GPU Parallel Implementation of Modified K- Nearest Neighbor Classifier

Prof. Sangita . Lade[1], Prof. Priyadarshan Dhabe[2]
Department of Computer Engineering
[1,2]Vishwakarma Institute of Technology, Pune, 411037, India

Abstract*-In this paper, GPU parallel implementation of Modified K- nearest neighbor classifier (MKNN) is proposed. MKNN was serially implemented in [1]. As the dataset grows, the time required for classification increases proportionally when implemented serially. Therefore it is implemented on GPU. GPU parallel implementation of MKNN is compared with its serial counterpart and observed that GPU+CUDA implementation is 200 times faster than its serial counterpart. The speed up achieved on GPU is proportional to the number of cores and number of threads scheduled on each core.*

Keywords-Modified K- nearest neighbor classifier (MKNN), Graphics Processing Unit (GPU), Compute Unified Device Architecture(CUDA), Group Prototypes.

I. INTRODUCTION

Instance-based classifiers such as the k-Nearest Neighbor classifier (KNN) [2] operate on the premises that classification of unknown instances can be done by relating the unknown to the known according to some distance/similarity function. Implementation of k-Nearest Neighbor classifier algorithm on a Central Processing Unit (CPU) for an enormous dataset, one containing instances in lakhs, would take hours to compute. The time it would take for the dataset to train and for an instance to classify would both be large. With the advancement in technology there was a great need to improve the speed of this algorithm without having a great effect on its accuracy in classification. The previous technique of implementation of the k-Nearest Neighbor algorithm had two major downsides. Firstly, CPU being a serial processor would compute the data serially, thereby taking enormous amount of time. Secondly, to classify an unknown instance – the instance would have to be compared with each and every instance in the training dataset so as to find the closest matching instance. Number of training instances for a large dataset would be greater than a lakh and comparing with each and every instance every time an unclassified instance appears, it would be an erroneous task. Hence this method of implementation doesn't seem robust and reliable in terms of speed and time and a newer approach to this problem was necessary.

Two major solutions to the above problem would be

i. As highly parallel structure of GPUs make them more effective than general-purpose CPUs for processing of large blocks of data. So parallelizing the implementation of KNN on GPU speeds up the operations of calculating nearest neighbors.

ii. Modifying the KNN algorithm such that the number of comparisons to be performed to label an unknown instance decreases drastically, which is achieved using MKNN [1]. So the GPU implementation of Modified K-Nearest Neighbor Classifier (GMKNN) is proposed.

II. MKNN

MKNN uses group prototypes. A group prototype is a prototype of a group of patterns from the same class and falling close to each other by a user defined Euclidean distance d, where $0 < d \leq 1$. There can be multiple group prototypes from the same pattern class. Instead of using training patterns as it is to reason about the testing patterns, these group prototypes are used. This small modification can eliminate all the drawbacks of original KNN stated in [1].

Advantages of MKNN over KNN:

The MKNN algorithm has the following Advantages over KNN:

1. Knowledge is represented in a generalized way so that it can be used in great many situations, such representation is called generalized representation.[1]
2. Scope of its applicability is increased.[1]
3. The recall time per pattern for testing patterns is very high and it is proportional to the size of the data set.[1]

The rest of paper is organized as follows. Section III describes NVIDIA's CUDA-enabled parallel Computing model, section IV describes the MKNN algorithm, section V describes the GPU parallel implementation of MKNN and section VI includes experimentation and results.

III. NVIDIA'S CUDA-ENABLED PARALLEL COMPUTING

Nowadays GPUs has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth.

A. GPU architecture

GPUs are suitable to compute-intensive, highly parallel computations. NVIDIA's GPU with the CUDA programming model provides an adequate API for non-graphics applications. CPU sees 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. GeForce 8800GTX has 128 SPs and Tesla C1060 has 240 SPs. Each SM contains a fast shared memory, which is shared by all of its SPs as shown in Fig. 1. It also has a read-only constant cache and texture cache which is shared by all the SPs on the GPU. 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 or written by the host and is persistent across kernel launches by the same application. Shared memory is managed explicitly by the programmers. Compared to the CPU, more transistors on the GPU are devoted to computing, so the peak floating-point capability of the GPU is an order of magnitude higher than that of the CPU as well as the memory bandwidth due to NVIDIA's efforts on optimization. [6]

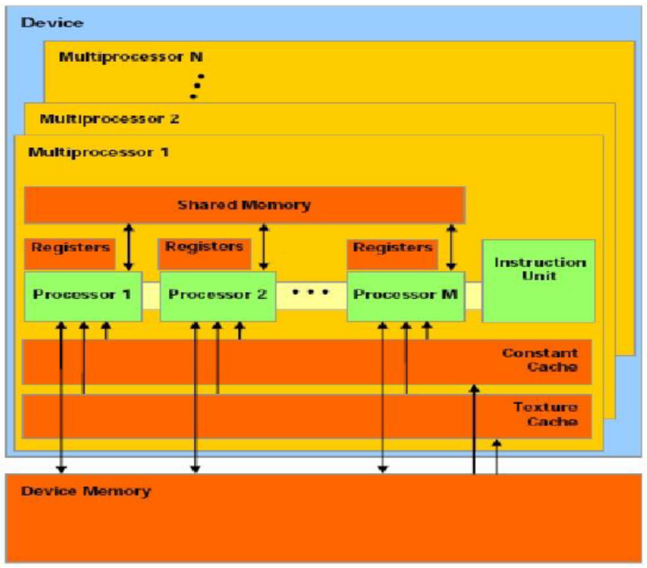
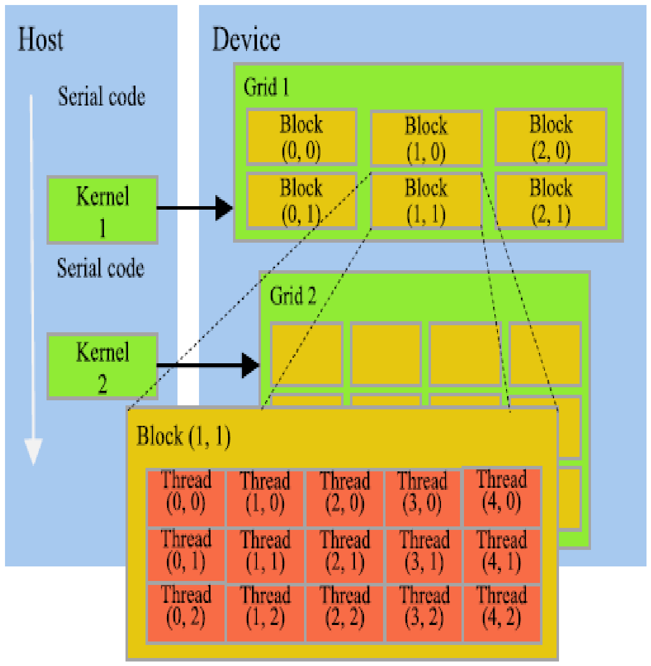


Fig1. A set of SIMD stream multiprocessors with memory hierarchy

B.The CUDA Paradigm

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel kernels, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A grid is a set of thread blocks that may be executed independently and thus may execute in parallel. When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks making up the grid. Each thread is given a unique thread ID number threadIdx within its thread block, numbered 0, 1, 2, ..., blockDim–1, and each thread block is given a unique block ID number blockIdx within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields. [4]

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel. Parallel execution and thread management are automatic. All thread creation, scheduling, and termination are handled for the programmer by the underlying system. [4]



IV. SERIAL MKNN

The original KNN is modified to overcome the drawbacks of KNN, using group prototypes. A group prototype is a prototype of a group of patterns from the same class and falling close to each other by a user defined

Euclidean distance d , where 0 < d ≤ 1 . There can be multiple group prototypes from the same pattern class. Instead of using training patterns as it is to reason about the testing patterns, these group prototypes are used. This small modification can eliminate all the drawbacks of original KNN.

A. Algorithm to calculate group prototypes

i. Let D be the set of K, n-dimensional training patterns along with their class labels and belonging to P classes.
Thus
$D = \{(x_1, c_i) \quad (x_2, c_i) \quad (x_3, c_i), ..., \quad (x_k, c_i)\}$
,
where c_i is the class label, where $i = 1,2,...., P$.

ii. Each pattern x_q is a n-dimensional normalized vector as $x_q = \{x_{q1}, x_{q2}, ..., x_{qn}\}$

iii. Initialize d , such that $0 < d \leq 1$

iv. Normalize the patterns x_q such that each component $0 < x_{q_j} \leq 1$. For $j = 1,2,.., n$ while(! all patterns groups are created)

v. Select any pattern v from the data set. For all the patterns belonging to the same class w of v do
   a. Find Euclidian distance between v and w
   b. If Euclidian distance between v and w is < = d add the pattern t w to the corresponding group of pattern v. Call this group of patterns as g.
   c. Find the group prototype for each group g.
   d. Let group g has m patterns, and then the group prototype for it can be calculated as an average of the m patterns from the group g. Remove all these patterns in g from the data set and use updated data set and go to step iii.

B. Testing Phase of MKNN

Use these group prototypes for the testing purpose instead of using actual patterns in the training set like original KNN. Find the value of for which 100% classification rate is achieved. If not then reduce the value of , which will result in creation of more number of group prototypes. Use group prototypes created such that they gives 100% classification and then go for testing the patterns present in the testing set.

V. GPU PARALLEL IMPLEMENTATION OF MKNN

Let CPU to be the master and the other N CUDA cores as slaves. Master launches the kernel for calculating the group prototypes for the training instances. Each core now computes the distance measures independently and stores the distance measures in a local array (using Euclidian distance).

The Euclidian distance is calculated using a formula

$$D = \sum_{i=o}^{n} (xi - yi)2$$

Group prototypes are calculated by each threads using distance d. Master then notes the end of processing for the sender processor and acquires the computed distance measures by copying them into its own array. After the master has claimed all distance measures from all processors, the following steps are performed:

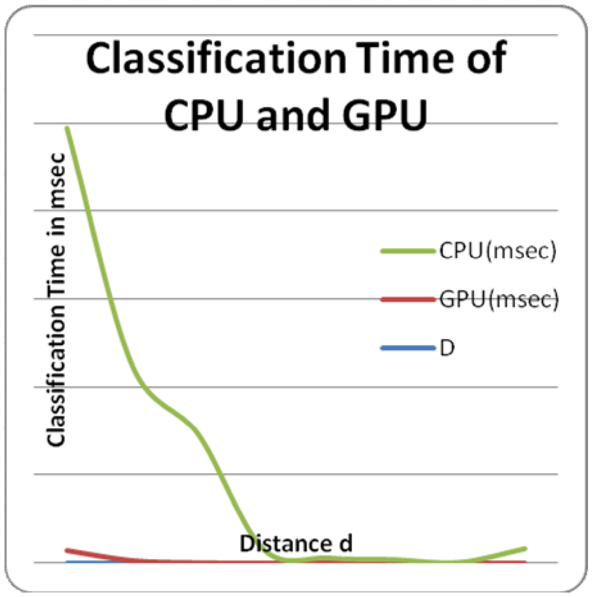
i. Sort all distance measures in ascending order.
ii. Select top k measures and count the number of classes in the top k measures.
iii. The test sample's class will assign the class having the higher count among top k measures.

Data set: The actual forest cover type for a given 30 x 30 meter cell was determined from US Forest Service (USFS) Region 2 Resource Information System data. Independent variables were then derived from data obtained from the US Geological Survey and USFS. The data is in raw form (not scaled) and contains binary columns of data for qualitative independent variables such as wilderness areas and soil type. This study area includes four wilderness areas located in the Roosevelt National Forest of northern Colorado. These areas represent forests with minimal human-caused disturbances, so that existing forest cover types are more a result of ecological processes rather than forest management practices. The training set contains both features and the Cover_Type. The test set contains only the features. The number of instances used for training is 290506 & another set of 290506 is used for testing.

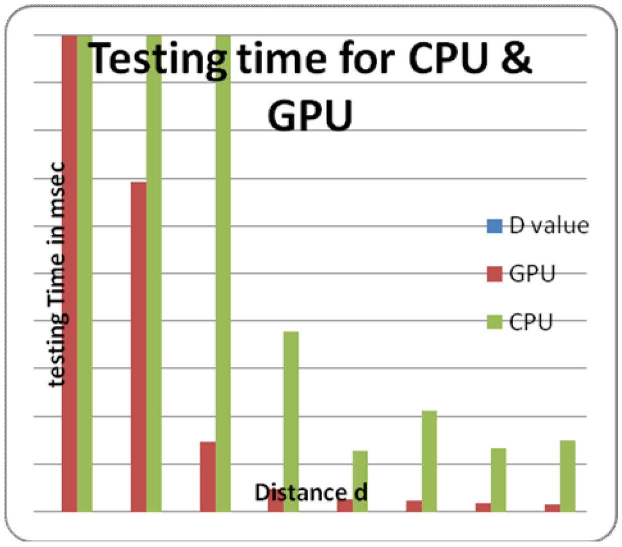
VI. EXPERIMENTATION AND RESULTS

In this section, we present the performance evaluation of MKNN on CPU and parallel implementation of MKNN on GPU.

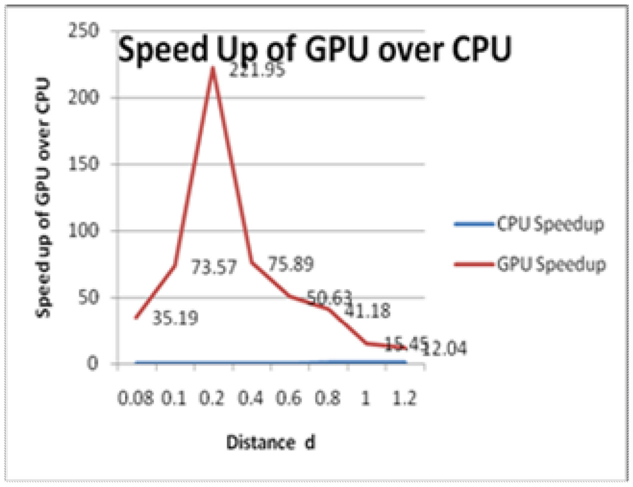
Classification Time: Gpaph 1 shows the classification time for MKNN on GPU & CPU. It is found that GPU takes less time as compared to CPU for any value of d. The classification time is % reduced overall on GPU.



Graph 1 : Classification Time of CPU v/s GPU



Graph 2: Testing Time for CPU v/s GPU



Graph 3: Speed of GPU over CPU

VI. CONCLUSION

Writing kernel for GPU is a challenging task as synchronization of all threads is to be ensured to get maximum accuracy. The GPU parallel implementation provides better results than its serial counterpart & also decreases the classification time as it works in parallel. The value of d plays very important role in classification. For 0.2 value of d for the given forest cover type dataset, 221% of speedup on GPU is achieved over CPU. For any other value of d, the overall speed up of 66%.is achieved.

REFERENCES

[1] P. S. Dhabe, S G. Lade, Snehal Pingale , Rachana Prakash and M.L. Dhore "Modified K- Nearest Neighbor Classifier Using Group Prototypes and Its Application To Fault Diagnosis.".CIIT International Journal of Data Miming and Knowledge Engineering,2010.

[2] Thomas M. Cover and Peter E. Hart, "Nearest neighbor pattern classification," IEEE Transactions on Information Theory, (1967) Vol. 13 (1) pp. 21-27

[3] Stratton, J.A., Stone, S. S., Hwu, W. W. 2008. "M-CUDA: An efficient implementation of CUDA kernels on multicores." IMPACT Technical Report 08-01, University of Illinois at Urbana-Champaign, (February)

[4] NICKOLLS, IAN BUCK, AND MICHAEL GARLAND, NVIDIA,KEVIN SKADRON, "Scalable Parallel PROGRAMMING with CUDA" UNIVERSITY OF VIRGINIA, March/April 2008 ACM QUEUE.

[5] NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA.

[6] NVIDIA.2007.CUDA Programming Guide 1.1; http:// developer . Download . nvidia.com/compute/cuda/1_1/ NVIDIA_CUDA_Programming_G uide_1.1.pdf.

[7] Liheng Jian · ChengWang Ying Liu · Shenshen Liang· Weidong Yi · Yong Shi " Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA) " Published online: 26 August 2011 © Springer Science+Business Media, LLC 2011