Master Thesis
Electrical Engineering
September 2015

Performance Analysis of kNN on large datasets using CUDA & Pthreads

comparing between CPU & GPU

Sriram Kankatala

School of Electrical Engineering
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Sriram Kankatala
E-mail(s):
kasr13@student.bth.se

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

School of Electrical Engineering	Internet	:	www.bth.se/ing
Blekinge Institute of Technology	Phone	:	+46 455 38 50 00
SE–371 79 Karlskrona, Sweden	Fax	:	+46 455 38 50 57

Abstract

Several organizations have large databases which are growing at a rapid rate day by day, which need to be regularly maintained. Content based searches are similar searched based on certain features that are obtained from various multi media data. For various applications like multimedia content retrieval, data mining, pattern recognition, etc., performing the nearest neighbor search is a challenging task in multidimensional data. The important factors in nearest neighbor search (kNN) are searching speed and accuracy.

Implementation of kNN on GPU is an ongoing research from last few years, focusing on improving the performance of kNN. By considering these aspects, our research has been started and found a gap in this research area. This master thesis shows effective and efficient parallelism on multi-core of CPU and GPU to compare the performance with single core CPU.

This paper shows an experimental implementation of kNN on single core CPU, Mutli-core CPU and GPU using C, Pthreads and CUDA respectively. We considered different levels of inputs (size, dimensions) to evaluate the performance. The experiment shows the GPU outperforms for kNN when compared to CPU single core with a factor of approximately 5.8 to 16 and CPU multi-core with a factor of approximately 1.2 to 3 for different levels of inputs.

Keywords: GPU, Multicore CPU, Parallel computing, Performance, Single core CPU.

Acknowledgments

Firstly, I would like to express my deep sense of thanks and gratitude to my supervisor Prof.Lars Lundberg for his valuable time and guidance to complete my thesis successfully.

I am grateful to Stefan Peterson (Institute for Creative Technologies) 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 Akash Kiran and Gautham Krishna Chavalli to their continuous support and help. I would like to take this opportunity to thank my best friends and well wishers Harish Mamidi and Sai Sree Keerthi 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.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
kNN	k Nearest Neighbors
MIMD	Multiple Instructions Multiple Data
OSPF	Open Shortest Path First
Pthreads	POSIX threads
SIMD	Single Instruction Multiple Data

Chapter 1
Introduction

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

1.1 Overview

Now-a-days manufacturers of multi-core processors are integrating CPU with GPU and implementing on the same chip set. But the performance of the multi-core processors of data streams may be different for CPU and integrated GPU. Due to their relative performance multi-core processors has become an attractive platform for high performance computing. NVIDIA released the Compute Unified Device Architecture (CUDA) [5] platform for GPU's, which is an extension of C programming language for writing kernels directly aiming the GPU.

CUDA programming involves allocating memory in the device to store input data, then copy data from CPU memory into allocated memory in GPU, execute the kernel in GPU by selecting necessary grid dimension, and then copying results back to CPU and finally free the allocated memory on GPU. To achieve better performance using CUDA requires very careful consideration of GPU architecture. CUDA works on the principle of Single Instruction Multiple Data (SIMD).

Developments in GPU give a way to parallel computing. Parallel computing is a process to speed up the process by dividing a particular task into sub-tasks. In parallel computing a large amount of data is divided into small chunks of data which are processed by each thread independently, whereas each thread executes the same instruction with its own set of data. After, all the data is executed the output from all the threads forms the final output data.

Classification is one of the important task of data mining in which an algorithm is used to find the class of unknown objects. kNN is used as a data mining algorithm for feature extraction and pattern recognition. kNN algorithm is a widely used algorithm for classification as it is simple to implement and has a feature of low error rate. kNN algorithm is proved to be practical and feasible for huge datasets.

This algorithm is also known as lazy learning and simplest one of all other machine learning algorithms. The main challenging tasks in kNN are to increase

the speed of search and to improve the accuracy of search. In multidimensional space finding nearest neighbors to a given vector is the problem.
Nearest Neighbor Search:

There are two different varieties of nearest neighbor search:[7].

1. The search for k close vectors to given vector (k-Nearest Neighbor).

2. The search for vectors within a range of distance £ (£- Nearest Neighbor).

Large number of applications like Finite Difference Time Domain (FDTD), Computational Fluid Dynamics (CFD), Magnetic Resonance Imaging (MRI), Neural Network, Support Vector Machine (SVM), Intrusion Detection, etc., have been implemented to perform parallel computing on CPU and GPU, where GPU is used as computation accelerator. Parallel computing plays a vital role to speedup the process of kNN.

As the technology has become a key element to support various infrastructure service in different organizations. Hence there is a increase in network complexity and growing emphasis on the internet, made network security a major problem to various organizations. So the aim is to prevent, network security approaches prevention, network intrusion detection. In recent years data mining plays a major role in network intrusion detection. In [23], author proposed kNN classifier as an intrusion detection model. This thesis can be implemented for intrusion detection in terms of prediction accuracy.

1.2 Aim and Objectives

The aim of thesis is to evaluate and compare the performance of CPU and GPU on data mining algorithm kNN.

- To develop a C-Program for generation of random datasets for experiment.
- To develop a sequential C-Program on CPU for kNN and to record the execution time.
- To develop a parallel C-program on CPU for kNN and to record the execution time.
- To develop CUDA program on GPU for kNN and to record the execution time.
- To compare the performance of CPU and GPU.

1.3 Research Questions

RQ1. What is the execution time of nearest neighbor search for large datasets on CPU with sequential C program?

RQ2. What is the execution time of nearest neighbor search for large datasets on CPU with parallel C program?

RQ3. What is the execution time of nearest neighbor search for large datasets on GPU using CUDA?

RQ4. Compare the performance of CPU and GPU on execution time of nearest neighbor search for large datasets?

1.4 Main Contributions

- This thesis work adds an advantage to ongoing research on parallel implementation on CPU using Pthreads and on GPU using CUDA.
- The experimental results show the performance of CPU and GPU by considering different levels of input and all possible ways of programming to find nearest neighbor by varying the number of nearest neighbors.
- The results obtained in this thesis can be used by any researcher for reference in their work.
- The results of this thesis work gives the better approach for implementing kNN algorithm.

1.5 Document Organization

The thesis report is organized as follows, which makes the reader to understand in easier way.

Chapter 2 gives an overview of the work done by various researchers on this research area. Brief explanation of kNN algorithm and introduction about Pthreads and CUDA programming followed by implementation of kNN using CUDA 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 also 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 *k*-Nearest Neighbor

kNN is one of the simplest of all 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 [6]. 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 [13].

Euclidean distance [13] (d) between two points x and y of M dimensions is given by:

$$d(x, y) = \sqrt{\sum_{i=1}^{M} (x_i - y_i)^2} \tag{2.1}$$

2.1.1 Classification

kNN classification is to classify an unknown object from known objects. Let us consider a simple example, there are plus and minus signs and the query point with red circle as shown in figure 2.1. Now we need to find the class of red circle, whether it belongs to plus or minus by calculating the distance from query point(red dot) to each and every point(plus and minus). Depending on the k value, the class of the object changes accordingly [13].

2.1.2 Regression

kNN regression is the simple implementation to calculate the average value of the k-Nearest Neighbors. Both kNN regression and kNN classification uses the same formula [13].

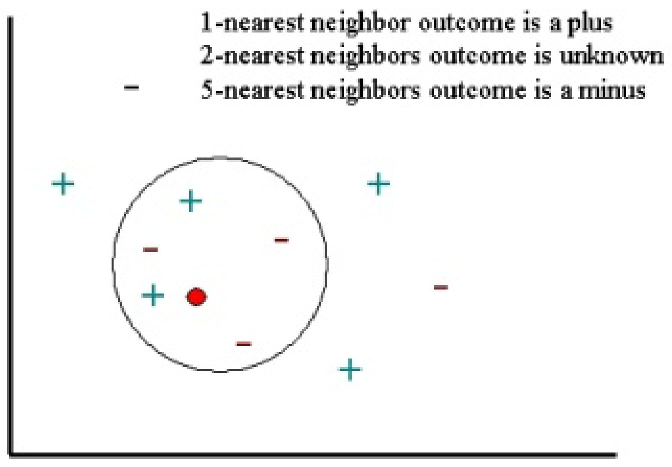

1-nearest neighbor outcome is a plus
2-nearest neighbors outcome is unknown
5-nearest neighbors outcome is a minus

Figure 2.1: kNN Classification [13]

2.2 Parallel Computing

There are many hardware and software improvements that are made in order to achieve high performance, which results in increase of clock speed, pipelined functional units, hyper threading, etc., and this leads to additional cooling system in the hardware and increases the cost of the hardware. Due to these challenges computer industries shifted to multi-core CPU's, which establishes a platform for parallel computing.

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. Parallel computing adds an speedup to an application using certain method. One of the method is by dividing the data into number of chunks and executing these chucks of data simultaneously.

2.2.2 Pthreads

In a single core CPU, computation is done using single core, where as 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 works 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

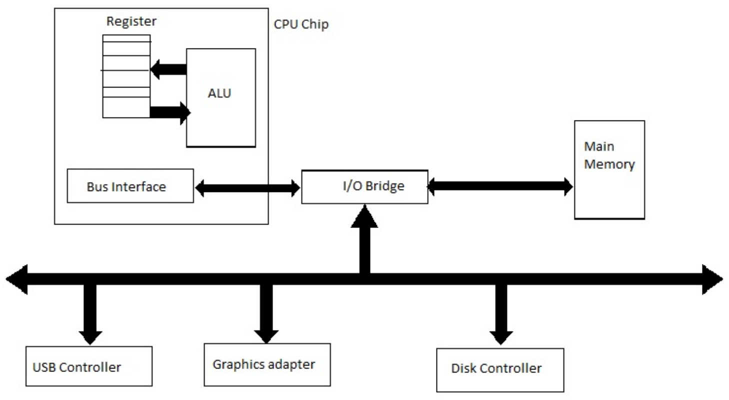


Figure 2.2: Single core CPU [14]

are run independent of each other which results in the increase of the speed of the process.

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 independently with 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 executes 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.
- Disturbing the tasks to all the available threads.
- Synchronisation of execution.

POSIX threads, popularly known as Pthreads by which the implementation of parallelism became easy on multi-core CPU's. This is an standard C-language library, specified by IEEEPOSIX 1003.1C standard [16] and can be downloaded for free from IEEE and other sites online.In thread library all the identifiers are begin with **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.

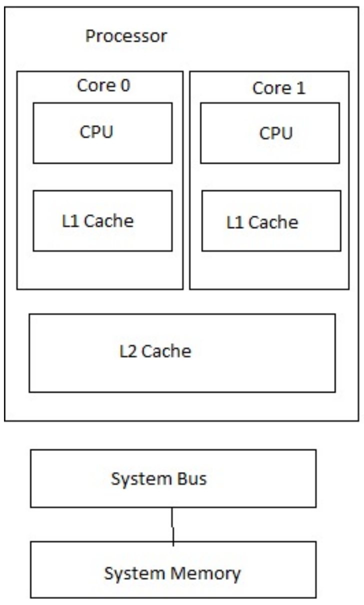


Figure 2.3: Multi-core CPU [15]

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

- pthread_create creates a new thread and this can called many number times within code.The following is syntax:

  pthread_ create(thread,attr,start_routine,arg)

- pthread_create arguments:

  thread: This is an unique identifier for the new thread returned by subroutine.

  attr: An attribute used to set the thread attributes. Thread attribute can be specified or NULL for default.

  start_routine: Thread will execute once it is created.

  arg: This is a single argument that may be passed to start_routine by reference as a pointer or NULL if no there are arguments to pass.

- The number of threads that can be launched in program is dependent but if the program that exceed the maximum number of threads may results in wrong results.
- pthread_attr_init and pthread_attr_destroy are used to initialize and destroy the thread attribute.

- pthread_exit() is used to terminate the thread that launched and it also checks whether its work is done or not.
- pthread_clsoe() routine is used to cancel the thread via another thread.
- pthread_join() is used to call the thread until specified threadid thread terminates.

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

iii. **Condition variables:** Condition variables is a routine that is used for communication between threads that share a mutex.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

In early days GPU's were mainly used for graphic applications. While developing TESLA GPU architecture NVIDIA realized its potential if programmers could think of GPU like a processor. Then NVIDIA selected a programming approach in which programmers would explicitly declare data parallel aspects of their work load. NVIDIA added memory load and store instruction with random bytes addressing capability to support the requirements of compiled C program. Programmers no longer need to use the graphics API to access the GPU parallel computing capabilities.

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 [17].

2.3.2 Overview and architecture of GPU

The reason why there is a large performance difference between many-core GPU and general purpose multi-core CPU's is because the differences in fundamental design of two processors and are shown in the figure 2.4. As shown in the 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 [17]. 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 parallezied is perfectly suitable to implement on GPU for faster outputs [22].

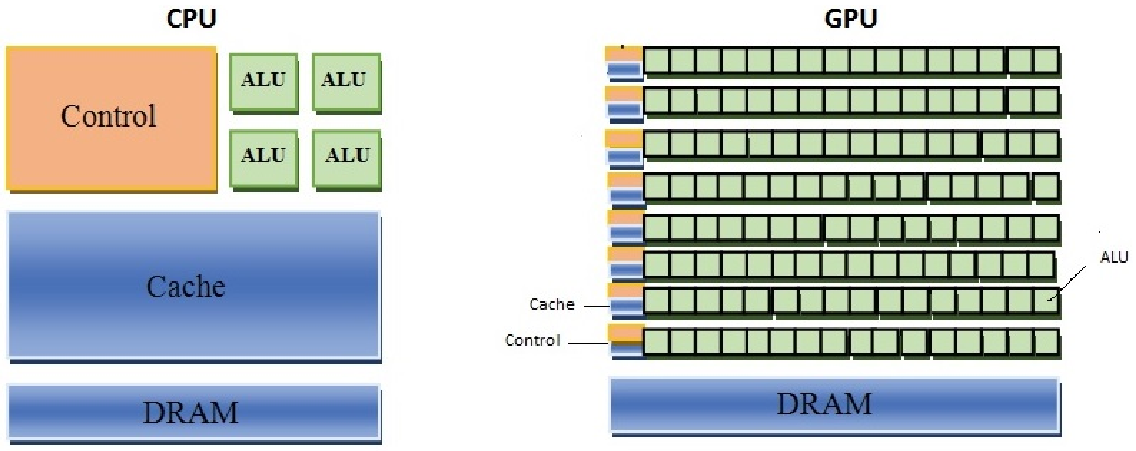


Figure 2.4: Fundamental designs of CPU and GPU [17]

The design of CPU is more efficient for sequential code performance and an other important aspect is the large cache memories are provided to reduce the instruction and data access lantenices of large complex applications.Memory bandwidth is the most important issue in CPU's because graphic chips have approximately 10 times the bandwidth of that of CPU [17].

The design of GPU is shaped by fast growing video gaming industry, which adds an 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 [17].

2.3.3 CUDA programming model

Many software applications that process a large amount of data and requires longer execution times. To develop such large data a parallel programming model is needed. CUDA is a parallel computing platform, which is used in GPU's manufactured by NVIDIA. The CUDA C consists a compiler which manages the hardware resources and provides libraries, which makes the programmer to implement parallelism in the program without any knowledge on low level hard architecture of GPU processor. This CUDA program can be easily written by any programmer who have a good knowledge in working with C programming language [17].

CUDA program structure

A CUDA program consists of different phases that are executed on either host (CPU) or a device (GPU) [17]. On host, little or no data which can not be parallelised are implemented whereas large data which can be parallelised are implemented on GPU.

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 complied by nvcc (NVIDIA C compiler) and executed on GPU. The kernel function generates a large number of threads in device to apply data parallelism [17]. The figure 2.5 shows the execution of CUDA program.

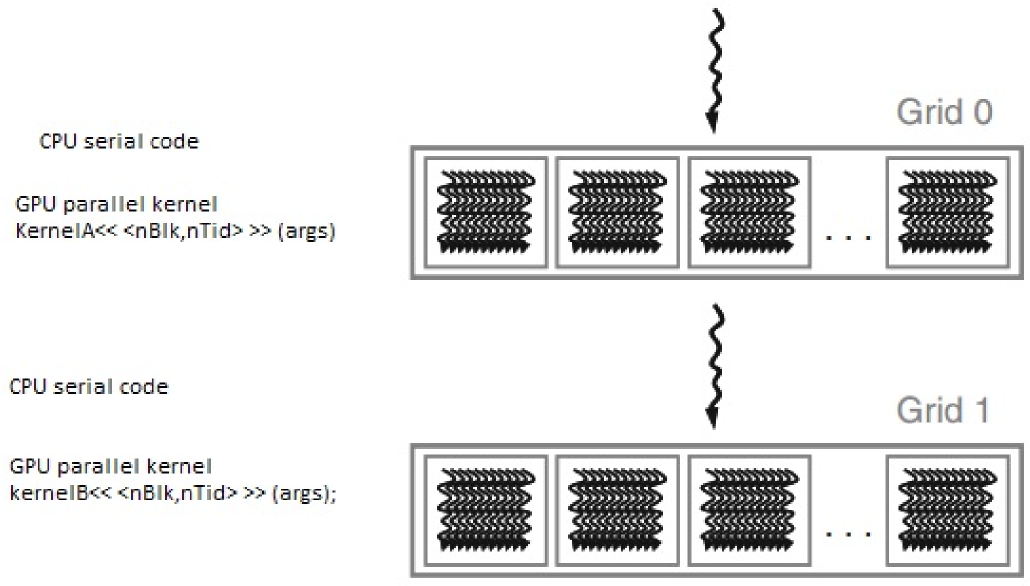


Figure 2.5: Execution of CUDA program [17]

The execution starts with host code and then when kernel is launched, the execution will 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.

CUDA threads

The GPU works on principle of single instruction multiple thread computing for CUDA programming. Each and every 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.

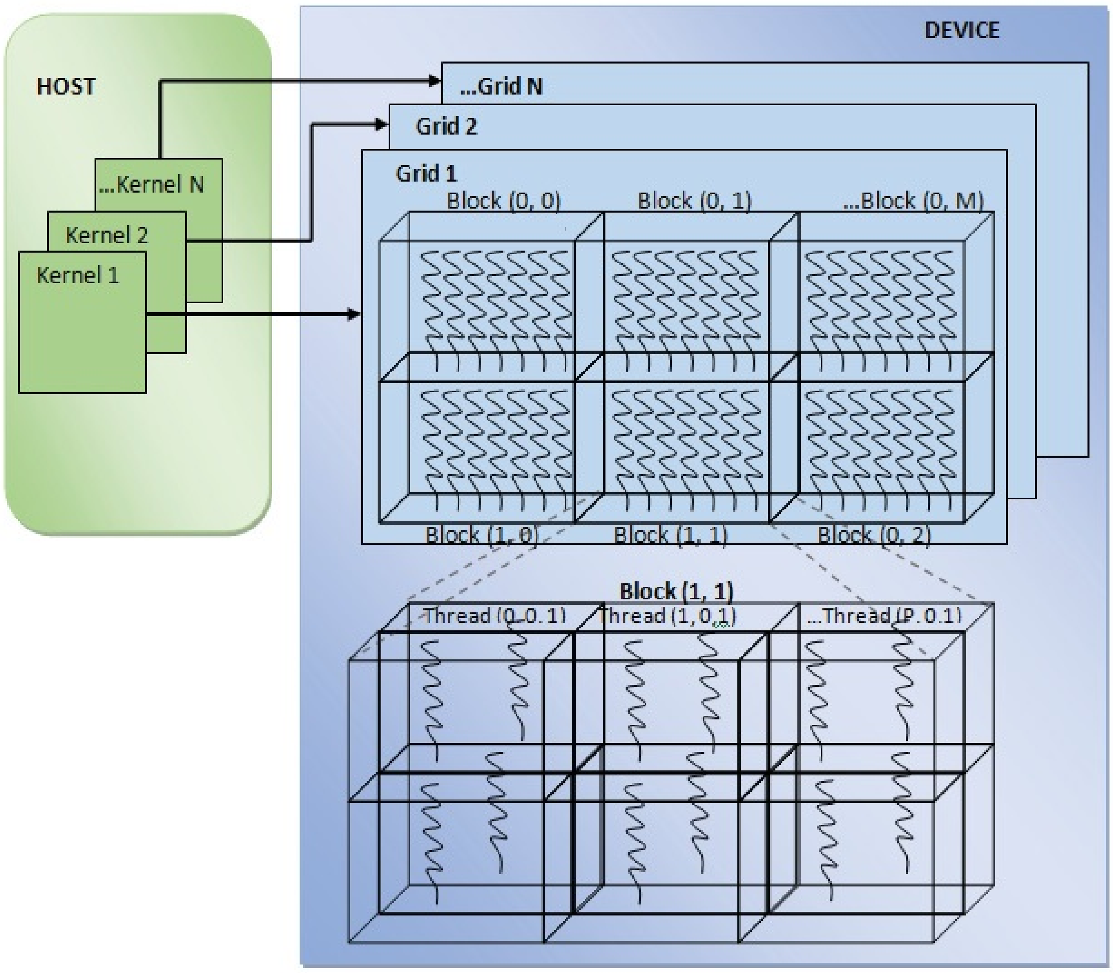


Figure 2.6: Threads Organization in CUDA [17]

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 [17] (2.2)

$$threadID = blockIdx.x * blockDim.x + threadIdx.x \tag{2.2}$$

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.

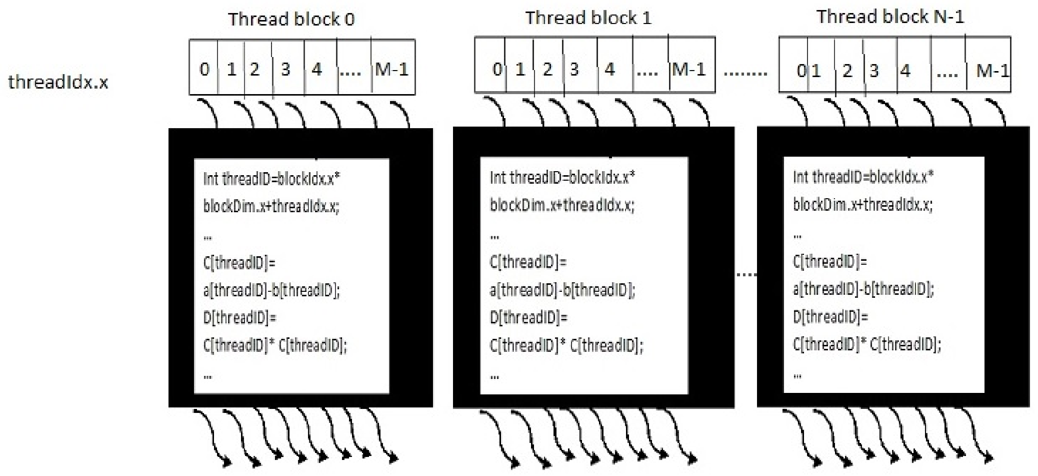

Thread block 0
Thread block 1
Thread block N-1
threadIdx.x
0 1 2 3 4 .... M-1
Int threadID=blockIdx.x* blockDim.x+threadIdx.x;
...
C[threadID]= a[threadID]-b[threadID];
D[threadID]= C[threadID]* C[threadID];
...

Figure 2.7: Overview of CUDA thread organization [17]

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. To launch a kernel in GPU one has to allocate the required memory to store the data for execution, which 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.

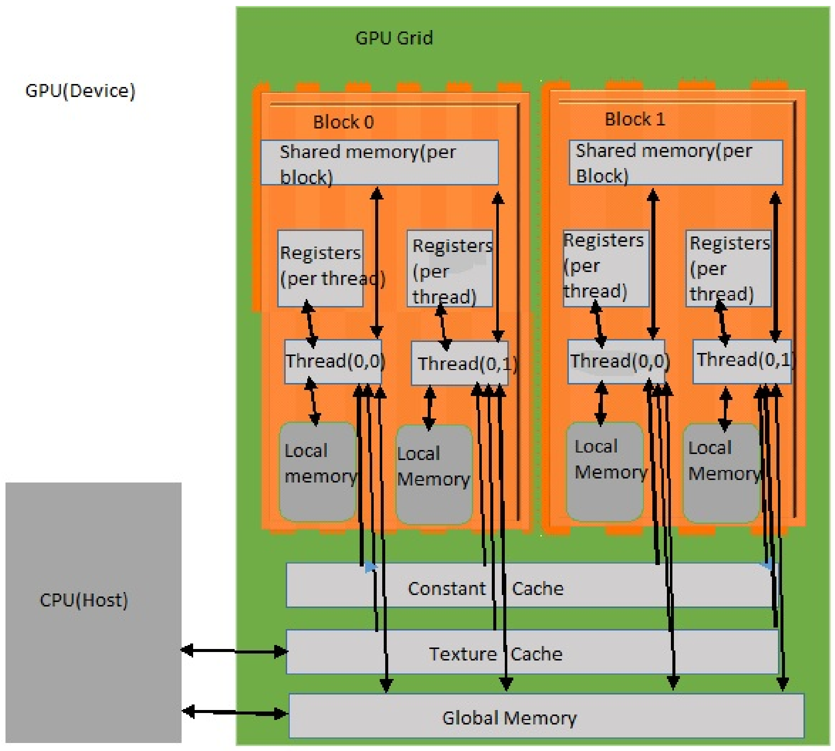


Figure 2.8: Overview of CUDA memory model [17]

CUDA program steps

The following are steps to launch a kernel in CUDA [17]:

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

2.4 Previous work

Several parallel data mining algorithms are implemented such as parallel decision tree, parallel ARM and parallel clustering. By using these algorithms and various optimization techniques the performance can be increased.

In [1] the authors, implemented a CUDA based kNN algorithm and the data segmentation method has been introduced to adapt to the CUDA thread model and memory. The authors have used adult data from UCI Machine Learning Repository as a dataset to compare the performance of CUDA based GPU with ordinary CPU and authors have also suggested that kNN method is efficient for applications with large amount of data.

In [2] the authors, implemented a CUKNN which constructs two multi-thread kernels such as distance calculation kernel and sorting kernel. By using CUKNN, 15.2X of execution speed for dataset is observed.

In [3] the authors, implemented kNN Classification using P-trees on spatial data streams. The authors proposed a new approach called closed-kNN which finds closure of the kNN set, which includes all the points on the boundary even if the size of nearest neighbor set becomes larger than k. The authors have used a new distance metric Higher Order Bit (HOB) that provides an easy and accurate way of computing closed-kNN.

In [4] the authors, proposed a fast and parallel k nearest neighbor and showed the impact on content based image retrieval applications. The proposed technique is implemented in C and MATLAB using GPU with CUDA.

In [7] the authors, proposed a new and efficient algorithm for high-dimensional nearest neighbor search based on ellipsoid distance, which uses Cholesky decomposition thereby improving the efficiency by skipping the unnecessary calculations.

In [8] the author, implemented a AES encryption of different cipher key length 128-bits, 192-bits and 256-bits on GPGPU(General Purpose Graphics Processing Unit) and CPU in order to evaluate the performance in different levels of optimization using Pthreads, CUDA and CUDA STREAMS. The author concluded that GPU outperforms both single and multi threads of CPU whereas use of CUDA STREAMS does not show impact on small data size, it shows the impact on large data size.

In [9] the authors implemented a fast SVM using CUDA on GPU and the performance is 100X when compared to CPU.

In [10] the authors implemented a CUDA based intrusion detection system where maximum throughput of 2.3 GB/s is achieved.

In [11] the authors, proposed a traffic classification scheme based on machine learning. The authors have considered different Machine Learning(ML) algorithms like TAN, L4.5, NBTree, Random form and the distance weighted kNN, which are used to reach high classification accuracy. The authors have collected the required information from the internet which are labeled by payload or by port. The authors have setup a local experiment for 100 hosts to generate simulated traffic. The hosts are running on specific applications like HTTP, SMTP, POP3, FTP and P2P which are very close to real internet traffic.

Chapter 3
Methodology

This chapter describes about 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™ i7-950 Processor
  - No of cores:4
  - No of threads: 8
  - Clock speed: 3.2 GHz
- A CUDA-enabled GPU
  - NVIDIA GeForce GTX 550 Ti
  - 192 processing cores
- Windows 7 Operating System
- A CUDA supported Microsoft Visual studio 2013
- NVIDIA CUDA toolkit

3.2 Experimental Setup

Figure 3.1 shows the flow of experiment.

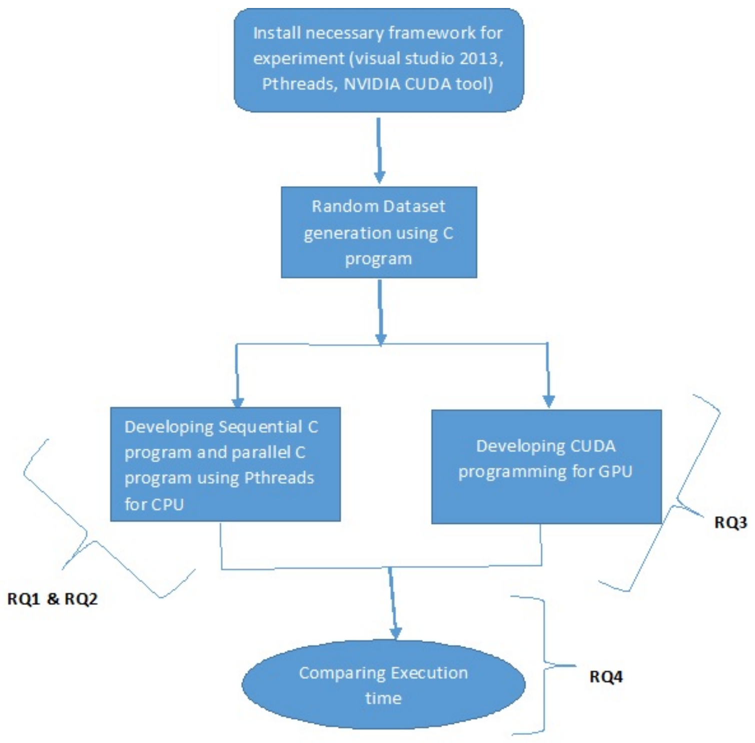

Install necessary framework for experiment (visual studio 2013, Pthreads, NVIDIA CUDA tool)
Random Dataset generation using C program
Developing Sequential C program and parallel C program using Pthreads for CPU
Developing CUDA programming for GPU
RQ1 & RQ2
RQ3
Comparing Execution time
RQ4

Figure 3.1: Experiment setup

3.3 Experiment

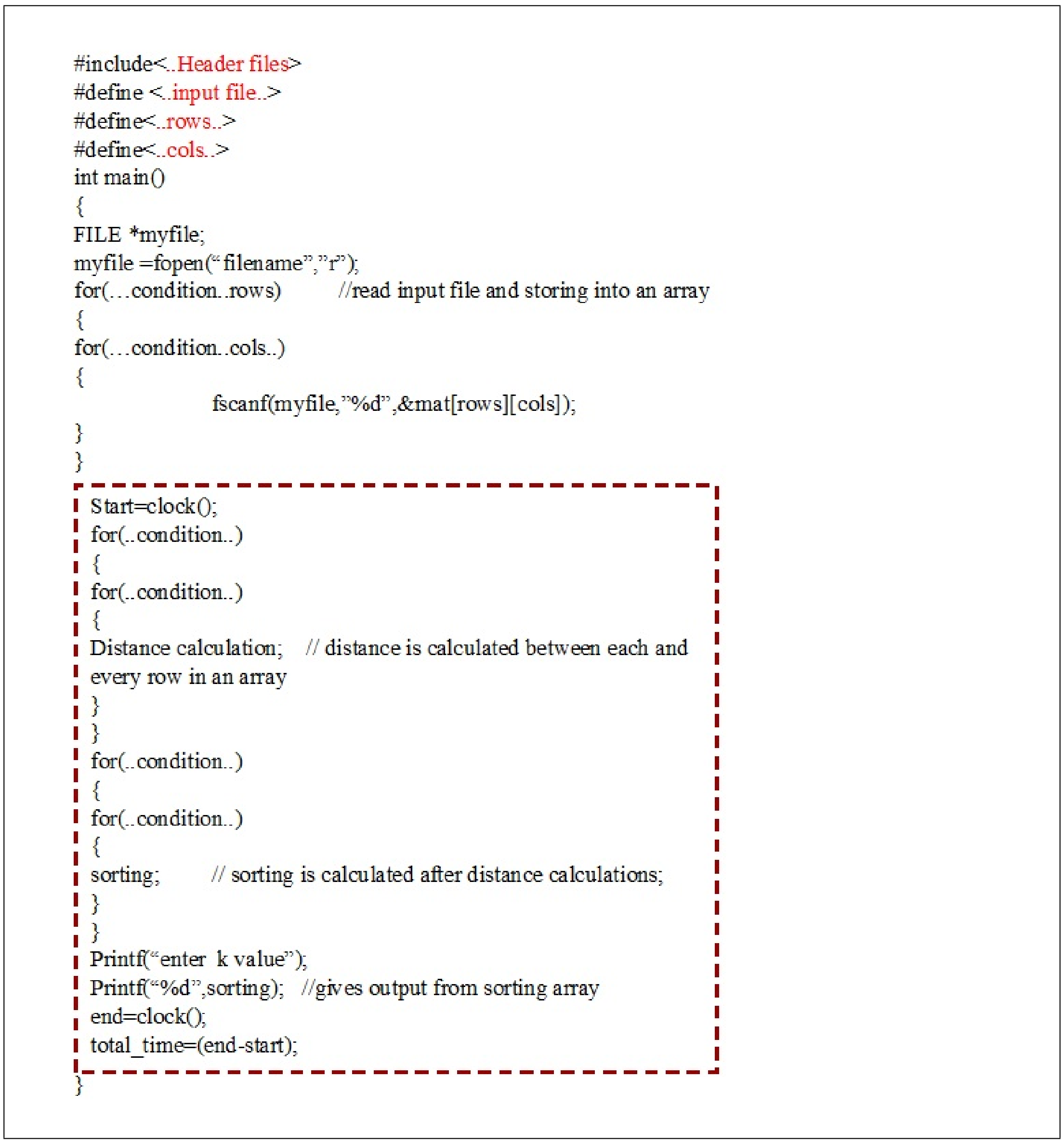
To evaluate the performance of kNN algorithm on CPU and GPU, a random dataset generated using C program is considered. The generated dataset can be varied according to the requirements, like size (no of datasets) and dimensions of dataset. For each and every implementation the execution time is calculated in order to search nearest neighbors for each input, which means if the dataset consists of N samples with M dimensions then the time taken to find the nearest neighbors for each sample is measured using Euclidean distance. The distance is calculated between each sample of data to the remaining samples. First a sequential program is implemented to find nearest neighbors for each sample of data. A parallel program is implemented on CPU using multi threads and the execution time is recorded. Also a parallel program is developed on GPU using CUDA.

In this experiment, we are going to calculate the execution time by varying the size of dataset (N), dimensions (M) and number of nearest values(k) for each implementation, which are considered from previous studies [2] [4].

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 C 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 and different values of *k*.



```
#include<..Header files>
#define <..input file..>
#define<..rows..>
#define<..cols..>
int main()
{
FILE *myfile;
myfile =fopen("filename","r");
for(….condition..rows)        //read input file and storing into an array
{
for(….condition..cols..)
{
                fscanf(myfile,"%d",&mat[rows][cols]);
}
}
Start=clock();
for(..condition..)
{
for(..condition..)
{
Distance calculation;   // distance is calculated between each and
every row in an array
}
}
for(..condition..)
{
for(..condition..)
{
sorting;        // sorting is calculated after distance calculations;
}
}
Printf("enter  k value");
Printf("%d",sorting);   //gives output from sorting array
end=clock();
total_time=(end-start);
}
```

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.

3.3.3 Implementation on GPU using CUDA

The most time taking part of kNN is distance calculations and k nearest neighbor selecting. Since GPU has many cores it is suitable for these type of applications. So this can be 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 varies with size of input and number of the threads launched per block. 64 threads are launched in each block.

Distance kernel

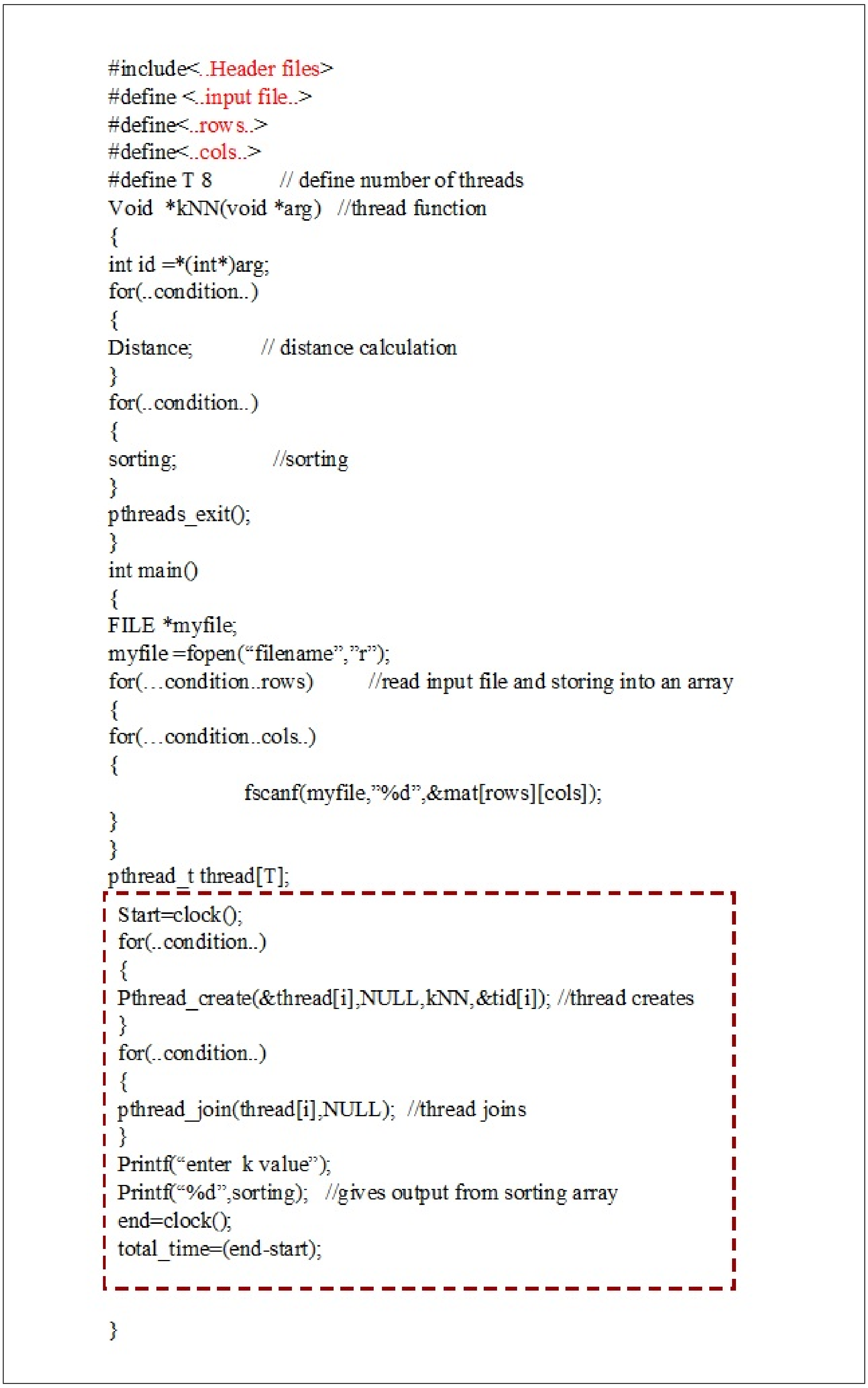
The aim of implementing this kernel is to calculate the distance between all the points which is done by different threads. Distance calculations can be fully parallelized since it is independent between pairs of objects. The data is transferred from CPU to GPU, whereas each and every thread involved 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:

```
distance << <numblocks,threadsperblock>> > (d_a,d_c);
```

Sorting kernel

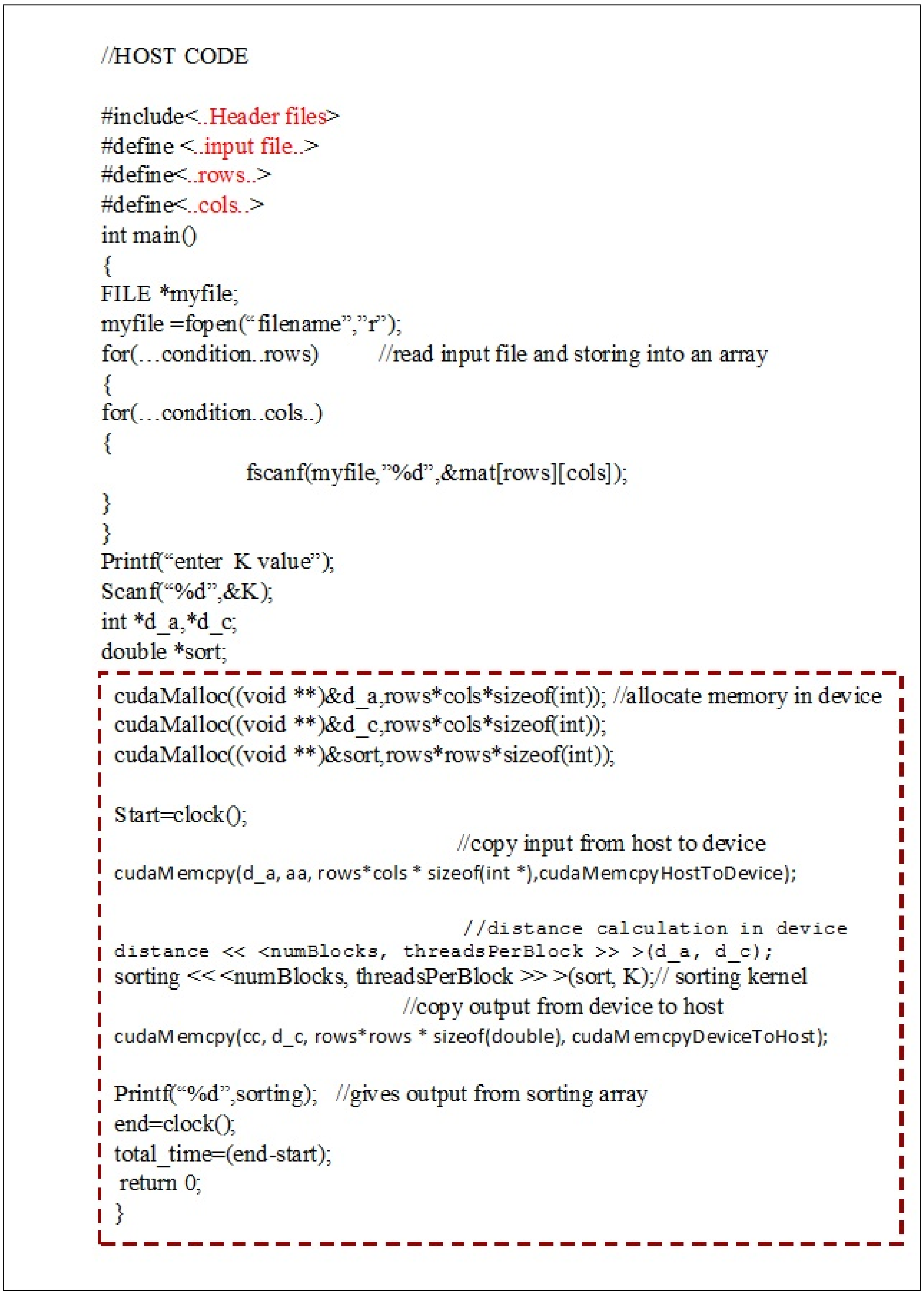
This kernel is to find the k nearest neighbors. Once the distance kernel execution is completed this kernel is invoked. The calculated distances are stored in shared memory. Each thread takes care of one distance. Now the challenging task is to find the k nearest neighbors. The distances in the shared 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:

```
sorting << <numblocks,threadsperblock>> > (d_c,k);
```



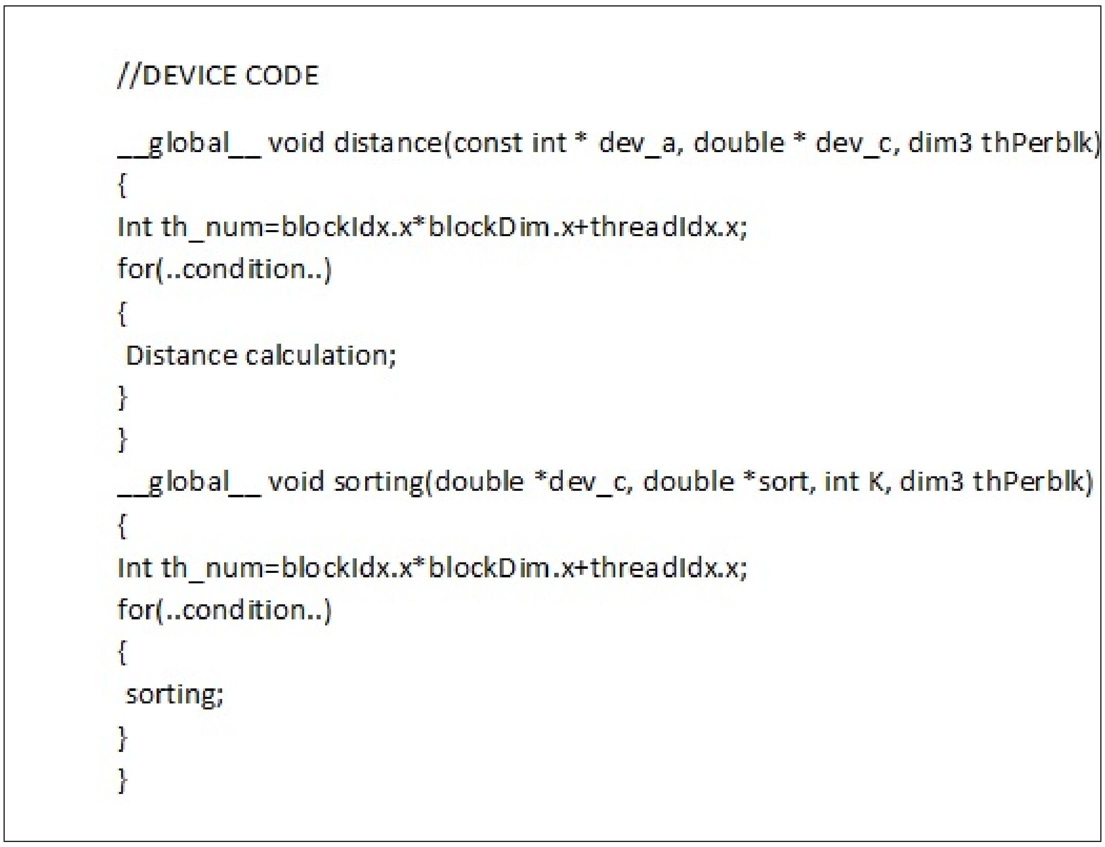
```
#include<..Header files>
#define <..input file..>
#define<..rows..>
#define<..cols..>
#define T 8          // define number of threads
Void  *kNN(void *arg)   //thread function
{
int id =*(int*)arg;
for(..condition..)
{
Distance;          // distance calculation
}
for(..condition..)
{
sorting;              //sorting
}
pthreads_exit();
}
int main()
{
FILE *myfile;
myfile =fopen("filename","r");
for(…condition..rows)         //read input file and storing into an array
{
for(…condition..cols..)
{
                fscanf(myfile,"%d",&mat[rows][cols]);
}
}
pthread_t thread[T];
 Start=clock();
 for(..condition..)
 {
 Pthread_create(&thread[i],NULL,kNN,&tid[i]); //thread creates
 }
 for(..condition..)
 {
 pthread_join(thread[i],NULL);  //thread joins
 }
 Printf("enter  k value");
 Printf("%d",sorting);   //gives output from sorting array
 end=clock();
 total_time=(end-start);

}
```

Figure 3.3: programming structure for CPU multi-core



```
//HOST CODE

#include<..Header files>
#define <..input file..>
#define<..rows..>
#define<..cols..>
int main()
{
FILE *myfile;
myfile =fopen("filename","r");
for(…condition..rows)        //read input file and storing into an array
{
for(…condition..cols..)
{
                fscanf(myfile,"%d",&mat[rows][cols]);
}
}
Printf("enter  K value");
Scanf("%d",&K);
int *d_a,*d_c;
double *sort;
 cudaMalloc((void **)&d_a,rows*cols*sizeof(int)); //allocate memory in device
 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 calculation in device
 distance << <numBlocks, threadsPerBlock >> >(d_a, d_c);
 sorting << <numBlocks, threadsPerBlock >> >(sort, K);// sorting kernel
                              //copy output from device to host
 cudaMemcpy(cc, d_c, rows*rows * sizeof(double), cudaMemcpyDeviceToHost);

 Printf("%d",sorting);   //gives output from sorting array
 end=clock();
 total_time=(end-start);
  return 0;
 }
```

Figure 3.4: programming structure for GPU(Host code)



```
//DEVICE CODE
__global__ void distance(const int * dev_a, double * dev_c, dim3 thPerblk)
{
Int th_num=blockIdx.x*blockDim.x+threadIdx.x;
for(..condition..)
{
 Distance calculation;
}
}
__global__ void sorting(double *dev_c, double *sort, int K, dim3 thPerblk)
{
Int th_num=blockIdx.x*blockDim.x+threadIdx.x;
for(..condition..)
{
 sorting;
}
}
```

Figure 3.5: programming structure for GPU(Device code)

Chapter 4

Results

The research work in this section gives the execution times in terms of performance (better approach) for kNN 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 algorithm.Various input samples are taken for different k values. 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 and table 4.2 shows the standard deviation of execution time. In appendix A, the results are shown for k=1, k=5 and k=10 for different input samples. Where n is number of inputs and d is dimension of each input.

4.2 Implementation on CPU Multi-core processor using Pthreads

Multi-core program is implemented for kNN algorithm. Various input samples taken for different k values. To get accurate results, 20 iterations of each sample

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

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
n=1200	0.2761	0.55775	1.11115	0.2774	0.55815	1.11355	0.276	0.55965	1.11435
n=2400	1.09785	2.23345	4.15715	1.09915	2.23325	4.2379	1.1047	2.2271	4.25215
n=4800	4.3835	8.7832	17.49	4.385	8.77575	17.48782	4.38435	8.781	17.48602

Table 4.2: Standard deviation of execution time on CPU single core processor

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
n=1200	0.005418	0.005739	0.006467	0.00915	0.005499	0.005596	0.005026	0.002925	0.004782
n=2400	0.004246	0.018208	0.450279	0.008067	0.023668	0.03201	0.023245	0.005739	0.045188
n=4800	0.009356	0.035585	0.020926	.008974	0.011443	0.020184	0.009965	0.028079	0.016311

Table 4.3: Avergae execution time in seconds on CPU multi-core processor using Pthreads

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
1200	0.05465	0.12025	0.22345	.05615	0.12045	0.2246	0.05635	0.1211	0.2305
2400	0.223	0.4212	0.7661	0.2246	0.4225	0.76105	0.2276	0.42275	0.76825
4800	0.7752	1.49	2.87865	0.77575	1.4873	2.87515	0.777	1.4855	2.8603

of inputs are taken. Table 4.3 shows the average execution time on CPU multi-core processors and table 4.4 shows Standard deviation of execution time. In appendix B, the results are shown for k=1,k=5 and k=10 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.

Table 4.4: Standard deviation of execution time on CPU multi-core processor using Pthreads

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
n=1200	0.006459	0.009034	0.011293	0.00488	0.009449	0.011518	0.007191	0.009593	0.011459
n=2400	0.010809	0.01102	0.019287	0.012141	0.010195	0.014515	0.011213	0.007405	0.011876
n=4800	0.014443	0.020926	0.073878	.011443	0.019437	0.042019	0.016059	0.015313	0.033655

Table 4.5: Avergae execution time in seconds on GPU using CUDA

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
1200	.02025	0.0817	0.1459	0.0305	0.08825	0.1673	0.0436	0.091775	0.19065
2400	0.11165	0.197	0.3439	0.1176	0.1957	.36725	0.1212	0.2107	0.3904
4800	0.3301	0.55495	1.03415	0.32945	0.5619	1.0395	0.343	0.56605	1.04395

Table 4.6: Standard deviation of execution time on GPU using CUDA

	k=1			k=5			k=10		
	d=32	d=64	d=128	d=32	d=64	d=128	d=32	d=64	d=128
n=1200	0.002984	0.00389	0.003919	0.003204	0.004051	0.005079	0.003068	0.002696	0.001565
n=2400	0.04221	0.005675	0.013669	0.009332	0.0049	0.0067735	0.003708	0.010687	0.004978
n=4800	0.004064	0.011803	0.003468	.007316	0.006008	0.06126	0.009695	0.005799	0.003762

4.3 Implementation on GPU using CUDA

A CUDA program is implemented for kNN algorithm. Various input samples are taken for different k values. To get accurate results, 20 iterations of each sample of inputs are taken. Table 4.5 shows the average execution time on GPU and table 4.6 shows the standard deviation of execution time. In appendix C, the results are shown for k=1,k=5 and k=10 for different input samples. Where n is number of inputs and d is dimension of each input. Here the number of blocks launched depends on the input and the threads per block are launched. These are passed as arguments in kernel function.

Chapter 5

Analysis

In this chapter, analysis of the results obtained from the experiment is presented and explicated 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 and slightly increases (in most of the cases) when the k value increases. Since the program is implemented as serial program using single core processor the execution time also increases. The following graph shows the average execution time for different input size (n), different dimensions (d) and different k values.

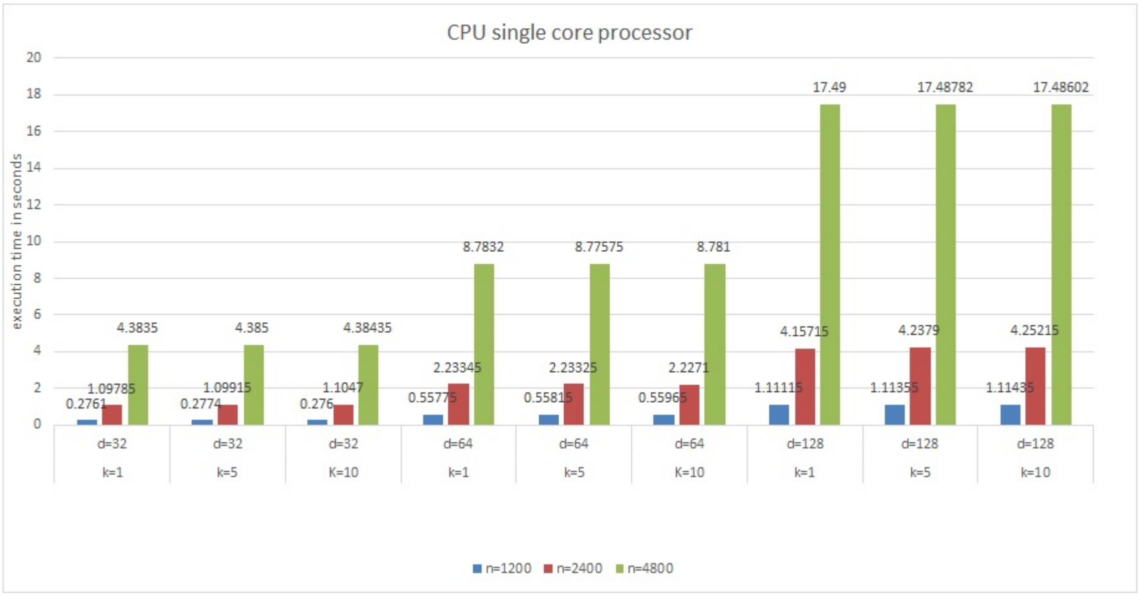


Figure 5.1: Average execution times for different input size n, dimensions d and k nearest neighbor on CPU single core processor

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 and slightly increases (in most of the cases) when the k value increases. This program is implemented using Pthreads. The CPU used in these experiments is having 4 cores which means it can launch 8 threads to improve performance when compared to single core CPU. For example if the input data of size 2400, each thread can handle 350 inputs. By comparing the results that are obtained, implementation on multi-core CPU outperforms the implementation on single core CPU. Because here the work load is shared by the threads. The following graph shows the average execution time for different input size (n), different dimensions (d) and different k values.

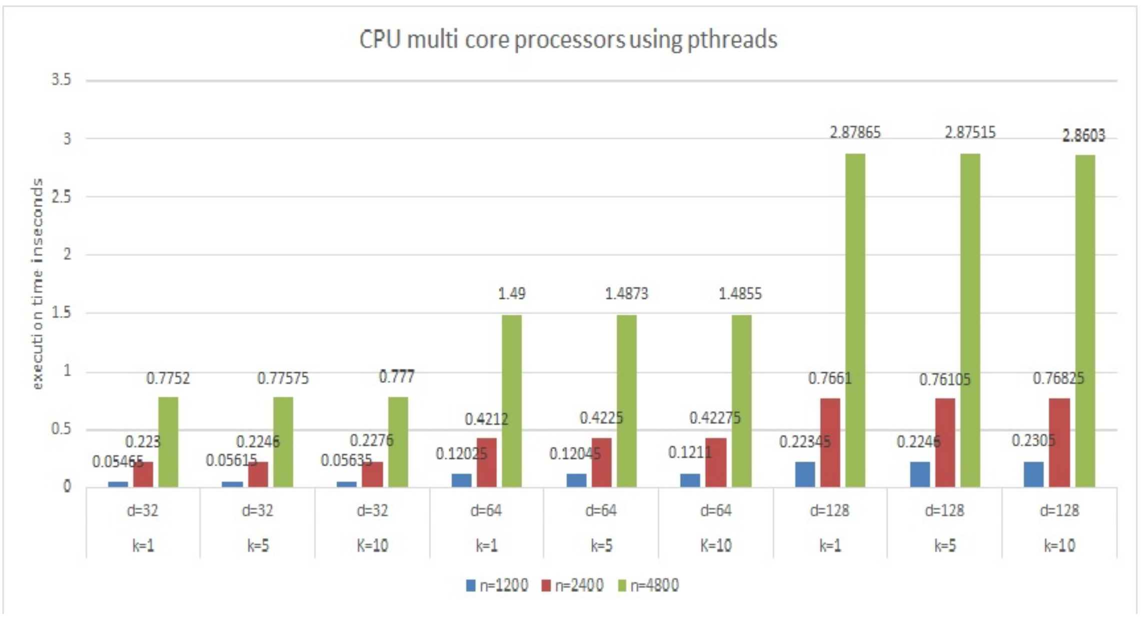


Figure 5.2: Average execution times for different input size n, dimensions d and k nearest neighbor on CPU multi-core processor

5.3 For GPU using CUDA

The results obtained for kNN implementation on GPU shows that the increase in the execution time as the size of input increases and also increases with increase in k values. 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 CPU by 5.8 to 16 times on single core CPU and 1.2 to 3 times on multi-core CPU.

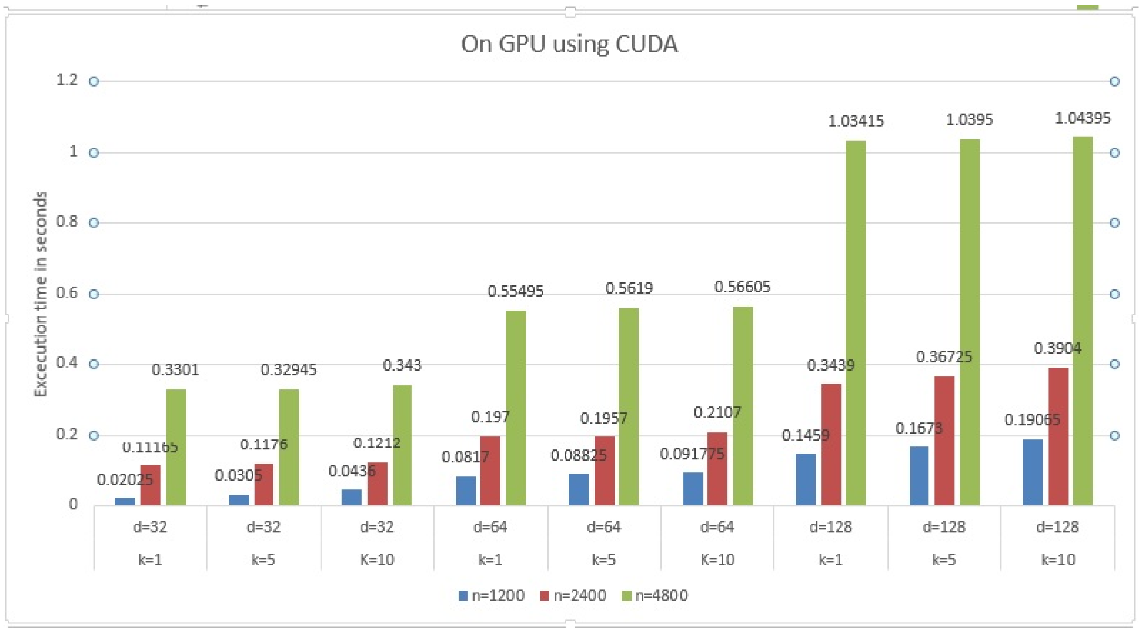


Figure 5.3: Average execution times for different input size n, dimensions d and k nearest neighbor on GPU using CUDA

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 [18]. Deviation from this validity may affects the accuracy of the results. To overcome the deviation, the following precautions should be taken:

- 20 iterations are considered in each and every 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 particular k nearest neighbors 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 outside the study [18]. This research work is carried out for parallel implementation of kNN. 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 550 Ti 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

6.1 Answer to research questions

RQ1. What is the execution time of nearest neighbor search for large datasets on CPU with sequential C program?

Ans. The execution times of one nearest neighbor search when the input size of 1200 and dimensions of 32, 64, 128 are 0.2761 sec, 0.55775 sec and 1.11115 sec respectively. And the execution times for one nearest neighbor when input size of 2400 and dimensions of 32, 64 and 128 are 1.075 sec, 2.23345 sec, 4.15715 sec respectively. Similarly the input of 4800 are 4.385 sec, 8.7832 sec, 17.49 sec. 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.

RQ2. What is the execution time of nearest neighbor search for large datasets on CPU with parallel C program?

Ans. The execution times of one nearest neighbor search when the input size of 1200 and dimensions of 32, 64, 128 are 0.05465 sec, 0.12025 sec and 0.22345 sec respectively. And the execution times for one nearest neighbor when input size of 2400 and dimensions of 32, 64 and 128 are 0.223 sec, 0.4241 sec and 0.7661 sec respectively. Similarly for the input of 4800 are 0.7752 sec, 1.49 sec, 2.87865 sec. Remaining values are tabulated in table 4.3. From all the results we can say the execution time increased with increase in size and dimension of data. These results outperforms then single core processor approximately 5 times.

RQ3. What is the execution time of nearest neighbor search for large datasets on GPU using CUDA?

Ans. The execution times of one nearest neighbor search when the input of 1200 and dimensions of 32, 64, 128 are 0.02025 sec, 0.0817 sec and 0.1459 sec respectively. And the execution times for one nearest neighbor when input size of 2400 and dimensions of 32, 64 and 128 are 0.223 sec, 0.4241 sec and 0.7661 sec respectively. Similarly for the input of 4800 are 0.03301 sec, 0.55495 sec, 1.03415 sec. 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.

RQ4. Compare the performance of CPU and GPU on execution time of nearest neighbor search for large datasets?

Ans. By comparing all the above results we can conclude that GPU outpeforms in all the condition when compared implementation on CPU single core with an factor of approximately 5.8 to 16 and implementation on CPU multi-core with an factor of approximately 1.2 to 3. And the impact of k values is very small for same number of input (n) and dimensions (d).

6.2 Future Work

- Due to limited time, this is research work is implemented on only one GPU (NVIDIA GeForce GTX 550 Ti), 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.
- This implementation can be used for intrusion detection for more accuracy and in faster way.
- OSPF is a routing protocol, which is used to find shortest path in the network [24]. Since kNN is used to find nearest neighbor, this can be implemented on OSPF to find in faster way.

References

[1] Quansheng Kuang and Leizhao, *A Practical GPU Based KNN Algorithm*, Proceedings of the Second Symposium International Computer Science and Computational Technology(ISCSCT '09), Huangshan, P.R.China, 26-28,(Dec 2009), pp.151-155.

[2] Shenshen Liang and Cheng Wang and Ying Liu and Liheng Jian , *Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU* , Web Society (SWS), 2010 IEEE 2nd Symposium on, Aug 2010. pp.53-60.

[3] Maleq Khan and Qin Ding and William Perrizo, *k-nearest Neighbor Classification on Spatial Data Streams Using P-trees*,Pacific-Asian Conf. On Knowledge Discovery and Data Mining, (2002), pp.517-528.

[4] Vincent Garcia and Eric Debreuve and Michel Barlaud, *Fast k nearest neighbor search using GPU*,2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, (2008), pp.1-6.

[5] CUDA C Programming Guide
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3IjV99bfV`

[6] KNN algorithm
`http://www.saedsayad.com/k_nearest_neighbors.htm`

[7] Tadashi Uemiya and Yoshihide Matsumoto and Daichi Koizumi and Masami Shishibori and Kenji Kita, *Fast Multidimensional Nearest Neighbor Search Algorithm Based on Ellipsoid Distance*, International Journal of Advanced Intelligence Volume 1, Number 1, (Nov,2009), pp.89-107.

[8] Akash Kiran Neelap, *Performance analysis of GPGPU and CPU on AES Encryption*, Master's Thesis(March,2014).

[9] Catanzaro Bryan and Sundaram Narayanan and Keutzer Kurt, *Fast Support Vector Machine Training and Classification on Graphics Processors*, Proceedings of the 25th International Conference on Machine Learning, 2008, pp.104-111.

[10] Vasiliadis Giorgos and Antonatos Spiros and Polychronakis Michalis and Markatos EvangelosP and Ioannidis Sotiris, *Gnort: High Performance Network Intrusion Detection Using Graphics Processors*, Recent Advances in Intrusion Detection,2008, pp.116-134.

[11] Li Jun and Zhang Shunyi and Lu Yanqing and Zhang Zailong, *Internet Traffic Classification Using Machine Learning*, Communications and Networking in China, 2007. CHINACOM '07. Second International Conference on,(Aug 2007), pp.239-243.

[12] Euclidean distance
`http://mathworld.wolfram.com/Distance.html`

[13] k-Nearest Neighbors
`http://www.statsoft.com/textbook/k-nearest-neighbors`

[14] Huong Nguyen:Computer systems
`http://cnx.org/contents/611fa6c7-a16d-460a-a221-ae57ff2379be@1`

[15] M.Domeika, *"Development and Optimization Techniques for Multi-Core Processors*,"Dr.Dobb's
`http://www.drdobbs.com/development-and-optimizationtechniques/212600040`

[16] "POSIX Threads Programming."
`https://computing.llnl.gov/tutorials/pthreads/`

[17] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, Second Edition: A Hands-on Approach, 2 edition. Amsterdam; Boston; Waltham, Mass.: Morgan Kaufmann, 2012.

[18] Steven Taylor and Gordon J.G. Asmundson, *"Ineternal and External Validity In Clinical Research"*
`http://www.sagepub.com/upm-data/19352_Chapter_3.pdf`

[19] "On Fair Comparison between CPU and GPU."
`http://www.eecs.berkeley.edu/~sangjin/2013/02/12/CPU-GPU-comparison.html`

[20] Lican Hunag and Zhilong Li*"A Novel Method of Parallel GPU Implementation of KNN Used in Text Classification"*, Networking and Distributed Computing (ICNDC), 2013 Fourth International Conference, Los Angeles, CA, 2013, pp.6-8.

[21] Liao Wei, Zhang Zhiming, Yuan Zhimin, Fu Wei and Wu Xiaoping*"Parallel Continuous k-Nearest Neighbor Computing in Location Based Spatial Networks*

on GPUs", Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on 21-23 June 2013, shiyang, pp. 271-274.

[22] Hong Zhang, Da-Fang zhang and Xia-An Bi *"Comparison and Analysis of GPGPU and Parallel Computing on Mutli-Core CPU"*,International Journal of Information and Education Technology, vol. 2, No.2, April 2012.

[23] M.Govindarajan and RM.Chandrasekaran *"Intrusion Detection Using k-Nearest Neighbor"*,Advanced Computing, 2009. ICAC 2009. First International Conference on, Chennai, pp. 13-20.

[24] OSPF Design Guide
`http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/7039-1.html`

Appendices

Appendix A
Program & Results for CPU single core processor

A.1 program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#define filename "test.csv"
#define rows 4800
#define cols 128
int main(){

        int f;
        printf("Enter the number of nearest neigbours");
        scanf("%d", &f);
        double total_time;
        clock_t start, end;
        double **b = malloc(rows*sizeof(double*));
        int mat[rows][cols];
        for (int i = 0; i < rows;i++)
        b[i] = malloc(rows* sizeof(double));
        srand(0);
        int l;
        FILE *myFile;
        myFile = fopen("test.csv", "r");

        //read file into array
        if (myFile == NULL)
        {
                printf("Error Reading File\n");
                exit(0);
        }
        for (int i = 0; i < rows; i++)
        {
                for (int j = 0; j < cols; j++)
                {
                        fscanf(myFile, "%d ", &mat[i][j]);
                }
        }
        fclose(myFile);

        start = clock();
        for (int m = 0; m < rows; m++)
        {
                for (int t = m + 1; t < rows; t++)
                {
                        double sum = 0;
                        for (int k = 0; k < cols; k++)
                        {

                                double p = mat[m][k] - mat[t][k];
                                double s = p*p;
                                sum += s;

                        }

                        b[m][t] = sqrt(sum);

                }

        }
        double temp;
for (int k = 0; k < rows; k++)
        {

                for (int i = k + 1; i<rows; i++)
                {
                        for (int j = 0; j< cols; j++)
                        {
                                if (b[k][i]<b[k][j])
                                {
                                        temp = b[k][i];
                                        b[k][i] = b[k][j];
                                        b[k][j] = temp;
                                }
                        }
                }

        }

        end = clock();

        total_time = ((double)(end - start)) / CLK_TCK;
        printf("\n\ntime taken %f \n", total_time);



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

        free(mat[i]);
        }


        free(mat);
        return 0;
}
```

A.2 Results

Number of inputs = 1200 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.286	0.28	0.28
2	0.28	0.28	0.28
3	0.27	0.28	0.28
4	0.27	0.27	0.27
5	0.281	0.27	0.28
6	0.28	0.271	0.27
7	0.28	0.28	0.27
8	0.28	0.27	0.27
9	0.27	0.274	0.27
10	0.278	0.27	0.28
11	0.27	0.27	0.28
12	0.274	0.283	0.28
13	0.282	0.28	0.28
14	0.27	0.28	0.27
15	0.28	0.31	0.27
16	0.28	0.27	0.27
17	0.271	0.28	0.28
18	0.27	0.28	0.28
19	0.27	0.27	0.28
20	0.28	0.28	0.28

Number of inputs = 1200 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.56	0.55	0.56
2	0.55	0.55	0.56
3	0.56	0.562	0.56
4	0.561	0.56	0.566
5	0.561	0.561	0.56
6	0.55	0.555	0.55
7	0.55	0.56	0.56
8	0.55	0.56	0.56
9	0.56	0.55	0.56
10	0.571	0.56	0.555
11	0.56	0.562	0.56
12	0.56	0.563	0.56
13	0.56	0.56	0.56
14	0.55	0.55	0.562
15	0.56	0.56	0.56
16	0.56	0.56	0.56
17	0.55	0.56	0.56
18	0.56	0.55	0.56
19	0.562	0.56	0.56
20	0.56	0.57	0.56

Number of inputs = 1200 & dimension =128

No.of.iterations	k=1	k=5	k=10
1	1.11	1.11	1.112
2	1.11	1.12	1.12
3	1.12	1.11	1.11
4	1.11	1.121	1.12
5	1.122	1.122	1.122
6	1.112	1.11	1.111
7	1.11	1.101	1.12
8	1.11	1.113	1.12
9	1.12	1.12	1.112
10	1.11	1.111	1.11
11	1.11	1.111	1.112
12	1.11	1.112	1.11
13	1.098	1.118	1.11
14	1.118	1.11	1.12
15	1.11	1.11	1.112
16	1.11	1.12	1.112
17	1.1	1.11	1.112
18	1.111	1.121	1.11
19	1.102	1.11	1.11
20	1.12	1.111	1.122

Number of inputs = 2400 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	1.102	1.09	1.101
2	1.09	1.101	1.1
3	1.1	1.1	1.1
4	1.1	1.09	1.096
5	1.1	1.112	1.2
6	1.1	1.092	1.116
7	1.1	1.1	1.096
8	1.1	1.09	1.11
9	1.102	1.1	1.1
10	1.092	1.092	1.09
11	1.1	1.102	1.091
12	1.1	1.099	1.1
13	1.096	1.09	1.1
14	1.101	1.11	1.096
15	1.1	1.12	1.1
16	1.09	1.1	1.101
17	1.091	1.092	1.101
18	1.101	1.1	1.09
19	1.1	1.103	1.104
20	1.092	1.1	1.102

Number of inputs = 2400 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	2.249	2.238	2.23
2	2.271	2.33	2.223
3	2.266	2.225	2.23
4	2.265	2.228	2.225
5	2.262	2.221	2.222
6	2.22	2.224	2.222
7	2.222	2.23	2.214
8	2.224	2.23	2.233
9	2.232	2.22	2.231
10	2.22	2.23	2.219
11	2.223	2.231	2.235
12	2.23	2.223	2.23
13	2.222	2.224	2.225
14	2.216	2.227	2.23
15	2.222	2.24	2.223
16	2.233	2.23	2.228
17	2.222	2.227	2.222
18	2.22	2.221	2.233
19	2.22	2.244	2.235
20	2.23	2.222	2.232

Number of inputs = 2400 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	4.52	4.238	4.33
2	4.233	4.33	4.223
3	4.233	4.225	4.23
4	4.22	4.228	4.225
5	4.22	4.221	4.222
6	4.23	4.224	4.222
7	4.233	4.225	4.314
8	4.232	4.33	4.233
9	4.232	4.22	4.335
10	4.249	4.23	4.219
11	4.271	4.231	4.235
12	2.266	4.223	4.23
13	4.265	4.224	4.225
14	4.262	4.225	4.33
15	4.22	4.23	4.223
16	4.242	4.23	4.228
17	4.34	4.227	4.222
18	4.232	4.231	4.33
19	4.22	4.244	4.235
20	4.223	4.222	4.232

Number of inputs = 4800 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	4.403	4.381	4.4
2	4.395	4.391	4.392
3	4.384	4.389	4.387
4	4.378	4.392	4.383
5	4.374	4.376	4.394
6	4.392	4.382	4.396
7	4.375	4.398	4.374
8	4.393	4.369	4.362
9	4.38	4.375	4.382
10	4.384	4.379	4.377
11	4.375	4.391	4.388
12	4.374	4.392	4.376
13	4.374	4.387	4.364
14	4.383	4.383	4.386
15	4.394	4.388	4.39
16	4.374	4.377	4.393
17	4.37	4.382	4.392
18	4.391	4.378	4.381
19	4.384	4.382	4.386
20	4.393	4.408	4.384

Number of inputs = 4800 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	8.762	8.762	8.762
2	8.772	8.782	8.792
3	8.792	8.772	8.762
4	8.762	8.762	8.772
5	8.772	8.772	8.782
6	8.782	8.782	8.782
7	8.882	8.792	8.792
8	8.882	8.762	8.752
9	8.792	8.782	8.792
10	8.772	8.762	8.762
11	8.752	8.771	8.782
12	8.782	8.762	8.792
13	8.762	8.792	8.762
14	8.772	8.782	8.772
15	8.782	8.792	8.772
16	8.764	8.76	8.742
17	8.782	8.79	8.782
18	8.752	8.782	8.792
19	8.772	8.772	8.882
20	8.774	8.782	8.792

Number of inputs = 4800 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	17.492	17.472	17.502
2	17.482	17.482	17.452
3	17.542	17.472	17.462
4	17.492	17.472	17.482
5	17.482	17.5172	17.492
6	17.522	17.492	17.5172
7	17.472	17.472	17.492
8	17.502	17.472	17.482
9	17.472	17.492	17.482
10	17.472	17.492	17.472
11	17.532	17.492	17.482
12	17.482	17.496	17.482
13	17.472	17.462	17.502
14	17.502	17.492	17.472
15	17.462	17.462	17.472
16	17.482	17.482	17.482
17	17.492	17.482	17.492
18	17.482	17.5172	17.492
19	17.482	17.492	17.5172
20	17.482	17.544	17.492

Appendix B

Program & Results for CPU multi core processor

B.1 program

```
#include <stdio.h>
#include <pthread.h>
#include<math.h>
#include<time.h>
#define T 8
#define N 4800
#define M 128
int A[N][M];
double C[N][N] = { { 0 } };


void *kNN(void *arg) {
        int id = *(int*)arg;
        int i, j,k;

        for (i = id; i < N; i += T)
        {
                //printf("id=%d \t",id);

                for (j = i + 1; j < N; j++)
                {
                        int sum = 0;
                        for (k = 0; k < M; k++)
                        {

                                int temp = A[i][k] - A[j][k];
                                int s = temp * temp;
```

```
                                        sum += s;
                                }

                                C[i][j] = sqrt(sum);

                        }
                }

                double temp;
                for (int k = id; k < N; k+=T)
                {

                        for (int i = k + 1; i<N; i++)
                        {
                                for (int j = 0; j< M; j++)
                                {
                                        if (C[k][i]<C[k][j])
                                        {
                                                temp = C[k][i];
                                                C[k][i] = C[k][j];
                                                C[k][j] = temp;
                                        }
                                }
                        }

                }

                pthread_exit(NULL);
}

int main()
{

        printf("Enter the number of nearest neigbours");
        scanf("%d", &f);
        srand(0);
        int l;
        FILE *myFile;
        myFile = fopen("test.csv", "r");
        double total_time;
        clock_t start, end;
        pthread_t thread[T];
        int tid[T];
int i, j;
int f;
//read file into array


if (myFile == NULL)
{
        printf("Error Reading File\n");
        exit(0);
}
for (int i = 0; i < N; i++)
{
        for (int j = 0; j < M; j++)
        {
                fscanf(myFile, "%d ", &A[i][j]);
        }
}
fclose(myFile);

start = clock();
//time recording starts

for (i = 0; i < T; i++) { //launching threads
        tid[i] = i;
        pthread_create(&thread[i], NULL, kNN, &tid[i]);
}

for (i = 0; i < T; i++)
        pthread_join(thread[i], NULL);


end = clock();//time recording stops

total_time = ((double)(end - start)) / CLK_TCK;



printf("\n\ntime taken %f \n", total_time);
printf("\n");
```

```
        return 0;
}
```

B.2 Results

Number of inputs = 1200 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.05	0.06	0.05
2	0.06	0.06	0.05
3	0.056	0.06	0.05
4	0.05	0.05	0.05
5	0.06	0.05	0.06
6	0.05	0.05	0.06
7	0.06	0.06	0.07
8	0.05	0.05	0.06
9	0.05	0.06	0.05
10	0.067	0.06	0.05
11	0.05	0.06	0.05
12	0.05	0.05	0.05
13	0.05	0.06	0.06
14	0.05	0.06	0.06
15	0.07	0.05	0.05
16	0.05	0.06	0.05
17	0.05	0.053	0.06
18	0.06	0.06	0.067
19	0.05	0.06	0.06
20	0.06	0.05	0.07

Number of inputs = 1200 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.133	0.112	0.11
2	0.13	0.133	0.12
3	0.11	0.11	0.13
4	0.12	0.13	0.1
5	0.121	0.11	0.11
6	0.11	0.11	0.12
7	0.12	0.13	0.11
8	0.1	0.13	0.13
9	0.12	0.122	0.12
10	0.12	0.11	0.12
11	0.11	0.133	0.12
12	0.13	0.11	0.112
13	0.12	0.119	0.13
14	0.13	0.12	0.12
15	0.13	0.11	0.12
16	0.13	0.13	0.12
17	0.12	0.11	0.14
18	0.12	0.13	0.13
19	0.11	0.12	0.13
20	0.121	0.13	0.13

Number of inputs = 1200 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	0.22	0.24	0.22
2	0.22	0.22	0.24
3	0.21	0.22	0.22
4	0.21	0.23	0.22
5	0.239	0.22	0.24
6	0.22	0.23	0.24
7	0.22	0.21	0.23
8	0.22	0.24	0.22
9	0.24	0.232	0.22
10	0.23	0.22	0.22
11	0.21	0.21	0.24
12	0.22	0.23	0.24
13	0.24	0.22	0.22
14	0.23	0.25	0.26
15	0.22	0.22	0.22
16	0.22	0.22	0.23
17	0.24	0.23	0.23
18	0.24	0.2	0.24
19	0.21	0.23	0.22
20	0.21	0.22	0.24

Number of inputs = 2400 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.23	0.22	0.24
2	0.23	0.24	0.21
3	0.22	0.2	0.24
4	0.21	0.23	0.24
5	0.21	0.24	0.23
6	0.24	0.23	0.24
7	0.23	0.202	0.22
8	0.21	0.22	0.22
9	0.21	0.23	0.232
10	0.23	0.23	0.24
11	0.21	0.24	0.22
12	0.24	0.23	0.21
13	0.22	0.22	0.22
14	0.21	0.22	0.24
15	0.22	0.23	0.22
16	0.22	0.24	0.21
17	0.23	0.21	0.22
18	0.22	0.22	0.23
19	0.23	0.23	0.23
20	0.24	0.21	0.24

Number of inputs = 2400 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.454	0.42	0.42
2	0.422	0.432	0.436
3	0.422	0.42	0.422
4	0.422	0.412	0.432
5	0.412	0.43	0.42
6	0.42	0.422	0.422
7	0.422	0.41	0.42
8	0.412	0.42	0.422
9	0.432	0.422	0.42
10	0.412	0.412	0.42
11	0.422	0.42	0.422
12	0.41	0.442	0.426
13	0.412	0.43	0.422
14	0.422	0.41	0.422
15	0.41	0.432	0.42
16	0.422	0.442	0.422
17	0.412	0.432	0.435
18	0.422	0.412	0.4
19	0.42	0.42	0.422
20	0.442	0.41	0.43

Number of inputs = 2400 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	0.782	0.752	0.782
2	0.752	0.752	0.772
3	0.762	0.772	0.772
4	0.762	0.752	0.762
5	0.792	0.752	0.762
6	0.752	0.762	0.762
7	0.772	0.773	0.762
8	0.762	0.742	0.772
9	0.762	0.782	0.752
10	0.762	0.772	0.762
11	0.832	0.762	0.782
12	0.752	0.732	0.78
13	0.772	0.752	0.769
14	0.742	0.782	0.8
15	0.752	0.752	0.762
16	0.752	0.782	0.752
17	0.772	0.772	0.774
18	0.764	0.762	0.752
19	0.762	0.742	0.772
20	0.762	0.772	0.762

Number of inputs = 4800 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.762	0.762	0.762
2	0.772	0.782	0.792
3	0.792	0.772	0.762
4	0.762	0.762	0.772
5	0.772	0.772	0.782
6	0.782	0.782	0.782
7	0.802	0.792	0.792
8	0.802	0.762	0.752
9	0.792	0.782	0.792
10	0.772	0.762	0.762
11	0.752	0.771	0.782
12	0.782	0.762	0.792
13	0.762	0.792	0.762
14	0.772	0.782	0.772
15	0.782	0.792	0.772
16	0.764	0.76	0.742
17	0.782	0.79	0.782
18	0.752	0.782	0.792
19	0.772	0.772	0.802
20	0.774	0.782	0.792
21	0.7752	0.77575	0.777

Number of inputs = 4800 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	1.492	1.472	1.502
2	1.482	1.482	1.452
3	1.542	1.472	1.462
4	1.492	1.472	1.482
5	1.482	1.512	1.492
6	1.522	1.492	1.512
7	1.472	1.472	1.492
8	1.502	1.472	1.482
9	1.472	1.492	1.482
10	1.472	1.492	1.472
11	1.532	1.492	1.482
12	1.482	1.496	1.482
13	1.472	1.462	1.502
14	1.502	1.492	1.472
15	1.462	1.462	1.472
16	1.482	1.482	1.482
17	1.492	1.482	1.492
18	1.482	1.512	1.492
19	1.482	1.492	1.512
20	1.482	1.544	1.492

Number of inputs = 4800 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	2.862	2.842	2.852
2	2.852	2.872	2.872
3	2.822	2.862	2.872
4	2.853	2.842	2.866
5	2.852	2.952	2.832
6	2.854	2.812	2.842
7	2.902	2.962	2.832
8	2.872	2.902	2.832
9	2.902	2.862	2.952
10	2.842	2.852	2.832
11	2.852	2.942	2.922
12	2.862	2.852	2.852
13	3.182	2.852	2.822
14	2.862	2.862	2.902
15	2.872	2.862	2.842
16	2.862	2.862	2.842
17	2.882	2.842	2.833
18	2.852	2.852	2.873
19	2.882	2.943	2.852
20	2.852	2.874	2.882

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 rows 4800
#define cols 128


int aa[rows][cols];
double cc[rows][rows];
double cpu_out[rows][rows];
int K;

__global__ void distance(const int * dev_a, double * dev_c, dim3 thPerbl
{

        int th_num = blockIdx.x*64 + 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_a[k*cols +
                dev_c[th_num] = sqrt(sum);
        }

        //printf("Sum: %d \t", sum);

}
__global__ void sorting(double *dev_c, double *sort, int K, dim3 thPerbl
{
        int temp;
        int i;
        i = blockIdx.x*64 + 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
                        aa[i][j] = atoi(record);
                        record = strtok(NULL, ",");
                        j++;
                }
                i++;
        }

        fclose(myFile);

        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 CU
                goto Error;
        }
        else
                printf("Working \n");

        clock_t start;
    start = clock();

        int *d_a = 0;
        cudaStatus = cudaMalloc((void **)&d_a, rows*cols * sizeof(int));
        if (cudaStatus != cudaSuccess) {
                fprintf(stderr, "cudaMalloc failed!");
                goto Error;
```

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

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

	double *d_c = 0;
	cudaStatus = cudaMalloc((void **)&d_c, rows* rows * sizeof(doubl
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}
	else
		printf("Success!!! \n");
	double *sort = 0;
	cudaStatus = cudaMalloc((void **)&sort, rows* rows * sizeof(doul
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}
	else
		printf("Success!!! \n");

	int threads = 64;
	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);
	sorting << <numBlocks, threadsPerBlock >> >(sort, K);

	cudaStatus = cudaGetLastError();
```

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

        cudaStatus = cudaDeviceSynchronize();
        if (cudaStatus != cudaSuccess) {
                fprintf(stderr, "cudaDeviceSynchronize returned error co
                goto Error;
        }

        //return cudaStatus;
        cudaStatus = cudaMemcpy(cc, d_c, rows*rows * sizeof(double), cud
        if (cudaStatus != cudaSuccess) {
                fprintf(stderr, "addKernel launch failed: %s\n", cudaGe
                goto Error;
        }
        for (int i = 0; i <= K; i++){

        }

        printf("GPU Time Taken: %f \n", (double)(clock() - start) / CLK_
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");
        cudaFree(d_c);
        cudaFree(d_a);


        return cudaStatus;


}
```

C.2 Results

Number of inputs = 1200 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.023	0.04	0.045
2	0.02	0.03	0.045
3	0.02	0.03	0.043
4	0.023	0.03	0.045
5	0.01	0.035	0.045
6	0.019	0.03	0.04
7	0.019	0.03	0.04
8	0.02	0.03	0.048
9	0.019	0.03	0.047
10	0.02	0.03	0.046
11	0.025	0.03	0.047
12	0.019	0.035	0.04
13	0.019	0.03	0.039
14	0.021	0.03	0.048
15	0.021	0.03	0.044
16	0.023	0.03	0.045
17	0.019	0.03	0.045
18	0.021	0.025	0.04
19	0.021	0.025	0.04
20	0.023	0.03	0.04

Number of inputs = 1200 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.08	0.093	0.1
2	0.081	0.09	0.093
3	0.08	0.082	0.09
4	0.08	0.09	0.09
5	0.08	0.09	0.09
6	0.082	0.09	0.092
7	0.08	0.09	0.0901
8	0.079	0.09	0.09
9	0.081	0.09	0.09
10	0.081	0.08	0.092
11	0.082	0.09	0.092
12	0.08	0.08	0.09
13	0.08	0.09	0.0902
14	0.078	0.09	0.09
15	0.078	0.09	0.09
16	0.083	0.09	0.092
17	0.089	0.09	0.0902
18	0.089	0.08	0.096
19	0.089	0.09	0.096
20	0.082	0.09	0.092

Number of inputs = 1200 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	0.14	0.17	0.19
2	0.145	0.16	0.189
3	0.145	0.176	0.19
4	0.15	0.17	0.19
5	0.149	0.17	0.191
6	0.15	0.16	0.189
7	0.149	0.16	0.188
8	0.145	0.17	0.189
9	0.14	0.16	0.193
10	0.15	0.16	0.19
11	0.149	0.17	0.19
12	0.14	0.17	0.19
13	0.148	0.17	0.19
14	0.14	0.17	0.193
15	0.144	0.17	0.193
16	0.14	0.17	0.19
17	0.149	0.17	0.191
18	0.149	0.16	0.193
19	0.148	0.17	0.191
20	0.148	0.17	0.193

Number of inputs = 2400 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.11	0.11	0.12
2	0.11	0.12	0.13
3	0.11	0.124	0.12
4	0.11	0.11	0.12
5	0.11	0.117	0.121
6	0.11	0.13	0.12
7	0.12	0.132	0.12
8	0.11	0.144	0.12
9	0.108	0.12	0.112
10	0.11	0.11	0.12
11	0.11	0.12	0.121
12	0.108	0.11	0.121
13	0.11	0.11	0.121
14	0.11	0.11	0.12
15	0.11	0.12	0.13
16	0.12	0.115	0.122
17	0.12	0.11	0.12
18	0.108	0.12	0.123
19	0.11	0.11	0.12
20	0.119	0.11	0.123

Number of inputs = 2400 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.2	0.193	0.21
2	0.2	0.192	0.2
3	0.2	0.2	0.22
4	0.2	0.2	0.22
5	0.198	0.2	0.202
6	0.2	0.204	0.203
7	0.198	0.204	0.2
8	0.19	0.197	0.2
9	0.19	0.195	0.2
10	0.201	0.194	0.2
11	0.19	0.201	0.22
12	0.2	0.195	0.223
13	0.191	0.195	0.223
14	0.201	0.187	0.233
15	0.201	0.197	0.2
16	0.2	0.19	0.2
17	0.21	0.19	0.21
18	0.19	0.191	0.21
19	0.19	0.199	0.22
20	0.19	0.19	0.22

Number of inputs = 2400 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	0.35	0.37	0.396
2	0.33	0.359	0.39
3	0.353	0.365	0.393
4	0.3	0.369	0.397
5	0.353	0.375	0.39
6	0.335	0.37	0.391
7	0.33	0.36	0.39
8	0.35	0.371	0.393
9	0.33	0.374	0.391
10	0.333	0.363	0.396
11	0.353	0.366	0.39
12	0.351	0.355	0.389
13	0.35	0.36	0.38
14	0.351	0.363	0.38
15	0.351	0.364	0.391
16	0.35	0.366	0.393
17	0.351	0.373	0.38
18	0.357	0.367	0.392
19	0.35	0.385	0.393

Number of inputs = 4800 & dimension =32			
No.of.iterations	k=1	k=5	k=10
1	0.33	0.333	0.35
2	0.32	0.34	0.348
3	0.33	0.35	0.33
4	0.33	0.332	0.331
5	0.321	0.33	0.34
6	0.33	0.335	0.33
7	0.33	0.32	0.351
8	0.33	0.33	0.35
9	0.34	0.33	0.34
10	0.33	0.33	0.34
11	0.332	0.324	0.35
12	0.332	0.33	0.35
13	0.33	0.33	0.33
14	0.33	0.321	0.35
15	0.33	0.33	0.35
16	0.332	0.322	0.36
17	0.332	0.33	0.35
18	0.334	0.32	0.35
19	0.329	0.332	0.33
20	0.33	0.32	0.33

Number of inputs = 4800 & dimension =64			
No.of.iterations	k=1	k=5	k=10
1	0.552	0.56	0.56
2	0.556	0.56	0.562
3	0.55	0.56	0.56
4	0.553	0.57	0.565
5	0.554	0.56	0.576
6	0.556	0.56	0.561
7	0.534	0.57	0.566
8	0.56	0.56	0.56
9	0.55	0.56	0.57
10	0.56	0.56	0.57
11	0.53	0.56	0.57
12	0.53	0.56	0.572
13	0.56	0.55	0.572
14	0.566	0.572	0.57
15	0.568	0.571	0.559
16	0.57	0.572	0.57
17	0.56	0.557	0.562
18	0.56	0.56	0.576
19	0.56	0.562	0.56
20	0.57	0.554	0.56

Number of inputs = 4800 & dimension =128			
No.of.iterations	k=1	k=5	k=10
1	1.033	1.039	1.044
2	1.04	1.041	1.04
3	1.041	1.04	1.044
4	1.033	1.032	1.044
5	1.03	1.058	1.043
6	1.032	1.044	1.04
7	1.033	1.04	1.044
8	1.03	1.042	1.042
9	1.033	1.033	1.042
10	1.037	1.03	1.052
11	1.037	1.036	1.044
12	1.033	1.041	1.042
13	1.03	1.034	1.041
14	1.03	1.045	1.05
15	1.037	1.042	1.041
16	1.033	1.044	1.04
17	1.037	1.042	1.043
18	1.037	1.035	1.051
19	1.03	1.036	1.042
20	1.037	1.036	1.05