Performance Analysis of kNN Query Processing on large datasets using CUDA & Pthreads

comparing between CPU & GPU

Preetham Kalakuntla

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
This thesis is submitted to Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Electrical Engineering with emphasis on Telecommunication Systems. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Preetham Kalakuntla
Email(s):
prka14@bth.se

University advisor:
Prof. Lars Lundberg
lars.lundberg@bth.se
Dept. Computer Science & Engineering

Dr. Yulia Sidorova
ysi@bth.se
Dept. Computer Science & Engineering

Faculty of Computing
Blekinge Institute of Technology
SE_371 79 Karlskrona, Sweden
Internet : www.bth.se :
Phone : +46 455 38 50 00 :
Fax : +46 455 38 50 57
Abstract

Telecom companies do a lot of analytics to provide consumers a better service and to stay in competition. These companies accumulate special big data that has potential to provide inputs for business. Query processing is one of the major tool to fire analytics at their data.

Traditional query processing techniques which follow in-memory algorithm cannot cope up with the large amount of data of telecom operators. The k nearest neighbour technique (kNN) is best suitable method for classification and regression of large datasets. Our research is focussed on implementation of kNN as query processing algorithm and evaluate the performance of it on large datasets using single core, multi-core and on GPU.

This thesis shows an experimental implementation of kNN query processing on single core CPU, Multicore CPU and GPU using Python, P-threads and CUDA respectively. We considered different levels of sizes, dimensions and k as inputs to evaluate the performance. The experiment shows that GPU performs better than CPU single core on the order of 1.4 to 3 times and CPU multi-core on the order of 5.8 to 16 times for different levels of inputs.

Keywords: GPU, Multicore CPU, Parallel computing, Performance, Single core CPU, kNN, Query Processing
Acknowledgements

Firstly, I would like to express my deep sense of thanks and gratitude to my supervisors Prof. Lars Lundberg and Dr. Yulia Sidorova for her valuable time and guidance to complete my thesis successfully.

I am grateful to Telenor for providing me necessary equipment to complete this thesis work.

I would like to thank my examiner Prof. Kurt Tutschku and others for their valuable suggestions and encouragement.

Special thanks to Saisree Junuthula for her continuous support and help. I would like to take this opportunity to thank my best friends and well-wishers for their love and care.

A very special thanks to all my friends for their valuable suggestions and support.

Finally, I would like to thank to my parents for their love and support.
List of Figures

2.1. kNN classification ................................................................. 5
2.2. Single core CPU ................................................................. 7
2.3. Multi-core CPU ................................................................. 7
2.4. Fundamental designs of CPU and GPU .................................... 9
2.5. Execution of CUDA program. ............................................... 10
2.6. Threads Organization in CUDA ............................................. 11
2.7. Overview of CUDA thread organization. ............................... 11
2.8. Overview of CUDA memory model ....................................... 12
3.1. Experiment setup .............................................................. 15
3.2. Programming structure for CPU single core ............................ 16
3.3. Programming structure for CPU multi-core ............................. 17
3.4. Programming structure for GPU (Host code) .......................... 18
3.5. Programming structure for GPU (Device code) ...................... 19
3.6. Sample telecom data. .......................................................... 19
5.1. Average execution times for different input size n, dimensions d and k=5 on CPU single core processor ........................... 23
5.2. Average execution times for different input size n, dimensions d and k=5 on CPU multi-core processor ............................ 24
5.3. Average execution times for different input size n, dimensions d and k=5 on GPU using CUDA ........................................ 24
5.4. Average execution time for kNN query processing for varying values of k on GPU and GPU . . 25
List of Tables

4.1. Average execution time in seconds on CPU single core processor. ............... 20
4.2. Average execution time in seconds on CPU multi-core processor using Pthreads. ................................................................. 21
4.3. Average execution time in seconds on GPU using CUDA ......................... 21
4.4. Average execution time in seconds for varying K on CPU and GPU. .......... 22
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>kNN</td>
<td>k Nearest Neighbors</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instructions Multiple Data</td>
</tr>
<tr>
<td>Pthreads</td>
<td>POSIX threads</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter describes about the overview of the research work, aim and objectives and contributions towards research and organization of the thesis.

1.1 Overview

In modern day computers, CPU and GPU are integrated and implemented on the same chip set by the manufacturers of multi-core processors. The performance of multi-core processors for data streams is different for CPU and integrated GPU. The multi-core processors have become a platform for high performance computing due to their relative performance. NVIDIA released a platform for GPU. This platform helps in writing kernels directly aiming the GPU. Compute Unified Device Architecture (CUDA) is an extension of C programming language[1].

In CUDA programming, memory is allocated in CPU to store input data, then copy data from CPU to GPU memory. The kernel is executed in the GPU by selecting necessary grid dimensions. The results are then copied back to CPU memory and the allocated memory in GPU is released. CUDA requires a careful consideration of GPU architecture to achieve better performance. CUDA works on the principle of Single Instruction Multiple Data (SIMD).

Parallel computing is possible due to the developments in GPU. The speed of a task is improved by parallel computing which divides the task into sub-tasks. When large amount of data must follow the same instruction, the data is divided into several individual threads, where each thread executes the instruction separately with its own set of data. The final output data consists the combination of data from all threads which are executed separately.

Query processing is the process of retrieving information from a database according to a set of retrieval criteria. Query processing is an important tool used by the companies to fire analytic requests at their data. With ever increasing size of their data, companies are unable to use traditional query processing techniques. When datasets are large and high dimensional, traditional query processing techniques cannot cope. The k-nearest neighbour (kNN) algorithm is a suitable method for query processing in large datasets[2].

kNN algorithm is a simple, effective and efficient algorithm for pattern recognition, categorisation and object recognition. Of all machine learning algorithms, kNN is known to be simplest one. Improving the speed of search and accuracy are min challenges to kNN.

Nearest Neighbour Search:

- The search for k close vectors to given vector n (k-nearest neighbour).
- The search for vectors within a range of distance k (k-nearest neighbour).

Extensive number of uses like Computational Fluid Dynamics (CFD), Neural Network, Finite Difference Time Domain (FDTD), Support Vector Machine (SVM), Magnetic Resonance Imaging (MRI), Intrusion Detection, and so on., are being used to perform parallel
processing on CPU and GPU, where GPU is used as computation accelerator. Parallel computing played a vital role in speeding up the process of kNN.

1.2 Aim and Objectives

The aim of thesis is to evaluate and compare the performance of CPU and GPU on query processing using kNN algorithm and to implement it on telecom data.

- To develop a Python program for generation of random datasets for experiment.
- To develop a sequential Python program on CPU for query processing using kNN and to record the execution time.
- To develop a parallel Python program on CPU for query processing using kNN and to record the execution time.
- To develop CUDA program on GPU for query processing using kNN and to record the execution time.
- To compare the performance of CPU and GPU.
- Implement the better performing query processing method on telecom data and record the time.

1.3 Research Questions

Research Question 1.
What is the execution time of kNN query processing for large datasets on single core CPU with Python program?

Research Question 2.
What is the execution time of kNN query processing for large datasets on multi core CPU with Python program?

Research Question 3.
What is the execution time of kNN query processing for large datasets on GPU using CUDA?

Research Question 4.
How can kNN query processing be implemented on Telecom data? What is the execution time?

Research Question 5.
Compare the performance of CPU and GPU on execution time of kNN query processing for large datasets?

1.4 Main Contributions

- This thesis work adds an advantage to ongoing research on parallel implementation on CPU using P-threads and on GPU using CUDA.
- The experimental results show the performance of query processing using kNN algorithm on CPU and GPU on different levels of size and dimensions and processing different kinds of queries on the available data.
• The results obtained from this thesis can be used by other researchers for reference of their work.
• The results of this thesis can be used for a better approach to query processing on large datasets.

1.5 Document Organization

The report has been meticulously organized to provide the reader the ease to understand and follow. Hereafter, the report is organized as follows.

Chapter 2 gives an overview of the work done by various researchers on this research area. Brief explanation of kNN query processing algorithm and introduction about P-threads and CUDA programming. It is followed by implementation of kNN query processing and use of CUDA parallelisation proposed by various authors.

Chapter 3 describes about the methodology followed to implement the experiment and specification of experimental setup.

Chapter 4 is about the results obtained from the experiments conducted by varying the input (size and dimension), which are used to compare the performance.

Chapter 5 is about analysis of the results obtained and presents the verification and validation of the experiment and outcomes. The analysis helps to answer the research questions.

Chapter 6 gives the conclusion which means solutions to research questions mentioned and future work to our research.
Chapter 2

Background

2.1 Query processing

Query processing is an important tool used by companies to fire analytics requests at their data. Query processing is the process of retrieving information from a database according to a set of retrieval criteria[4]. Applications of query processing may include vehicle navigation, wildlife social discovery, and squad/platoon searching on the battlefields. Query processing is implemented in telecommunication field for traffic estimation and analysis of usage patterns of the user.

Identifying the queried object, from a large volume of given uncertain dataset, is a tedious task which involves time complexity and computational complexity. Traditional Query processing techniques like B+tree, R-trees etc., follow in-memory algorithms to get the result. When the datasets are large, traditional query processing techniques cannot cope. To solve these complexities, various research techniques were proposed. Among these, the simple, highly efficient and effective technique is, K-Nearest Neighbor (kNN) algorithm. It is a technique which has applications in various fields such as pattern recognition, text categorization, moving object recognition etc. It has so many advantages like simplicity, robust to noisy training data, improved query time and memory requirements etc., and have disadvantages like Computation Complexity, Memory limitation and high cost in execution of algorithm [4].

2.1.1 K-Nearest Neighbor

K-Nearest Neighbor (kNN) is one of the simplest of machine learning algorithms. kNN is used in pattern recognition for classification and regression. For both classification and regression, the input consists of the k nearest training examples in sample space and the output depends on classification or regression. kNN has been used in statistical estimation and pattern recognition [4]. There are different measures for distance calculation like Euclidean, Euclidean Squared, City-block and Chebyshev. Among all these Euclidean is most popular choice to measure the distance between to the two points.

The nearest neighbour (NN) rule identifies the category of unknown data point based on its nearest neighbours whose class is already known. This rule is widely used in pattern recognition, text categorization, ranking models, object recognition and event recognition applications. Several methods have been proposed for efficient processing of nearest neighbour queries for stationary points. The k- nearest neighbour classifies the whole data into training data and sample data point. Distance is evaluated from the sample point to all training points and the training points are sorted based on distance. The k training points with least distance are called k-nearest neighbours. The value of K affects the results in some cases. The distance calculation in kNN is done by using the Euclidean distance formula[5].
\[ E(x, y) = \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2} \]

Equation 2.1: Euclidean distance formula

In figure 2.1, the figure explains classification of an object using kNN algorithm. The Red star is the point of unknown class. Green squares are class 1 and the blue hexagons are class 2. kNN algorithm is used to find the class of the unknown point. Distances are calculated from the sample point to all the known points and are sorted. When \( K = 3 \), then the point belongs to class 2, when \( k = 5 \), then the point belongs to class 1. From the example, it is clear that the value of \( k \) determines the class of the point. The value of \( K \) should not be a multiple of the number of classes present because if all the classes get an equal number of nearest neighbours, the point cannot be classified.

The kNN query processing can be classified into two different categories:
1) Structure based
2) Non-structure based

The structure-based kNN algorithms are Ball Tree, k-d Tree, Principal Axis Tree (PAT), Orthogonal Structure Tree (OST), Nearest Feature Line (NFL), Centre Line (CL). Non-structure based kNN algorithms are Weighted kNN, Condensed NN, Model based k-NN, Ranked NN (RNN), Pseudo/Generalized NN, Clustered kNN (CkNN), Mutual kNN (MkNN), Constrained RkNN. The structure-based algorithms suffer due to memory limits. So, structure-based kNN algorithms can be used for small volumes of data. The non-structure-based kNN algorithm suffers due to computation complexity. It can be used for large volumes of data [5].
To improve speed of classical kNN, many techniques such as ranking, false neighbour information, clustering etc., are used. Non-structured k-NN techniques are further amplified in the areas of moving object which uses query indexing technique instead of object indexing. Object indexing technique is not suited for the database where the objects are continuously moving and so uses Query indexing. Most of the indexing techniques such as R-trees, B+ trees etc., are based on the disk-based indexing which is not well-suited for moving objects for the following reasons [5]:

(i) updating the index when the object moves;
(ii) Frequent revaluation of queries when any object moves; and
(iii) achieving minimum execution times for large number of moving objects and queries.

Also, the cost of executing these algorithms from main memory is high. In both the cases, kNN algorithm is well suited for finding the solution.

In this research, we are considering large datasets which have high input samples and are high dimensional. So, Non-structured kNN query processing technique is used. kNN algorithm is used for query processing by setting a set of retrieval criteria. According the criteria, the distances are calculated only in the required dimensions. Let us consider a dataset has 100,000 sample points and 15 dimensions. If the requirement according to the criteria is to calculate 10 dimensions among 15, remaining 5 dimensions are not considered during calculation of the distance.

2.2 Parallel Computing

The invention of multi-core CPU’s in early 2000’s, helped parallel computing make a huge leap forward. Earlier many hardware and software improvements had to be made to achieve high performance. Pipelined functional units, hyperthreading etc., were used which lead to addition of cooling systems in hardware which increases cost. With the invention of multi-core processors, the performance is increased without much addition to hardware costs.

2.2.1 Use of parallel computing

Complex problems can be solved with faster computers and the computations can be performed in a better way in the same or less amount of time. When same instruction must be followed by large number of inputs, the inputs can be divided into number of threads. The threads are executed parallelly which takes less time than when executed sequentially.

2.2.2 Pthreads

In a single core CPU, computation is done using single core, whereas multi-core has more than one core for computation. The figure 2.2 shows the single core architecture and figure 2.3 shows multi-core architecture.

Multi-core processors work on Multiple Instructions Multiple Data (MIMD), in which different threads can be launched on different parts of the memory. The aim is to convert sequential program into a simultaneous process where the tasks are run independent of each other which results in the increase of the speed of the process [6]. Parallelism can be done in various ways like domain decomposition, task
decomposition and pipelining. In domain decomposition method, large data is divided into small chunks of data and the data is processed by each thread independent of other threads. Each thread executes the same instruction with its set of data and after, all the data execution, output from each thread combines to produce final output. Where as in task decomposition method large task is divided into subtasks and each thread is executed independently. In pipelining, two threads execute two tasks concurrently overlapping the execution and making it faster. The following are important to make the task parallel:

- Check if the algorithm is suitable for parallelism.
- Find which part of the algorithm can be parallelised.
- Distributing the tasks to all the available threads.
- Synchronisation of execution.

**Single-core CPU chip**

![Single core CPU chip](image)

**Figure 2.2: Single core CPU[7]**

**Multi-core CPU**

![Multi-core CPU](image)

**Figure 2.3: Multi-core CPU [6]**
POSIX threads, popularly known as Pthreads by which the implementation of parallelism became easy on multi-core CPU’s. This is a standard Python language library and can be downloaded for free from IEEE and other sites online. In thread library, all the identifiers begin with a Pthread. The Pthreads API can be informally classified into four major groups, namely thread management, mutexes, condition variables and synchronization.

i. **Thread management**: The routines that works on threads for creating, detaching, joining, etc.

Any program by default runs on single thread. Threads can be launched explicitly by the programmer using routines.

The syntax for creating threads and running is:

```python
threads = X
pool = ThreadPool(threads)
function = pool.map(aa, bb)
pool.close()
pool.join
```

ii. **Mutexes**: Mutexes is an abbreviation for mutual exclusion, which deals with synchronization. This function provides for creating, destroying, locking and unlocking mutexes.

iii. **Condition variables**: Condition variables are a routine that are used for communication between threads that share mutexes. This group also includes functions like create, destroy, wait, etc.

iv. **Synchronization**: Synchronization routine is used for managing read/write locks and barriers.

### 2.3 GPU Computing

#### 2.3.1 History of GPU

Early GPUs were designed specifically for graphics applications. It settled out somewhere around early 90s with a fixed pipeline/function. Those were the devices that had specific silicon for specific operation like shading a triangle and so on. At one point, NVIDIA realizes that there is much more flexibility if programmability is added on these devices. And eventually there were the vertex shaders and the pixel shaders which were basically either pieces of programs that operated on 2D data structures or pieces of programs that operated on 3D data structures. In 2008, when developing the TESLA GPU, it was realized that one can use the same programming interface on the same silicon to do both 2D and 3D operations. And that is basically where the Compute Unified Device Architecture (CUDA) was born. Since then all the newer generation GPUs are CUDA capable and will remain CUDA capable in the future. Size of the device is proportional to the number of transistors that is located on the device.

NVIDIA also developed the CUDA with C/C++ compiler, libraries and run time software which helps programmers to access the new data parallel computation model and
develop applications. Efficient threading support has been provided which allows the applications to handle large amount of parallelism than the available hardware execution. A graphic program or CUDA program is written once and runs on a GPU with many number of processor cores[8].

2.3.2 Overview and architecture of GPU

The fundamental difference in design of multi-core CPU and many-core GPU is the reason for large performance difference. The differences are shown in the picture below 2.4. As shown in figure 2.4, GPU consists more number of ALU’s than typical CPU and fewer components for cache and flow control, which implies high arithmetic intensity of operation and capacity to process parallel arithmetic operations, this helps in same operation can be performed on many different data elements. The hardware in GPU takes an advantage of large number of executions which means threads to do work, where some of them waiting for long latency[8]. In GPU, small cache memories are used to help control the bandwidth requirements of application hence multiple threads that access same memory data do not need to go back to DRAM. Any application whose function can be parallelized is perfectly suitable to implement on GPU for faster outputs [9].

The design of CPU is efficient for sequential code performance and another important aspect is the large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Memory bandwidth is the most important issue in CPU. This is because graphic chips have approximately 10 times the bandwidth of that of CPU[8].

The design of GPU is shaped by fast growing video gaming industry, which adds a tremendous performance on a massive number of floating point calculation per video frame in advanced graphic video games. The hardware takes an advantage of executing the threads to do the work load when some of the threads are waiting for long latency memory access. Here small cache memory plays a very important role because the applications need bandwidth. The threads using same memory data do not need to go all the way to the DRAM for floating point calculations [6].

![Fundamental designs of CPU and GPU](image-url)
2.3.3 CUDA programming model

The speed of execution of the CUDA program is accelerated only if the software efficiently exploits the parallelism provided by the hardware architecture underlying the multiprocessor. There is hence a need to develop the algorithm in such a way, that it can effectively use the multithreaded structure and contribute towards potentially increasing speed of execution. To develop such an algorithm, a parallel programming model is required that would support the parallel programming environment. CUDA is a parallel computing platform and programming model introduced by NVIDIA to be used in the GPUs that they produce. This model provides developers access to the memory of CUDA compatible GPUs and the virtual instruction set. The CUDA C compiler efficiently manages hardware resources and provides functions libraries which hide the hardware resources making it possible for the programmer to efficiently exploit parallelism in the program without being an expert in the lower-level hardware architecture of GPU. Programmers who are familiar with C programming language can develop the GPU applications easily [8].

CUDA program structure

A CUDA program consists of different phases that are executed on either host (CPU) or a device (GPU). On host, little or no data which cannot be parallelised are implemented whereas large data which can be parallelised are implemented on GPU [8].

The host code is implemented in straight ANSI C code, which complies on CPU whereas device code is written using extended ANSI C with some keywords for indicating data-parallel function called kernels, which is compiled by nvcc (NVIDIA C compiler) and executed on GPU. The kernel function generates a large number of threads in device to apply data parallelism [8].

The figure 2.5 shows the execution of CUDA program.

![Figure 2.5: Execution of CUDA program [8]](image)

The execution starts with host code and then when kernel is launched, the execution will be processed on device where large number of threads are generated to get data parallelism. When all threads complete their task the corresponding grid (grid means collection of blocks...
and blocks means collection of threads) terminates and execution continues on the host until another kernel in launched [3].

CUDA threads

The GPU works on principle of single instruction multiple thread (SIMD) computing for CUDA programming. Each thread executes same CUDA kernel. These threads are hierarchically organized. The figure 2.6 show the organization of threads in blocks and blocks in grid.

![Threads Organization in CUDA](image)

A grid consists of one or more blocks and a block consists of one or more threads. Every block within a grid have unique block index to differentiate with other blocks and similarly every thread within a block have unique thread id to differentiate with other threads. The size of the grid is defined as M*N where M is the number of blocks and N is the number of threads in each block. The thread id is given by following equation [8] (2.2)

\[
\text{threadID} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}
\]  

(2.2)

Let us consider grid has 128 blocks (M=128) and each block has 32 threads (N=32). The total number of threads is 128*32=4096 in the grid and blockDim is 32 in the kernel. Thread 4 of block 5 has a threadID value of 5*32+4=164 and thread 16 of block 102 has threadId value of 102*32+16=3280.

![Overview of CUDA thread organization](image)
CUDA memory

Figure 2.8 shows the CUDA memory model. It consists of registers, global memory, shared memory, constant memory and texture memory. Since GPUs are hardware cards that are come with inbuilt memory. For host and device have different memory spaces in CUDA programming. GPU should allocate required memory in order to launch kernels. It is transferred using PCI bus using global memory, the communication between host and device is established. The data which is stored in global memory can be accessed by GPU for execution and after execution the output is transferred to CPU using PCI bus. All the blocks in the device shares the global and constant memory. All the threads in every block shares the memory of block, which have its own shared memory. Each thread within a block has its own private memory and registers.

CUDA program steps

The following are steps to launch a kernel in CUDA [8]:

1. To store the input in the device, allocate the sufficient memory.
2. Copying the input data from host memory to device memory which is allocated in GPU.
3. Launch the kernel in device by selecting grid dimension.
4. Copying the output from device memory to host memory (i.e to CPU).
5. Empty the allocated memory in the device.

The instructions shown in the figure below is executed in a sequential manner. The kernel execution should wait until all the data is stored into the GPU memory, and after execution it needs to send all the output data to the CPU before it can be displaced for the user.

Figure 2.8: Overview of CUDA memory model [8]
2.4 Previous Work

In [10], the author analysed the merits and de-merits of kNN query processing algorithm. It is found that the structure based kNN techniques suffer due to memory limit whereas the Non-structure based kNN techniques suffer due to computation complexity. Hence, structure based kNN techniques can be applied to small volume of data whereas Non-structure kNN techniques can be applied to large volume of data.

In [11], the author divided the search into multiple cone shaped areas centered at the query point. It then performs a query dissemination and response collection itinerary in each of the cone-shape areas in parallel. The results show that DIKNN yields substantially better performance and scalability over previous work. It outperforms the second runner with up to 50% saving in energy consumption and up to 40% reduction in query response time, while rendering the same level of query result accuracy.

In [2], authors used a set of algorithms, based on different strategies. The algorithms have distinctive performance characteristics and are desirable under different contexts. The performance of these algorithms has been evaluated under several sensor network scenarios and application requirements.

In [12], author proposed four different methods for solving the problem of finding k nearest neighbors of a moving query point. In the experiments, the algorithms always outperform the existing ones by fetching 70% less disk pages. In some settings, the saving can be as much as one order of magnitude.

In [13], author investigated the benefits of network aware query processing for widely distributed environments. The results quantify the significant benefits of the network-aware approaches and reveal the fundamental trade-off between bandwidth efficiency and result latency that arises in networked query processing.

In [3], author analysed the performance of kNN algorithm on single-core and multi-core CPU and GPU. The execution time is measured by varying the number of datasets and number of dimensions. It is found out that the performance of GPU is significantly better than the performance of single-core and multi-core CPU.

In [14], the author executed the AES algorithm on CPU and GPU to compare the performance. The experiment has been conducted by using different granularity levels and different grid dimensions which effect the performance. The results showed that GPU performed significantly better than single-core and multi-core CPU.
Chapter 3

Methodology

This chapter describes the methodology used for the research. This is the crucial part of the research which gives the results to the research questions and problem statement. This research is carried out in two stages, literature review and in experiment. In the stage of literature review, researcher gains a knowledge over the research domain and it helps to give hypothetical output of the research. This is also helps to fill the knowledge gap on kNN algorithm, parallel programming on CPU and GPU. In the stage of experimentation, the gained knowledge from literature review is used to implement to validate the hypothesis. In this section the details or prerequisite for experiment, experiment setup and experiment are explained briefly.

3.1 Pre-Requisites for experiment

- CPU
  - Intel Core i7-6700HQ Processor
  - No of cores: 4
  - No of threads: 8
  - Clock speed: 2.6GHz

- A CUDA-enabled GPU
  - NVIDIA GeForce GTX 960M
  - 640 processing cores

- Windows 10 Operating System
- Microsoft Visual studio 2015
- Python toolkit
- NVIDIA CUDA toolkit

3.2 Experimental Setup

Figure 3.1 shows the flow of experiment.
3.3 Experiment

To evaluate the performance of k NN query processing algorithm on CPU and GPU, a random dataset generated using Python program is considered. The dataset can be generated according to required number of elements(M) and number of dimensions(N). For each implementation of nearest neighbour search, execution time is noted. First the distance is calculated by measuring the Euclidian distance from the query point to each and every sample point. The sample points are sorted according to distances. The sample points with least k distances are the K nearest neighbours of the query point.

First a sequential program is implemented to find nearest neighbours. A parallel program is implemented on CPU using multi threads. Also, a parallel program is developed on GPU using CUDA the execution time is recorded for all the execution types.

In this experiment, we are going to calculate the execution time by varying the size of dataset (N), dimensions (M) and value of k for each implementation.

Euclidean distance [13] (d) between two points \( x \) and \( y \) of M dimensions is given by equation 2.1.
3.3.1 Implementation on CPU single core processor

A single core processor is used to execute the implemented program in CPU. A sequential code is written using Python language and to store the input data, memory is allocated. The structure of the program is showed in figure 3.2. The program is attached in appendix A. The execution time is taken for distance calculations and sorting them for different input samples.

```python
import time
import csv
import operator
import math

def main():
    scanf k
    with open('testfine.csv')

    scanf input for query
    start clock

    for(...condition...)
        distance calculation
            //distance is calculated between query point and every row in the array/
    for(...sort...)
        sorting
            //sorting is done based on the distance calculated and k/
    end clock
    total time=end-start
    printf(total time taken= %d, total time)

Figure 3.2: Programming structure for CPU single core
```

3.3.2 Implementation on CPU Multi-core processor using Pthreads

Multiple cores are used to execute the implemented program in CPU. In our experiment, CPU consists of 4 physical cores and 2 virtual cores. Using all the available cores efficiently one can achieve high performance. So, total 8 threads are used to process the entire dataset. The program is developed in such way that entire input data is divided equally between the thread to execute. The structure of the program is showed in figure 3.3. The program is attached in appendix B
The most time taking part of kNN query processing algorithm is distance calculations and sorting the distances. Since GPU has many cores it is suitable for these types of applications. So, this has been implemented using two kernels, namely distance kernel and sorting kernel. The structure of the program on host is showed in figure 3.4 and program on device is showed in figure 3.5. The program is attached in appendix C. Here grid dimensions vary with size of input and number of the threads launched per block. 128 threads are launched in each block [3].

3.3.3 Implementation on GPU using CUDA

The most time taking part of kNN query processing algorithm is distance calculations and sorting the distances. Since GPU has many cores it is suitable for these types of applications. So, this has been implemented using two kernels, namely distance kernel and sorting kernel. The structure of the program on host is showed in figure 3.4 and program on device is showed in figure 3.5. The program is attached in appendix C. Here grid dimensions vary with size of input and number of the threads launched per block. 128 threads are launched in each block [3].

```python
import time
import csv
import operator
import math
import multiprocessing

def main()

    scanf k
    with open(testfile.csv)

    define threads=8
    scanf input for query
    start clock

    for(...condition...)

        for(...condition...)
            distance calculation
            //distance is calculated between query point and every row in the array/

        for(...sort...)
            sorting
            //sorting is done based on the distance calculated and k/

    end clock
    total time=end-start
    printf(total time taken= %d, total time)

    Figure 3.3: programming structure for CPU multi-core
```
Distance kernel
The aim of implementing this kernel is to calculate the distance between query point and the sample points which is done by different threads. Distance calculations can be fully parallelized since it is independent between pairs of objects. The data is transferred from CPU to GPU, whereas each thread involved in calculating the distance. If the number of points are large, a large number of threads and blocks are launched to execute this kernel. The following kernel is launched to calculate distance:

\[
\text{distance} \ll<\text{numblocks}, \text{threadsperblock}>> (d_a, d_c);
\]

Sorting kernel
This kernel is to sort the distances and to find the k nearest neighbors. Once the distance kernel execution is completed this kernel is invoked. The calculated distances are stored in shared memory. Each thread takes care of one distance. Now the challenging task is to find the k nearest neighbors. The distances in the shared memory are sorted and then copied to global memory, where the k nearest neighbors are calculated and copied back to CPU. The following kernel is launched for sorting the distances:

\[
\text{sorting} \ll<\text{numblocks}, \text{threadsperblock}>> (\text{sort}, k);
\]

```c
#include<br> #define<br> #define<br> #define<br> int main()<br> {
    FILE *myfile;
    myfile=fopen("testfile.c");
    for(, condition rows) {
        for(, condition cols) {
            fscanf("myfile", "%f", &mut[rows][cols]);
        }
        Scanf (k);
        Scanf (input for query)
        CudaMalloc((void**)&d_a, rows*cols*sizeof(int)); // allocate memory
        CudaMalloc((void**)&d_c, rows*cols*sizeof(int));
        CudaMalloc((void**)&sort, rows*rows*sizeof(int));
    }
    start=clock();
    //copy input from host to device
    cudaMemcpy(d_a, aa, rows*cols*sizeof(Int*), cudaMemcpyHostToDevice);
    distance<<<numblocks, threadsperblock>>>(d_a, d_c); //Distance
    sorting<<<numblocks, threadsperblock>>>(sort, k); //Sorting
    end=clock();
    total time=end-start;
    return 0;
}
```

Figure 3.4: programming structure for GPU(Host code)
3.4 Implementation of kNN query processing algorithms on Telecom data

Implementation of kNN query processing algorithm on the telecom data can be done for location aware queries for a given query point. Sample telecom data is given in the picture 3.6 below. The data consists of id, day, time and location of the user. The data can be used for traffic estimation in the network and analysis of usage patterns of the user.

<table>
<thead>
<tr>
<th>key</th>
<th>UserId</th>
<th>Weekday</th>
<th>Time</th>
<th>ProfileID</th>
<th>SegmentID</th>
<th>Originalroworder</th>
<th>EASTING</th>
<th>NORTHING</th>
<th>LATITUDE</th>
<th>LONGITUDE</th>
<th>BEAMDIRECTION</th>
<th>GSM</th>
<th>UMTS</th>
<th>LTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1210113812</td>
<td>25592353</td>
<td>Mon</td>
<td>08:20:00</td>
<td>130</td>
<td>0</td>
<td>1</td>
<td>1293120</td>
<td>6468310</td>
<td>58.28941</td>
<td>12.277146</td>
<td>160</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1210113781</td>
<td>25592353</td>
<td>Mon</td>
<td>13:05:00</td>
<td>130</td>
<td>0</td>
<td>2</td>
<td>1293100</td>
<td>6464280</td>
<td>58.25229</td>
<td>12.28404</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1210117842</td>
<td>25592353</td>
<td>Mon</td>
<td>14:05:00</td>
<td>130</td>
<td>0</td>
<td>3</td>
<td>1295890</td>
<td>6469100</td>
<td>58.2978</td>
<td>12.325387</td>
<td>130</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108120681</td>
<td>41985671</td>
<td>Mon</td>
<td>13:00:00</td>
<td>203</td>
<td>0</td>
<td>4</td>
<td>1297457</td>
<td>6314305</td>
<td>56.91104</td>
<td>12.4796</td>
<td>200</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>108121887</td>
<td>41985671</td>
<td>Mon</td>
<td>13:40:00</td>
<td>203</td>
<td>0</td>
<td>5</td>
<td>1304740</td>
<td>6315200</td>
<td>56.96064</td>
<td>12.751915</td>
<td>330</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>108120941</td>
<td>41985671</td>
<td>Mon</td>
<td>13:40:00</td>
<td>203</td>
<td>0</td>
<td>6</td>
<td>1308860</td>
<td>6320230</td>
<td>56.9689</td>
<td>12.657761</td>
<td>150</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1210119143</td>
<td>322596203</td>
<td>Mon</td>
<td>13:15:00</td>
<td>135</td>
<td>0</td>
<td>41</td>
<td>1289830</td>
<td>6459000</td>
<td>58.20443</td>
<td>12.229584</td>
<td>270</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>107129220</td>
<td>413147127</td>
<td>Mon</td>
<td>07:25:00</td>
<td>133</td>
<td>0</td>
<td>45</td>
<td>1301809</td>
<td>6461893</td>
<td>58.2359</td>
<td>12.430501</td>
<td>200</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>107129215</td>
<td>413147127</td>
<td>Mon</td>
<td>07:25:00</td>
<td>133</td>
<td>0</td>
<td>46</td>
<td>1301809</td>
<td>6461893</td>
<td>58.2359</td>
<td>12.430501</td>
<td>200</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.6: Sample telecom data

The telecom data considered for experiment consists of 1,000,000 elements and 15 dimensions. Queries are fired at the data using kNN query processing algorithm. The queries are executed to measure the time for different values of k on single core CPU and on GPU using CUDA. The execution time is measured and compared.
Chapter 4

Results

The work in this section gives the execution times in terms of performance (better approach) for k NN query processing algorithm on different level of parallelism on CPU and GPU.

4.1 Implementation on CPU single core processor

A single core program is implemented for kNN query processing algorithm. Various input samples are taken by varying number of sample points and number of dimensions. To get accurate results, 20 iterations of each sample of inputs are taken. Table 4.1 shows the average execution time on CPU single core processor. In appendix A, the results are shown for different input samples. Where n is number of inputs and d is dimension of each input.

<table>
<thead>
<tr>
<th>Rows</th>
<th>D=10</th>
<th>D=32</th>
<th>D=64</th>
<th>D=128</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>0.633</td>
<td>1.68</td>
<td>3.57</td>
<td>6.27</td>
</tr>
<tr>
<td>6000</td>
<td>0.807</td>
<td>2.02</td>
<td>4.23</td>
<td>7.38</td>
</tr>
<tr>
<td>7500</td>
<td>1.002</td>
<td>2.5</td>
<td>5.19</td>
<td>9.14</td>
</tr>
<tr>
<td>10000</td>
<td>1.388</td>
<td>3.18</td>
<td>6.89</td>
<td>12.11</td>
</tr>
<tr>
<td>12000</td>
<td>1.58</td>
<td>3.91</td>
<td>8.24</td>
<td>14.44</td>
</tr>
<tr>
<td>15000</td>
<td>2.01</td>
<td>4.95</td>
<td>10.18</td>
<td>17.94</td>
</tr>
<tr>
<td>20000</td>
<td>2.61</td>
<td>6.42</td>
<td>15.2</td>
<td>23.98</td>
</tr>
<tr>
<td>30000</td>
<td>3.77</td>
<td>9.54</td>
<td>22.9</td>
<td>36.08</td>
</tr>
<tr>
<td>50000</td>
<td>6.13</td>
<td>15.77</td>
<td>31.02</td>
<td>68.34</td>
</tr>
<tr>
<td>60000</td>
<td>7.28</td>
<td>19.41</td>
<td>48.26</td>
<td>77.03</td>
</tr>
<tr>
<td>75000</td>
<td>9.09</td>
<td>23.57</td>
<td>67.61</td>
<td>82.4</td>
</tr>
<tr>
<td>100000</td>
<td>12.1</td>
<td>31.87</td>
<td>88.83</td>
<td>130.6</td>
</tr>
<tr>
<td>150000</td>
<td>17.65</td>
<td>47.77</td>
<td>108.4</td>
<td>191.1</td>
</tr>
<tr>
<td>200000</td>
<td>24.58</td>
<td>63.22</td>
<td>148.7</td>
<td>280.2</td>
</tr>
</tbody>
</table>

Table 4.1: Average execution time in seconds on CPU single core processor

4.2 Implementation on CPU multi core processor

Multi-core program is implemented for kNN query processing algorithm. Various input samples taken by varying number of sample points and number of dimensions. To get accurate results, 20 iterations of each sample of inputs are taken. Table 4.2 shows the average execution time on CPU multi-core processors. In appendix B, the results are shown for different input samples. Where n is number of inputs and d is dimension of each input. Here 8 threads are launched, where entire input data is divided equal in such way that each can handle same number of inputs.
4.3 Implementation on GPU using CUDA

A CUDA program is implemented for kNN query processing algorithm. Various input samples are taken by varying number of sample points and number of dimensions. To get accurate results, 20 iterations of each sample of inputs are taken. Table 4.3 shows the average execution time on GPU. In appendix C, the results are shown for different input samples. Where n is number of inputs and d is dimension of each input.

<table>
<thead>
<tr>
<th>Rows</th>
<th>D=10</th>
<th>D=32</th>
<th>D=64</th>
<th>D=128</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1.9205</td>
<td>4.665</td>
<td>8.139</td>
<td>15.043</td>
</tr>
<tr>
<td>6000</td>
<td>2.3371</td>
<td>5.63</td>
<td>9.985</td>
<td>18.088</td>
</tr>
<tr>
<td>7500</td>
<td>3.037</td>
<td>7.08</td>
<td>12.44</td>
<td>22.766</td>
</tr>
<tr>
<td>10000</td>
<td>4.2765</td>
<td>9.49</td>
<td>16.905</td>
<td>30.5245</td>
</tr>
<tr>
<td>12000</td>
<td>5.4167</td>
<td>11.95</td>
<td>20.551</td>
<td>36.337</td>
</tr>
<tr>
<td>15000</td>
<td>7.31256</td>
<td>15.22</td>
<td>26.329</td>
<td>46.244</td>
</tr>
<tr>
<td>20000</td>
<td>11.0129</td>
<td>20.21</td>
<td>36.693</td>
<td>62.193</td>
</tr>
<tr>
<td>30000</td>
<td>21.167</td>
<td>38</td>
<td>60.378</td>
<td>98.845</td>
</tr>
<tr>
<td>50000</td>
<td>58.4451</td>
<td>92.2</td>
<td>129.89</td>
<td>201.7</td>
</tr>
<tr>
<td>60000</td>
<td>99.2984</td>
<td>129</td>
<td>187.15</td>
<td>262.5</td>
</tr>
<tr>
<td>75000</td>
<td>173.653</td>
<td>206</td>
<td>297.142</td>
<td>358.68</td>
</tr>
<tr>
<td>100000</td>
<td>331.46</td>
<td>385</td>
<td>498.33</td>
<td>527.6</td>
</tr>
<tr>
<td>150000</td>
<td>880.465</td>
<td>954</td>
<td>1055</td>
<td>1142</td>
</tr>
<tr>
<td>200000</td>
<td>1683.015</td>
<td>1818</td>
<td>1986</td>
<td>2146</td>
</tr>
</tbody>
</table>

Table 4.3: Average execution time in seconds on GPU using CUDA
4.4 Implementation of kNN query processing algorithms on Telecom data

Queries are fired at telecom data where number of rows are 1,000,000 and number of columns are 15. The experiment is conducted on single core CPU and GPU using CUDA by varying K from 1 to 1000. The execution time for various values of k are:

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
<th>k</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>1</td>
<td>162</td>
<td>87.516</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>5</td>
<td>168</td>
<td>88.704</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>20</td>
<td>171</td>
<td>90.288</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>100</td>
<td>187.8</td>
<td>93.456</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>500</td>
<td>208</td>
<td>96.624</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15</td>
<td>1000</td>
<td>234</td>
<td>97.812</td>
</tr>
</tbody>
</table>

Table 4.4: Average execution time in seconds on CPU and GPU for varying K
Chapter 5

Analysis

In this chapter, analysis of the results obtained from the experiment is presented and explicates the effect of results obtained from various implementations.

5.1 For single core processor

The results obtained on single core processor shows that increase in the execution time as the size of the input increases. Since the program is implemented as serial program using single core processor, the execution time increases with size of input. The following graph shows the average execution time for different input size (n), different dimensions (d).

![Graph showing average execution time for different input size n, dimensions for k=5 on CPU single core processor.](image)

Figure 5.1: Average execution times for different input size n, dimensions for k=5 on CPU single core processor.

5.2 For multi core processor

The results obtained on CPU multi-core processor shows that increase in the execution time as the size of the input increases. This program is implemented using P-threads. The CPU used in these experiments is having 4 cores which means it can launch 8 threads. The following graph shows the average execution time for different input size (n), different dimensions (d).
5.3 For GPU using CUDA

The results obtained for implementation of kNN query processing algorithm on GPU shows that the increase in the execution time as the size of input increases. The program is implemented in such a way that the distance is calculated by one kernel and sorting is done by another kernel which results in better performance when compared to implementation on both single-core CPU and multi-core CPU.
5.4 Comparison of CPU and GPU

The results obtained for implementation of kNN query processing using telecom data on CPU and GPU for varying values of k. The results show that the increase in k increases the execution time. From the graph, we can get that the increase in GPU is less than the increase in CPU.

![Graph showing execution time for varying k on GPU and CPU](image)

**Figure 5.4: Average execution times for varying values of k on GPU and CPU**

5.4 Validity Threat

There are two types of validity threats that need to be considered while conducting research, which are internal validity and external validity.

5.4.1 Internal Validity Threat

Internal validity threat means the ability of research paper to be able to correlate the cause and effect. Deviation from this validity may affect the accuracy of the results. To overcome the deviation, the following precautions should be taken:

- 20 iterations are considered in each case to get accuracy of the results. Standard deviation also calculated to check the consistency of the results.
- The output generated in all the implementations are same, which means the results for query are same.
- The performance variations are observed for different combinations of inputs.

5.4.2 External Validity Threat

External validity threat means the generalization of results of research work out-side the study. [15] This research work is carried out for parallel implementation of k NN. Hence it depends on the dataset size, which effects the performance of devices and not the algorithm complexity. The execution time depends on the type of GPU used in implementation. In this
research work, NVIDIA GeForce GTX 960M GPU is used. The performance may vary in comparison factor between different types of GPUs used. From the general observation from the results and literature study, NVIDIA GPUs using CUDA holds best results.
Chapter 6

Conclusion and Future Work

In the research, we compared the performance of kNN query processing on large datasets using single core CPU, multi-core CPU and GPU. Python program is used to execute kNN query processing on single core CPU and multi core CPU. CUDA program is used to execute kNN query processing on GPU. Python program is used to generate datasets of required number of elements and dimensions.

It is expected that the multi-core CPU outperforms single core CPU because of launching number of threads which parallelize the execution. From the experiment, we can see that the performance of single core CPU is better than multi-core CPU. This is because of an element present in python called as Global Interpreter Lock (GIL) [16]. It prevents multi thread execution in python. Threads are created are task is assigned for individual threads, GIL does not allow the threads to execute at once. It in turn creates overhead delay for each thread execution which results in high execution time for multicore processing in python.

From the experiment, we can conclude that GPU outperforms in all the condition when compared implementation on CPU single core with a factor of approximately 1.2 to 3 and implementation on CPU multi-core with a factor of approximately 5.8 to 16. The overhead due to copy of data from CPU to GPU and vice versa is 0.002 seconds which is negligible. Since the size of datasets are ever increasing in the companies, using CUDA program in GPU provides an advantage in execution of queries.

Execution for k=1000 in CPU took 45% more time than k=1. Execution for k=1000 in GPU took 12% more time than k=1. From there results we can concur that the impact of increase in k value is less on GPU than on CPU.

From the above results, it is safe to say that with proper parallelization of task, GPU performs better than CPU on large datasets.

6.1 Answers to research questions

Research Question 1.

What is the execution time of nearest neighbour search for large datasets on CPU with sequential python program?

Answer.

The execution times of five nearest neighbour search when the input size of 5000 and dimensions of 10, 32, 64, 128 are 0.633 sec, 1.68 sec, 3.57 sec and 6.27 sec respectively. Execution times for one nearest neighbour when input size of 50,000 and dimensions of 10, 32, 64 and 128 are 6.13 sec, 15.77 sec, 31.08 sec and 68.34 sec respectively. Similarly, the input of 200000 and dimensions of 10, 32, 64 and 128 are 24.58 sec, 63.22 sec, 148.7 sec and 280.42 sec respectively. Remaining values are tabulated in table 4.1. From all the results, we can say the execution time increased with increase in size and dimension of data.
Research Question 2.
What is the execution time of nearest neighbour search for large datasets on CPU with parallel Python program?

Answer.
The execution times of one nearest neighbour search when the input size of 5000 and dimensions of 10, 32, 64, 128 are 1.92 sec, 4.66 sec, 8.13 sec and 15.04 sec respectively. And the execution times for one nearest neighbor when input size of 50,000 and dimensions of 10, 32, 64 and 128 are 58.44 sec, 92.2 sec, 129.9 and 201 secs respectively. Similarly, for the input of 200,000 and dimensions of 10, 32, 64 and 128 are 1683 sec, 1818 sec, 1986 sec and 2146 sec respectively. Remaining values are tabulated in table 4.2. From all the results, we can say the execution time increased with increase in size and dimension of data. These results outperform than single core processor approximately 5 times.

Research Question 3.
What is the execution time of nearest neighbor search for large datasets on GPU using CUDA?

Answer.
The execution times of one nearest neighbor search when the input of 5000 and dimensions of 10, 32, 64, 128 are 0.645 sec, 0.99 sec, 1.96 and 2.7 secs respectively. And the execution times for one nearest neighbor when input size of 50000 and dimensions of 10, 32, 64 and 128 are 6.19 secs, 9.84 secs, 19.5 secs and 21.6 secs respectively. Similarly, for the input of 200,000 are 24.7 secs, 39.3 secs, 77 secs and 84 secs respectively. Remaining values are tabulated in table 4.5. From all the results, we can say the execution time increased with increase in size and dimension of data.

Research Question 4.
How can kNN query processing be implemented on telecom data? What is the execution time for kNN query processing on telecom data?

Answer.
kNN query processing can be implemented on large telecom data for firing location aware queries. The data can be used for traffic estimation in the network and analysis of usage patterns of the user. The execution time where k is 5, 20, 500, 1000 on CPU is 168, 171, 208, 234 secs respectively and execution time where k is 5, 20, 500, 1000 on GPU is 88.7, 90.2, 96.6, 97.8 secs respectively.

Research Question 5.
Compare the performance of CPU and GPU on execution time of nearest neighbor search for large datasets?

Answer.
By comparing all the above results we can conclude that GPU outperforms in all the condition when compared implementation on CPU single core with a factor of approximately 1.2 to 3 and implementation on CPU multi-core with a factor of approximately 5.8 to 16.
6.2 Future Work

- Due to limited time, this is research work is implemented on only one GPU (NVIDIA GeForce GTX 960M), since the performance may vary with another version of NVIDIA GPU's.
- This implementation can be done on GPU using CUDA STREAMS, which may give more optimized results then using CUDA.
- The multi-core python can be executed by eliminating GIL which could improve its performance.
References


Appendices

Appendix A

Program & Results for CPU single core processor

import time
import csv
import operator
import math

def euclideanDistance(instance1, instance2, length):
    distance = 0
    for x in range(length):
        distance += pow((instance1[x] - instance2[x]), 2)
    return math.sqrt(distance)

def getNeighbors(trainingSet, testInstance, k):
    distances = []
    length = len(testInstance)-1

    for x in range(len(trainingSet)):
        dist = euclideanDistance(testInstance, trainingSet[x], length)
        distances.append((trainingSet[x], dist))
        #print(distances)
    distances.sort(key=operator.itemgetter(1))
neighbors = []
for x in range(k):
    neighbors.append(distances[x])
return neighbors

def main():
s=raw_input()
testInstance = map(float,s.split())

k=raw_input("k:")
k=int(k)

start=time.clock()
with open('C:/Users/preetham/Desktop/thesis/trial/gentest/gentest/994439.13.csv') as csvfile:
    lines = csv.reader(csvfile)
    c=[]
    for row in lines:
        a=list(row)
        c.append(a)
#trainSet = [[3, 0, 0, 'a'], [0, 4, 0, 'b']]  
#testInstance = [0,0,0,'b']
#k = 1

e=[]
print(c)

for x in c:
h = len(x)
for z in range(h-1):

    x[z] = long(x[z])

#******************************
# item = map(int, x)
# l = len(item)
# item[l-1] = str(item[l-1])
# e.append(item)
# print(e)
# print(c)
#******************************

for y in range(len(testInstance)):
    if testInstance[y] == 1001001:
        print(testInstance[y])
        a[y] = 1001001

# neighbors = getNeighbors(e, testInstance, k)
neighbors = getNeighbors(c, testInstance, k)
print("*******preetham********")
# print(neighbors)

v = []
v1 = []

for x in neighbors:
    m = len(x)
    v.append(x[0][m-1])
v1=list(set(v))
f=[]
for i in v1:
    c=0
    for j in v:
        if i==j:
            c=c+1
    f.append([i,c])

print("here the c vLUES")
print(f)
print(v)
# print(v1)

k = max(f)
print(k)

# for value in f:
#      if value==k:
#          print(x)
#      w=v1[x]
#      print(w)

print((time.clock() - start))

main()
Results for CPU single core processor

<table>
<thead>
<tr>
<th>K=5</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>D=10</td>
<td>D=32</td>
<td>D=64</td>
<td>D=128</td>
</tr>
<tr>
<td>5000</td>
<td>0.633</td>
<td>1.68</td>
<td>3.57</td>
<td>6.27</td>
</tr>
<tr>
<td>6000</td>
<td>0.807</td>
<td>2.02</td>
<td>4.23</td>
<td>7.38</td>
</tr>
<tr>
<td>7500</td>
<td>1.002</td>
<td>2.5</td>
<td>5.19</td>
<td>9.14</td>
</tr>
<tr>
<td>10000</td>
<td>1.388</td>
<td>3.18</td>
<td>6.89</td>
<td>12.11</td>
</tr>
<tr>
<td>12000</td>
<td>1.58</td>
<td>3.91</td>
<td>8.24</td>
<td>14.44</td>
</tr>
<tr>
<td>15000</td>
<td>2.01</td>
<td>4.95</td>
<td>10.18</td>
<td>17.94</td>
</tr>
<tr>
<td>20000</td>
<td>2.61</td>
<td>6.42</td>
<td>15.2</td>
<td>23.98</td>
</tr>
<tr>
<td>30000</td>
<td>3.77</td>
<td>9.54</td>
<td>22.9</td>
<td>36.08</td>
</tr>
<tr>
<td>50000</td>
<td>6.13</td>
<td>15.77</td>
<td>31.02</td>
<td>68.34</td>
</tr>
<tr>
<td>60000</td>
<td>7.28</td>
<td>19.41</td>
<td>48.26</td>
<td>77.03</td>
</tr>
<tr>
<td>75000</td>
<td>9.09</td>
<td>23.57</td>
<td>67.61</td>
<td>82.4</td>
</tr>
<tr>
<td>100000</td>
<td>12.1</td>
<td>31.87</td>
<td>88.83</td>
<td>130.6</td>
</tr>
<tr>
<td>150000</td>
<td>17.65</td>
<td>47.77</td>
<td>108.4</td>
<td>191.1</td>
</tr>
<tr>
<td>200000</td>
<td>24.58</td>
<td>63.22</td>
<td>148.7</td>
<td>280.2</td>
</tr>
</tbody>
</table>
Appendix B

Program & Results for CPU multi core processor

```python
import time
import csv
import operator
import math
import multiprocessing

from multiprocessing.dummy import Pool as ThreadPool

def euclideanDistance1(everytrainingset):
    instance2=everytrainingset
    global testInstance
    instance1=testInstance
    length=len(everytrainingset)-1
    distance = 0
    distances=[]
    for x in range(length):
        distance += pow((instance1[x] - instance2[x]), 2)
    p= math.sqrt(distance)
    distances.append((instance2, p))
    return distances

def getNeighbors(trainingSet, testInstance):
    threads=8
```
pool=ThreadPool(threads)
distances=pool.map(euclideanDistance1,trainingSet)

pool.close()
pool.join
return distances

if __name__ == "__main__":

s=raw_input()
testInstance = map(int,s.split())
# print(testInstance)
k=raw_input("k:")
k=int(k)

start=time.clock()
with open('C:/Users/preetham/Desktop/thesis/trial/gentest/gentest/994439.13.csv') as csvfile:
    lines = csv.reader(csvfile)
    c=[]
    for row in lines:
        a=list(row)
        for y in range(len(testInstance)):
            if testInstance[y]==1001001:
                a[y]=1001001
                # print(a[y])
        c.append(a)
    # print(c)
c.append(a)

# print(c)

for x in c:
    h = len(x)
    for z in range(h - 1):
        x[z] = int(x[z])

print(c)

neighbors = []
distances = getNeighbors(c, testInstance)
print(distances)

# distances.sort(key=operator.itemgetter(1))
# distances.sort(key=lambda tup: tup[1])

d = reduce(operator.concat, distances)

# print(d)
d.sort(key=lambda tup: tup[1])

for x in range(k):
    neighbors.append(d[x])

# print("*******nikhil mara*******")
# print(neighbors)
print((time.clock() - start))

Results for CPU multi core processor

<table>
<thead>
<tr>
<th>k=5</th>
<th>Rows</th>
<th>D=10</th>
<th>D=32</th>
<th>D=64</th>
<th>D=128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5000</td>
<td>1.9205</td>
<td>4.665</td>
<td>8.139</td>
<td>15.043</td>
</tr>
<tr>
<td></td>
<td>6000</td>
<td>2.3371</td>
<td>5.63</td>
<td>9.985</td>
<td>18.088</td>
</tr>
<tr>
<td></td>
<td>7500</td>
<td>3.037</td>
<td>7.08</td>
<td>12.44</td>
<td>22.766</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>4.2765</td>
<td>9.49</td>
<td>16.905</td>
<td>30.5245</td>
</tr>
<tr>
<td></td>
<td>12000</td>
<td>5.4167</td>
<td>11.95</td>
<td>20.551</td>
<td>36.337</td>
</tr>
<tr>
<td></td>
<td>15000</td>
<td>7.31256</td>
<td>15.22</td>
<td>26.329</td>
<td>46.244</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>11.0129</td>
<td>20.21</td>
<td>36.693</td>
<td>62.193</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>21.167</td>
<td>38</td>
<td>60.378</td>
<td>98.845</td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>58.4451</td>
<td>92.2</td>
<td>129.89</td>
<td>201.7</td>
</tr>
<tr>
<td></td>
<td>60000</td>
<td>99.2984</td>
<td>129</td>
<td>187.15</td>
<td>262.5</td>
</tr>
<tr>
<td></td>
<td>75000</td>
<td>173.653</td>
<td>206</td>
<td>297.142</td>
<td>358.68</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>331.46</td>
<td>385</td>
<td>498.33</td>
<td>527.6</td>
</tr>
<tr>
<td></td>
<td>150000</td>
<td>880.465</td>
<td>954</td>
<td>1055</td>
<td>1142</td>
</tr>
<tr>
<td></td>
<td>200000</td>
<td>1683.015</td>
<td>1818</td>
<td>1986</td>
<td>2146</td>
</tr>
</tbody>
</table>
Appendix C
Program & Results for GPU
C.1 program

#include"cuda_runtime.h"
#include"device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

#define row 500000
#define col 15
#define rows 5000
#define cols 15

int aba[row][col];
int aa[rows][cols];
double cc[rows][rows];
double cpu_out[rows][rows];
int k;
int a, b, c, d;
double *d_c = 0;
int *d_a = 0;
int *d_i = 0;

__global__ void distance(const int * dev_a, double * dev_c, const int * dev_i)
{
    int th_num = blockIdx.x * 128 + threadIdx.x;
    double sum = 0;
    for (int k = th_num + 1; k<rows; k++)
    {
        for (int c = 0; c < cols; c++)
        {
            sum += (dev_a[th_num*cols + c] - dev_i[k*cols + c])*(dev_a[th_num*cols + c] - dev_i[k*cols + c]);
        }
        dev_c[th_num] = sqrt(sum);
    }
    // printf("Sum: %d \n", sum);
}

__global__ void sorting(double *dev_c, double *sort, double K)
{
    int temp; int i;
    i = blockIdx.x * 128 + threadIdx.x;
    for (int r = 0; r < rows; r++)
    {
        for (int c = 0; c < cols; c++)
        {
if (dev_c[cols*r + c] < dev_c[cols*i + c])
{
    temp = dev_c[cols*r + c];
    dev_c[cols*r + c] = dev_c[cols*i + c];
    dev_c[cols*i + c] = temp;
}

int main()
{
    printf(" enter the number of K nearest neighbors :");
    scanf("%d", &k);
    FILE *myFile;
    myFile = fopen("test.csv ", "r");
    if (myFile == NULL)
    {
        printf("Error Reading File\n");
        exit(0);
    }

    char buffer[1024];
int i = 0, j = 0;
char *record, *line;

while ((line = fgets(buffer, sizeof(buffer), myFile)) != NULL)
{
    j = 0;
    record = strtok(line, " ,");
    while (record != NULL)
    {
        // printf("%d \t %d \t %d \n", ( cols * i ) + j , i , j );
        aba[i][j] = atoi(record);
        record = strtok(NULL, " ,");
        j++;
    }
    i++;
}

fclose(myFile);

int input[cols];

for (int i = 0; i < cols; i++)
{

    int x;
    printf("enter input %d\n", i);
    scanf("%f\n", &input[i]);
if (input[i] == 1001001)
{
    for (x = 0; x < row; x++)
    {
        aba[x][i] = 1001001;
    }
}

for (i = 0; i < cols; i++)
{
    printf("%d", input[i]);
}

clock_t start; start = clock();

for (a = 0; a < 199; a++)
{
    for (b = 0; b <5000; b++)
    {
        c = a * 5000 + b;
        for (d = 0; d <cols; d++)
        {
            aa[b][d] = aba[c][d];
            //printf(" %d\n", aa[b][d]);
        }
        //scanf("%d", &k);
    }
}

cudaError_t cudaStatus;
cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaDeviceReset failed !");
    return 1;
}

cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaSetDevice failed ! Do you have a CUDA-compatible GPU installed ?");
    goto Error;
}
else printf(" Working \n");

cudaStatus = cudaMalloc((void **)&d_a, rows*cols * sizeof(int));
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMalloc failed !"); goto Error;
}
else
{
    printf(" Success ! ! ! \n");
}

else
{
    printf(" Success ! ! ! \n");
}

cudaStatus = cudaMalloc((void **)&d_i, cols * sizeof(int));
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMalloc failed!"); goto Error;
}

else

    printf(" Success ! ! ! \n");

    cudaStatus = cudaMemcpy(d_i, input, cols * sizeof(int *), cudaMemcpyHostToDevice);

    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    else printf(" Success ! ! ! \n");

    cudaStatus = cudaMemcpy(d_a, aa, rows*cols * sizeof(int *), cudaMemcpyHostToDevice);

    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    else printf(" Success ! ! ! \n");
else printf(" Success ! ! ! \n");

cudaStatus = cudaMemcpy((void **)&d_c, rows* rows * sizeof(double));
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMalloc failed !");
    goto Error;
}
else printf(" Success ! ! ! \n");

double *sort = 0;

cudaStatus = cudaMemcpy((void **)&sort, rows* rows * sizeof(double));
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMalloc failed !");
    goto Error;
}
else
    printf(" Success ! ! ! \n");

int threads = 128;
while (rows%threads != 0) threads++;
printf("TH: %d \n", threads);
//return 0;
dim3 threadsPerBlock(threads);
dim3 numBlocks(rows / threadsPerBlock.x);

distance << <numBlocks, threadsPerBlock >> > (d_a, d_c, d_i);
sorting << <numBlocks, threadsPerBlock >> > (d_c, sort, k);

cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "addKern launch failed : %s\n",
            cudaGetErrorString(cudaStatus));
    goto Error;
}

cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching addKernel !\n", cudaStatus);
    scanf("%d", &k);
    goto Error;
}

//return cudaStatus ;
cudaStatus = cudaMemcpy(cc, d_c, rows*rows * sizeof(double),
cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "addKernel launch failed : %s\n",
            cudaGetErrorString(cudaStatus));
    goto Error;
}
printf("GPU Time Taken: %f\n", (double)(clock() - start) / CLK_TCK);
scanf("%d", &k);

for (int l = 0; l <= k; l++)
{
    for (i = 0; i < rows; i++)
    {
        for (int j = 0; j < rows; j++)
        {
            printf("%f\t", cc[(rows * i) + j]);
        }
    }
}

Error:

printf("Exiting . . \n");
scanf("%d", &k);
cudaFree(d_c);
cudaFree(d_a);
cudaFree(d_i);
## Results for GPU

<table>
<thead>
<tr>
<th>k=5</th>
<th>Rows</th>
<th>D=10</th>
<th>D=32</th>
<th>D=64</th>
<th>D=128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5000</td>
<td>0.645</td>
<td>0.99</td>
<td>1.96</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>6000</td>
<td>0.752</td>
<td>1.97</td>
<td>2.68</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>7500</td>
<td>0.946</td>
<td>1.977</td>
<td>3.24</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>1.246</td>
<td>1.983</td>
<td>3.91</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>12000</td>
<td>1.485</td>
<td>2.384</td>
<td>4.42</td>
<td>5.8</td>
</tr>
<tr>
<td></td>
<td>15000</td>
<td>1.865</td>
<td>2.886</td>
<td>5.85</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>2.482</td>
<td>3.94</td>
<td>7.84</td>
<td>8.6</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>3.719</td>
<td>5.98</td>
<td>11.7</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>6.19</td>
<td>9.84</td>
<td>19.5</td>
<td>21.6</td>
</tr>
<tr>
<td></td>
<td>60000</td>
<td>7.433</td>
<td>11.8</td>
<td>23.4</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>75000</td>
<td>9.289</td>
<td>14.72</td>
<td>29.3</td>
<td>32.5</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>12.37</td>
<td>19.66</td>
<td>39</td>
<td>43.3</td>
</tr>
<tr>
<td></td>
<td>150000</td>
<td>18.52</td>
<td>29.47</td>
<td>58</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>200000</td>
<td>24.7</td>
<td>39.3</td>
<td>77</td>
<td>84</td>
</tr>
</tbody>
</table>