A parallel implementation of outlier detection over data streams

Kangqing YU
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
kangqingyu@cmail.carleton.ca

November 28, 2017

Abstract

Outlier detection has become an increasing challenging task in modern applications due to the fact that the data may come in the form of streams rather than statically as it was before. A lot of algorithms have been modified so that they can work in the streaming environment. Among all of them, a very popular technique called *sliding window* is used, which only keep a portion of streaming data in memory and detect outliers purely based on those data. A main drawback with this approach is that the decision of outliers is only based on the data in current window and historical data are simply dropped and not considered in decision making of outlier data in current window. Therefore, this method fails to address the challenge of *concerpt drift* in data streams. Another challenge in detecting outliers in data streams is that since the data may come at a very high rates and it is impossible to store all data in memory, the decision should be made in a timely manner with only one pass over data stream. This will pose a very harsh requirement in computational power and in most case, it is impossible for CPU programming to achieve. In this project, I purposed a novel solution to detect outliers in streaming environment, powered by GPU and MapReduce framework, based on a very efficient classical algorithm called LOF(Local Outlier Factor) to address the increasing challenge of outlier detection over continuous data streams. The proposed method is able to address the *concerpt drift* in data streams, as well as detecting outliers in high-dimensional, high-rates data streams and produces timely results without compromising performances. Since the LOF algorithm is very computational expensive, very few works(almost none) have been conducted to extend the algorithm to work in streaming environment. As far as I know, the solution I purposed is the first algorithm that try to modify the LOF algorithm in the streaming environment and therefore it is also the first parallel implementation of LOF algorithm in the streaming context.

1 Introduction

An outlier in a dataset is a data point that is considerably different from the rest of the data as if it is generated by a different mechanism[12]. Applications of outlier detections vary in numerous fields, including fraud detection, network intrusion detection, medical image screening, environment monitoring etc. A stream environment is where data come constantly at a high volume and may change over time. This can impose a very high requirement for computation power as decisions need to be made in real time within limited

amount of time among all data. In addition to that, since the applications do not have random access to the underlying data in the streaming environment, when building an application that process data stream, these three key characteristics of data stream need to be taken into consideration: uncertainty, transiency, and incompleteness[23]. Uncertainty means the data distribution of the model may change over the time as new data coming in a unpredictible way. This term is sometimes known as *concept drift* in some literatures. Dealing with concept drift is a main challenge in most streaming applications. Transiency means it is not possible to store the entire dataset from data stream in the memory. The data can only be accessed in one pass and when it is processed, it should be deleted from memory. Completeness assume that the data will come indefinitely and they will never stop. These all make outlier detection in data streams extremely challenging from both algorithms and hardware perspective.

To be more specific, suppose you have a data stream that keeps coming indefinitely, at one time t_i, you identify object o_k as being outlierness in the current window. And after some time W at time $ti + W$ when the whole recent history is considered, object o_k may become an inliner. And vice versa. This can be illustrate in Figure 1
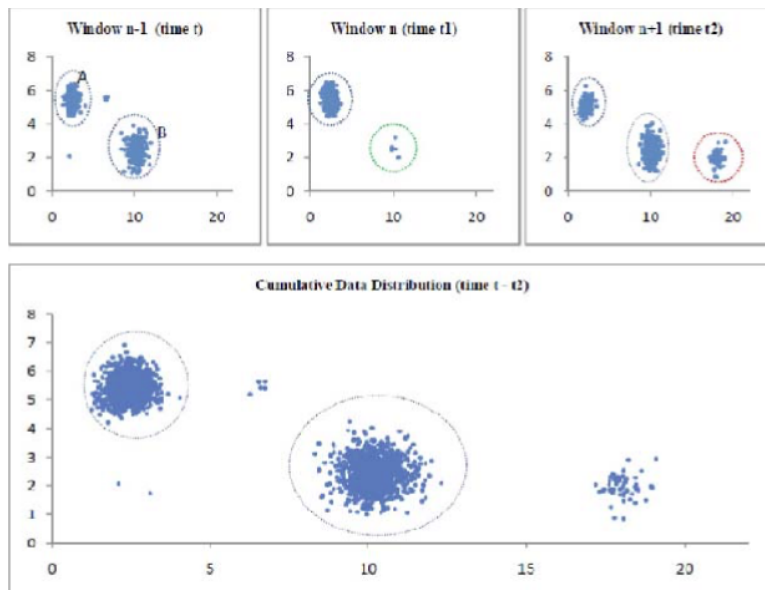

Figure 1: Evolving 2D data stream

In this project, I developed an novel, modified version of *Local Outlier Factor* algorithm, called Cumulative Local Outlier Factor, with the support of statistical binned summary, where an outlier decision is considered not only based the data points in the current data window, but also taking consideration of historical data without the necessity storing the entire dataset in the secondary memory. In doing so, I kept a statistical binned summary of all the observed data to help making decisions on outlier for data points within current processing window, and gradually fade away the impact of the obsolete data when making decisions on current data. Thus the proposed algorithm should provide a more accurate results compared to the widely used sliding window approach.

Note that most density-based approaches in outlier detection, including LOF, despite being accurate, are notorious of being computational expensive. And therefore, it is almost

impossible to detect outliers with high volume, high speed data streams without introducing parallelism. Other computational in-expensive algorithms exists but they either need to sacrifice on the accuracy or assuming fixed distribution over the underlying data, which fails to capture the nature of *concept drift* in data streams. To address the expensive computation demands of the LOF algorithms over data streams, I employed parallelism in this project as an implementation of the proposed algorithm. To be more specific, the Cumulative Local Outlier Factor algorithm is implemented on GPU with CUDA framework to accelerate the computational time in order to provide timely results over the high input rates of streaming data and maintenance of binned summary is implemented using the MapReduce programming paradigm, which to a great extent, simplify the problem as well as leaving a lot of rooms for parallelizing.

The accuracy of the result in this method is compared with another GPU accelerated approach, SOD_GPU as proposed in [12], which is based on *kernel density estimator*. And the speedup on performance is compared with distance-based method based on sliding window, which runs in a multi-core CPU to further illustrate that LOF_GPU can practically be used in a streaming environment to detect outliers at a high rate of data volume where it is otherwise incapable to handle by CPU.

2   Literature Review

A lot of techniques have been introduced in last decades to solve the outlier detection problem. And these techniques can be briefly summarized into three different categories:

1. Supervised approaches

2. Semi-supervised approaches

3. Unsupervised approaches

Supervised approaches typically require building a prediction model for rare events based on manually labelled data(the training set), and use it to classify new event based on the learnt model[13, 14]. In other words, the outlier problem in this case becomes a classification problem where we are only interested in the minority class whose data deviate largely from the rest. The main problem with this approach is that in order to ensure accuracy, a large number of labelled data need to be generated which is unpractical in most cases.

Compared to supervised approaches, Semi-supervised approaches[6, 26] only require a small number of training data with some unlabeled data to obtain better predictions. One approach introduced by Jing Gao et all.[11] takes advantage of K-mean clustering in unsupervised learning by adding penalties to the objective function for mislabelled data points and optimize the overall objective function.

Although some of those techniques may generate very promising results, they work well only in static data and typically don't fit into the context of dynamic streaming environment. In other words, both supervised and semi-supervised methods will assume that they will have *random access* to the underlying data while this is not possible for streaming data when you can only have portion of it at one time and they also fails to address the problem of the potential change of data distribution.

In contrast, unsupervised learning methods don't require labelled input and typically don't assume a fixed data distribution as the model can be dynamically built based on

variations of data. Many best-known techniques of outlier detection fall into this category and based on the context, they can mostly be classified into two categories: **Unsupervised outlier detection on static data** and **Unsupervised outlier detection on streaming data**.

Distance-based approach was among the very first outlier detection method introduced by Knorr and Ng[15] that detect outliers in a static environment. It calculates the pair-wise Euclidian Distance between all data and if one data point has less than k neighbors within distance R, it is considered an outlier. There are some variants of the static distance-based approaches, and their ideas are quite similar. For instance, Ramaswamy et all.[22] purposed a method where an outlier is defined by considering the total number of objects whose distance to its kth nearest neighbor is smaller than itself. Angiulli and Pizzuti[5] introduced a method where an outlier is defined by taking into account of the sum of the distances from $1^{s}t$ up to the kth nearest neighbors. These methods are sometimes referred as KNN and it should be noted that it is different from the term KNN in supervised machine learning.

Density-based approach is another way to detect outlier on static data. The basic idea is to assign a degree of being outlier(a score) based on the density of local neighbourhood, given some predefined restrictions. A popular example of this approach is Local Outlier Factor(LOF) algorithm[7], which is what this proposed algorithm is based on, use the concept called *reachability* to coin the density of data point. Another popular density-based outlier detection method is called LOCI(Local Correlation Integral)[20]

There are some other types of outlier detection techniques on static data. *Statistics approach*, which is based on the probability theory and normally models the underlying data using a stochastic distribution(e.g. Gaussian distribution). One of the most popular one used is *auto-regression* model or sometimes being referred as Gaussian mixture model[8]; *Deviation approach*, which is first introduced by Arning et al.[1], where an outlier is detected if feature space of one data point deviates largely from other data points; Another recent technique introduced by Harkins et al.[2] takes advantage of replicator neutral network(RNN) to detect outliers. There might be other techniques used for unsupervised outlier detection but due to the limitation of this paper, I can only list some of them. The ones mentioned above are those best-known so far to detect outliers statically.

As modern applications have an increasing demands to process streaming data in real-time, a lot of these static methods mentioned before have been extended to work in the dynamic streaming environments. The all based on the same ideas in static approaches but algorithms have been modified in an incremental fashion to address the **concept drift** of the data stream properties.

Distance-based outlier detection approach was among the first which start to apply the method in the streaming context. In the last decade, there are several studies which focus on *distance-based outlier detection in data streams(DODDS)*. Due to the fact that the distance-based require random access on the data and this is not possible with stream data, *sliding window* technique was introduced which only keep a number of active objects in current window. When objects expire, they are deleted from memory as new object comes in. There are mainly two window models in data streams: count-based window and time-based window

Numerous algorithms have been invented to process stream data using sliding window on outlier detection. And based on the benchmark among all DODDS algorithms given by Luan Tran et al.[25], the MCOD algorithm introduced by M.Kontaki et al.[16] seems to have the most satisfying performance. Its basic idea is to pre-compute the *safe inliers*

that have more than k neighbors which arrived after p_i by using an event queue, which can reduce greatly on space complexity. Because the neighbors which arrives before p_i may expire, by declaring the neighbors which arrived after p_i to be larger than k, we can safely mark p_i as inlier. The time complexity of this algorithm is guaranteed to be $O(n \log k)$ while maintaining the space complexity to be $O(nk)$

Most of DODDS algorithms are based on the original definition of distance-based technique given by Knorr and Ng[15]. The other distance-based techniques in outlier detection such as KNN remain unsolved in the streaming context. It would be interested to see if those methods can be extended to work in the data streaming context. Since all DODDS methods only have access to only a portion of data, they all lack a global view of the entire dataset and in most cases, there accuracies are not guaranteed.

Clustering is another technique to outlier detection on stream data. Since clustering is a technique in unsupervised machine learning, it inspired the ideas of outlier detection in streaming environment. Two main algorithms exists for clustering-based approaches. One of them is called *K-Mean clustering*[10], which also use the idea of sliding window and cluster the data in each window. But different than the distance based approach, the detected outliers are not reported immediately but rather be considered as a *candidate outlier*. A metric which measure the mean value of each clusters is maintained and carried over to next window in stream to further compare with data in other chunks. If the candidate outlier passed a given number of windows, it is then identified as *real outlier*. Compared to K-Mean clustering, *K-Median clustering*[9] clusters each chunk of data into variable number of clusters(from k to $k log(n)$), and it passes the weighted medians found in current window into next one for testing outlierness rather than the mean and candidate outliers. Both of these two approaches will require users' input of value k but K-Median clustering theoretically is better since the number of clusters is not fixed.

To address this problem of storage and users' input parameters, an ideal method need to find a efficient way to efficiently mine its historical data or gradually fade old data away without users' intervention. An technique inspired from *sensor network* is mentioned in[24], where it use a *kernel density estimator* to model the distribution of the sensor data. When used for outlier detection, the number of neighbors of a given data point p_i is estimated by the distribution function $f(p_i)$. In [21], D. Pokrajac et al illustrated that the LOF algorithm can be applied incrementally and the insertion of new data as well as deletion of obsolete data does not depend on the total number of N in dataset and therefore the time complexity of incremental LOF algorithm can theoretically be $O(N \log N)$. These two methods are both based on computing of the densities of local neighbourhood. However, there are still a lot of noise around these algorithms and many researchers argue that it is still computational expensive in practice. Also the LOF approach proposed by D. Pokrajac et al will need to store the entire dataset, which is not applicable in the streaming context.

Even though there are many studies in the last decade that focusing on detecting outlier online in data stream, only a few tried to solve this problem using a parallel implementation. In [12] C. HewaNadungodage et al.implemented a so-called SOD_GPU[1] algorithm which is based on kernel density estimator powered by GPU to effectively detect outlier in continuous data streams. The results seems very promising as it is 20X faster compared to a multicore CPU implementation and even a higher accuracy rate compared to the sliding window approach. And this is the only work, I found by the time of writing, which tried to solve the outlier detection problem in streaming environment using parallel computing approach.

---

[1]Stream Outlier Detector-GPU

Other parallel implementations for outlier detection exist but they only work in a static fashion, in which case, it does not seem to be quite necessary. In [4], Angiulli et al. proposed a distance-based KNN algorithm powered by GPU and similarly, Matsumoto and Hung[19] introduced a GPU-accelerated approach to detect the outliers in uncertain data. Another GPU-accelerated approach to detect outlier in static data was purposed in [3], where the LOF(Local outlier factor) algorithm is used. Anna Koufakou et all.[18] developed a parallel outlier detection strategy based on *Attribute Value Frequency(AVF)*[17][2] algorithm using MapReduce programming paradigm. Other incremental parallel methods exist for online outlier detection but they either require storing the entire history of data stream or giving a set of user-defined parameters, which is hard to define in most cases.

3 Proposed Approach

Unlike the classical LOF algorithm, the proposed LOF_GPU method proposed here is capable of processing outlier in data stream $S = X_1, X_2, X_3, ..., Xn, ...$, where each data point S_i can be arbitrary d-dimension such that $X_i =< x_{i1}, x_{i2}, ..., x_{id} >$ without storing all observed data. This is achieved by keeping a binned summary of all processed data and these summary statistics is used to help decide the Local Outlier Factor of future data points. Specifically, the binned summary acts as *virtual data points*, whose **local reachability density(lrd)**(explained later) is pre-calculated based on the frequencies of previous data that fall into that bin, and need not calculate its own Local Outlier Factor. The full details of this algorithm is explained in following:

3.1 Local Outlier Factor

The main idea of LOF algorithm is to assign each data point a degree(score) of being outlier. And this degree(score) is called *Local Outlier Factor(LOF)* of this data point. This metrics measure the density of a data point compared with its neighbourhood(K-nearest neighbor). The computing of LOFs for all data points typically comprise of following steps[7]:

1. For each data point p, compute its **k-distance(p)** as distance to its k-th nearest neighbor of p

2. For each data point p, find its **k-distance-neighbourhood** of p, which contains every object o whose distance to p, noted as $d(o, p)$ is not greater than **k-distance(p)**

3. For each data point q in the **k-distance-neighbourhood** of p, calculate its reachability distance with respect to data record p as follows:

$$\textbf{reach-distk(p,q)} = max(d(p,q), k - distance(q)) \tag{1}$$

4. For each data point p, calculate its **local reachability density(lrd)** of q as inverse of the average reachability distance over k-nearest neighbor of p:

$$\textbf{lrd(p)} = \frac{1}{\sum\limits_{k \in knn(p)} reach - distk(p,q)/k} \tag{2}$$

---

[2]Note that the AVF algorithm can only detect outlier in categorical data.

5. And finally, for each data point p, calcuate its **LOF** as ratio of average **lrd** over k-nearest neighbor of p and **lrd** of p it self

$$\textbf{LOF(p)} = \frac{\frac{1}{k}\sum_{k\in knn(p)} lrd(p)}{lrd(p)} \tag{3}$$

The outlierness is detected once the LOF of point p is greater than a pre-defined threshold θ. To avoid generating a large amount of false positives, we dynamic update the value of θ based on the average of LOF over all observed data since outliers are minorities and rarely seen.

$$\theta = \frac{1}{N}\sum_{p=1}^{N} LOF(p)$$

To extend the LOF algorithm in context of data stream, I borrowed the idea of *sliding window* from DODDS(distance-based outlier detection in data stream) and maintain chunks of data points in memory. But different from DODDS which take no consideration of historical data, I apply the LOF algorithm over data points in current window plus the virtual data points from the binned statistical summary(explained shortly). This will give a more accurate estimate of outliers since some synopsis of previous data points is maintained.

3.2 Maintaining Binned Statistical Summary

Since it is not possible to store all data in the streaming environment, a binned statistical summary is maintained in order to mine the previous observed data. The method of maintaining binned statistical summary is inspired from the literature in [12]. But different than that method where an mean value vector and standard deviation over the entire dataset need to be maintained, the method I proposed only need to maintain the aggregated bin count(C_j) and aggregated mean value(M_j) vector for each bin. The following sections describe how this is achieved:

Assume there are N data point and each consists of d dimensions. For each dimension, we calculate the upper bound and lower bound in order to derive the length of that dimension. And for each of the dimension length, we divide it by a pre-defined value k to get its width, Δ. Therefore:

$$\Delta = \sum_{j=0}^{d} [max(x_j) - min(x_j)]/k$$

To find the corresponding bin index(where this data point belongs to) for each data point x_i. Firstly, convert the input values in each dimension of x_{ij} into interval [0, 1] using the following function:

$$x_{ij} = \frac{x_{ij} - min(x_j)}{max(x_j) - min(x_j)}$$

Then, encode the data point x_i as:

$$< I_{i1}, I_{i2}, I_{i3}, ..., I_{id} >$$

7

where $I_{ij} = x_{ij}/\Delta$. And after that, use the following formula to find the corresponding bin index for data point X_i

$$B_{X_i} = (I_{id} - 1)k^{d-1} + (I_{i(d-1)} - 1)k^{d-2} + ... + (I_{i2} - 1)k + I_{i1} \qquad (4)$$

For each bin, we maintain the count number of all data points that fall into this bin(C_j) and the mean value vector(M_j) that correspond to the mean value of this bin across all dimensions d. In addition, the upper bound and lower bound values also need to be maintained in order to derive the width of all data in each dimension. These bins can serve as *virtual points* that are used when deciding the LOF for future window of data points.

3.3 Update the binned summary

Once we calculate the bin index for each data point in current window n, we can now proceed to update the binned summary which based on previous window $n - 1$. One thing to note is that the update of binned summary can happen concurrently when calculating the LOF factors for data points in current window and they should not interfere with each other. After calculating the binned summary, the data in current window can be safely discarded and make space for future data. The bin is kept in memory throughout the application and it keeps a historical summary of all the data points that have observed. It plays a key role in deciding the future outliers and therefore it need to be updated constantly to keep up with the change of patterns in outliers(concept drifts). The following describe the procedure on how binned summary is updated:

Let C_j^{n-1} denotes the number of data points , and M_j^{n-1} denotes the mean value vector of bin B_j, at the time of which, we have observed $n - 1$ windows of data. c_j^n denotes the number of data points in bin B_j in window n and $\mu_{\mathbf{j}}^{\mathbf{n}}$ denote the mean value vector in bin B_j in window n. To update the value C_j^{n-1} and M_j^{n-1}, we first look into the number of data points in all bins of current window. If the number of data points in a particular bin(c_j^n) in the current window is significantly smaller than the average number of data points(c_{avg}^n) in other non-empty bins in the current window, we update its C_j^{n-1} as follows, where $0 < \alpha < 1$ and $0 < \delta < 1$

$$C_j^{n-1} = \alpha \times C_j^{n-1} if(c_j^n << c_{avg}^n * \delta)$$

This is necessary in order to keep the binned summary update to data with the latest change of distribution from data streams.

Then update C_j^n, M_j^n as follows:

$$C_j^n = c_j^n + C_j^{n-1}$$

$$M_j^n = \frac{(c_j^n \times \mu_{\mathbf{j}}^{\mathbf{n}} + C_j^{n-1} \times M_j^{n-1})}{c_j^n + C_j^{n-1}}$$

Note that the number of binned item may grow exponential as number of dimension grows, and it is impossible to keep such large number of bins(array) in memory. Instead, we only keep a non-empty bins in memory. In fact, most of the bins are empty and the number of non-empty bins should be much smaller than the number of k^d.
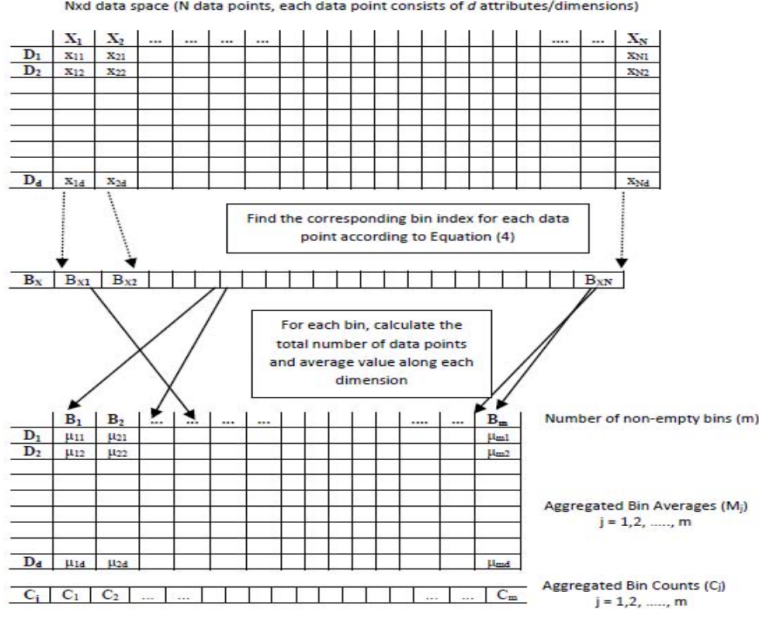
Figure 2: How to update the binned summary

3.4 Cumulative Local Outlier Factor

When the binned statistical summary is calculated, it can be used together with the data points in the next window to decide the outliers in that window. The algorithm to calculate the Local Outlier Factor for new data points based on binned summary is as follows:

Algorithm 1 Cumulative LOF:
(1) Combine the data points X with the virtual points in binned summary B, noted as $X + B$
(2) Calculate the k-distance for each pair of data points within $X + B$
(3) Calculate the k nearest neighbourhood, but this time only for data points X within all data points $X + B$
(4) Calculate the lrd values for data points in X, noted as Lrd, with the k-distance information obtained in step 2 and the given virtual points in binned summary B
(5) Calculate the syntactic lrd values for virtual data from binned summary B, noted as $vLrd$ based on equation 5
(6) Combine the lrd values from both data points X and binned summary B, noted as $Lrd + vLrd$ and calculate the LOF values for data points in X
based on $Lrd + vLrd$
— End of Algorithm —

Note that virtual points from binned summary are only presented to help calculating LOF values for future data. Hence, they do not need to calculate LOF values of their own, nor their k-distance-neighbourhood. But they do need to calculate their k-distances and lrd values since these are the required information in order to calculate the LOF values of their neighbourhood, which can be a real data point rather than virtual ones. The mean vector M_j can serve as feature vector X_j for virtual point from bin B_j and therfore, its k-distance

9

as well as its distance to any data point can be calculated. The bin count C_j value can be used to simulate the lrd value of the virtual point from bin B_j. Since the local reachability density(lrd) measure the crowding of point p within its k-distanced objects, the higher the lrd is, the more crowed it is within the region of p. This imply that the larger the data points fall within that region of p, the higher the lrd value should be. Therefore, the lrd value of virtual point from B_j can be given as:

$$lrd(B_j) = \alpha \cdot logC_j \tag{5}$$

where α is a predefined parameter.

4   Parallel implementation

As we can see from the definition of LOF algorithm in 3.1, it is very computational consuming since It evolves first getting the k nearest neighbors for each data point in dataset. If we use the brute-force approach to find the neighbourhood, the time complexity is $O(N^2)$ since it needs to calculate the distance matrix of size N. This can also grow as the dimension of data is increased. The same can be true when calculating the statistical binned summary and update it based on the bin index. When the data is coming at a very high rates with each data point having a very large dimension, this algorithm will probably not be able to give results in a timely manner and will hit the bottleneck.

To avoid this bottleneck, I implemented a parallel-based approach, taking advantages of both NVIDIA CUDA programming platform and the MapReduce programming paradigm to fully accelerate this framework. Specifically, the CUDA framework is used to calculate the LOF values in current window based on the proposed cumulative LOF algorithm in GPU and the update of statistical binned summary is implemented using MapReduce which can potentially run in a cluster. These two steps can be achieved simultaneously, which gives another level of parallelism.

For the calculation of LOF values, the most expensive part of the algorithm is on the calculation of KNN. Specifically, it includes the calculation of distance matrix on query points and reference points and a sorting algorithm for each query points. In other words, if we have a query points of size N, and reference points of size R, we will get a $N \times P$ dimension of distance matrix and sort the distances across each columns. GPU programming follows SIMD(Single Instruction Multiple Data) architecture and the performance of the GPU code depends really on how to manage memory access and threads resources efficiently. NVIDIA CUDA support something called **shared memory**, which means all threads within a block can share a same memory space and avoid unnecessary data transfer between global memory to each individual threads' local memory. To further take the advantages of coalesces memory access in GPU, the input matrices are converted to a column-major format$(d \times N)$ and each thread block will contain a multiple of 32 threads since threads in NVIDIA GPUs are executed in warp, which is a set of successive 32 scheduled threads together. This can be taken a great advantages of to give a significant boost in our KNN algorithm as well as the following algorithms to calculate the lrd and LOF values for each data.

For the update of statistical binned summary, I basically divide the algorithm into two steps. The Map step and the Reduce step. And depending on whether the data point is from the current window or it is a virtual one from the binned summary, they are treated differently. Therefore, the MapReduce framework fit naturally to this problem and it is also a parallel programming paradigm by its nature.

The following section explains this in details:

4.1 Calculate cumulative Local Outlier Factor with GPU acceleration

The CUDA program consists of a host program, which runs on CPU and a device program which is executed on GPU. The device program is written using a so-called *kernel functions*. Each kernel function is run by thousands of threads concurrently. Threads are organized in terms of block and grids. Each block contain multiple threads and they each have a unique *threadIdx(id)* within the block. Each grid contain multiple blocks and they each have a unique *blockIdx(id)* within the grid. Both *threadIdx* and *blockIdx* can be identified using 1D, 2D, or 3D indeces. This give a very nature way to process mathematical data such as matrix, vector etc. In order to let GPU process the input data, they need to be transferred from host memory to device memory first. To fully utilize the power of coalesced memory access in CUDA, some input matrices are transformed in the column-major fashion($d \times N$ matrix). Depending on the number of data and dimensions of data point, we can adjust the number of threads per blocks and number of blocks per grids to fully utilize resources power on GPU.

Based on the definition of LOF described in 3.1 and binned statistical summary described in 3.1, I have defined the following host and kernel functions to process data in each window:

Inputs: All data points from current window X^n, bin count C^{n-1}, mean value vector M^{n-1} from previous window

Outputs: LOF values for all data points from current window LOF^n

- Kernel 1: Compute the distances(Euclidian) matrix T between $X^n + M^{n-1}$ and itself(query points = reference points)

- Kernel 2: Sort the table T column wise. Get the kth row(k-distances)

- Kernel 3: Sort the sub-table $T[:, 0 : N]$ column wise, get the indices range from $0 - k$(k-nearest neighbors)

- Kernel 4: Compute the lrd(local reachability density) for X, based on equation 2

- Host function 1: Compute the virtual lrd for M^{n-1}, based on equation 5

- Kernel 5: Compute the LOF(local outlier factors) for N, with lrd appended by virtual lrd as reference points.

- Host function 2: Update the average LOF values α as threshold.

While processing the values for the LOF, the computing and updating of binned summary can execute simultanously since they do not rely no each other, as long as the input of the binned summary for computing LOF is from the previous window. For sorting which takes places in Kernel 2 and Kernel 3, I use a built-in library called cuBLAS[3], which implement RadixSort algorithm on GPU. Since sorting on each query point does not depend on each other, I assigned the sorting task on each column in T to different *CUDA streams* to concurrently run the sorting kernels. Other kernels are implemented from scratch, which is described in details in the following:

---

[3]https://developer.nvidia.com/cublas

1. *Kernel 1 - Calculate the distance matrix*

   Before calculate the distance matrix, the data points X need to be merged with the mean value vector M from non-empty binned summary of previous window. To take the advantage of coalesces memory access in GPU, the query matrix need to be transposed to column-major format of dimension $d \times N'$ where N' is the total number of data points in current window and the number of non-empty bins from previous window. The reference matrix remain the same as row-major, which has dimension of $N' \times d$. The details of the structure is explained in Figure 3 Since the query and reference points are the same in this case, the goal of Kernel 1 is to calculate any pair-wise combination of the distances within size of N' data points and to achieve this, $N' \times N'$ computation is needed. The result of this computation will be a $N' \times N'$ dimension matrix. Hence, I divided the computation into $B \times B$ dimension blocks, each consisting of $T \times T$ dimension threads., where $B = N'/T$ and T must be a multiple of size 32 to achieve the coalesces memory access. Additionally, the size of matrix N' and dimension of data point d needs to be rounded(padded) to the multiple of warp size of 32 to achieve further more coalescing.
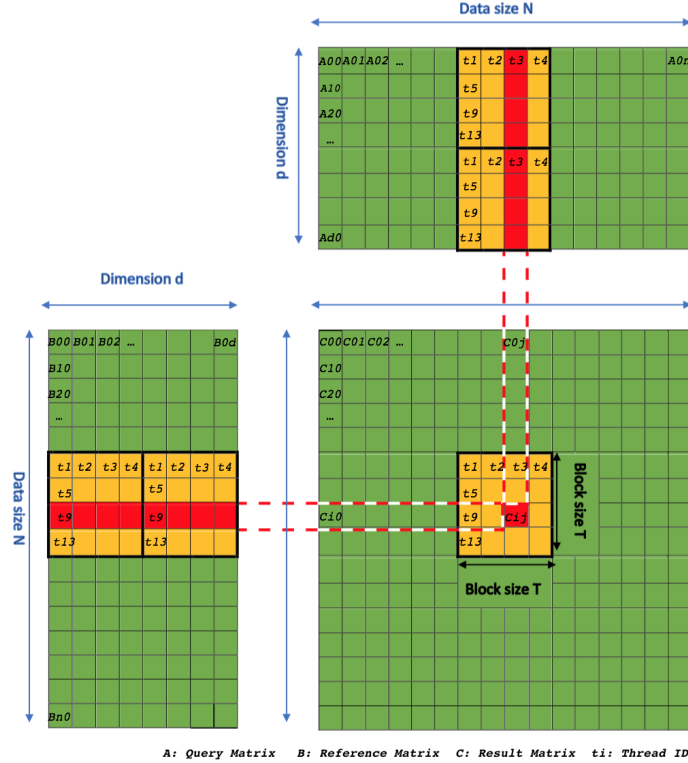

Figure 3: To achieve the coalesces memory access, the query matrix is converted to column-major format and the block size T as well as input data size N need to be multiple of the warp size 32

   Since each data point from N' is access multiple times during the calculation and in GPU, retrieving the same data from global memory is a very expensive operation, we therefore divide the $d \times N'$ matrix into size of $d \times T$ sub-matrices and copy two of them each time(one from query points, the other from reference points) to the

block-wise *shared memory* to get the partial squared matrix of size $T \times T$ in the final result matrix of size $N' \times N'$. Then, we perform the distance calculations among all possible combinations in these two sub-matrices, one from query points, the other from reference points, across all dimensions. The result will be partial distances between data points of size T from query points and data points of size T from reference points. The is repeated among all thread blocks to get the final distance matrix.

2. *Kernel 3 - Local Reachability Density Estimation*

   The computation of lrd depends on the KNN indices achieved in Kernel 3 and the K-distances achieved in Kernel 2 as well as the total data points from current window and the mean value vector from non-empty binned summary of previous window, $X + M$. KNN is a $N \times K$ dimension matrix, where N is the number of data point and K is the number of selected neighbourhood. Based on the definition of lrd in 2, we need to calculate the average reachability distances among all K neighbors from all data points in X. This involve a total number of $N \times K$ calculations. To take the advantage of the *shared memory* in thread block in order to calculate the average of reachability distances among K instances, in Kernel 3, the thread layout has been organized as N blocks, with K threads for each. The KNN matrix should be padded to multiple of 32 for each dimension in order to achieve coalesced memory access.

   For each thread in a block, it first loads centroid point p based on the block index, for which we want to calculate the lrd value of, into shared memory since it needs to be loaded by other threads in the block as well. Then, it loads the corresponding neighbor data p as well as its k-distance by the thread index in order to calculate the reachability distance between data p and q, based on the equation 1. After all threads in a block finishing loading and calculating their reachability distances, all threads are synchronized and the inverse of average reachability distances in this block is calculated, which is the lrd value for the data point given by this block index

3. *Kernel 4 - Local Outlier Factor Estimation*

   After getting the lrd values for each query data points from the window data X and mean vector value M, the calculation of LOF values are very similar to that of lrd in Kernel 3. We still need the KNN matrix, but this time, we only need the lrd values calculated from previous step as reference data points. Still, the kernel will require $N \times K$ calculations and therefore, it have N thread block with K threads in each. The KNN matrix should also be padded to be in the dimension of multiple of 32. As for the shared memory, we need to store the lrd value for the current query point as well as the lrd values for all its neighbors into shared memory to speedup the data accessing in final step. And finally when all data is loaded, the LOF value can be calculated based on 3

4.2 Update the binned summary with MapReduce

Instead of using GPU to update the binned summary, it is updated with MapReduce(ideally in a cluster). The reason is simply because of the curse of dimension problem as I mentioned in 3.3. More specifically, it would be physically impossible to allocate an array with dimension k^d. Even if it does, it would incur a lot of bandwidth to transfer this huge array back and force between host and device memory. And the length of bins needed to store data points in each window is unknown until they have been processed. Therefore,

using GPU to compute the binned summary would not be beneficial and most likely to be disadvantageous.

On the other hand, by decomposing the steps of updating binned summary into the corresponding Map and Reduce stages, it becomes mush simpler, and it is also very easy to parallelize. The following describe what need to be calculated in each step, regarding to both actual data points in current window and virtual data points from binned summary:

Inputs: All data points from current window X^n, bin count C^{n-1} and mean value vector M^{n-1} from previous window

Outputs: The new binned summary with bin count C^n and mean value vector M^{n-1}

1. *Update upper and lower bounds* Get the minimum and maximum number for current data points in X^n across each dimensions. Update the upper and lower bounds if necessary. The bounds are necessary in order to calculate the bin index in the following step.

2. *Map stage* The Map stage focus on the calculation of the bin index. For virtual data points from the binned summary j, their bin indices are already calculated. All we need to do is just to emit it with the bin count and the mean value vector of this bin, given as $(j, C_j^{n-1}, M_j^{n-1})$. And for newly arrived data point x_i, we need to calculate their bin indices j' based on equation 4 and emit the calculated indices with its bin count as 1 and the data point itself, given as $(j', 1, x_i)$

3. *Reduce stage* The reducer has two tasks: One to calculate the bin count C_j^n and the other to calculate the mean value vector M_j^n for each bins. Calculation on the bin count C_j^n can simply be achieved by summation over all the second emit argument from the Map stage. For calculating the mean value vector M_j^n, we need to first multiply the second and the third emit arguments from the map stage and sum over all the result of application of the multiply. And finally divide it by the total bin count C_j^n we get previously. Then return the bin index j with the aggregated results of bin count and mean value vector, given as (j, C_j^n, M_j^n)

Note that there is a side product in updating the binned summary process which is the upper and lower bounds for each dimension of the data. This measures span of the data points in each dimension and have a direct impact of k, which will decide the maximum number of bins needed(k^d)

5 Conclusion

The "moral of the story": What have we learned? What did we achieve? What did we not achieve? What would we do better next time? Possibilities for future research...

References

[1] A linear method for deviation detection in large databases. In *In: Proc of KDD96*, pages 164–169, 1996.

[2] Outlier detection using replicator neural networks. In *Proc of DaWaK02*, pages 170–180, 2002.

[3] Malak Alshawabkeh, Byunghyun Jang, and David Kaeli. Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 104–110, New York, NY, USA, 2010. ACM.

[4] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. Fast outlier detection using a gpu. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 143–150, July 2013.

[5] F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):203–215, Feb 2005.

[6] Sugato Basu, Mikhail Bilenko, and Raymond J. Mooney. A probabilistic framework for semi-supervised clustering. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 59–68, New York, NY, USA, 2004. ACM.

[7] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM.

[8] D. I. Curiac, O. Banias, F. Dragan, C. Volosencu, and O. Dranga. Malicious node detection in wireless sensor networks using an autoregression technique. In *Networking and Services, 2007. ICNS. Third International Conference on*, pages 83–83, June 2007.

[9] Parneeta Dhaliwal, M. P. S. Bhatia, and Priti Bansal. A cluster-based approach for outlier detection in dynamic data streams (KORM: k-median outlier miner). *CoRR*, abs/1002.4003, 2010.

[10] M. Elahi, K. Li, W. Nisar, X. Lv, and H. Wang. Efficient clustering-based outlier detection algorithm for dynamic data stream. In *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery*, volume 5, pages 298–304, Oct 2008.

[11] Jing Gao, Haibin Cheng, and Pang-Ning Tan. Semi-supervised outlier detection. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 635–636, New York, NY, USA, 2006. ACM.

[12] C. HewaNadungodage, Y. Xia, and J. J. Lee. Gpu-accelerated outlier detection for continuous data streams. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1133–1142, May 2016.

[13] Mahesh V. Joshi, Ramesh C. Agarwal, and Vipin Kumar. Mining needle in a haystack: Classifying rare classes via two-phase rule induction. *SIGMOD Rec.*, 30(2):91–102, May 2001.

[14] N.Chawla A.Lazarevic L.Hall K.Bowyer. Smoteboost:improving the prediction of minority class in boosting. 2003.

[15] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *Int. Conf. on Very Large Databases (VLDB98)*, pages 392–403, 1998.

[16] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *2011 IEEE 27th International Conference on Data Engineering*, pages 135–146, April 2011.

[17] A. Koufakou, E. G. Ortiz, M. Georgiopoulos, G. C. Anagnostopoulos, and K. M. Reynolds. A scalable and efficient outlier detection strategy for categorical data. In *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*, volume 2, pages 210–217, Oct 2007.

[18] A. Koufakou, J. Secretan, J. Reeder, K. Cardona, and M. Georgiopoulos. Fast parallel outlier detection for categorical datasets using mapreduce. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 3298–3304, June 2008.

[19] Takazumi Matsumoto and Edward Hung. *Accelerating Outlier Detection with Uncertain Data Using Graphics Processors*, pages 169–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[20] S. Papadimitriou, H. Kitagawa, P. B. Gibbons, and C. Faloutsos. Loci: fast outlier detection using the local correlation integral. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 315–326, March 2003.

[21] D. Pokrajac, A. Lazarevic, and L. J. Latecki. Incremental local outlier detection for data streams. In *2007 IEEE Symposium on Computational Intelligence and Data Mining*, pages 504–515, March 2007.

[22] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 427–438, New York, NY, USA, 2000. ACM.

[23] Shiblee Sadik and Le Gruenwald. Online outlier detection for data streams. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, pages 88–96, New York, NY, USA, 2011. ACM.

[24] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 187–198. VLDB Endowment, 2006.

[25] Luan Tran, Liyue Fan, and Cyrus Shahabi. Distance-based outlier detection in data streams. *Proc. VLDB Endow.*, 9(12):1089–1100, August 2016.

[26] S. Schroedl S Wagstaff K.Cardie, C. Rogers. Constrained k-means clustering with background knowledge. page 577 584, 2001.