Proposal Report On
Parallel Implementation of K-NN Classifier

Kajol Jain
NITK
Computer Science Engineering
Roll No- 17CS014
Email: kajoljain379@gmail.com

Priyanka Gupta
NITK
Computer Science Engineering
Roll No- 17CS019
Email: prigupta9875@gmail.com

I. Problem statement

k-Nearest Neighbor (k-NN) is a classification algorithm used in machine learning and data mining applications such as email spam filtering, content retrieval , customer segmentation in online shopping websites etc.In the algorithm, the variable k is defined by the user.k-NN classifiers find the k number for training set, which has similar or is closest to the test data.The k-NN algorithm can be used for classification, based on the distance to the k nearest members ,the algorithm decides which class the given input should belongs to.

II. Objectives

High parallelism can be achieved using GPU and in comparatively lesser cost than CPU.
So we are trying to do the same task in GPU which speedup the execution by 99% .We will do the implementation consists of different level of parallelism like searching ,sorting and other parallely executable task.

III. Work Distribution among team members

This algorithm has three main steps and they are,
1) distance calculation 2) nearest neighbor search 3) class prediction
So,we will apply the parallelism on each above steps and each teammate will work on one algorithm and together merge it with class prediction step .

IV. Implementation

We have done the serial implementation of KNN algorithm which focuses on how to classify a large dataset which takes significant amount of time in a conventional CPU.We have done this via following steps i.e.

A. Distance calculation:

We have considered the SHUTTLE numeric dataset from the website of Machine Learning Repository. The dataset consists of 43,500 training data and 14,500 testing data, each having nine attributes without any missing components.The distance calculation should measure how close the given vector (also known as the test vector) is to all of the training vectors.We used the Euclidian distance (D) method to calculate the distance between two vectors X and Y of size n can be calculated using Equation 1.

$$D = \sqrt{\sum_{i=1}^{n}(Xi - Yi)^2} \tag{1}$$

B. Nearest Neighbor Search:

In the second step, minimum k distance values will be sorted with their corresponding class values for the test data.So,these k values can we used to predict final class.

C. Class prediction:

The final phase locates the majority class value in the sorted training set and predicts that class to the test data.So, occurance of each class is calculated that whichever class occur max , that class is taken as corresponding class for that point and this procedure for all test data.

V. Result of Serial implementation

We implement Serial algorithm of KNN classifier with all steps mentioned on dataset taken from website Machine learning repository. Here we run each step serially on 14500 test data by calculating its Euclidian distance from all 43500 train dataset. We also check our result from given data result and it predicts resultant class correctly for 99.9% of test data. We check it using different values of k.

VI. Proposed Improvements

In refered paper by us,they apply sorting on full distance array but as we required only k nearest neighbours so, in our implementation we apply selection sort k times which improve execution time from O(nlogn) to O(n).

VII. Parallel Implementation

KNN algorithm is data parallel application,so we can improve its performance drastically using parallel computing which improves it near by 100 times in compare to serial implementation on CPU. So,there we make steps parallel like distance calculation and Nearest neighbour classification This serial implementation with k value 12 takes 43 second on CPU .

A. Distance calculation

We designed the process as each block responsible for calculating the distance between one test and all training data set.This kernel returns all the distances as a 1D array to find the k-nearest neighbors. The length of the array is the multiplication of the number of test records and the number of training records.

B. Nearest Neighbour

In this kernel, we define the number of threads as equal to the number of testing records in the testing dataset. we send only the distance array one test data per thread. It finds the index of minimum k distance for each test data.

C. Class Prediction

We find the corresponding class values from class array and replace the index with the corresponding class value and we return the k size array for each test record.This kernel returns k-dimensional array for each test record to class prediction function which is implemented in CPU.

VIII. Experimental Setup

The CPU program is tested on a machine with Intel core i5-3470 CPU @ 3.20GHz processor and 8GB memory. The GPU program is tested on Tesla K40, which consists of 2880 cores and supports up to 30720 threads. The GPU memory was 12GB. We have considered the SHUTTLE numeric dataset from the website of Machine Learning Repository. The dataset consists of 43,500 training data and 14,500 testing data, each having nine attributes without any missing components. The dataset was stored in the CSV format. We have divided the dataset into various sizes of training sets and test sets to evaluate the execution time of the distance calculation phase and the nearest neighbor search of the k-NN algorithm across GPU and CPU. The number of neighbors (k value) is taken as 5 for all the test cases.

A. First Implementation

First implementation was the serial implementation as we discussed above.In this we calculate the Euclidian distance of 14500 test point with all the train data i.e. 43500 trained data points in a serial manner .Each data point has 9 attributes . After finding the distance we extract the nearest k value from the distance arrray and then findout the nearest neighbour class. This serial computation on CPU takes more time whenever data increases fastly. To minimize the time taken by serial implementation we implement the parallel version of implementation by using GPU supported architecture .when the test data size increases, the execution time of both CPU and GPU increases linearly. When the test data size is 14,500, the CPU take about 43 seconds.

B. Second Implementation

we implement the parallel program of K-NN classifier .We write the two kernel module one for calculating the Euclidian distance and other for findind k nearest neighbour .While calculating the distance we designed the process as each block responsible for calculating the distance between one test and all trainingdata set. Inside the block, each thread calculates the distance between one test data and one training data. The blocks are arranged in a 2D thread organization.The corresponding attributes between testing and training dataset are subtracted first. After that, it takes the square root of summation of subtracted values and store that value in a correct index of the distance array. Index identifying like $index = TrainRecordIndex * NumberofTestRecord + TestRecordIndex$.

This kernel returns all the distances as a 1D array to find the k-nearest neighbors. The length of the array is the multiplication of the number of test records and the number of training records.
This array will pass to the other kernel i.e. K-NN nearest neighbour for predicting the corresponding class.First it sort the first k nearest distance and then findout that from which class majority of data points belong that will be the resultant class of the corresponding test point. This will take the approx 20 sec for more than 14500 test point with 9 attributes.This implementation will speed up the execution more than 70 $\%$.

C. Third Implementation

We have presented a GPU based implementation of k-NN classification algorithm with two kernels, the distances calculation kernel and the nearest neighbor search kernel.There are some limitations in the current implementation of k-NN algorithm in GPU. Though the program results for a large and common machine learning dataset, but the program fails to handle larger data, i.e. when it is exceed the device hardware memoryand loaded into device memory. The reason is that the program tries to do all the work at once loading the whole training and testing data set into the memory.
Therefore, we try to develope the chunk concept .We divide the test data in the form of chunk of small size according to device hardware memory available .
We are doing the Euclidian distance calculation and k nearest neighbour parallely but the chunkwise . After that we store the result of each chunk which is the storing of k nearset neighbour of each chunk data and at last we combine all the data of each chunk and at last we do class prediction of all the stored data combinely .We use 100000 training data and 40000 test data in chunk implementation .It will take 135 sec in serial implementation and 56 sec in chunk implementation .Which is not possible with these much data in our first parallel implementation due to device hardware implementation because of no of thread created by these much data.

```
__global__ void euclideanDistance() {

 int row = blockIdx.x * blockDim.x +
threadIdx.x;
 int col = blockIdx.y * blockDim.y + threadIdx.y
 distanceId = row * testRecords+col;

 if (row less than trainRecords and col less
than
testRecords)
{
 for (i = 0 to attributes - 1) {
 diff <= (d_trainSet [row*attributes + i] –
d_testSet[col*attributes +
i])
 sum += diff * diff;
 }
 distanceArray [distanceId] <= sqrt(sum)
 }
}
```

Fig .1: Kernel code for Euclidian distance

```
__global__ void NearestNeighbour(){

int threadId = blockIdx.x * blockDim.x +
threadIdx.x;

if(threadId less than testRecords){
for(b = 0 to < k){
Min <= distanceArray[threadId]
index = 0;
for(c = 1 to c){
if(Min greater than distanceArray[c*testRecords
+ threadId]){
Min <= distanceArray[c*testRecords + threadId]
index <= c
}
if((Max less than distanceArray[c*testRecords +
threadId]) and (b equal to 0)){
Max <= distanceArray[c*testRecords + threadId]
}
}
distanceArray[index*testRecords + threadId] <=
Max
class[testRecords*b +threadId] <=
trainClass[index]
}
}
}
```

Fig.2 : kernel code for nearest neighbour

```
__global__
void class_classification(int *index_chunks,double **set, int
total_chunks_train ,int k,int *d_kneighbours,int set1,int
*res_class){
int i=blockDim.x*blockIdx.x+threadIdx.x;
int set_i;
int min=0;
for(int f=0;f<total_chunks_train;f++)
index_chunks[f]=0;
if(i<test_row)
{
for(int l=0;l<k;l++)
{
min=0;
for(int j=1;j<total_chunks_train;j++)
{
if(set[min][(index_chunks[min]*test_row+i)*2]>set[j][(index_ch
unks[j]*test_row+i)*2])
min=j;
}
set_i=i*set1;
d_kneighbours[(int)(set[min][(index_chunks[min]*test_row+i)*2
+1])-1+set_i]+=1;
index_chunks[min]++;
}
set_i=i*set1;
int max=0;
for(int l=1;l<set1;l++)
{
if(d_kneighbours[set_i+l]>d_kneighbours[set_i+max])
max=l;
}
res_class[i]=max+1;
}
}
```

Fig.3 : kernel code for nearest neighbour in chunk implementation

X. CONCLUSION

We have presented three implementation first is serial implementation ,second is a GPU based implementation of k-NN classification algorithm with two kernels, the distances calculation kernel and the nearest neighbor search kernel.

We tested the solution using a standard dataset in both CPU and GPU and compared the results separately. We could find a significant speedup of the algorithm specially when the test data size becomes larger. For a test data set of 14,000, we could observe around 70x speedup in GPU.In the third implementation we tryed to remove the device memory hardware limitation by introducing the chunk concept.So that we can the big data with the same hardware