Design and Evaluation of classification algorithm on GPU

Kinjal Shah, Prashant Chauhan, Dr. M. B. Potdar

Abstract— Recent developments in Graphics Processing Units (GPUs) have enabled inexpensive high performance computing for general-purpose applications. Due to GPU's tremendous computing capability, it has emerged as the co-processor of CPU to achieve a high overall throughput. CUDA programming model provides the programmers adequate C language like APIs to better exploit the parallel power of the GPU. K-nearest neighbor (KNN) is a widely used classification technique and has significant applications in various domains. The computational-intensive nature of KNN requires a high performance implementation. In this paper, we propose GPU based parallel implementation of KNN. We evaluate our implementation by using different size of images. Our parallel implementation gives us 1.68X speed up, while working with GPU gives us 8.413Xspeedup.

Index Terms— data mining, classification, decision tress, KNN, MPI, CUDA, CPU, OPEN MP

1 INTRODUCTION

The size of datasets are increasing tremendously as there is a large speed up in processing and communication have improved the capability of data generation and collection in areas such as scientific experimentation, business and government transaction, as well as internet. The current system can not handle current data sets because of the large size of data. Now a days new technique that automatically handles this large size of data into meaningful information is required. Data mining, process of analysing data for different perspective and summarising into useful information. Data mining is used in fields like crediting, banking, research areas, marketing, transportation, insurance, WWW, scientific simulation etc.

Classification is a data mining technique that assigns items in order to target class. Main goal of classification is to accurately predict the target class for each case of data. For example, a classification model could be used to identify loan application such as low, medium or high credit risk. The task of classification begins with the data set for which the classes are known. As the classifications are discrete so they do not imply order.

K-Nearest Neighbor (KNN) is one of the most widely used classification algorithms. For each unknown query object, KNN scans all the objects in the reference dataset. Assume that number of reference object is m and number of query object is n then total time taken for computing is O(nm) , which is computationally intensive. Therefore serial program handling large data set will take a lot of timing, hence parallelisation is needed.

- *Kinjal Shah is currently pursuing masters degree program in computer engineering in NIRMA University, India, . E-mail: 13mcen33@nirmauni.ac.in*
- *Prashant chauhan is working as project scientist at Bhaskaracharya Institute for Space Applications and Geo-informatics, Gandhinagar. Email:Prashant.mecs@gmail.com*
- *Dr. M. B. Potdar is working as project director at BISAG, Gandhinagar. Email:mbpotdar11@gmail.com*

The rest of this paper is organized as follows. Section 2 presents the background knowledge of this parallelism. Section 3 overviews the CUDA related work. In section 4, we present CUKNN. Experimental results are presented in section 5 and we summarize our work in section 6.

2 PARALLEL APPROCHES

Many scientific and computer related and large problems can be bettlerly solved using parallel programming. The memory limits faced by serial classifiers and need of classifying larger data set in shorter time, make classification algorithm to solve the task using parallel approach. The parallel formulation, must address issue of efficiency and scalability both in terms of memory requirements and parallel run time. Data mining can be executed in a highly parallel environment over multiple processors [6].

Modern programming languages are also structured so as to efficiently utilize novel architectures. There exist dedicated parallel programming paradigms for parallelizing the algorithms over multiprocessor and networked systems. OPENMP and MPI are used to achieve shared and distributed memory parallelization. CUDA is a programming language that is designed for parallel programming on NVIDIA GPU [7]. In CUDA, thread access different memories of GPU. CUDA offers a data parallel programming model.

General purpose programming can also be done on the GPU where multi cores can be used for highly parallel processing. Many data mining algorithms have been specifically designed in CUDA and they show drastic improvement in performance. Parallel programming is incomplete without discussing the most recent approach called MAP Reduce. It can process large sized data in highly parallel manner [8]. Map Reduce was introduced by Google in 2004. Map Reduce has

become the most popular framework for mining-large scale datasets in parallel as well as distributed environment. Different computing environments' require different programming paradigms depending upon the problem type. As data mining techniques are data and compute intensive both, it can be exploited better by using any one or combination of parallel programming approaches given in the table [9].

MPI	OPENMP	CUDA	Map Reduce
Framework for distributed memory parallelism.	Framework for threaded parallelism.	A parallel programming model for multiprocessing environment for GPUs.	Multi-threaded frameworks. Threads assigned for Map or reduce task.
Each has own private memory.	Shared memory.	Both shared and private memory.	Map task run on slave nodes and Reduce task work on slave nodes.
Multiple task run concurrently	Multiple threads run concurrently	Multiple light weight threads run concurrently on light weighted GPU	Library expresses two functions: Map And Reduce
Message based : send and receive	pragma omp directives	Kernel functions runs on GPUs	Based on key-value pair
On distributed network	On multicore processor	Specially designed for GPUs.	On multicore CPU, GPU, Grids and on Clouds.
Flexible and expressive.	Easier to program and debug than MPI.	Based on C- language.	Used when data size is too large.
Each processor has its own local variable.	Directives can be added.	A kernel functions has its own local variables.	Map task is highly scalable.

Table 1: Different Parallel Approach[1]

3 CUDA PARALLEL COMPUTING ARCHITECTURE

3.1 NVIDIA GPU Architecture

GPUs have a parallel architecture with massively parallel processors. The graphics pipeline is well suited to rendering process because it allows the GPU to function as a stream processor. NVIDIA's GPU with the CUDA programming model provides an adequate API for non-graphics applications. The CPU treats a CUDA device as a many-core co-processor.

At the hardware level, CUDA-enabled GPU is a set of SIMD stream multiprocessors (SMs) with 8 stream processors (SPs) each. Each SM contains a fast shared memory, which is shared by all its processors as shown in Figure. A set of local 32-bit registers is available for each SP. The SMs communicate through the global/device memory. The global memory can be read from or written to by the host, and are persistent across the kernel launches of the same application, however, it is much slower than shared memory. Shared memory is managed explicitly by the programmers. Compared to the CPU, the peak floating-point capability of the GPU is an order of magnitude higher, as well as the memory bandwidth.

3.2 CUDA programming model

At the software level, CUDA model is a collection of threads running in parallel. The unit of work issued by the host computer to the GPU is called a kernel. CUDA program is running in a data-parallel fashion. Computation is organized as a grid of thread blocks. Each SM executes one or more thread blocks concurrently. A block is a batch of SIMD-parallel threads that runs on the same SM at a given moment. For a given thread, its index determines the portion of data to be processed. Threads in a common block communicate through the shared memory.

CUDA consists of a set of C language extensions and a runtime library that provides APIs to control the GPU. Thus, CUDA programming model allows the programmers to better exploit the parallel power of the GPU for general-purpose computing.

4 K- NEAREST NEIGHBOUR ALGORITHM

4.1 K-Nearest Neighbor

KNN algorithm is a method for classifying an object based on the closest reference object. KNN is a lazy leaner. Here a query object is classified by a majority vote of its k nearest neighbors in the reference objects. Given an unknown object *p*, a KNN classifier searches the reference dataset for the *k* objects that are closest to *p,* and then, *p* is assigned to the majority class among the k-nearest neighbors. "Closeness" is usually defined as a distance metric, such as Euclidian distance, Cosine distance etc. The most time consuming part of KNN is distance calculation component and sorting component. Therefore, the work focuses on accelerating these two components:

1.) Distance Calculation Kernel

The computation of the distance can be fully parallelized since the distances between pair of tuples are independent. This property makes KNN suitable for a GPU parallel implementation. After transferring the data from CPU to GPU, each

thread performs the distance calculation between the query object and a reference object. Threads in a common block share the reference objects with others. Since the number of objects is large, a large number of threads and blocks are launched in this kernel. In distance calculation kernel, both reference data and query data are loaded from global memory into shared memory. Each Stream Processor in a block fetches data from shared memory [4].

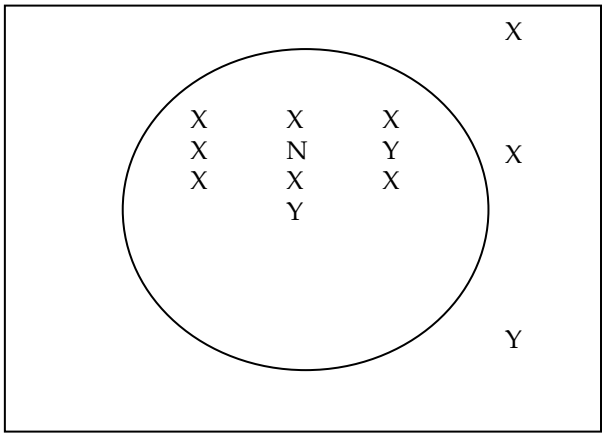


Figure 1: k-nearest neighbor. n is a new case. It would be assigned to the class x because the seven x's within the ellipse outnumber the two y's

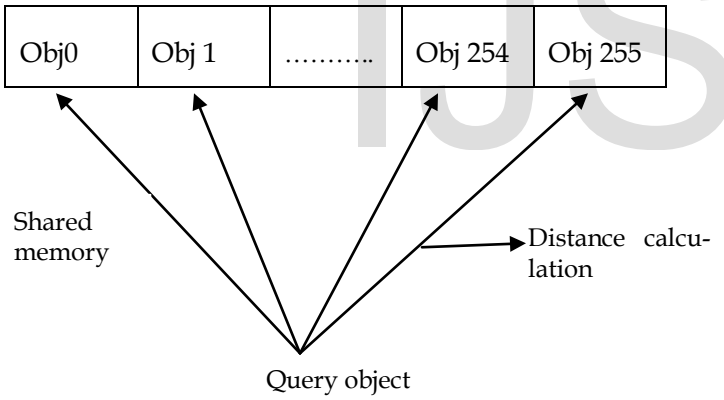


Figure 2: Parallel Approach for KNN

2.) Sorting Kernel

After calculating distance between query object, *p*, and reference object, sorting is performed to find KNN of *p*. The distances calculated by threads in common block are stored in shared memory. Threads ti takes care of one distance di. By comparing distance calculated by other threads, ti obtains the rank of di. All the threads in common block generate such task simultaneously, called *local-k* nearest neighbours of *p*. We use only one thread *t* to find the *global-k* nearest neighbors across all the blocks from the *local-k* neighbors on each block. We launch *m* blocks and each block stores *k* shortest distance in ascending order. In first iteration, thread *t* selects the global shortest one from the *m local-k* neighbors. Multiple times this step is repeated until *k* global nearest neighbors of *p* are selected. This is shown in the figure. Once the *global-k* nearest neighbors is obtained, it is easy to figure out the class label of *p*.

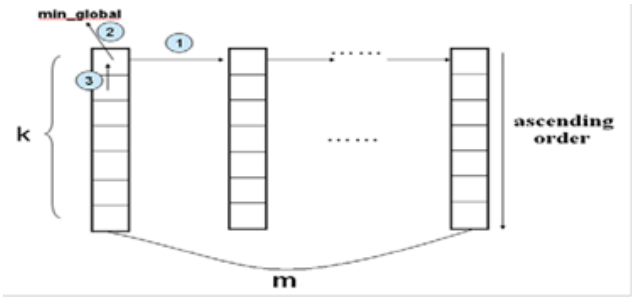


Figure 3: Global- KNN from m local –KNN [9]

4.2 Multiple Query Objects

The number of query objects to be classified is usually more than one in real application. The performance of performing classification for multiple query objects in a parallel manner is definitely superior to the sequential manner. Method is as follows:

Distance calculation kernel. In addition to a portion of reference objects, all the query objects are loaded into the shared memory of every block in the GPU kernels from the CPU. Then, the kernel is launched where each thread performs the distance calculation between the query objects and a reference object. Thus, at the end of this kernel, the distances between each query object and the reference object is calculated, respectively.

Sorting kernel. For a given query object, the sorting process is applied. Assume *q* query objects to be classified, *q* threads are used to generate the *global-k* nearest neighbors with one query per thread. In this way, the sorting process of the *q* query objects is performed in parallel. Notice that all the operations are performed in shared memory, which is highly efficient.

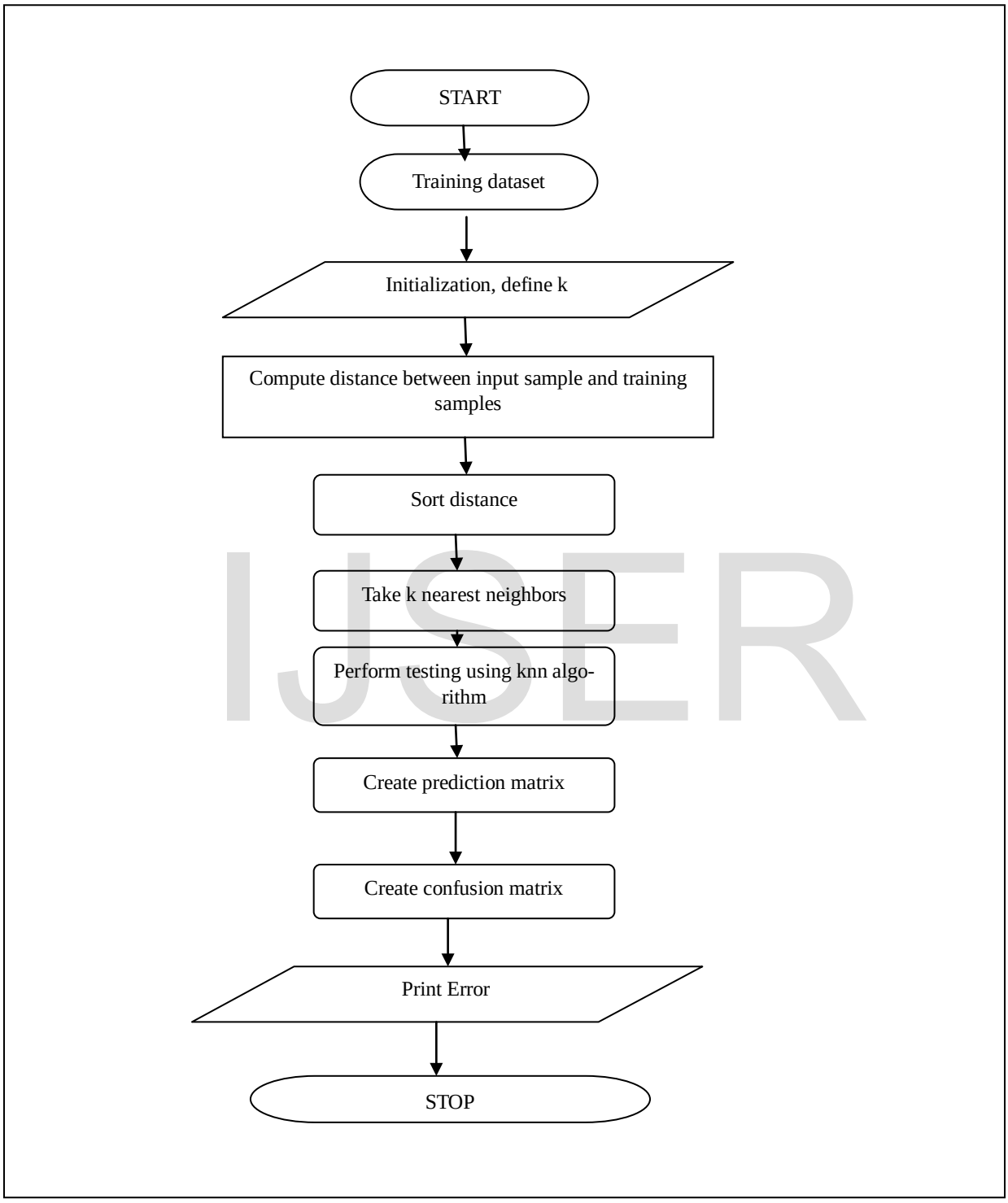
5 EXPERIMENTAL EVALUATION

The following shows the flow chart of KNN algorithm.

The device used in our experiment is NVIDIA's GeForce GTX 70 ti with 2 GB of device memory. CUDA v5.5. We implemented KNN algorithm using matlab and gpu. Initially we trained 100 images of different sizes and then tested for 2 images using KNN algorithm. The following table shows the performance comparison while working with 10, 50, 100, 150 images. This gives us 8.413Xspeed up.

The following two graphs show the comparison performance of while working with sequential algorithm to parallel algorithm and while working with GPU.

Flowchart for KNN algorithm



No. Of images	Working with sequential approach	Working with 2 parpool supported by matlab	Working with gpu
10	8. 2453s	6.50s	2.56s
50	29.51s	19.15s	3.89s
100	54.69s	32.43s	6.63s
150	87.28s	47.56s	13.20s

Table 2: Experimental time

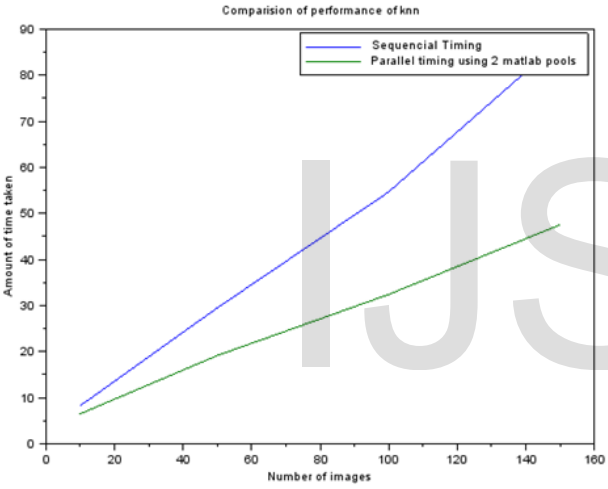


Figure 4: performance comparison with sequential timing with parallel timing

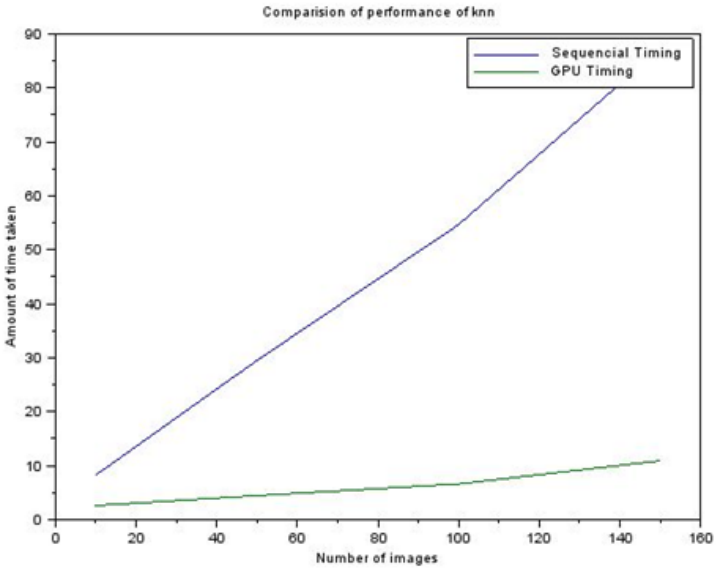


Figure 5: Performance comparison of sequential timing with GPU timing

6 Conclusion

In this paper, we presented a parallel implementation of KNN. It is hybrid implementation of CPU and GPU. We created an array on GPU which reads number of images and process on it. Experiment showed good scalability on image data. KNN presented up to 8.413X speedup. The result shows that KNN is suitable for large scale dataset.

Acknowledgment

This paper is carried out with the full support from Bhaskaracharya Institute for Space Applications and Geo-informatics and the director of institute Mr T. P. Singh. I am also thankful to all the members of the institute for supplying the precious data and resources.

References

[1] Kinjal Shah, M. B. Potdar, Prashant Chauhan, Sapan Mankad, Classification Techniques in parallel environment- A comprehensive Survey, International Journal of Computer Applications, Volume 108, issue 1

[2] Shenshen Liang; Ying Liu; ChengWang; Liheng Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," Web Society (SWS), 2010 IEEE 2nd Symposium on , vol., no., pp.53,60, 16-17 Aug. 2010 doi: 10.1109/SWS.2010.5607480

[3] Jia Tse, "Image processing with cuda," Master's Thesis, University Of Nevada, Las Vegas, August 2012.

[4] http://de.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html?nocookie=true

[5] Garcia, V.; Debreuve, E.; Nielsen, F.; Barlaud, M., "K-nearest neighbor search: Fast GPU-based implementations and application to

high-dimensional feature matching," Image Processing (ICIP), 2010 17th IEEE International Conference on , vol., no., pp.3757,3760, 26-29 Sept. 2010

[6] Wang, Lizhe, et al. "G-Hadoop: MapReduce across distributed data centers for data-intensive computing." Future Generation Computer Systems 29.3 (2013): 739-750

[7] Nickolls, John, et al. "Scalable parallel programming with CUDA." Queue 6.2 (2008): 40-53.

[8] K. Bhaduri, R. Wolf, C. Giannella, and H. Kargupta. "Distributed decision-tree induction in peer-to-peer systems." Stat. Anal. Data Min., 1(2):85–103, 2008.

[9] Yike Guo and R. Grossman, "HIGH PERFORMANCE DATA MINING Scaling Algorithms, Applications and Systems", A Special Issue of DATA MINING AND KNOWLEDGE DISCOVERY, Volume 3, No. 03(1999).