GPU-Enabled Brute Force k-Nearest Neighbors on CUDA

Raza Hussain (shussain11@student.gsu.edu)

CSc 8530, Parallel Algorithms
Spring 2018

***Abstract*—In data mining and machine learning the k-Nearest Neighbors (kNN) algorithm is used for many applications including classification, regression, clustering, image processing, information retrieval, and language translation, to name just a few. As its name suggests, kNN finds the k most similar samples (neighbors) from a reference date set for a given query. However, in high-dimensional spaces, the kNN algorithm suffers from poor performance due to its computational complexity. Therefore, parallel implementations of kNN are desirable since they can help to mitigate the effects that high-dimensional data sets have on its performance. The Graphics Processing Unit (GPU) provides parallel computation capabilities that result in significant performance improvements in image and video processing tasks by making use of the NVIDIA CUDA API [1]. In this paper, a GPU-based kNN algorithm is implemented using CUDA and the squared matrix is considered to compute the distances of kNN queries. Moreover, we ported the GPU-based brute force implementation in [2] to a CPU-based serial implementation and we compare the performance of the two implementations. We obtain a considerable performance increase compared to previous state of the art methods.**

***Index Terms*—K-nearest neighbor (kNN), Graphics Processing Units (GPU), Classification.**

I. Introduction

Tasks such as image analysis, information retrieval, forecasting, and video security systems are becoming increasingly difficult to process sequentially due to the ever-increasing amount of data that must be processed. Applying data mining and machine learning algorithms can speed up processing tasks for certain data sets. Many different classification and clustering techniques have been developed over the years. kNN is an algorithm that is widely used in the fields of machine learning, image processing, bioinformatics, statistical analysis for classification and regression analysis of a data set. kNN is an instance-based algorithm that compares the points in the data set with the points in a query. The similarity of the samples in the data set compared to the query is measured by calculating the distance metric to classify the query. The distance function can improve the classification accuracy depending on the specific data set. It has been shown that the distance between the points in a query and the points in a large data set can be effectively calculated locally [3], [4]. The kNN algorithm is illustrated on a set of points in Figure 1.

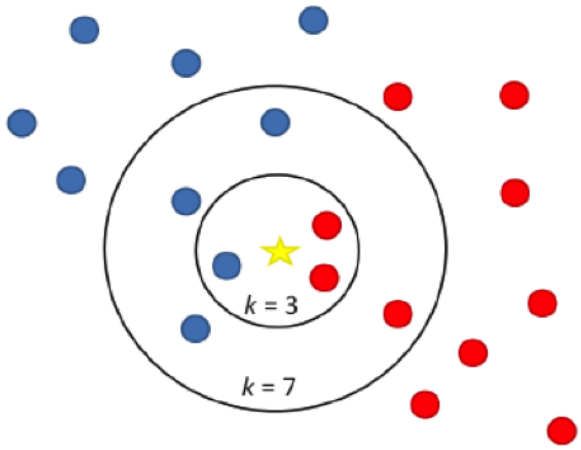


Fig. 1: kNN for $k = 3$ and $k = 7$ nearest neighbors.

The query point is denoted by the star in the center of the figure. The coordinates of this point are used to find the desired number of nearest neighbors. As can be seen in the figure, when the value of k is set to 3, the nearest neighbors are shown by the three points enclosed in the innermost circle. Similarly, when the value of k is set to 7, the nearest neighbors are shown by the seven points enclosed in the outermost circle, which includes the points found when k was 3. Intuitively, when

the value of k increases, the running time of kNN also increases. The overall complexity of kNN can be described as follows. To compute the distance to $k = 1$ nearest neighbors for d dimensions, the running time is $O(d)$. To compute the pairwise distance of all d-dimensional points in a dataset consisting of n points, the running time is $O(nd)$. If we then choose a different value of k, the overall runnning time becomes $O(nk + nd)$ [5].

The so-called "curse of dimensionality" makes searching for the nearest neighbors of the points in a query extremely time consuming as the number of dimensions increases. Efforts to index the structure of the data tend to increase the complexity of kNN with little improvement to show for the increased complexity. Therefore, parallel computation of distances can be an effective solution to improve the performance of kNN, especially on large data sets. Because their massively parallel architectures dwarf the parallel capabilities of contemporary CPUs, GPUs can play a key role in computing distances between points [6]. Moreover, GPUs have become much more affordable in recent years, leading to their increased adoption within the research community.

In this paper, the brute force kNN implementation in [2] is examined using a CPU-based serial implementation as well as a GPU based implementation. The brute force method of computing distances works by computing the squared distances of the points in the data set and then searching for the k-smallest squared distance for each query. The effectiveness of the brute force GPU implementation depends on minimizing the copying of data between the host and the GPU. We compare the performance of the brute force algorithm using data sets composed of millions of elements with varying numbers of attributes for different sizes for k. In all cases, the Euclidian distance metric was used to compute the distance between points. All the distances are computed in a row by row basis. We show that a significant performance increase can be obtained using a GPU implementation for finding the K-nearest neighboring elements compared with other methods.

The rest of the paper is organized as follows: Section II briefly reviews the other related work of GPU implementation for finding k-nearest neighboring elements. Section III describes the brute force kNN search. Section IV describes about the CUDA platforms. Our experimental setup and the results we obtained are described in sections V and VI, respectively. Finally, Section VII concludes the paper with some future work.

II. Related Work

The use of GPUs to aid parallel processing of massive amounts of data quickly and efficiently continues to increase. Many indexing techniques have been proposed to avoid the redundant computation of distances between points. K-d trees [4] have been used as one approach for solving the indexing problem. The Minimum Spanning Tree (MST) is used in [7] for kNN classification. The authors propose a convex data structure optimization technique to calculate the minimum distance of each query using a GPU implementation. A heap based structure which makes use of a priority queue and a controller to exchange information in between CPU and GPU is proposed in [8], [7]. Truncated insertion sort was proposed to handle the query efficiently and to delete the processed items from the list and eventually obtain the minimum value. The authors show that a heap tree structure with priority queue utilization are a promising technique that improve the performance and accuracy rate of kNN.

Non-tree structures and coarse-grained MPI-based structures have also been considered previously, however, their use increased the complexity of the index, causing it to be less effective [6], [9]. A new distance measure between images was proposed in [9] to simplify the indexing and retrieval of similar images in large data sets.

The GPU based implementation in [2] of brute force kNN simplifies the indexing problem. However, CPU-based parallel implementations of brute force kNN do not inherit the same reductions in complexity due to the inherent memory contention issues. Optimization techniques have been proposed in order to maximize CPU utilization. A sorting algorithm was proposed in [8] that works by sorting all the distances of the queries based on GPU radix sort.

III. Brute Force kNN

The kNN classification algorithm is a widely used technique to find the nearest neighbor for the purpose of classifying an input query. Let $X = u_1, u_2, \cdots, u_n$ be the set of n elements with $u_i \in U$

For K = 20							
Serial C Time	n = 65536	n = 262144	n = 1.04 M	n = 4.19 M	n = 16.78 M	n = 67.11 M	n = 268.43 M
attributes = 1	0.013436	0.030902	0.087871	0.210881	0.827032	3.309748	13.409107
attributes = 4	0.004114	0.015899	0.062302	0.246896	0.981026	3.917369	15.807368
attributes = 16	0.006395	0.025149	0.098902	0.392032	1.561678	6.385722	25.171505
attributes = 64	0.016032	0.063511	0.251774	1.013237	4.011605	16.313342	65.039414
attributes = 256	0.052603	0.209976	0.842106	3.359452	13.459978	53.786060	217.314409
CUDA C Time	n = 65536	n = 262144	n = 1.04 M	n = 4.19 M	n = 16.78 M	n = 67.11 M	n = 268.43 M
attributes = 1	0.421919	0.419270	0.420871	0.430118	0.468458	0.613930	1.196618
attributes = 4	0.420371	0.419469	0.426368	0.429201	0.469669	0.620023	1.223312
attributes = 16	0.417614	0.414482	0.422215	0.432828	0.485006	0.679875	1.435813
attributes = 64	0.420379	0.419494	0.425001	0.450100	0.528774	0.844008	2.080894
attributes = 256	0.418640	0.421336	0.437356	0.493969	0.707720	1.520078	4.793522

For K = 100							
Serial C Time	n = 65536	n = 262144	n = 1.04 M	n = 4.19 M	n = 16.78 M	n = 67.11 M	n = 268.43 M
attributes = 1	0.048013	0.110143	0.218016	0.848374	3.414924	13.633248	54.660771
attributes = 4	0.014903	0.057748	0.226309	0.895016	3.559934	14.239883	57.797932
attributes = 16	0.017143	0.066682	0.262630	1.051911	4.145434	16.578274	66.866427
attributes = 64	0.026711	0.104834	0.415321	1.651571	6.684300	26.369912	107.412841
attributes = 256	0.063512	0.251780	1.001679	4.012104	16.036892	64.310819	257.472167
CUDA C Time	n = 65536	n = 262144	n = 1.04 M	n = 4.19 M	n = 16.78 M	n = 67.11 M	n = 268.43 M
attributes = 1	0.419826	0.418529	0.424296	0.429826	0.471889	0.616870	1.198858
attributes = 4	0.420330	0.418197	0.422653	0.430647	0.471199	0.619459	1.212209
attributes = 16	0.418604	0.418555	0.424506	0.435407	0.485678	0.692184	1.432241
attributes = 64	0.419286	0.420082	0.424993	0.446868	0.528990	0.849035	2.079278
attributes = 256	0.416149	0.424112	0.434249	0.495339	0.707413	1.523641	4.798463

TABLE I: Aggregate Test Results

and let $q_i \in Q, Q \subseteq U$ be a query. The distance between the two points can be represented by a function. The Euclidian or manhattan distance metrics are commonly used for determining the distance between the points for a given query. Other distance metrics such as Chebyshev and Mahalanobis are widely used and provide similar performance. The brute force algorithm is sometimes referred to as exhaustive search. Once all of the distances between points have been calculated the points are sorted. This process is performed for each query point.

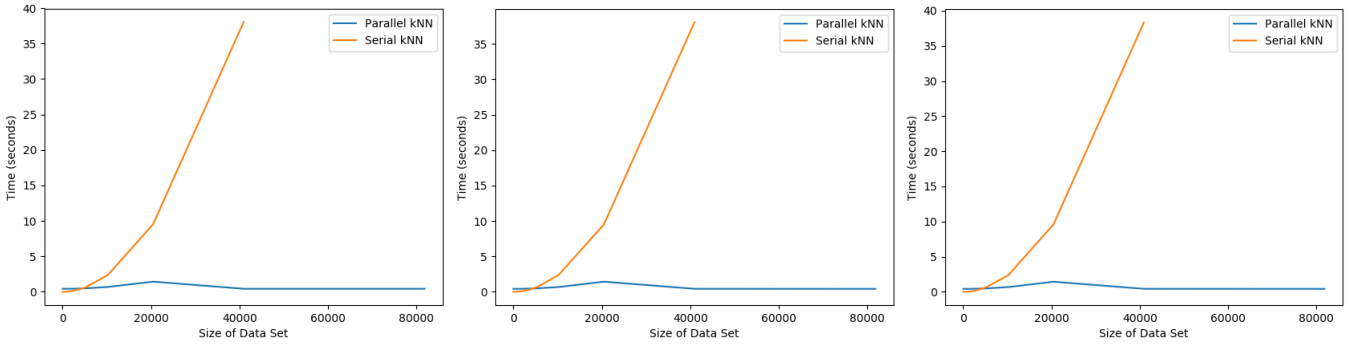
As Section II shows, several kNN algorithms have been proposed to find the nearest element with minimum distance. As the number of dimensions increases, distance calculations become a bottleneck in terms of performance. Effective sorting of the points is also crucial. Insertion sort and quicksort have been used in previous approaches to sort the distances. Although quicksort has a better running time than insertion sort, in terms of the brute force method, radix sort has also shown good performance.

IV. CUDA Implementation

CUDA is a widely used platform for parallel computing using GPUs. In particular, the use of CUDA on data mining and machine learning algorithms can provide substational performance improvements over CPU-based implementations. Additionally, the field of deep learning has experienced a revival due to the ability to deploy very deep neural networks on GPUs and execute them in reasonable running times. As a result, GPU computing using CUDA has drawn immense interest in scientific and research circles.

The brute force process of computing distances can be peformed in parallel due to the independence of the points in the data set. This property enables parallel GPU based implementations. As mentioned previously, the brute force method works by computing the distance of neighboring elements, then sorting the distances using some sorting algorithm. The distance calculation is performed for each query element.

The texture memory and the global memory are used in parallel processing of data. The texture memory is a read only memory and sometimes it is more effective in terms of memory accesses when it becomes non-coalesced. Global memory is a read and write memory. Curiously, the texture memory performs better than global memory in certain situ-



(a) 5 attributes, $k = 1$. (b) 5 attributes, $k = 10$. (c) 5 attributes, $k = 20$.

Fig. 2: Comparison between serial and parallel brute force kNN for varying values of k.

ations such as non-coalesced memory access which can decrease the performance of global memory. In most other cases, however, global memory performs better as it has much greater bandwidth and it resides in CUDA environment. Moreover, the texture memory is used for representing the reference elements set but the global memory is used for representing the query element set. The sorting and distance computation processes can be parallelized in CUDA using kernels for the respective processes.

V. Experimental Setup

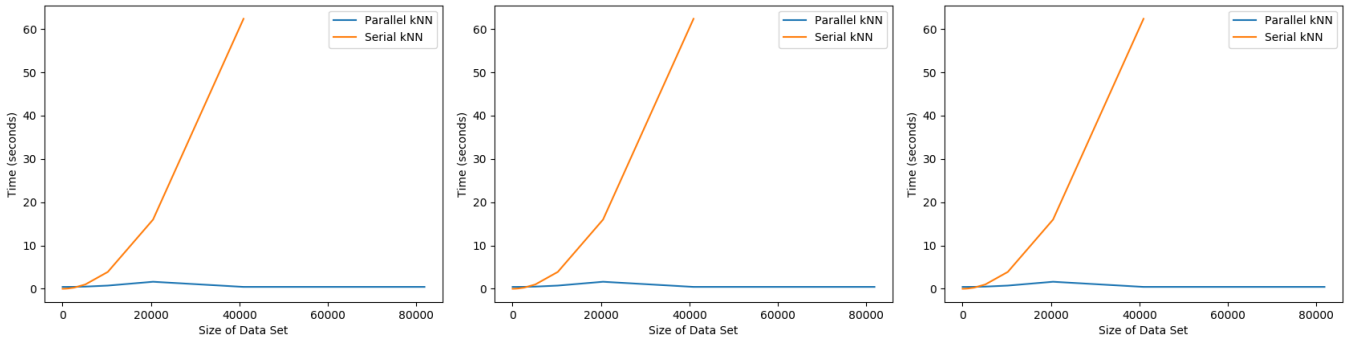
The hardware we used to compare the performance of the brute force algorithm between the CPU-based serial implementation and the GPU-based implementation consists of an Intel Core i7-4770k CPU, 8 GB of main memory, and an Nvidia GeForce GTX 980 GPU. The operating system used for testing is Ubuntu Linux 16.04 LTS and the version of CUDA used was 8.0.

VI. Results

The aggregate results for our testing are shown in Table I. The table shows the time taken (in seconds) for the program to run for the given values of parameters k, n, and $attributes$. The number of nearest neighbors that we need to find for each object is given by k. The $attributes$ parameter is the number of dimensions of each object. n is the total size of the data matrix that is computed, which is the square of the total number of objects. Each floating point entry in the table is the time taken (in seconds) for the program to run for a particular value of k, $attributes$, n, and $ProgramType$, where $ProgramType$ is either serial or CUDA.

Figure 2 shows the results of our comparison between the serial and paralellel implementations of brute force kNN when the value of k is increased. As shown in the figure, as k is increased, neither the shape of the graphs nor the running time changes much. This suggests that the value of k is not a factor in the performance of the algorithm. This makes sense intuitively since the portion of the algorithm that is most computationally expensive is computing the squared distance matrix. In comparing the serial implementation to the GPU-based parallel implementation, we see that once the size of the data set exceeds 20,000 instances, the running time of the serial implementation sharply increases while the parallel implementation remains mostly the same throughout for all values of k.

In order to measure the effect of the number of attributes on the performance of the algorithm, we ran another set of tests with k held constant at 100 and varied the number of attributes. Figure 3 shows the results of these tests. Similar to the tests shown in Figure 2, the serial implementation exhibits an exponential increase in running time. However, the overall running time is increased much faster. As shown in Figure 3a, when doubling the number of attributes, the running time of the serial algorithm increases by 150% over the previous tests. In Figure 3b the running time of the serial algorithm increases by nearly an order of five times over Figure 3a and increases by nearly an order of ten times over 3a in Figure 3c. This suggests that the performance of the serial algorithm is directly related to the number of attributes in the data. The performance of the GPU-based implementation is similar to the tests with varying k, suggesting that computing the squared distance matrix in parallel provides the best



(a) 10 attributes, $k = 100$. (b) 50 attributes, $k = 100$. (c) 100 attributes, $k = 100$.

Fig. 3: Comparison between serial and parallel brute force kNN for varying numbers of attributes.

performance gain over the serial implementation.

For both figures, the X-axis in each figure represents the squared distance matrix for the given data set size, and the values should be squared to give a better sense of the amount of computations required. We decided against showing the squared values since doing so negatively impacted the resolution of the graphs, making them more difficult to understand.

VII. Conclusions

In this paper, we have demonstrated that a GPU-based brute force K-nearest neighbor algorithm provides a significant improvement in performance over a CPU-based implementation. Because we can compute the squared distance matrix in parallel, the theoretical parallel speedup is approximately $\frac{O(nk+nd)}{O(k+d)} = n$.

Further optimizations are possible. Computing the squared distance matrix involves many redundant distance computations since the result is a symmetric matrix with zeros down the main diagonal. Limiting distance calculations to either the upper or lower portion of the matrix could improve the running time, especially on larger data sets. Finally, the selection of the k neighbors can be sped up by using a heap instead of an iterative search.

References

[1] J. Minar, K. Riha, and H. Tong, "Intruder detection for automated access control systems with kinect device," in *Telecommunications and Signal Processing (TSP), 2013 36th International Conference on*. IEEE, 2013, pp. 826–829.

[2] S. Li and N. Amenta, "Brute-force k-nearest neighbors search on the gpu," in *International Conference on Similarity Search and Applications*. Springer, 2015, pp. 259–270.

[3] S. Boltz, E. Debreuve, and M. Barlaud, "High-dimensional statistical distance for region-of-interest tracking: Application to combining a soft geometric constraint with radiometry," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.

[4] N. Corporation, "cublas in cuda toolkit 6.5." *URL https://developer.nvidia.com/cuda-toolkit-65*, 2014.

[5] O. Veksler, "knn: Computational complexity," *URL http://www.csd.uwo.ca/courses/CS9840a/Lecture2_knn.pdf*, 2015.

[6] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009.

[7] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato, "knn-borúvka-gpu: a fast and scalable mst construction from knn graphs on gpu," in *International Conference on Computational Science and Its Applications*. Springer, 2012, pp. 71–86.

[8] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matias, and M. Marin, "knn query processing in metric spaces using gpus," in *European Conference on Parallel Processing*. Springer, 2011, pp. 380–392.

[9] P. Piro, S. Anthoine, E. Debreuve, and M. Barlaud, "Image retrieval via kullback-leibler divergence of patches of multiscale coefficients in the knn framework," in *Content-Based Multimedia Indexing, 2008. CBMI 2008. International Workshop on*. IEEE, 2008, pp. 230–235.