High Performance GPU Implementation of k-NN Based on Mahalanobis Distance

Mohsen Gavahi, Reza Mirzaei, Abolfazl Nazarbeygi, Armin Ahmadzadeh[i], Saeid Gorgin[ii]
High Performance Computing Laboratory of Institute for Research in Fundamental Sciences (IPM), Tehran, Iran
hpc@ipm.ir

Abstract— the k-nearest neighbor (k-NN) is a widely used classification technique and has significant applications in various domains. The most challenging issues in the k-nearest neighbor algorithm are high dimensional data, the reasonable accuracy of results and suitable computation time. Nowadays, using parallel processing and deploying many-core platforms like GPUs is considered as one of the popular approaches to improving these issues. In this paper, we present a novel and accurate parallel implementation of k-NN based on Mahalanobis distance metric in GPU platform. We design and implement k-NN for GPU architecture and utilize mathematic and algorithmic techniques to eliminate repetitive computations. Moreover, in addition, to taking advantage of different parallelism techniques, we improve warp management to gain maximum speed up in this implementation. Via Compute Unified Device Architecture (CUDA)-enabled GPUs, the acceleration is considerable as experimental results show the 110X speedup with respect to the single core CPU implementation. Furthermore, we measure the energy and power consumption of this algorithm for both CPU and GPU platforms, where GPU is more energy efficient regarding this application.

Keywords— k-NN algorithm; Mahalanobis distance; High throughput; CUDA; GPU.

I. INTRODUCTION

The k-nearest neighbor (k-NN) search algorithm is a problem in many research and industrial domains such as data mining, machine learning, business intelligence, scientific simulation, and bioinformatics. Nowadays, large dataset computations are common for the aforementioned applications. Also, the k-NN search algorithm is needed to handle a huge amount of data in many pragmatic approaches.

Let us consider a set $DATASET$ contains C points $DATASET$=$\{d_1, d_2, \dots, d_C\}$, where each point is described in a D-dimensional space and an input set can be defined by $INPUTSET$ in the same dimension. The k-NN search algorithm finds k nearest points to input points ($INPUTSET$) among the set $DATASET$. There are several metrics to compute the distance between two specified points, such as Euclidean, Manhattan, Mahalanobis and Kullback-Leibler. Since the results based on Mahalanobis metric is consequence of effects of all points in $DATASET$, it has high demand in various applications such as stereo matching [1], texture classification [2], object tracking [3] and gene selection [4]. Therefore, in this paper, we take advantage of this distance metric to find k-nearest neighbors. The Mahalanobis distance measures as a difference between two vectors $\vec{x}$ and $\vec{y}$ of the same distribution with the covariance matrix S. The distance can be calculated by the following equation where $\vec{x}$ and $\vec{y}$ are D-dimension input vectors and S^{-1} is $D\times D$ dimension inverse covariance matrix.

$$dist(\vec{x},\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})} \qquad (1)$$

k-NN classifier scans all the elements in the dataset points ($DATASET$) for each element of input set ($INPUTSET$). The complexity required for brute force scanning is $O(C \times N)$ where C and N are dataset size and input set size, respectively. Hence, the k-NN algorithm is computationally intensive [5] and for large datasets, the computation time on a single core machine is not reasonable. For this type of algorithm, parallel processing is considered as a conventional approach to managing the computation complexity and achieve a practical response time. Independent data computations in k-NN algorithm make it a good candidate for many core GPU platforms. In addition, extensive computations of Mahalanobis distance convince us to utilize massive parallelism and computation power of GPUs. To this end, we use methods like reforming of warp arrangement, revising reduction technique and improve management of memory hierarchy.

The rest of this paper is organized as follows: Section II is dedicated for background, where the previous works on implementation of k-NN algorithm and CUDA architecture as a dominant framework for parallel programming on graphics processing units are reviewed. We present a mathematic reformulation of Mahalanobis distance in Section III. The proposed design for parallel implementation of the k-NN algorithm based on Mahalanobis distance is presented in Section IV. The experimental results on various multi-core CPU and many-core GPU are shown in Section V. The paper is concluded in Section VI.

II. BACKGROUND

The required background is presented in this section. First, the previous works on implementation of the k-NN algorithm are revised, after that CUDA architecture, which is chosen the framework for implementation of proposed method, is concisely reviewed.

A. Related works

The simplest approach for the k-NN algorithm, named linear search, explores all points and computes distances to find nearest neighbors [6]. Since, searching a large dataset needs a huge amount of computing power, some implementation of k-

i A. Ahmadzadeh is also affiliated with Sharif University, Tehran, Iran
ii S. Gorgin is also affiliated with Iranian Research Organization for Science and Technology (IROST), Tehran, Iran.
978-1-4673-9181-8/15/$31.00 ©2015 IEEE

nearest neighbors uses approximate methods such as KD-tree [7], k-NN graph methods [8] and hashing [9] which were introduced in order to decrease complexity and computation time [6]. However, these methods do not handle high dimensional data appropriately.

One of the approximate algorithms is space partitioning [6]. This algorithm classifies the feature space into several parts and each part includes a subset of dataset points. The parts with the maximum possibility of neighbor's existence are searched to find k-nearest neighbors. This type of algorithms reduces the search space, subsequently the computation decrease. Some of the algorithms use specific data structures such as K-D-B tree [7] and quad tree [10], in order to better access dataset points.

The K-D-B tree structure combines properties of K-D tree and the binary tree. There are several ways to classify the points; for example the K-D-B tree structure uses rectangular bounding and R* tree [11] uses spheres bounding. Although spheres bounding based algorithms save 50% disk space, but for multi-dimensional data points get more volume than rectangular bounding structure. To overcome this drawback, the SR-tree structure was introduced [12]. In fact, the SR-tree structure by combining R* tree and SS-tree find nearest neighbors points for non-uniform and high dimensional data points. This structure takes advantage of a rectangular bounding structure by dividing the search space to smaller rectangular bounding. However, specifying a region by the intersection of the bounding sphere and bounding rectangle of underlying points is the highlighted feature of the SR-tree. To divide the feature space based on the data distribution, these structures create unbalanced partitions which reduce the performance of an algorithm. On the other hand, k-means clustering algorithm, like other space partitioning algorithms, needs a well-defined data structure [13]. This algorithm generally used with the Euclidian distance metric and is usually not propitious in GPUs, while global memory's latency may have the negative impact on performance.

Garcia et al. [14] introduced a fast k-NN algorithm. It is implemented in C and MATLAB using GPU with CUDA. It uses two GPU-kernels, one for computing distances and the other for sorting calculated distances in parallel. It uses comb sort and insertion sort method and compares the results for different Ks. In another work, in Ref. [15], They use CUDA's CUBLAS library with Euclidian parameter. Also, in [16] parallel implementation of the k-NN search algorithm is done with two different sorting methods: insertion sort and quick sort, while stream computing to gain a better performance is used.

B. CUDA Architecture

General-purpose computing on graphics processing units (GP-GPU) is the concept of using graphics processing unit (GPU) to execute the tasks which is usually done by CPUs. The parallel computing makes possible to achieve prominent speed-ups by computational power of the GPU. GPUs using massively parallel architecture with a large number of multiprocessors, named Stream Multiprocessors (SM). These SMs are well suited to handle huge computations.

Each SM has a fast shared memory that is shared among all the processors. For communication between SMs, global memory can be used which is much slower than shared memory, however, is more spacious for holding data.

The CPU, which is called host in this context, can read (write) from (to) the global memory which is persistently accessed by the kernel launches of the same application. Both global and shared memory can be manipulated by the programmers. The rapid increases in the performance of graphics hardware have made GPU a strong candidate for high-performance computing applications. GPUs now include fully programmable processing units that follow a stream programming model.

CUDA is NVIDIA`s C-like language appropriate for programming GPUs that focus on massively data-parallel and task-parallel kernels. GPUs use hundreds of thousands of organized threads into a grid of thread blocks. After 2006, all of NVIDIA's GPUs can provide a suitable API for non-graphics applications by CUDA programming model. The CPU treats a CUDA device as a many-core coprocessor.

NVIDIA's GPUs perform sequential threads in SIMT (Single Instruction, Multiple Thread) fashions; all cores in the same SM execute the same instruction simultaneously. Threads are grouped into blocks and each thread has a unique local index in its block. The blocks are grouped in a grid and each block has a unique local index in the grid. Each 32 threads are actually executed the same instruction at the same time, this group of thread are called warp.

III. MATHEMATIC REFORMULATION

Recently, we implemented a cost-efficient version of k-NN search algorithm using Mahalanobis distance metric on multicore CPUs [17].We have used some optimization methods such as mathematic optimization of [17] in present work to improve this algorithm for GPU implementation.

Computing Mahalanobis distance includes matrix multiplication, which needs a huge amount of addition, multiplication and memory access. In serial programming, these computations consist of two nested loops that the algorithm complexity is $O(n^2)$. These two nested loop can be decreased to one simple loop by exploiting multi-threads in GPU platform.

One of the approaches that presented in [17] is reformulating Mahalanobis distance to decrease the computation. Let $\vec{x}$ be a member of the $INPUTSET$ and $\vec{y}$ be a member of the $DATASET$. Mahalanobis distance equation can be reformulated as:

$$dist_M^2(\vec{x},\vec{y}) = (\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})$$

$$\xrightarrow{(A\pm B)^T = A^T \pm B^T} (\vec{x}^T - \vec{y}^T)S^{-1}(\vec{x}-\vec{y})$$

$$= (\vec{x}^T S^{-1} - \vec{y}^T S^{-1})(\vec{x}-\vec{y})$$

$$= (\vec{x}^T S^{-1}\vec{x} - \vec{y}^T S^{-1}\vec{x} - \vec{x}^T S^{-1}\vec{y} + \vec{y}^T S^{-1}\vec{y}) \quad (2)$$

All parts of the equation (2) are scalar values. In addition, $\vec{y}^T S^{-1}\vec{x}$ and $\vec{x}^T S^{-1}\vec{y}$ are equal, since S^{-1} is a positive semi-definite and symmetric matrix. So, the equation (2) can be rewritten as:

$$dist_M^2(\vec{x},\vec{y}) = \vec{x}^T S^{-1}\vec{x} - 2\vec{x}^T S^{-1}\vec{y} + \vec{y}^T S^{-1}\vec{y} \qquad (3)$$

In equation (3), $\vec{x}^T S^{-1}\vec{x}$ is executed N times and is not a large portion of the computations. Moreover, $\vec{y}^T S^{-1}\vec{y}$ can be computed in the first iteration and then use it for other ones. The middle expression in (3) is executed C times for each input, however, the $S^{-1}y$ portion of it can also be calculated only one time. This part is the bottleneck of the algorithm and consumes most of the computing power. Consequently, this method reduces the D-dimensional computation of last part to just a scalar pre-computation and decreases memory access about D times. Finally, we decreased $N \times C$ times computation to just N times ($C \gg N$).

Moreover, S^{-1} is a covariance matrix which is symmetric and the native implementation of matrix multiplication can be considered:

$$(xs^{-1})_i = \sum_{j=1}^{n} x_j S_{ij}^{-1}, i = 1, \dots, n$$

This equation can be rewritten with two partial sums $(xs^{-1})_i = (P_1)_i + (P_2)_i$ where:

$$(P_1)_i = 2 \times \sum_{j=i+1}^{n} x_j S_{ij}^{-1}, i = 1, \dots, n$$

$$(P_2)_i = \sum_{j=1}^{n} x_j S_{jj}^{-1}, i = 1, \dots, n$$

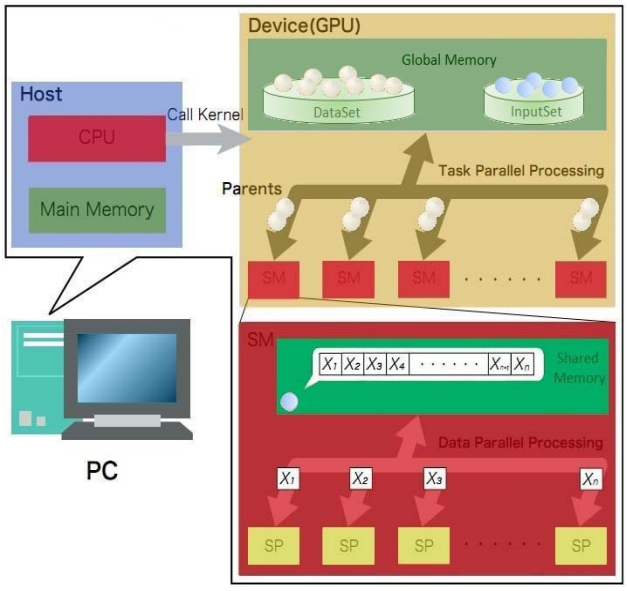


Fig. 1. Distribution of data in GPU

With using above mathematic optimization, the access memory and computation will be reduced.

IV. PROPOSED IMPLEMENTATION

In this section, we present a high throughput parallel implementation of k-NN search algorithm with Mahalanobis distance metric using CUDA-enabled GPU. In this algorithm, multiple blocks of threads are launched, where each thread computes the distance between the input set and a data set. A kernel needs to compute distance and sort the k closest neighbors as results.

As shown in Figure 1, each member of $INPUTSET$, which is called $Input$, is placed in shared memory of GPU's block. Each $Input$ includes D different dimensions that is assigned to a thread of GPU, and is referred by its thread index (i.e. $threadIndex$). Similarly each member of $DATASET$, which is called $Data$, include D different dimension. As mention in section III, to avoid redundant repetitive computation, we compute some parts only one time, e.g. $\vec{y}^T S^{-1}\vec{y}$ in equation (3), and call them $PreComputedData$. In functions which are used in this design, such as $PARALLEL\_SUM$, we use various optimization methods like warp managing and reduction technique. The pseudo code of this part is presented in

Algorithm 1: Parallel k-NN

```
Thread ← Input[threadIndex];
FOR i=1 TO D DO
    temp[threadIndex] ← Thread × S^-1[i][ threadIndex];
    var ← PARALLEL_SUM (temp[1..D]);
    IF (threadIndex ==i)
        Resp[i] ← var × Thread;
END DO
Result_1 ← PARALLEL_SUM (Resp[1..D]);
FOR Next=1 TO DATASETSIZE DO
    Distance ← 0;
    temp_product[threadIndex] ← Thread × Data [Next × D+ threadIndex];
    Result_2 ← PARALLEL_SUM (temp_product [1..D]);
    IF (threadIndex ==0)
        Distance ← Result_1 – 2×Result_2 + PreComputedData[Next];
        Sort (Distance); // sort new distance between K nearest
    END IF
END DO
RETURN;
```

Algorithm 1.

A. Reforming of warp arrangement

There are many methods for implementing the kernel. In the simplest one, a member of $INPUTSET$ (i.e. $Input$), is allocated to each block of threads and assign one thread to each block. This thread does computation on one of $Input$ dimension and in the next step, next dimension of $Input$ would be computed. This process continues until the distance computations for all dimensions are done. The problem of this method is that in each step, the GPU platform resources are not fully utilized. In fact, in each step, a block of threads gets resources of a warp, however just one thread is used. In this condition, other resources become idle and this would lead to performance reduction and increasing of the total time of computation. Moreover, there are so many call and rewrite operations which forces a heavy time overhead on the kernel.

TABLE I. SPECIFICATION OF MULTI-CORE AND MANY-CORE PLATFORMS

Commercial Name	Micro Architecture	Core freq (MHz)	Max freq (MHz)	Max Memory Bandwidth(GB/s)	Fab Process (nm)	# of Cores	TDP (Watt)	Memory clock (GHz)
corei7 960	Nehalem	3200	3460	25.6	45	4	130	1.066
Xeon E5-2650 v3	Haswell	2300	3000	68	22	10	105	2.133
GTX480	Fermi	700	1401	177.4	40	480	250	1.84
K20x	Kepler	732	784	250	28	2688	235	2.6

In another method, several inputs are assigned to each block. In this method, the number of threads which assigned to each block is equal to the number of *Inputs* assigned to that block. Like the previous method, each thread does computation of one of the *Inputs*. All the threads are executed in parallel. The problem which is observed in this method is branch divergence. Branch divergence occurs when at least one thread of a warp, chooses a different path in conditional commands. In this situation, all threads need to wait for this distinguished thread. Therefore, occurrence of divergence forces too much time overhead to computation time.

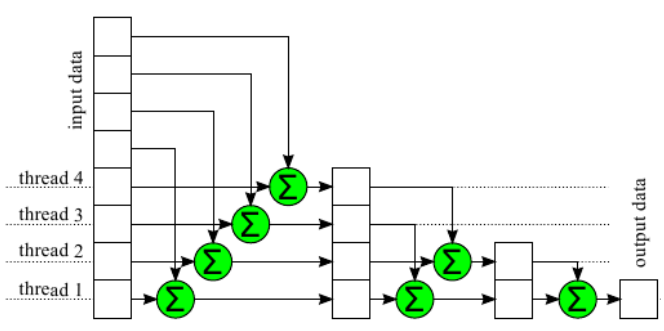
We can better occupy GPU by assigning each *Input* to one block and distribute different dimensions to different threads of the same block. Since the computation of each dimension is independent of other ones and all thread in the same block can be executed in parallel. In this method, if *Input* dimension is less than maximum warp size, part of warp resources would be underutilized. Thus, for maximum utilization of warp resources the *Input* dimension should be equal to warp size.

B. Utilization of reduction technique

The computation time of sum operation is considered as a bottleneck. To fully utilize the parallel infrastructure of summation, parallel reduction technique should be used. The process of combining multiple parallel threads results into one overall result is called reduction technique.

The sequential execution of summation takes N steps; however, as shown in figure 2, parallelization of reduction operation takes just $logN$ steps. Since there is a data dependency between steps, in order to make all threads synchronize, we should insert barriers.

There are two options for applying barriers: the first one is using built-in API (such as *__synchthread()* in CUDA) between critical sections. This option will cause to increase computational time. The second option is using inherent synchronization feature of threads of a warp. In this method, if the number of threads in the block is less than warp size, built-in API can be removed and threads will be synchronized intrinsically. The pseudo code of modified parallel sum reduction, which is called *PARALLEL_SUM*, for Input with 32 dimensions is presented as follows:




```
PARALLEL_SUM
    IF (Index<16) THEN
        Thread [Index] += Thread [Index+16];
        Thread [Index] += Thread [Index+8];
        Thread [Index] += Thread [Index+4];
        Thread [Index] += Thread [Index+2];
        Thread [Index] += Thread [Index+1];
    END IF
```

If maximum warp size is greater than *Input* dimension, it is possible to put more than one *Input* in each block. For example, if the maximum warp size and *Inputs* dimension are considered 32 and 8, respectively, then 4 *Inputs* can be put simultaneously in each warp.

C. Memory management

Since shared memory is faster than global memory, we tried to place more data on the shared memory, in order to better manage the memory.

Moreover, another challenge in this research was a high amount of memory access. It is solved by optimizing memory usage which is handled by big bandwidth between global memory and shared memory.

V. EXPERIMENTAL EVALUATION

For the performance evaluation, we implement the presented method in various many-core platforms with two different architectures. Table I shows a list of many-core platforms which are used for evaluation. We took advantage of GPU platforms via CUDA programming model. To gain better efficiency, we used NVidia Toolkit and CUDA 7.0 API which was the most up to date Toolkit.

At first, we compared our approach implemented in CUDA with the one implemented in CUBLAS library (CUDA

TABLE II. COMPARE COMPUTATION TIME OF SERIAL, CUBLAS AND CUDA IMPLEMENTATION OF PRESENTED APPROACH BASE ON SECOND

Platform	8 Dim.	16 Dim.	32 Dim.
Intel corei7 960	845	2650	9240
NVidia K20x CUBLAS	567	883	1423
NVidia K20x CUDA	538	809	1316

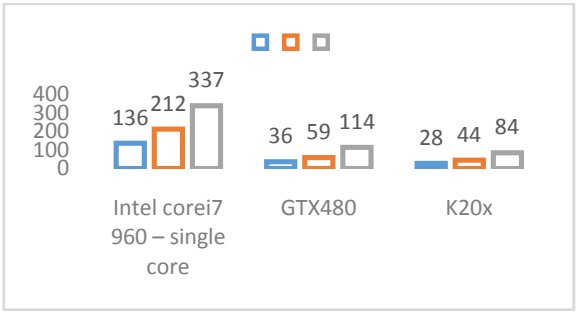


Fig. 3. Execution time for different dimension – X axis show different platforms and Y axis show time base on second

implementation of linear algebra library). We use double precision BLAS3 function to achieve high performance. CUBLAS library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary. This library attaches to a single GPU and does not auto-parallelize across multiple GPUs. CUBLAS provides separate functions to multiply matrices and vectors that has high-speed functionality and uses power optimization techniques. CUDA and CUBLAS are penalized by the time needed to transfer data from host memory (CPU) to device memory (GPU) and back. For high dimensions and a large number of inputs, this penalty becomes more negligible. Computation of matrix multiplication $(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})$ represented the main part of the computation time, so we implement this section by using NVIDIA CUBLAS library. Table II shows computation times of different implementations of proposed algorithm. In this paper, we find 10 nearest neighbors $(k = 10)$ for the large input set $(N = 1{,}000)$ and dataset has $(C = 10{,}000{,}000)$ members and the dimension is variable from 8 to32. Also, insertion sort is used as sorting algorithm. As Table II shows, our CUDA implementation has better execution time even than CUBLUS implementation. Since, we use GPU resource more efficient in CUDA implementation rather than CUBLAS library.

By checking table II, one can note that the parallelization (CUDA and CUBLAS) has achieved better performance in terms of the dimension than CPU single core implementation. The speed-up achieved by CUDA or CUBLAS in comparison with CPU implementation increased significantly with the number of points and dimension. The speed-up achieved by CUBLAS in comparison with CUDA increased much less. However, CUBLAS computes the distances more efficiently. In this experiment, CUDA and CUBLAS were up to 7X and 6.45X faster than the serial implementation in CPU, respectively.

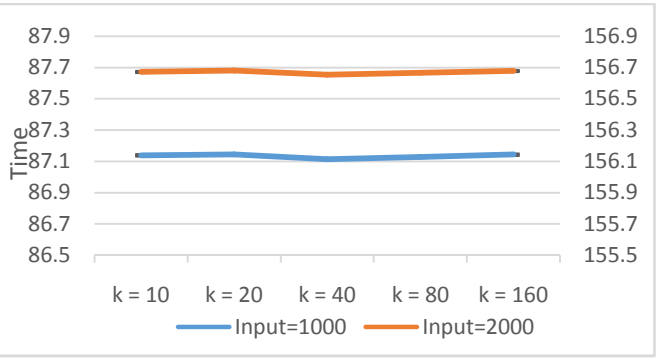


Fig. 4. Execution time for different k - X axis shows different K and Y axis shows time base on second

For evaluation of our mathematic modification, we run two different versions of code on Intel corei7 960. One of these codes exploits our mathematic modification and the other one does not. We use a single core of this processor to compare this modification in a sequential approach. As shown in Table III, the modified version has the better result. So after that we only use modified version.

We evaluate performance of proposed method by comparing the execution time of k-NN on a GeForce GTX 480 and Tesla K20x card with a modified serial version of k-NN program on an Intel corei7 CPU for the mentioned $INPUTSET$ and $DATASET(N{=}1{,}000$ and $C{=}10{,}000{,}000)$. Figure 3 shows the execution time of various platforms for different dimensions. Experimental results present 110X speedup of the modified and parallelized implementation which is exposed in Figure 3 rather than non-modified and sequential implementation which is showed in Table III.

TABLE III. COMPARE MATHEMATIC MODIFIED AND NON-MODIFIED VERSIONS BASE ON SECOND

Version	8 Dim.	16 Dim.	32 Dim.
Non-Modified	845	2650	9248
Modified	136	212	337

Figure 4 shows the variation of proposed algorithm base on different k value. To obtain these results, we consider the dimensions of data point 32 and computation is done by Nvidia Tesla K20x. In this figure, left vertical Y axis shows time for 1000 input and right vertical Y axis shows time for 2000 input. It is obvious that the time computation of algorithm for different k, between 10 and 160, is roughly constant and proposed algorithm is independent of the k variation.

We also use NVidia Management Library (NVML) for

TABALE IV. AVERAGE POWER CONSUMPTION (P_{avg}), COMPUTATION TIME (T) AND CORRESPONDING ENERGY (E) MEASURED FOR THE EXECUTION OF PRESENTED IMPLEMENTATION

Device	Idle	8 Dim.			16 Dim.			32 Dim.		
	(W)	$P_{avg}(W)$	$T(s)$	$E(J)$	$P_{avg}(W)$	$T(s)$	$E(J)$	$P_{avg}(W)$	$T(s)$	$E(J)$
Intel Xeon 2650	17	36	720	25961	56	1766	99736	69	4297	296493
NVidia Tesla K20x	25	80	28	2240	82	44	3608	84	84	7056

measure power consumption of GPU platform [18] and Intel Power Governor SDK [19] for evaluation power consumption of CPU platform. As mentioned in [18], the run-time power of a kernel is measured with the NVML library by running the kernel on a thread and NVML on another thread (by using POSIX Threads usually referred to as Pthreads). The results of this experiment for various dimensions are shown in table IV. Both platforms are ones of the best and the most up to date devices in their category and are designed and manufactured base on latest techniques for reducing energy consumptions of the device. As table IV exposes, the energy consumption of our implementation on GPU platform is much less than CPU platform.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a GPU implementation of the k-nearest neighbor algorithm based on the Mahalanobis distance metric. Our GPU implementation is 110 times faster than the sequential programming approach. This implementation needs less memory and decreases bandwidth requirements of the k-NN search algorithm. To accelerate overall execution time, we eliminate redundant repetitive tasks in the k-NN algorithm. Our comparison shows that GPU implementation has better energy consumption rather than CPU. This implementation can be scaled into several GPUs, that it is one of our future works.

ACKNOWLEDGMENT

We are grateful to Prof. Hamid Sarbazi Azad, Head of the school of computer science, for his support and useful guidance. We also would like to acknowledge Mr. Mohsen Mahmoudi Aznaveh and all member of HPC lab at IPM.

REFRENCES

[1] S. Kim, B. Ham, B. Kim, and K. Sohn, "Mahalanobis Distance Cross-Correlation for Illumination-Invariant Stereo Matching,," *IEEE Transactions on Circuits and Systems for Video Technology,* pp. 1844,1859, Nov. 2014.

[2] M.-C. L. Chi-Man Pun "Log-polar wavelet energy signatures for rotation and scale invariant texture classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 25, pp. 590-603, 2003.

[3] P. Gabriel, J. B. Hayet, J. Piater, and J. Verly, "Object tracking using color interest points," in *Advanced Video and Signal Based Surveillance, 2005. AVSS 2005. IEEE Conference on*, 2005, pp. 159-164.

[4] K. Z. Mao and W. Tang, "Recursive Mahalanobis Separability Measure for Gene Subset Selection," *IEEE/ACM Trans. Comput. Biol. Bioinformatics,* vol. 8, pp. 266-272, 2011.

[5] J. Han and M. Kamber, *Data mining: concepts and techniques*: Morgan Kaufmann Publishers Inc., 2000.

[6] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," presented at the Proceedings of the 24rd International Conference on Very Large Data Bases, 1998.

[7] J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," presented at the Proceedings of the 1981 ACM SIGMOD international conference on Management of data, Ann Arbor, Michigan, 1981.

[8] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," presented at the Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two, Barcelona, Catalonia, Spain, 2011.

[9] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," presented at the Proceedings of the thirtieth annual ACM symposium on Theory of computing, Dallas, Texas, USA, 1998.

[10] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Inf.,* vol. 4, pp. 1-9, 1974.

[11] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects," 1987.

[12] N. Katayama and S. i. Satoh, "The SR-tree: an index structure for high-dimensional nearest neighbor queries," *SIGMOD Rec.,* vol. 26, pp. 369-380, 1997.

[13] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, Berkeley, Calif., 1967, pp. 281-297.

[14] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, 2008, pp. 1-6.

[15] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *ICIP*, 2010, pp. 3757-3760.

[16] S. Liang, Y. Liu, C. Wang, and L. Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," in *Web Society (SWS), 2010 IEEE 2nd Symposium on*, 2010, pp. 53-60.

[17] A. Ahmadzadeh, R. Mirzaei, H. Madani, M. Shobeiri, M. Sadeghi, M. Gavahi, K. Jafari, M. M. Aznaveh, and S. Gorgin, "Cost-efficient implementation of k-NN algorithm on multi-core processors," in *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, 2014, pp. 205-208.

[18] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, "Power aware computing on GPUs," in *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, 2012, pp. 64-73.

[19] M. Dimitrov. (2012). *Intel Power Governor*. Available: https://software.intel.com/en-us/articles/intel-power-governor