**A KNN Optimization Based on GPU Parallel** **Computing Method**

Olusesi Balogun Varaprasad Kurra Sravanthi Malepati

**Abstract** — K Nearest neighbor algorithm is a simple and practical classification algorithm, but it is less efficient when handling high-dimensional data. Parallel computation is an effective way to perform huge data calculations. To improve the efficiency of KNN, an optimization method based on GPU parallel computation is proposed. Our approach involves a series of steps to perform KNN at its best. Firstly, in the distance calculation phase, the performance is increased to the numerical value, and the independent components are calculated in parallel before the thread collaborative computation. Secondly in the distance sorting stage, a method of judging order is proposed. This method based on shared memory gives sorting the parallel ability to determine whether the sequence is ordered. The results that we got in this research will show that the proposed method can obviously improve the execution efficiency of KNN algorithm in high-dimensional samples.

**Index Terms** — ***KNN, GPU, parallel optimization, CUDA***

——————  ——————

# 1Introduction

The project we worked on focused on Optimization of KNN based on GPU parallel computing method. KNN is a simple and practical classification algorithm widely used in local linear embedding dimension reduction [3], sample clustering [4] and personalized recommendation [5], but it is less efficient when dealing with massive high-dimensional data because its time complexity increases exponentially and its efficiency becomes a bottleneck for large-scale application with high number and dimension of samples. In recent years, reduction in the training time of the model, higher standards and requirements for algorithm efficiency and performance has gradually become a research hotspot. Parallelism has helped in providing some solutions to these challenges. The advent of high-performance computing (HPC) and graphic processing unit (GPU) provides enough resources for parallel computing [1] GPUs have enabled inexpensive high-performance computing for general-purpose applications.

In order to improve the efficiency of KNN algorithm, several methods have been used. These include matrix operations, proposed by T. Roweis et al, used to calculate distance and avoid cyclic calculation but there it has a limit of efficiency. Also, CUDAKNN method, proposed by S.S Liang, first finds the K nearest distances in the local block and then global K-nearest neighbor ordering is performed on the obtained results. However, the adaptability to change K is insufficient. Other methods include pre-screening method, proposed by P. Tianet al which was based on distance filtering to reduce the number of samples [8].

Similarly, P. L. Yang et al. proposed a parallel optimization method based on OpenCL, which utilizes fine-grained parallelization method to obtain distance, performs Bitonic sorting with optimized thread and memory model [9]. However, there is a preprocessing problem for sequence length.

Considering the paper, in our project, we looked at KNN parallel computing optimization based on CUDA to improve the efficiency of KNN algorithm. This method firstly increases the calculation parallelism of distance from the sample to the numerical value and then calculates the corresponding component result before calculating the distance. Secondly, a parallel judgment method is proposed in the distance sorting stage phase. The thread shared memory is used to make an orderly judgment on each round of sorting results. This orderly judgment is used to end the calculation of the basic ordered sequence ahead of time.

This paper proposes a KNN parallel computing optimization based on CUDA to improve the efficiency of KNN algorithm. This method firstly increases the calculation parallelism of distance from the sample to the numerical value and then calculates the corresponding component result before calculating the distance. Secondly, a parallel judgment method is proposed in the distance sorting stage phase. The thread shared memory is used to make an orderly judgment on each round of sorting results. This orderly judgment is used to end the calculation of the basic ordered sequence ahead of time.

# 2 Related Works

## CUDA in Brief

[CUDA](https://developer.nvidia.com/about-cuda)is a parallel computing platform and programming model developed by Nvidia for general computing on its own GPUs (graphics processing units). CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.

While there have been other proposed APIs for GPUs, such as [OpenCL](https://www.khronos.org/opencl/), and there are competitive GPUs from other companies, the combination of CUDA and Nvidia GPUs dominates several application areas, including deep learning, and is a foundation for some of the fastest computers in the world.

CUDA mainly includes the following models:

(1) **Thread model:** Threads are managed by grid, block, and thread. The grid contains blocks which contain threads. The main program identifies a thread with the unique ID through the three-dimensional index (e.g., [grid, block, thread]) and the dimension.

(2) **Memory model:** Each thread block has a shared memory. Only threads of the same block can access shared memory.

(3) **Thread synchronization:** The thread synchronization requires threads in the same block to wait others. The atomic function ensures that the variable is locked when operating.

## KNN Analysis

The k-nearest neighbor algorithm (kNN) is a widely used algorithm used for both classification and regression prediction/classification. The processing time of the kNN search remains the bottleneck in various application knowledge, especially in high dimensional dataset. In this paper, we will try to address this processing time issue by performing the kNN search using the GPU.

In theory, supposing there is a set S = {s0,s1,..,sn-1}, where si € S. The sample si can be represented as an m-dimensional vector (x0, x1, xm-1). Complex KNN algorithm (e.g., in the LLE algorithm [3]) is defined as follows. let si € S, then calculating the distance set DS(i) = {d0, d1, …, dj-1, dj+1, … dn-1} which comes from si to sj, here sj € S, sj ≠ si, i j € [1, n]. Then these n samples are calculated to obtain the distance matrix Ds. The line number indicates the index of sample sumi, the column number indicates the index of sample sumj. Dij represents the distance between sumi and sumj. This distance matrix is sorted in other to make necessary classification by KNN Algorithm.

Our goal is accelerating the calculation procedure for the KNN model well-defined above. The main computing consumption of the KNN algorithm is focused in distance calculation and distance sorting which control the computation time. As data dimension upsurges, the time increases. For example, in the distance calculation phase, Ds needs to achieve n2 distance calculations which needs to operate on m-dimensional components totally.  Thus, the total action amount is m\*n2, and the time complexity is O(mn2). Also, in the distance sorting stage, each row in Ds needs to be fully sorted. The middling time complexity of one row is O(nlogn), and thus the total sorting time difficulty is O(n2logn).

From the above, an enhanced parallel algorithm needs to be calculated to reduce the time. Although, some current parallel KNN algorithms have made some examinations, however, the paper we’re modelling describes a unique optimization way of solving this by considering the amplification effect of sample dimension in the distance calculation stage and further doing optimization for the parallel Odd-even sorting algorithm

## 3 OPTIMIZATION METHOD

**A. Calculating Distance Parallely**

The distance between two sample vectors designates how similar they are. The normally used distance is the Euclidean metric. The meaning as in (1):  x and y are sample points represented by n-dimensional vector xi and yi represent the i-th constituent. The idea of existing parallel algorithm is to implement the process which calculates distance between samples self-sufficiently. The optimization methods mainly use different memory or thread models to speed up the access. However, Howeverthe role of data dimension and the features of distance procedure are not considered.

If x and y are sample points represented by n-dimensional vector. xi and yi represent the i-th component, such that xi = (a0, a1 …, am-1), yi = (b0, b1 …, bm-1) and the square operation is denoted by the function g, then the Euclidian distance can be represented by d = f(g (xi - yi)) where d represents the spatial distance between xi and yi Therefore, r = g(xi - yi) is set as the first step of the distance calculation to obtain the m-dimensional vector r = (r0, r1 …,rm-1).

The parallelization method can be used to calculate r and the function f is implemented by other parallel method subsequently. The parallel distance calculation is thus completed by two steps and the sorting in one step. The below algorithms were used for the distance computation steps.

**1.** **Sample Distance Prefix Sum** – The built-in atomic Add function of CUDA can ensure the correctness of parallel accumulation. But when m is extremely large, the atomic add will take a lot of time. At present, the effective method of parallel summation is the prefix summation. So, we are increasing the calculation parallelism of distance from the sample to the numerical value by taking in, the corresponding distance vector r = (c0, c1 …, cm-1) and generating, a prefix sum S = {s0, s1, … sn-1}.

The built-in atomicAdd function of CUDA can ensure the correctness of parallel accumulation. But when m is extremely large, the atomic add will take a lot of time. At present, the effective method of parallel summation is the prefix summation which can be illustrated simply as: s0 = c0, s1 = s0 + s1, s2 = s0 + s1 + s2 and so forth. Where (sum0, sum1 …, summ-1) is the prefix sum of r, and sm-1 is d2 in (2). The algorithms for distance calculation are shown in below.

**Algorithm 1 Distance Prefix sum**

**Input:** distance vector ***s:*** = [*S0, S1 …, Sm-1*]

**Output:** distance sum *sm-1*

1. set *span:* = 1;

2. set *id:* = thread index

3. while: *span* < m do

4. if*id* > (*span* 1) do

5. *Sid:* = *Sid* + *S(id-span)*

6. \_\_syncthread ();

7. *span:* = *span* \*2;

8. *Summ-1:* = S*m-1*

9. end

After the successful execution of the Prefix sum it is time to investigate the Distance as a measure of matrix between the samples. We will be lookinginto a parallel algorithm that best performs the distance in the form a matrix. We name this as Distance matrix between the samples.

2. **Distance Matrix between Samples** –We generate a matrix of distances between the test data which is the key part of this algorithm and every other samples of the training data. Firstly, we compute the distance vector r and then call the Sample Distance Prefix Sum algorithm to compute distance d in parallel and lastly assign value dij to the distance matrix Ds. When calculating r = g (xi - yi), m threads (T0, T1 …, Tm-1) are enabled. Thread Ti is responsible for executing the calculated Ci = (ai - bi)2，the result ci is stored in the local memory and r = (C0, C1 …, Cm-1).

**Algorithm 2 Distance matrix between samples**

**Input**: sample set Sum = {Sum0, Sum1, … Sumn-1} and di- mension

**Output**: distance matrix Ds

1. Allocate memory and copy Sum to GPU, data size is m\*n;

2. Start n\*n\*m threads, each thread calculates the Ci in r;

3. for each thread: do begin

4. Set base as the x-dim value of the block index;

5. Set dest as the x-dim value of the thread index;

6. Set dim as the y-dim value of the thread index;

7. Read ai and bi according to index base, dest and dim;

8. Calculate a dimensional difference Ci = (ai - bi)2;

9. get distance vector r = (C0, C1 …, Cm-1);

10. use algorithm 1 to get the sum of r:

dij= ∑Si, m-1, i=0

**B. Sorting using odd even parallel sorting**

The next step after computing the distance matrix is sorting. To achieve efficient way of sorting, we implemented a parallel sort with order judgment as explained below:

**3. Parallel sort with order judgment** - A parallel order judgment method is proposed in the distance sorting phase. The thread shared memory is utilized to make an ordered judgment on each round of sorting results. This orderly judgment is used to end the calculation of the basic ordered sequence ahead of time.

***Algorithm: 3*** **Parallel odd-even sort with order judgment**

**Input**: distance matrix Ds

**Output**: ordered distance matrix Dsorted

1. Allocate memory and then copy the Ds to GPU, data size is n\*n;
2. start n\*n threads, each for one dij (i, j€ [0, n-1]) in distance matrix (Ds);
3. for each thread: do
4. set r as the x-dim value of the block index;
5. set c as the x-dim value of the thread index;
6. While(true) do
7. set the shared memory variable sorted = 1;
8. if c == odd do
9. if d (r, c) < d (r, c-1) do
10. set sorted = 0;
11. swap d (r, c) and d (r, c-1);
12. \_\_threadsync ();
13. if c is odd and is not n-1 do
14. if d (r, c) > d (r, c+1) do
15. set sorted = 0;
16. swap d (r, c) and d (r, c+1)
17. threads synchronization;
18. if sorted is 1 do
19. break the loop;
20. set Dsorted = Ds
21. end

# 4 Results

**A. Experimental Data**

1) **Fixed-dimensional data**: The data type is float and the dimension of example (m) is 3. The figures of sample (n) are 10, 20, 50, 100, 200, 500 and 1000 correspondingly.

2**) Variable-dimensional data**: The variable-point numbers are replicated arbitrarily. The model dimension (m) are 4, 8, 16, 32 and 64 respectively. The figures of sample (n) are 256, 128, 64, 32, 16 correspondingly.

**B. Results and Discussion**

***Experiment 1***: To test the competence of diverse parallel

distance procedures with constant-dimensional data.

1) The sample parallel algorithm is defined as using n\*n

threads, each thread calculates the distance between a pair of samples.

2) The numerical parallel (atomic sum) is defined as using n\*n\*m threads, each thread computes a component of the sample, then the atomicAdd is used to calculate the distance between samples.

3) The arithmetic parallel (prefix-sum) is well-defined as same as another algorithm but the prefix sum is used to estimate the distance. Tentative data 1 is used as input.

The normal time after 10 implementations is recorded. The process 2 and 3 have no result for 500 and 1000 samples due to the limitation of threads number per chunk.

In analysing the experiment results of testing the efficiency of three different parallel distance algorithms with constant-dimensional data, we derived that for low dimension of data, the sample parallel sum has a higher efficiency than numerical parallel prefix and numerical parallel atomic sums.

For example, after 10 executions, between 10 and 20 samples, the execution time variations between the three distance algorithms is insignigicant and it's about 0.02ms. As number of samples increases from 20 to 50, the execution time increases linearly, and the time variations becomes significant. At 50 samples, sample parallel, numerical parallel atomic sum and numerical parallel prefix sum have execution times of 0.021ms, 0.029ms and 0.3ms respectively. Similarly, at 200 samples, their execution times becomes 0.1ms, 0.19ms and 0.34ms respectively.

From the above analysis, it is proved that the sample parallel sum is more efficient and results into better optimization than numerical parallel prefix sum and numerical atomic sums algortihms.

***Experiment 2*:** To examine the competence of different parallel distance procedures by variable-dimensional data. The test algorithm used in this trial is the same as Experiment 1, but the input is replaced by experimental data 2. It is designed to test efficiency of the numerical parallel algorithm in different dimensional data. The calculation amount in the experiment is fixed. The average time after 10 executions is recorded.

From the experiments results, we discovered that for variable-dimension data, as the sample dimension increases, the efficiency of the numerical parallel algorithm surpasses the sample parallel algorithm.

After 10 executions, in analysiing the results, at a number\*dimesnion of 256\*4, the exceution time of sample parallel, numerical parallel atomic sum and numerical parallel prefix sums algorithms have execution times of 0.2ms, 0.4ms and 0.78ms respectively. As number\*dimension increases to 128\*8, exceution times drops to 0.09ms, 0.2ms and 0.48ms for sample parallel, numerical parallel atomic sum and numerical parallel prefix sums respectively.

However, as samples (number\*dimension) of data increases to 16\*64, the exceution times of sample parallel algorithm increases to 0.1ms while that of numerical parallel atomic sum and numerical parallel prefix sums reduces to 0.03ms and 0.13ms to respectively. This trend of reduction and increment in execution times continues as dimension of data increases for variable data.

From the above analysis, it can be inferred that for variabale dimension of data, the numerical prefix sum will be the most efficient algorithm as the dimension increases

***Experiment 3:*** To test the efficiency of different parallel distance sorting algorithms with fixed-dimensional data. The sorting algorithms used in this experiment are improved Odd-even sorting and Bitonic sorting. Each row in the distance matrix Ds, which is calculated from data 1 is sorted using the above two algorithms. The average time after 10 executions is recorded.

In analyzing results of our sorting experiments, we discovered that between 1 and 200 samples, the execution time is approximately 0.1ms for both parallel odd-even sort and parallel bitonic sort. A gradual increase in samples from 200 to 500 reults into execution time of both sorting algorithms increasing with parallel bitonic sort’s increasing more than that of parallel odd-even at a rate of 55%. However, as samples increased to 650, execution time of two algorithms crosses and an increment is obtained, henceforth, with parallel odd-even becoming more than parallel bitonic sort at a rate of 50%. The execution time of 200 and 500 samples, illustrates that if the data is basically ordered, it will get higher efficiency because of the ability to judge order in parallel.

In general, the sorting is extra efficient since it uses the half-interval method, but the odd-even categorization belongs to bubble sorting. The amount of calculation for Bitonic sorting is fixed so that the sorting result can only be obtained when the process is finished.

# 5 Conclusion And Future Work

In our project, we simulated a parallel KNN optimization method as proposed in the paper we referenced. The optimization uses numerical parallelism instead of sample to calculate distance and presents a parallel order judgment method. The improved algorithm has signifcant efficiency for high dimensional data and partially ordered data. But the practicality of this method is limited by the number of threads. When the sample size is much larger, this method cannot complete all work at one cycle. In fture, the scheduling collaboration mechanism between threads is considered to complete batch calculation.

**6 References**

1. K. Mivule, B. Harvey, C. Cobb, and H. E. Sayed, "A review of cuda, mapreduce, and pthreads parallel computing models," Eprint Arxiv, vol. 1, pp. 208-217, Oct 2014.
2. J.S. Bridle, “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition,” *Neurocomputing—Algorithms, Architectures and Applications,* F. Fogelman-Soulie and J. Herault, eds., NATO ASI Series F68, Berlin: Springer-Verlag, pp. 227-236, 1989. (Book style with paper title and editor)
3. S. Roweis, and L. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding," Science, vol. 290, pp. 2323-2326, Dec 2000.
4. S. Zhang, X. Li, M. Zong, X. Zhu, and R. Wang, "Efficient kNN Classification with Different Numbers of Nearest Neighbors," IEEE Trans. Neural Networks and Learning Systems, pp. (99)1-12, 2017.
5. N.P. Chen, "Research on an Improved Personalized Collaborative Recommendation Algorithm," Software Guide, vol. 16, pp. 45-47, 2017.
6. S. Roweis, and L. Saul, An introduction to locally linear embedding. Jan 2001, <https://cs.nyu.edu/~roweis/lle/publications.html>
7. S.S liang, C. Wang, Y. liu and L.H. Jian, "CUKNN: A Parallel Implementation of K-nearest neighbor on CUDA-Enabled GPU, " Proceedings of 2009 IEEE Youth Conference on Information. Computing and Telecommunication, 2009, pp. 437-440.
8. P. Tian, B. Hua, and L. Li, "Implementation of K-nearest Neighbor Algorithm Based on GPU," Computer Engineering, vol. 41, pp. 189- 192, Feb 2015.
9. P.L. Yang, B.M. Feng, Z.Y. Zhou, and X.H. Wen, "A parallel kNN algorithm based on OpenCL," Computer Engineering & Science, vol. 39, pp. 2198-2202, Dec 2017.
10. J. He, Y.L. Pu, H.B. Li, B. Yan, "A energy efficient parallel KNN algorithm evaluated on GPU using OpenCL," Application of Electronic Technique, vol. 42, pp. 14-16, 2016.