International Journal of Computer Applications (0975 - 8887)
Volume 181 - No.11, August 2018

Similarity Learning in Many Core Architecture

Arjeta Selimi-Rexha
Dept. of Computer Science, College of Computer
Qassim University
Qassim, Saudi Arabia

Ali Mustafa Qamar
Dept. of Computer Science, College of Computer
Qassim University
Qassim, Saudi Arabia

ABSTRACT

A lot of recent research works have pointed out that metric learning is far better as compared to using default metrics such as Euclidean distance, cosine similarity, etc. Moreover, similarity learning based on cosine similarity has been proved to work better for many of the data sets, which are not necessarily textual in nature. Nevertheless, similarity learning in nearest neighbor algorithms has been inherently slow, owing to their $O(d^3)$ complexity. This short-coming is addressed in this research and a similarity learning algorithm for many core architectures is proposed; whereby, Similarity Learning Algorithm (*SiLA*) is parallelized. The resulting algorithm is faster than the traditional one on many data sets because of its parallel nature. The results are confirmed on UCI data sets.

Keywords

Similarity learning, $SiLA$ algorithm, Parallel computing, Supervised machine learning, CUDA programming

1. INTRODUCTION

One of the oldest and widely applied classification rule is the k nearest neighbor (*kNN*) algorithm. The k nearest neighbor classification rule requires a training data set of labeled instances. It classifies a new data instance with the class the most represented in the set of k closest (labeled) training instances. The kNN rule has been studied from different points of view, such as in machine learning, pattern recognition, database etc. In an attempt to enhance the standard kNN rule, the researchers normally take one of the two lines of research: distance metric learning (e.g. [14]) or similarity learning (e.g. [12]). While distance metric learning is focused on learning a distance function between the points in the space in which the examples lie, similarity learning focuses on learning a measure of similarity between two pairs of objects. It has been shown that in several real cases, similarities between pairs may be preferable than distances. One of the pioneer works on similarity learning is Similarity Learning for Nearest Neighbor Classification (*SiLA*) [12], an approach that aims to learn a similarity metric for k Nearest Neighbor (kNN) classification rule. *SiLA* algorithm has shown better results on many data sets than distance metric learning. The aim is to move the nearest neighbors that belong to the same class nearer to the input example (known as *target* neighbors) while pushing away the nearest examples belonging to different classes (known as *impostors*). Despite its simplicity, *kNN* algorithm has a high computational complexity itself, so learning a metric for k-nearest neighbor is also slow (*SiLA* has got $O(d^3)$ complexity). As a result, a satisfactory performance in many applications cannot be accomplished because of this drawback. When training set and test set is large, the execution time may be a bottleneck for the application [8]. Therefore, in order to be practically efficient, there is a need to speed up the execution time of learning process. The recent developments in parallel computing have opened a new era of computing which delivers tremendous computational power. Thus, parallel computing provides a way to speed up the execution time taken to perform large tasks. Various works have partly addressed the execution issues for kNN, few works address this for distance metric learning such as [9] and [13], but no previous study is known that has completely addressed it in the similarity learning context. This is exactly the aim of this research. A CUDA implementation for learning a similarity metric is proposed, that is a parallel version of *SiLA* algorithm. Besides being faster on large data sets, its performance is also improved. The rest of this paper is organized as follows: Section 2 describes the state of the art, a parallel version of *SiLA* algorithm is presented in Section 3 whereas Section 4 covers the experimental results followed by conclusion and the future work.

2. STATE OF THE ART

Numerous machine learning approaches rely on some metric. On one hand, it incorporates unsupervised learning like clustering, grouping together close or similar objects. On the other hand, it incorporates supervised approaches such as nearest neighbor classification algorithm, which depends on a representation of the input data, i.e. the labels of nearest objects to decide on the label of a new object. Metric learning can be divided into two different types: distance metric learning and similarity learning. Distance metric learning is a method of learning a distance metric from a training dataset which consists of a given collection of pair of similar/dissimilar points that shows the distance relation among the training data. Speaking in general, distance metric learning aims to minimize the distance between a similar pair and to separate a dissimilar pair with a large distance. Toward this objective, many researches have been done and a variety of functions and algorithms were proposed. One of the first works on Metric Learning has been Xing's distance metric learning [15]. A widely known approach for metric learning is Large Margin Nearest Neighbor (*LMNN*) [14], which is one of the most widely used Mahanalobis distance learning methods. *LMNN* is an algorithm that learns the metric in a supervised fashion to improve the accuracy of the k nearest neighbor classification rule. During learning, differently labeled examples (the *impostors*) are pushed outside the perimeter established by the *target* neighbors. A more recent work on metric learning is Distance Metric Learning using Graph Convolutional Networks [7] that propose a metric learning method to evaluate the distance between irregular graphs that leverages the power of convolutional neural networks [7]. Another type of metric used to measure the pairwise relationship between two feature vectors is similarity metric. In several real world circumstances like information retrieval, text analysis etc., similarities were preferred over distances. Speaking in general, a similarity learning method intends to increase the similarity between a similar pair and to decrease the similarity between a dissimilar pair. One of the first works on similarity learning is Similarity Learning for Nearest Neighbor classification (*SiLA*) [12], based on the voted perceptron algorithm [5]. Like *LMNN* [14], *SiLA* also aims at moving the *target* points closer to the input point, and at the same time pushing away differently labeled examples. Other works on similarity learning include *good-edit similarity learning by loss minimization* [1], *Online Algorithm for Scalable Image Similarity (OASIS)* [2], *Generalized Cosine Learning Algorithm (gCosLA)* [11] etc.

In serial programming, a single processor executes program instructions step-by-step; a problem is broken into a series of instructions which are executed sequentially one after the other on a single processor. However, some operations have multiple steps that do not have time dependencies and therefore, they can be separated into multiple smaller tasks that can be executed concurrently. Parallel computing helps in performing large computations by typically breaking down a computational task into very similar sub-tasks that can be processed independently, and assigning these tasks to more than one processor, all of which perform the computation at the same time. The goal of parallel computing is to increase the performance of an application by executing it on multiple processors, thus reducing the execution time. There are two approaches: the first, multi-core approach, which integrates a few cores (between two and ten) into a single microprocessor, seeking to keep the execution speed of sequential programs [3]. Actual laptops and desktops incorporate this type of processor. The second approach is many-core which uses a large number of cores. This approach is exemplified by the Graphical Processing Units (GPUs) available today [3]. One kind of widely used GPUs are NVIDIA GPUs or CUDA GPUs. In GPU applications, the sequential section of the workload is executed on a CPU which is optimized for single-threaded performance, while CUDA is used to execute the computation intensive portion of the application on the GPU. CUDA-enabled GPUs have thousands of cores that can collectively run thousands of computing threads, thereby speeding up the processes. The parallel approach of *SiLA* algorithm leverages the parallel computing nature to speedup the learning process.

3. PARALLEL SIMILARITY LEARNING ALGORITHM

This section provides a parallel algorithm for learning the similarity metric, a parallel version of *SiLA* algorithm. The core of *SiLA* (and its parallel version) is an online update rule in which the current estimate of the learned metric is iteratively improved. In *SiLA*, when an input instance i is not separated from the impostors, an update is performed to the current A matrix by the difference between the coordinates of the target neighbors $T(i)$ and the impostors $B(i)$. If the input example under focus is correctly classified by the current A matrix, then A is left unchanged and its weight is incremented by 1, so that the final weights correspond to the number of instances properly classified by matrix A.

In order to speed up the learning process, the sets $T(i)$ and $B(i)$ are found in advance. Since the set $B(i)$ changes over time, a predetermined number of impostors (e.g. 100) is found for each example before the algorithm has been launched. Later, these impostors are analyzed to find the most similar one with the example under consideration [10]. The worst-time complexity of *SiLA* is $O(Mnp^2)$ where M is the number of iterations, n is the number of train examples while p stands for the number of dimensions. The most expensive steps consist of calculating the similarity s_A and f_{ml} [10]. Since $T(i)$ does not change over time, it is computed only once before the algorithm is launched, and the similarity with the target neighbors is calculated while considering those in advance. In the parallel approach, $T(i)$ is also found in advance, though in a parallel manner. A special CUDA kernel was written for this purpose. After transferring the arranged data as a matrix and its transpose from CPU to GPU, each thread performs the similarity calculation between a pair of points. Thus, each row of the resulting matrix will contain the similarity of a single instance from the training dataset with all other instances. In order to find the nearest neighbors, a `stable_sort_by_key` over *Thrust* library [6] was used to sort the similarity matrix. Since $B(i)$ changes with the passage of time, calculating the similarity function with all the training examples will be costly and time-consuming.

The independence of similarity function between pairs of data can be leveraged to speedup the learning process. For each example for which its nearest neighbors are to be found, its similarity with the every instance in the training set is calculated separately in parallel. Here, it is pertinent to mention that the parallel *SiLA* does not make the assumption that the nearest impostors must be from the first 100 calculated earlier. Therefore, instead of comparing only 100 impostors found in advance when finding the impostors online, parallel *SiLA* will make a comparison with all the training examples.

Also, it can be seen that the function f_{ml} can be parallelized because it does not have any dependency between the different target neighbors / impostors.

So for each target neighbor / impostor, f_{ml} function can be calculated concurrently. The parallel training algorithm of *SiLA* is given next:

p*SiLA* - Training

```
Input: training set
((x^{(1)}, c^{(1)}), ..., (x^{(n)}, c^{(n)})) of n vectors
in ℝ^p, number of epochs M; A_{ml} denotes the element of A
at row m and column l
Output: list of weighted (p × p) matrices
((A_1, w_1), ..., (A_q, w_q))
Initialization: t = 1, A^{(1)} = 0 (null matrix), w_1 = 0
Repeat M times (epochs)
  1. for $i = 1, \cdots, n$
    2. $B(i) = \text{kNN}(A^{(t)}, x^{(i)}, \bar{c}^{(i)})$
      2.1 $B(i)$ is computed in parallel - separate thread for
      calculating the similarity function with each example,
      and parallel sorting of nearest neighbors
    3. if $\sum_{y \in T(i)} s_A(x^{(i)}, y) - \sum_{z \in B(i)} s_A(x^{(i)}, z) \leq 0$
      4. $\forall (m,l), 1 \leq m,l \leq p$,
         $A^{(t+1)}_{ml} = A^{(t)}_{ml} + \sum_{y \in T(i)} f_{ml}(x^{(i)}, y)$
                    $- \sum_{z \in B(i)} f_{ml}(x^{(i)}, z)$
        4.1 Parallel computation of $\sum_{y \in T(i)} f_{ml}(x^{(i)}, y)$ and
        $\sum_{z \in B(i)} f_{ml}(x^{(i)}, z)$ for each nearest neighbor
    5. $w_{t+1} = 1$
    6. $t = t + 1$
7. else
    8. $w_t = w_t + 1$
```

where k nearest neighbors of example x belonging to class s, are given by *kNN*(A, x, s), $T(i)$ are the target neighbors of $x(i)$, and $B(i)$ is the set of impostors. Parallel *SiLA* is implemented in CUDA C/C++.
A special CUDA kernel was written for similarity calculation during the training, which calculates the similarity between pairs of example. Then a `stable_sort_by_key` is used over the similarity vector to find the nearest neighbors.
While calculating f_{ml}, the similarity of one example with the rest of the dataset is calculated in parallel. A special CUDA kernel is written for this purpose. During the prediction, two rules are considered by *SiLA*: the standard *kNN*, which classifies and finds the nearest neighbors that belong to the same class; and symmetric $kNN(SkNN)$ which finds nearest neighbors from different classes. It is important to note that each testing example is independent from the rest of the testing examples. Thus, for each testing point, its similarity calculation with all of the training examples, sorting of the similarity function to find the nearest neighbors, and assigning the class can be done by a separate thread for each example. Therefore, all the testing examples can be processed concurrently, eventually speeding up the testing process. However, in some cases there are only a small number of testing examples e.g. in face recognition we may have only one test example. In such cases, assigning the prediction algorithm for each test point to a separate thread would be inefficient. As a second approach to cover such cases, the similarity with all the training examples can be calculated in parallel, as well as the process of sorting to find the nearest neighbors. Lastly, assigning the class can be done easily in a sequential manner. The parallel version of the prediction algorithm is given next:

p*SiLA* - Prediction

Input: new example x in $\mathbb{R}^p$, list of weighted $(p \times p)$ matrices $((A_1, w_1), \cdots, (A_q, w_q))$; A is defined as: $A = \sum_{l=1}^{q} w_l A_l$
Output: list of classes

```
(1) Standard kNN rule
    First approach: for each thread, assign a test example and do
    the following:
      1. Compute the $k$ nearest neighbors based on $s_A$;
      2. Select the class with the highest weight (or the class which
         is the most represented in the nearest neighbor set)
    Second approach: for each thread, assign:
      1. pair similarity calculation
      2. sort in parallel the similarities to find the nearest neighbors
      3. assign a class to the test example in a sequential manner
(2) Symmetric classification rule
    Let $T(x, s) = \text{kNN}(A, x, s)$;
    First approach: for each thread, assign a test example and as-
    sign $x$ to the class for which $\sum_{z \in T(x,s)} s_A(x, z)$ is maximal.
    Second approach: for each thread assign
      1. pair similarity calculation
      2. sort in parallel the similarities to find the nearest neighbors
      3. assign a class to the test example in a sequential manner
```

For the first approach, a CUDA kernel was written, which performs the similarity calculation with each training example, sorting those similarities to get the nearest neighbors, and then choosing the class that is more present in the nearest neighbor set. Each thread follows these steps for one testing example separately from the other testing examples, and will run concurrently. A vector containing the predicted class for each example will be returned to the CPU, which will calculate the precision depending on these predicted classes. For the second approach, a CUDA kernel was written to do the similarity calculation, while the sorting is done through `stable_sort_by_key` from *Thrust* library as in the training part. Moreover, it is important to note that the parallel prediction algorithm is also used in the validation part of the training, from which some speedup is gained.

Table 1. Speedup for finding target neighbors in parallel for *Waveform* and *Letter* datasets

Dataset	Waveform	Letter
Finding target neighbors in advance	8.5	80

Table 2. Overall speedup results obtained for *Waveform* and *Letter* datasets

Dataset	Waveform		Letter	
	kNN-A	SkNN-A	kNN-A	SkNN-A
Training	2.8	2.8	4	3.5
Testing	8.3	8.3	27	31.4

4. EXPERIMENTAL RESULTS

Setup. The computer used to perform the experiments is an Intel Core i5-52000U CPU @ 2.20 GHz, and a NVIDIA GeForce 920M graphic card.

Performance. 3 UCI [4] datasets were used to assess the parallel version of SiLA, ranging from small to larger datasets. Datasets that were used are *Iris*, *Waveform*, and *Letter Recognition* dataset. The *Iris* dataset contains 150 instances, 3 classes and 4 attributes. Each class contains 50 instances. After 80:20 split, 120 examples were used for training whereas 30 examples for testing. *Waveform Database Generator* (Version 2) dataset has 5000 examples, 40 attributes and 3 classes. After performing the 80-20 split, 4000 examples were used for training and validation purposes, and 1000 examples for testing. *Letter Recognition* dataset consists of 20000 instances, 16 attributes and 26 classes. 16000 examples were used for training and validation while 4000 examples were employed for testing.
In all of the experiments, the number of iterations is 1, because with an increase in the iteration there is a uniform increase in the execution time, thus leading to the same speedup between two approaches as for 1 iteration. In the Iris dataset, there is no speedup for the parallel version. In contrast, there is a slowdown as a result of memory transfer overhead. Before the learning process is started, the target neighbors are found in advance. Table 1 shows the resulting speedup of parallel *SiLA* for *Waveform* and *Letter* datasets.
While observing separately the training and testing phases, the following results were obtained as depicted in Figure 1, Figure 2, Figure 3, and Figure 4. Table 2 shows the resulting speedup for training and testing, respectively, and Table 3 shows its associated accuracy. Note that during the training, the comparison of one example is done with all training set (e.g. 16000 in the case of *Letter* dataset). As the number of examples in the dataset is increased, more work needs to be done, and more data gets transferred between CPU and GPU. It is pertinent to mention that there is just a slight difference between the parallel algorithm with f_{ml} sequential and parallel algorithm with f_{ml} parallel. Therefore, the main speedup comes from

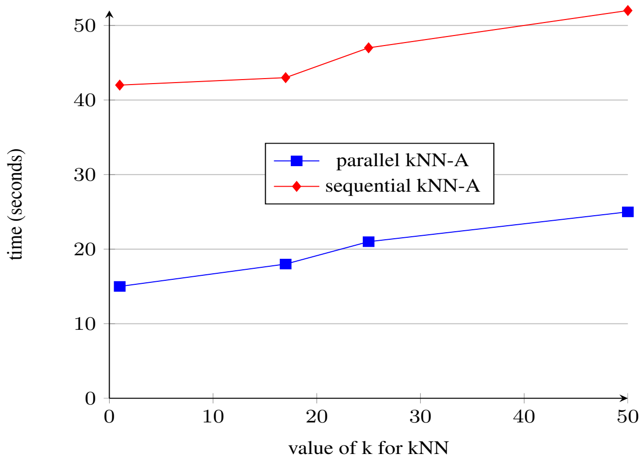

Fig. 1. Comparison of execution time between sequential and parallel kNN-A for training on *Waveform* (fold 1.0)

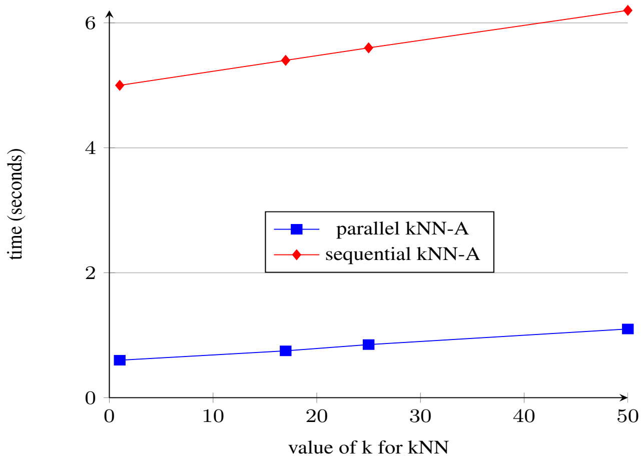

Fig. 2. Comparison of execution time between sequential and parallel kNN-A for testing *Waveform* (fold 1.0)

Table 3. Accuracy results for *kNN-A* rule when testing was done for 1 nearest neighbor

Dataset	Waveform		Letter	
	kNN-A	SkNN-A	kNN-A	SkNN-A
SiLA	0.792	0.792	0.959	0.959
pSiLA	0.792	0.792	0.959	0.959

finding the impostors in parallel and the validation part of training. Comparing the accuracy for *SiLA* and its parallel variant while the number of iterations is 1, results similar to sequential *SiLA* were observed while using its parallel version as shown in Table 3.

5. CONCLUSION AND FUTURE WORK

In this paper, a parallel version of *SiLA* algorithm is proposed. Experimental results have demonstrated that the proposed algorithm has faster execution time than the traditional one (excluding smaller datasets). It is important to note that the algorithm performs better on testing than on training because of the algorithm change, since the new algorithm needs to do more work than the traditional

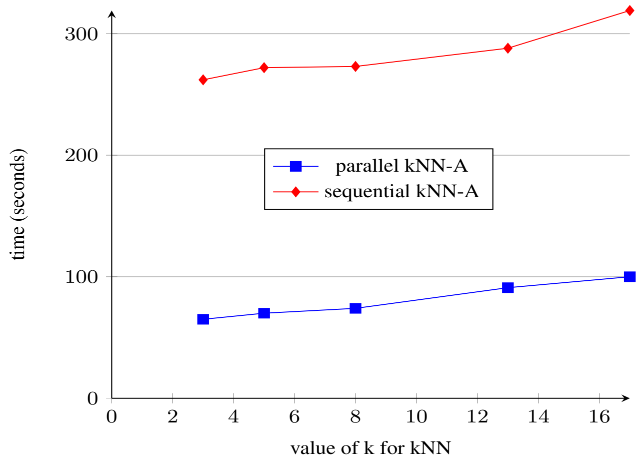

Fig. 3. Comparison of execution time between sequential and parallel kNN-A for training on *Letter* (fold 1.0)

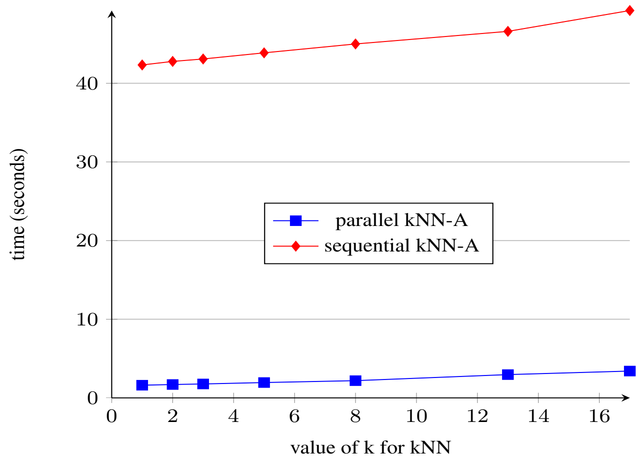

Fig. 4. Comparison of execution time between sequential and parallel kNN-A for testing *Letter* (fold 1.0)

one during training. In the future, larger data sets from UCI repository, or otherwise, could be added. It is expected that the accuracy will improve compared to the sequential algorithm as the number of iterations could be increased which can also be validated using more experiments. The algorithm could also be applied on bigger datasets of different type e.g. churn prediction, healthcare data etc.

6. REFERENCES

[1] Aurélien Bellet, Amaury Habrard, and Marc Sebban. Good edit similarity learning by loss minimization. *Machine Learning*, 89(1-2):5–35, 2012.

[2] Gal Chechik, Varun Sharma, Uri Shalit, and Samy Bengio. Large scale online learning of image similarity through ranking. *Journal of Machine Learning Research*, 11(Mar):1109–1135, 2010.

[3] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[4] D Dua and E Karra Taniskidou. UCI machine learning repository [http://archive. ics. uci. edu/ml]. 2017.

[5] Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

[6] Jared Hoberock and Nathan Bell. Thrust–parallel algorithms library. *Available: thrust. github. io*, 2015.

[7] Sofia Ira Ktena, Sarah Parisot, Enzo Ferrante, Martin Rajchl, Matthew Lee, Ben Glocker, and Daniel Rueckert. Distance metric learning using graph convolutional networks: Application to functional brain networks. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 469–477. Springer, 2017.

[8] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. In *Proceedings. The 2009 International Symposium on Computer Science and Computational Technology (ISCSCI 2009)*, page 151. Citeseer, 2009.

[9] Hamidreza Mohebbi, Yang Mu, and Wei Ding. Learning weighted distance metric from group level information and its parallel implementation. *Applied Intelligence*, 46(1):180–196, 2017.

[10] Ali Mustafa Qamar. *Generalized cosine and similarity metrics: a supervised learning approach based on nearest neighbors*. PhD thesis, Université de Grenoble, 2010.

[11] Ali Mustafa Qamar and Eric Gaussier. Online and batch learning of generalized cosine similarities. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 926–931. IEEE, 2009.

[12] Ali Mustafa Qamar, Eric Gaussier, Jean-Pierre Chevallet, and Joo Hwee Lim. Similarity learning for nearest neighbor classification. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 983–988. IEEE, 2008.

[13] Yuxin Su, Haiqin Yang, Irwin King, and Michael Lyu. Distributed information-theoretic metric learning in apache spark. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 3306–3313. IEEE, 2016.

[14] Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in neural information processing systems*, pages 1473–1480, 2006.

[15] Eric P Xing, Michael I Jordan, Stuart J Russell, and Andrew Y Ng. Distance metric learning with application to clustering with side-information. In *Advances in neural information processing systems*, pages 521–528, 2003.