements in some conditions, we do not need to do this. Here, we still use our naive way of finding the 1st minimal, 2nd minimal, ..., k-th minimal each at a time. Although this algorithm is $O(k^2)$, we do not waste time because here k is rather small. In our test cases, the largest k is only 32.

Take both time complexity and memory access speed, we see from **Table 7** that the algorithm above is suitable for GPU.

Table 7: Time cost of sorting using trick 3

sample	original time	optimised time	original speed up	optimised speed up
small	3.6265	1.6138	5.51	12.39
middle	96.7870	14.7265	3.93	25.80
large	3085.1785	226.2323	4.06	55.39

5.4 Conclusion

From **Table 8**, we see the overall optimising after applying the three tricks talked above.

Table 8: Time cost of both passes after optimising

sample	original time	optimised time	original speed up	optimised speed up
small	29.7710	6.0522	3.02	14.87
middle	1764.2568	298.9564	1.84	10.87
large	111086.0613	18846.3651	2.46	14.52

6 Contest Programs

In order to make better performance in contest, we do some simple modifications such as remove some useless flow control statements like `if-else` and avoid some unnessary parameters or computations in functions. Also, we keep the trade-off between optimising strategy and sample scale. We compare the performance with the genernal program including all optimising explained in the previous chapter and the results are shown in **Table 9**.

Table 9: Performance compare between genernal and contest programs

sample	genernal program	contest program
small	6.0522	5.8431
middle	298.9564	294.9636
large	18846.3651	18400.6765

7 Usage

Type

```
$ make
```

and programs will be compiled.

Note that the following programs do different things:

`knn` the genernal program suitable for all input data

`compute` the contest program suitable for contest **ONLY**

Please specify an input file path as argument.

References

[1] Peterson, L. E. (2009). “K-nearest neighbor” . In: Scholarpedia 4.2, p. 1883.

[2] Kuang, Quansheng and Lei Zhao (2009). “A practical GPU based kNN algorithm” . In: Interna- tional Symposium on Computer Science and Computational Technology (ISCSCT), pp. 151– 155.

[3] Vincent Garcia, Eric Debreuve, Michel Barlaud. Fast k Nearest Neighbor Search using GPU. Universite de Nice-Sophia Antipolis/CNRS. Laboratoire I3S, 2000 route des Lucioles, 06903 Sophia Antipolis, France.