GPU-accelerated Outlier Detection for Continuous Data Streams

Chandima HewaNadungodage, Yuni Xia
Department of Computer & Information Science
Purdue School of Science
IUPUI
Indianapolis, IN, USA
chewanad@iupui.edu, yuxia@iupui.edu

John Jaehwan Lee
Department of Electrical & Computer Engineering
Purdue School of Engineering & Technology
IUPUI
Indianapolis, IN, USA
johnlee@iupui.edu

***Abstract*-Outlier detection or anomaly detection is applied in numerous applications, such as fraud detection, network intrusion detection, manufacturing, and environmental monitoring. Due to the continuous and dynamic characteristics of streaming data, outlier detection over data streams becomes a very challenging task. When analyzing real-time data streams, it is typically impossible to store the entire set of data due to space limitations. Also due to the high data rates, it is necessary to produce the results in a limited amount of time. Parallel processing power of Graphics Processing Units (GPUs) can be used to accelerate the outlier detection process, and thus address the challenges of outlier detection over data streams. This paper proposes a GPU-accelerated outlier detection algorithm for continuous data streams using kernel density estimation approach. Experiments show that the proposed SOD_GPU algorithm is efficient for detecting outliers in high-dimensional, high-speed data streams, and produces results in a timely manner without compromising the outlier detection accuracy. The proposed method achieved up to 20X speedup compared to a respective multi-core CPU implementation, and the speedup increases with the number of data attributes and the input data rate.**

Keywords-Outlier detection; Data Streams; GPU; CUDA; kde

I. INTRODUCTION

An outlier in a dataset is a data point that is considerably different from the rest of the data points as if it is generated by a different mechanism. Outlier detection is applied in numerous applications, such as network intrusion detection, fraud detection, manufacturing, and environmental monitoring.

Stream data is generated continuously in a dynamic environment, with huge volume and infinite flow. Thus, it is typically impossible to store the entire set of data in memory due to space limitations. In this context, stream outlier detection algorithms have to work with only one pass over the dataset and with limited resources. In addition, the characteristics of a data stream may change over time. Because of these reasons, outlier detection over continuous data streams becomes a very challenging task.

Due to the memory constraints, mining data streams is often performed on batches of data called windows. In all window models, the main task is to analyze the portion of the data stream belonging to the current window. When detecting outliers in a streaming environment, it is not accurate to simply conclude that a data point is an outlier by only considering the statistics of the current window, because it can be a non-outlier when the recent history is considered. Similarly, a set of points that appear to be non-outliers in the current window can be outliers when the entire dataset is considered.

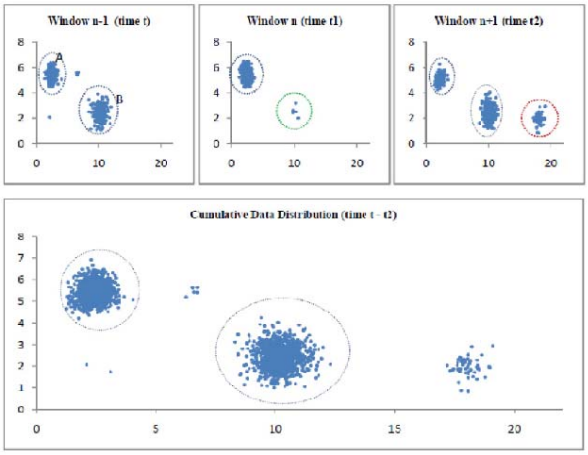

Figure 1. Evolving 2D data stream.

Let us illustrate such scenarios. Fig. 1 depicts the progress of a data stream consisting of 2-dimensional data points, over time. The top 3 charts in Fig. 1 show the individual data distribution of three consecutive windows, *n-1*, *n*, and *n+1*, at times *t*, *t1*, and *t2* respectively. The bottom chart shows the cumulative data distribution from time *t* to time *t2*. The horizontal axis and the vertical axis represent the two attributes of the data points. Note that in the n^{th} window, the points circled by the green dotted line appear to be outliers when we consider only the data points in that window; however, if we consider the previous window, those points actually belong to cluster B. Therefore, declaring those points as outliers is not accurate. On the other hand, the set of points circled by the red dotted line in the $(n+1)^{th}$ window forms a new cluster, hence

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.101

appear to be non-outliers; however, when we consider the entire dataset, these points do not have enough neighbors to consider them as non-outliers. This is visible from the cumulative distribution of the entire dataset shown in the bottom chart.

To address the above mentioned challenges in data streams processing, this paper proposes a novel streaming outlier detection algorithm that considers a global view of the data, only using a single pass over the dataset. We take a non-parametric approach based on kernel density estimation to detect outliers. The proposed method maintains a binned summary of previous statistics to help with the decisions of the current batch of data, and thus providing more accurate results compared to the sliding window approach.

Although the density-based outlier detection approaches are proven to be accurate, they are also known to be computationally demanding. Therefore, when using kernel density estimation to detect outliers in a high volume, high-speed data stream, we need to speed up the computation to keep up with the input rate of streaming data.

Graphics Processing Units (GPUs) are designed to handle highly parallel workloads and can execute thousands of concurrent threads. With the introduction of CUDA (Compute Unified Device Architecture)[1], which is a general purpose parallel computing architecture with a new parallel programming model, GPU computing became more and more popular in general-purpose data mining applications [1, 2, 3, 4]. To comply with the real-time processing requirements of streaming data, we use parallel processing powers of GPUs to accelerate kernel density estimation and generate results in a timely manner. Experimental results on real-world datasets show that our proposed method achieved 20X speedup compared to a respective multi-core CPU implementation.

The remainder of this paper is organized as follows: Section II briefly reviews related work. In section III, we describe the proposed GPU-accelerated streaming outlier detection algorithm SOD_GPU. Section IV presents the experiments and the results analysis, and Section V concludes the paper.

II. Related Work

Data stream outlier detection has been explored by many researchers over the past decade. For static data, k-nearest neighbor (kNN) based algorithms seem to be the most used and one of the best performing methods available. Several approaches have been proposed to extend nearest-neighbor outlier detection algorithms to data streams [5, 6, 7, 8]. The methods proposed in [5, 6] are based on sliding window concept and only consider the data belonging to the most recent window. Our proposed method maintains a statistical summary and makes decisions considering the historical data as well as the current data. Sadik and Gruenwald [7] proposed an incremental method for outlier detection using kernel density estimation; however, it can be only applied to one dimensional data, whereas our algorithm works on multi-dimensional data. In [8], D. Pokrajac et al. presents an incremental Local Outlier Factor (LOF) algorithm to detect outliers in streaming data. Although LOF algorithm produces very promising results; it is computationally intensive even for static datasets. Thus, performing incremental updates will not scale well for high-speed data streams. Our work is based on kernel density estimation, and we use GPUs to accelerate the computation and produce timely results.

Some outlier detection techniques make assumptions about the underlying data distribution, such as Gaussian mixture models or auto-regression models [9, 10, 11]. However, it is often difficult to predict the underlying data distribution, and the success of these techniques depends on the quality of the selected models. In contrast, our method takes a non-parametric approach and does not make any assumptions about the underlying data distribution. There are some works on clustering-based approaches for stream outlier detection. Elahi et al. proposed a K-mean clustering approach [12] and Dhaliwal et al. proposed a K-median clustering approach [13]. The main drawback in both of these approaches is that the user has to specify the number of clusters. In contrast, our method does not require such information and the underlying data distribution is discovered using kernel density estimation. In [14] Yu et al. proposed a method for anomaly detection in evolving data streams based on semi-supervised learning. This work assumes the presence of labeled data, whereas our method does not depend on labeled data.

Utilizing GPUs to speedup LOF outlier detection method is explored in [15] and [16]. In [18], Angiulli et al. proposed set of GPU accelerated distance-based kNN algorithms. Matsumoto and Hung [19] propose a GPU-accelerated approach to detect the outliers in uncertain data. All of those methods were designed for static datasets and cannot be directly extended to streaming data, whereas our method is designed for data streaming applications.

III. Proposed Approach

A data stream S can be defined as a set of continuous data points $S = \{X_1, X_2, X_3, \ldots\ldots\ldots, X_N, \ldots\ldots\ldots\}$, where each data point X_i consists of d-dimensional attributes $X_i = <x_{i1}, x_{i2}, \ldots\ldots\ldots, x_{id}>$.

A. Kernel Density Based Outlier Detection

As mentioned earlier we take a non-parametric approach to outlier detection where we do not make any assumptions about the data distribution. We use kernel density estimation to discover the underlying probability density function. In this work, we have used the Gaussian kernel density estimator as it gives smooth estimations. It should be noted that choosing the Gaussian kernel function is different from fitting the underlying distribution to a Gaussian pdf. More details on kernel density estimation techniques can be found in [17].

For each data point X, the density around that point is calculated according to the following formula,

$$p(X) = \frac{1}{N}\sum_{j=1}^{N} \frac{1}{(2\pi)^{d/2}\mathrm{H}} exp\left\{\frac{-1}{2}\sum_{i=1}^{d}\left(\frac{x_i - x_j}{h_i}\right)^2\right\} \quad (1)$$

where N is the number of data points and $H = (h_1 h_2 h_3 \ldots\ldots h_d)$ is the bandwidth of the kernel function.

[1] http://developer.nvidia.com/page/home.html

Using the Scott's rule [17], bandwidth is calculated as $h_i = \sigma_i N^{(1/d+4)}$, where σ_i is the standard deviation of the values in dimension i.

The *outlierness* of a particular point X can be defined as the inverse of the density around that point; points with high density values are considered as inliers and points with low density values are considered as possible outliers. Thus, the *outlier-factor* (f_o) of a data point X is defined as follows,

$$f_o(X) = \frac{1}{p(X)} \quad (2)$$

Since outliers are rare events, simply using f_o to declare a point as an outlier will generate a large amount of false positives. To avoid that, we define a cut-off threshold thr_o as follows,

$$thr_o = \frac{1}{p_{avg} * \delta} \quad (3)$$

where p_{avg} is the average probability of all data points, and $0 \leq \delta \leq 1$ is the user-defined cut-off ratio. A data point X is declared as an outlier point if $f_o(X) > thr_o$. Then the outlier points can be sorted in the descending order of their f_o values to get the top K outlier points.

B. Application to Data Streams

Due to the continuous nature of streaming data and limited availability of memory, it is impossible to collect the entire dataset and apply the above mentioned approach to detect outliers. In practice, mining data streams is often performed on batches of data which are called windows. In our approach, the data stream is divided into non-overlapping windows at regular time intervals. Let W be the window size and let T_0 denote the starting time; thus, the window boundaries will be $T_0 + W$, $T_0 + 2W$ and so on. We can apply the aforementioned kernel density estimation method to each window, to find outliers in that window. However, as explained before, if we declare a data point as an outlier only based on the statistics of the current window, that decision might not be accurate. Therefore, we need to maintain some synopsis of the previous batches.

1) Maintaining the Binned Statistical Summary

It is not possible to store all the historical data due to the memory limitations; therefore, we maintain a binned summary calculated from the previous data points.

Assume there are N data points and each data point consists of d attributes (dimensions). This Nxd data space is partitioned into k^d bins by dividing each dimension d into k slots of user-specified width Δ. Then, each data point is assigned to its corresponding bin B_j ($0 \leq j \leq k^d$) as follows. For each dimension, the range of the input values is mapped to an interval [0, 1], then, each data point X_i can be coded using d indices $<I_{i1}, I_{i2}, I_{i3}, \ldots\ldots, I_{id}>$ where $I_{ij} = x_{ij} / \Delta$. After that, using the below formula, we can calculate the corresponding bin index for each data point X_i.

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

For each bin B_j, we maintain the aggregated number of data points falling into that bin (bin count C_j), and the aggregated mean value vector ($M_j = <\mu_{j1}, \mu_{j2}, \mu_{j3}, \ldots\ldots, \mu_{jd}>$) of the points falling into that bin. Here, μ_{jd} is the respective mean value of dimension d. We also maintain the mean (μ) and standard deviation (Σ) of the entire dataset. Fig. 2 illustrates this process.

It is worth noting that the total number of possible bins (k^d), is exponential to the number of dimensions in the dataset. Therefore, it is possible that both the run-time, and the space utilization will increase with the number of dimensions. However, we are only interested in the number of non-empty bins (m). Although the total number of bins in the data grid exponentially increases with the number of dimensions, most of these bins does not contain any data points, and thus, the actual number of non-empty bins is very small compared to the total number of bins ($m <<<< k^d$).

The summary statistics needs to be updated with new data at the end of each batch. The update procedure will be described in sub-section III B 3 below; before that, in the next sub-section, we will discuss how to detect the outlier points by combining the stored summary statistics and the current data points.

2) Cumulative Kernel Density Estimation

For each data point X in the current window, we compute the probability density around that point using two components as given in the following equation,

$$p(X) = w * p_{curr}(X) + (1 - w) * p_{prev}(X) \quad (5)$$

where w is the user defined weight factor, $p_{curr}(X)$ is the probability density of point X calculated considering the data points of the current window, and $p_{prev}(X)$ is the probability density of point X calculated considering binned summary of previous data. $p_{curr}(X)$ is calculated by directly applying (1) to the current batch. To compute $p_{prev}(X)$ we modify (1) as follows,

$$p_{prev}(X) = \frac{1}{C}\sum_{j=1}^{m} \frac{1}{(2\pi)^{d/2}\mathrm{H}} C_j * exp\left\{\frac{-1}{2}\sum_{i=1}^{d}\left(\frac{x_i - \mu_{ji}}{h_i}\right)^2\right\} \quad (6)$$

where m is the total number of non-empty bins, and C is the total number of data points ($C = \Sigma\ C_j$) in the non-empty bins. The bandwidth of the kernel function $H = (h_1 h_2 h_3 \ldots\ldots h_d)$ is calculated according to the Scott's rule [17], using the aggregated standard deviation Σ. Once we have computed $p(X)$ for each data point, we can find the outlier points using the *outlier_factor* and cut-off threshold as described in Section III A.

3) Updating the Summary Statistics

At the end of each batch, summary statistics calculated from that batch needs to be aggregated with the stored summary of the previous data. Typically, in streaming environments, concept drifts occur making the recent data points more significant than the historical data. Hence, considering all the historical data is not necessary, and sometimes not accurate. Thus, outdated concepts should be discarded, while favoring the recent concepts. Therefore, we gradually discard the obsolete data and update the summary statistics as follows.

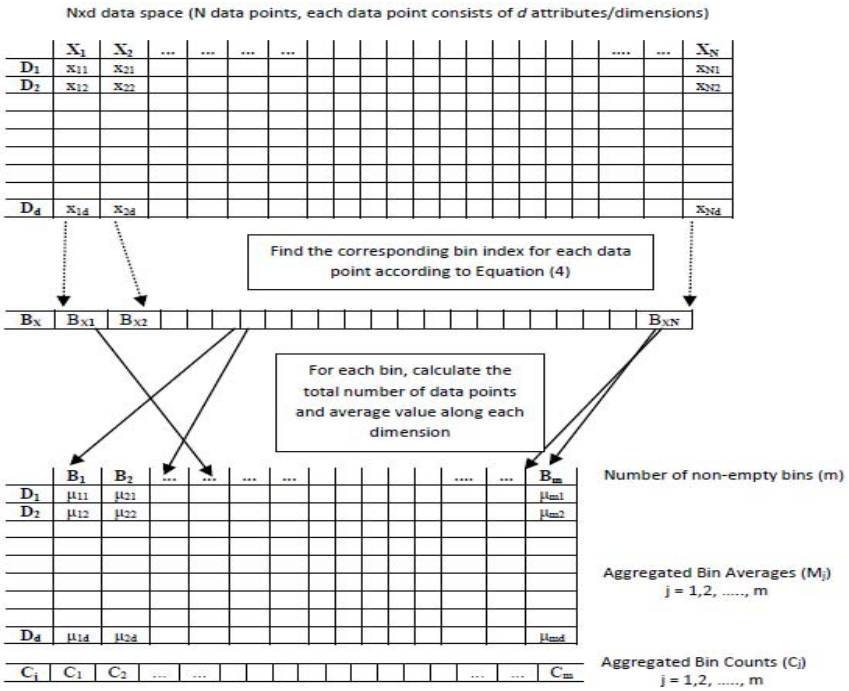

Figure 2. How to compute the binned synopsis.

Assume we are currently processing the n^{th} window, then, for each non-empty bin B_j, we have stored C_j^{n-1}, which is the aggregated number of data points in bin B_j up to the $(n\text{-}1)^{th}$ window and M_j^{n-1}, which is the aggregated mean value vector for bin B_j up to the $(n\text{-}1)^{th}$ window. We also have stored μ^{n-1} and Σ^{n-1} which are the cumulative mean and standard deviation of the entire dataset up to the $(n\text{-}1)^{th}$ window respectively.

When updating the aggregate values, we consider the data distribution of the current window and gradually discard the historical records that do not match with the current distribution. If the number of data points in a particular bin of the current window is significantly less than the average number of data points per bin in the current window, then we discount the aggregated count of that bin as follows,

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

where c_j^n is the number of data points in bin B_j of the n^{th} window, c_{avg}^n is the average number of data points per bin in the n^{th} window considering only the non-empty bins of the n^{th} window, $0 \le \delta \le 1$ is the user defined cut-off ratio, and $0 \le \alpha \le 1$ is the user defined decay ratio. After we update the aggregated count (C_j^{n-1}) for each bin the other statistics can be updated as follows,

- $M_j^n = \frac{(c_j^n * \theta_j^n + c_j^{n-1} * M_j^{n-1})}{(c_j^n + c_j^{n-1})}$
- $C_j^n = c_j^n + C_j^{n-1}$
- $C^{n-1} = \sum_{j=1}^{m} C_j^{n-1}$
- $\mu^n = \frac{(c^n * \theta^n + C^{n-1} * \mu^{n-1})}{(c^n + C^{n-1})}$
- $\Sigma^n = \sqrt{\frac{(c^n * (\sigma^n)^2 + C^{n-1} * (\Sigma^{n-1})^2)}{(c^n + C^{n-1})} + \frac{(c^n * C^{n-1})}{(c^n + C^{n-1})^2} (\theta^n - \mu^{n-1})^2}$

where θ_j^n is the mean value vector of bin B_j of the n^{th} window, θ^n is the mean value vector of all data points in the n^{th} window, σ^n is the standard deviation of all data points in the n^{th} window, and c^n is the total number of data points in the n^{th} window.

4) Why to use GPU

Although the above described approach will address the continuous nature of streaming data, it does not address the timely processing requirements. As can be seen from (1) and (6), computing probability density values is a compute intensive task. Each data point X needs to be compared with every other data point (and data bin) for each dimension. This gives polynomial time complexity in terms of the number of data points N in a given batch. When processing high-speed data streams, this will be a bottleneck, and thus, the algorithm will not be able to produce the results in a timely manner. This will create a backlog of input data and algorithm will not be able to keep up with the data stream. To avoid this bottleneck, in this paper we propose a parallel GPU-based implementation using NVIDIA CUDA programming platform.

The probability density $p(X)$ for each data point X can individually be calculated without depending on the probability density of the other data points. Therefore, this

step can be highly parallelized using GPUs to achieve significant speedup as compared to CPU implementation. We name our algorithm as SOD-GPU (Stream Outlier Detector using GPU), and in the next two sections, we will describe it in detail using the NVIDIA CUDA programing model.

C. *SOD-GPU (Stream Outlier Detector-GPU) Algorithm*

GPU programming model follows a SIMD (Single Instruction Multiple Data) architecture. A CUDA program consists of a host program running on the host CPU, and one or more *kernel functions* executed on a GPU. Execution of a CUDA *kernel* is handled by units called *blocks,* each *block* contains multiple threads, and these *blocks* are executed in parallel on different sections of the input data

We split the probability density estimation into four steps and use a separate GPU *kernel function* for each step. As described in Sections III A and III B, for each new batch of data following steps need to be performed.

- Kernel 1: Computes the standard deviation and estimates the bandwidth *H* according to the Scott's rule [17].
- Kernel 2: Computes the probability density for each data point using Gaussian kernel density estimation over the current data (1).
- Kernel 3: Computes the probability density for each data point using Gaussian kernel density estimation over the binned summary statistics (6).
- Kernel 4: Computes cumulative probability density and *outlier-factor* for each data point (5 and 6), and mark outliers.

After calculating the current bin statistics and transferring the necessary data to the GPU, we can discard the current batch of data and make space for the next batch of incoming data in the CPU (we do not have to wait till the GPU is done with the computation for the current batch). Also, while the GPU is processing the outliers, CPU updates the binned summary statics using the information of the current window as described in Section III B 3.

In CUDA, we can use *cuda streams* to concurrently execute two or more kernel functions. As Kernel 2 and Kernel 3 are independent of each other, they can be concurrently executed. Therefore, we assign these two kernels to two separate *cuda streams* and execute them in parallel. Kernel 1 should be completed before executing the other three kernels; and Kernel 4 should be executed after the completion of both Kernel 2 and Kernel 3. In CUDA, the *default stream* (known as stream 0) is always synchronized with the other *cuda streams*. Therefore, we assign the Kernel 1 and the Kernel 4 to the *default stream*. Also, by using *cuda streams*, we can hide most of the data transfer overhead from CPU to GPU by overlapping the data transfers with kernel executions. Fig. 3 depicts the kernel execution timeline for one batch. As shown in Fig. 3, while the compute engine executes Kernel 1 computation in Stream 0, the copy engine copies the data necessary for Kernel 2 in Stream 2. Similarly, when the compute engine executes Kernel 2 in Stream 1, the copy engine copies the data necessary for Kernel 3 in Stream 2. This process is repeated for each batch of data.

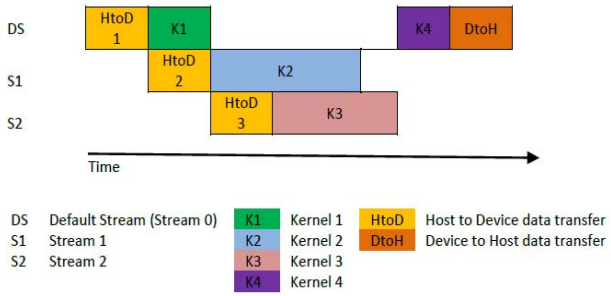

Figure 3. GPU kernel execution timeline.

Input: Incoming batch of stream data, cells-per-dim *k*, decay ratio α, cut-off ratio δ, weight factor *w*

Output: Set of outlier points

Begin:

- For each data tuple *X* in the current batch,
  - Calculate the corresponding bin index.
  - Update the stats for the corresponding bin.
- Compute ourlier_factors for the current batch in GPU.
  - Default Stream: Transfer *dxN* input data to GPU
  - Default Stream: Kernel 1
    - Computes the standard deviation and estimates the bandwidth H.
  - Stream 2: Transfer current window stats to GPU.
  - Stream 2: Kernel 2
    - Computes the probability density for each data point using Gaussian kernel density estimation over the current data.
  - Stream 3: Transfer aggregated stats up-to previous batch to GPU.
  - Stream 3: Kernel 3
    - Computes the probability density for each data point using Gaussian kernel density estimation over the binned summary statistics.
  - Default Stream: Kernel 4
    - Compute cumulative probability and outlier-factor for each data point.
    - Mark outlier points.
  - Default Stream: Transfer outlier information to CPU
- Update the binned summary statics using the information of the current window (Section III B 3)

End

Figure 4. Pseudo-code of the SOD_GPU algorithm.

Before starting the GPU computation, input data needs to be transferred from the host memory to the GPU's memory called global memory. In CUDA, if a warp (a set of successive 32 threads scheduled together) accesses successive 128 bytes in global memory, the memory access is coalesced and only takes a single fetch. To fully utilize the power of coalesced memory access, input data is transferred to GPU in a column-major format (i.e. data points are stored column-wise in a *d* x *N* data matrix). Depending on the window size *W*, the number of

dimensions *d*, and the task performed by each *kernel*, we adjust the number of blocks and the number of threads per block to fully utilize the available GPU resources. In the following sections, we will describe the implementation of each GPU kernel in detail. Fig. 4 presents the pseudo-code for the proposed SOD-GPU algorithm.

1) Kernel 1- Bandwidth Computation

Kernel 1 computes the standard deviation and estimates the bandwidth H according to the Scott's rule [17]. Here, the computation is divided into 1 x B_{K1} one-dimensional *blocks*, each consisting of T_{K1} one-dimensional threads. The value of T_{K1} needs to be a multiple of 32 to achieve coalesced global memory access. The value of B_{K1} is set to N/T_{K1}. Each block processes T_{K1} data points in parallel and calculates the standard deviation over d dimensions. Each thread computes the partial sum for each dimension in *shared memory* and atomically update total sum stored in the *global memory*. Then using total sum computes the standard deviation and the bandwidth for each dimension.

2) Kernel 2, 3 – Probability Density Estimation

Kernel 2 and Kernel 3 compute the probability densities of the data points in the current window with respect to the current data and the binned summary statistics, respectively. For these two kernels, we use the same block and thread layout as they implement the same function, difference is only in the values of the parameters. To compute the probability density around one data point, we need *N-1* computations (i.e. each data point needs to be compared with all the other data points in the current window over each dimension), then the result from these N computations are summed up to get the probability density value for that point. Therefore, in total we need to compute NxN values to get the probability density for all the data points. Hence, in these kernels, the computation is divided into B_X x B_Y two-dimensional *blocks*, each consisting of T x T two-dimensional threads. Fig. 5 depicts the block and thread arrangement for Kernels 2 and 3.

When computing the kernel density, each data point is accessed multiple times. In GPUs, retrieving the same data point multiple times from the GPU global memory is a costly operation which can limit the possible speedup of the application. On the other hand, GPU shared memory provides faster access but limited in size. Therefore, we divide the data into small sub-matrices, and copy two sub-matrices at a time to the shared memory. Then we perform all possible calculations on those two sub-matrices. Each *block* computes the partial sum assigned to that *block* and merge the results to a density matrix stored in global memory.

Similar to previous kernels, the value of T needs to be a multiple of 32 to achieve coalesced global memory access, and Tx T should not exceed 1024 (the maximum number of threads allowed per block). In Kernel 2, $B_X = B_y = N/T$ as the density is computed using current data points. In Kernel 3, density is computed using the binned summary, therefore, value of B_X is given by N/T and the value of B_y is given by C/T where C is the number of bins.

3) Kernel 4 – Outlier-factors Computation

Kernel 4 computes the cumulative probability by combining the results from Kernel 2 and Kernel 3 according to (5). Then it computes *outlier_factors* according to (2), and mark outlier points using the cut-off threshold. Here, the computation is divided into 1 x B_{K4} one-dimensional *blocks*, each consisting of T_{K4} one-dimensional threads. The value of T_{K4} needs to be a multiple of 32 to achieve coalesced global memory access. The value of B_{K4} is set to N/T_{K4}. Each block computes the *outlier_factor* for T_{K4} data points in parallel and mark outlier points.

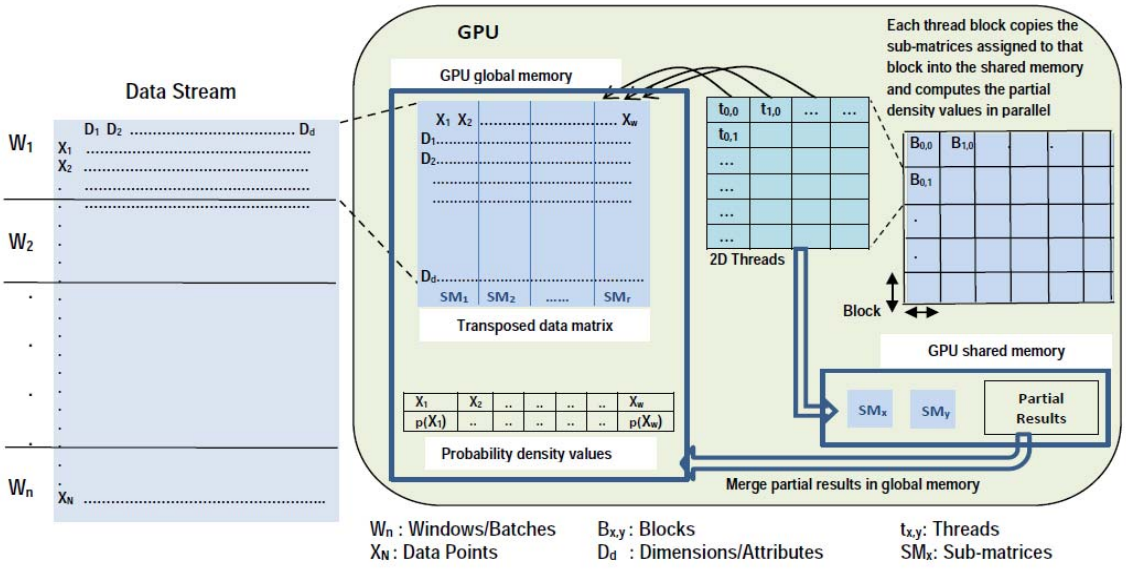

Figure 5. GPU thread arrangement for Kernel 2 and Kernel 3.

IV. Results and Observations

This section presents the performance evaluations for the proposed SOD_GPU algorithm in terms of accuracy and runtime.

A. Datasets

We performed our experiments on two real world datasets obtained from the UCI machine learning repository[2]. The first dataset is the *KddCup99* network intrusion detection dataset, and the second dataset is the *Covertype* forest cover dataset. Both of these datasets were originally designed for classification tasks; therefore, when using these datasets for outlier detection we used the data points belong to the classes with higher number of instances as normal data. For the Kddcup99 dataset, the points belonging to the classes "*smurf*" or "*neptune*" were selected as the normal data points. For outlier points, we randomly selected some records from the classes which have less than 1000 instances. The resulting dataset consists of ~3.8M records of 11 attributes each, including 0.05% outliers. For the Covertype dataset, the points belonging to the classes "*Spruce-Fir*" or "*Lodgepole Pine"* were chosen as the normal data points. Outlier points were randomly sampled from the class with the lowest number of instances "*Cottonwood/Willow*". The resulting dataset consists of ~495K records of 10 attributes each, including 0.1% outliers.

B. Evaluation Criteria

To evaluate the accuracy of the proposed method, we utilize the *Precision*, *Recall*, and *F-score* values, which are widely used metrics for accuracy evaluation. *Precision* is defined as the number of correctly detected outliers (true positives) divided by the total number of detected outliers (true positives + false positives). *Recall* is defined as the number of correctly detected outliers divided by the total number of outliers in the dataset (true positives + false negatives). *F-score* is a measure which combines *precision* and *recall* to give an un-biased evaluation of the detection accuracy of the algorithm; it is defined as (2**precision***recall*)/(*precision* + *recall*).

To evaluate the speedup and the scalability achieved by the proposed method, we compared the performance of the SOD_GPU algorithm with a respective multi-core CPU implementation named as SOD-CPU. Runtime of the GPU and CPU implementations was measured against increasing number of attributes in the datasets and also increasing input data rate.

All the experiments were carried out in a server running 64-bit Fedora 21, equipped with two Intel Xeon 3.3GHz quad-core CPUs and 64GB memory, and an NVIDIA GTX TITAN GPU card with 6GB memory, CUDA runtime version 7, and compute capability 3.0. The multi-core CPU implementation is configured to use 16 threads (2(CPUs) x 4(cores) x 2(hyper-threading) = 16). All the algorithms were implemented using C++. CUDA is used for GPU programming, and OpenMP is used for multi-core CPU programming, the GPU code was hosted by a single CPU process.

In all the experiments, each data dimension is divided into 100 slots, values greater than 100 did not make any noticeable improvement in the accuracy of the method, and thus 100 is chosen. To choose the values for cut-off ratio δ, decay-ratio α, and weight factor *w*, we tried different values in the range of (0-1] and pick the values which gave the best results for each dataset. For the *KddCup99* dataset, δ was set to 0.05 and α was set to 0.5, and for the *Covertype* dataset, δ was set to 0.01 and α was set to 0.4; *w* was set to 0.5 in both datasets.

For all the experiments, the reported results are the average values obtained by conducting the same experiment multiple times over the respective datasets.

C. Accuracy Evaluation

Although there are some algorithms proposed to utilize GPUs to accelerate outlier detection [15, 16, 18, 19], none of those algorithms addresses the challenges when detecting outliers for streaming data. To demonstrate how the proposed SOD_GPU algorithm performs well in the presence of streaming data, we compared it with the *GPU SolvingSet* (*GPU_SS*) algorithm proposed in [18]. GPU_SS is a parallel CUDA implementation of the *SolvingSet* algorithm [20], which is an optimized distance-based k-nearest neighbor outlier detection algorithm. We apply the GPU_SS algorithm for each batch of data in sliding window fashion (i.e. it only utilizes the current portion of data for outlier detection); after computing the outliers, current batch of data is discarded to make space to the next batch of data.

Fig. 6 and Fig. 7 present the accuracy comparison on the Covertype dataset and the KDDCup99 dataset, respectively. Each dataset was processed in batches of 60,000 records. The input parameters for GPU_SS algorithm are set as, number of candidates = 1% of the batch size, and number of neighbors = 50.

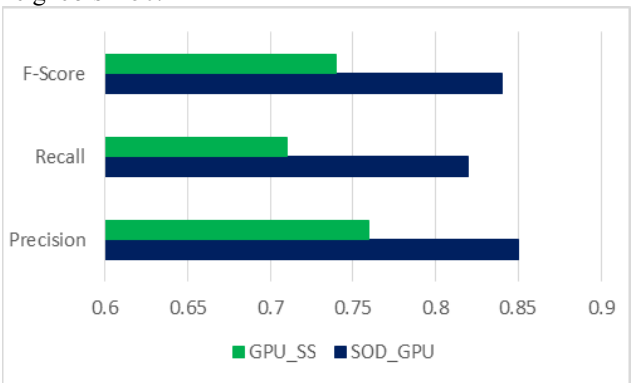

Figure 6. Accuracy comparison on the Covertype dataset.

It is visible that the *Precision*, *Recall*, and *F-score* values achieved by the SOD_GPU algorithm are higher than the respective values achieved by the GPU_SS algorithm in both datasets. This is due to the fact that GPU_SS algorithm is not capable of processing streaming data efficiently. As we mentioned earlier, detecting outliers using only the data points of the current window results in higher number of

[2] http://archive.ics.uci.edu/ml/datasets.html

false positives and/or false negatives, and thus reducing the detection accuracy; in contrast, SOD_GPU algorithm maintains an aggregated summary and makes the decisions based on a cumulative picture of the data stream seen so far.

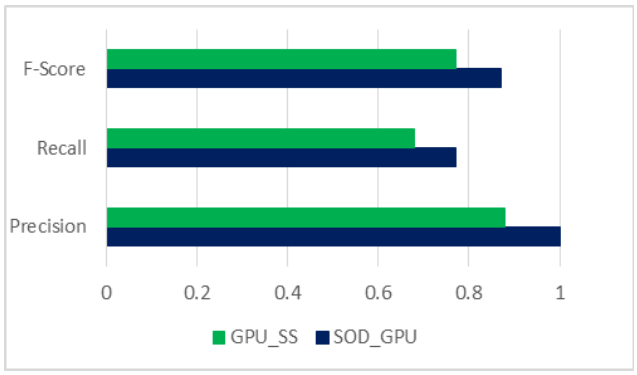

Figure 7. Accuracy comparison on the KddCup99 dataset.

To evaluate how the outlier detection accuracy of the SOD_GPU algorithm changes when the batch size is changed, we computed the *F-Score* values at different batch sizes. We used KddCup99 dataset for this experiment as it was the larger dataset. Fig. 8 depicts the *F-Score* achieved by the SOD_GPU algorithm for different batch sizes. It is visible that the *F-Score* improves with the increase of the batch size up to a certain limit and then drops again. The reason behind this is, when the batch size is small, there is not enough data to accurately compute the outliers; on the other hand, when the batch size is large, there is obsolete information which is not yet discarded, and thus, negatively affects the accuracy of the computation. This observation validates our argument that, for dynamic streaming data, the outlier detection process should be completed in a timely manner to avoid backlogging of input data.

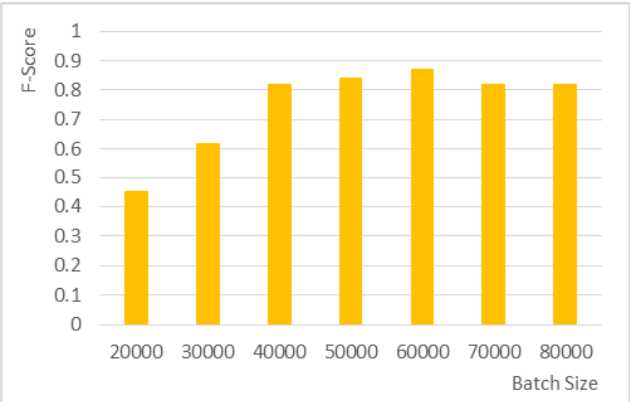

Figure 8. F-score w.r.t batch size on KddCup99 dataset.

D. *Runtime and Scalability Evaluation*

It is important for an outlier detection algorithm to scale well with the number of attributes in the dataset, so that it will be able to process high-dimensional data in a timely manner. To evaluate the scalability and the speedup achieved by the proposed method, we compared the runtime of the SOD_GPU algorithm with the runtime of its respective multi-core CPU implementation, SOD_CPU algorithm, at different values of the number of attributes (or dimensions) for each dataset. The runtimes reported here are the total runtimes for the respective versions, including the data transfer time between the CPU and the GPU, and the summary statistics update time.

Fig. 9 presents the average time (in seconds) spent by each algorithm to process a batch of 60,000 records w.r.t the increasing number of attributes on the Covertype dataset. Fig. 10 presents the speedup (SOD-CPU time/SOD-GPU time) achieved w.r.t the increasing number of attributes. Fig. 11 and Fig. 12 depict the same information for the KddCup99 dataset.

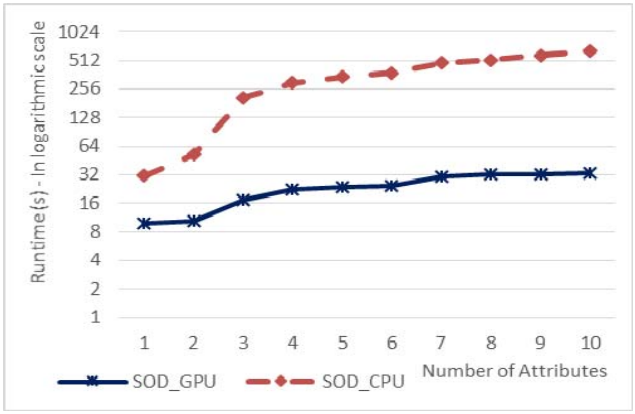

Figure 9. Runtime w.r.t. number of attributes on Covertype dataset.

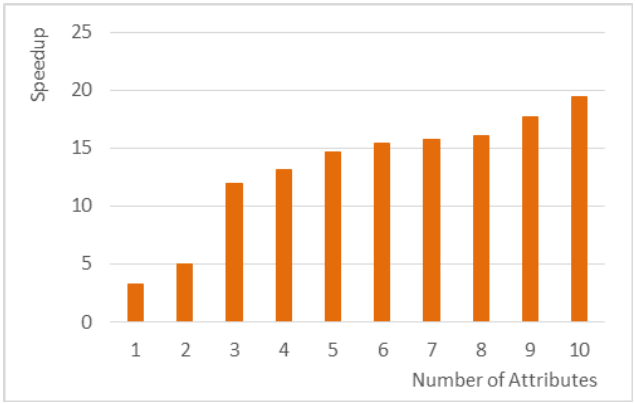

Figure 10. Speedup w.r.t. number of attributes on Covertype dataset.

It is visible that the runtime of the SOD_GPU algorithm scaled well with the increasing dimensionality of the data and the speedup increases with the number of attributes, achieving up to 20X speedup compared to the multi-core CPU implementation. Even at the highest value of the number of attributes, SOD-GPU algorithm processed the a batch of input data around in 30 seconds for the Covertype dataset and around in 15 seconds for the KddCup99 dataset; whereas, SOD_CPU algorithm took around 10 mins and 5 mins, respectively, to complete processing a batch of input data. Therefore, it is visible that the proposed GPU-based

algorithm is capable of producing results in a timely manner and scales well with the number of attributes in the dataset.

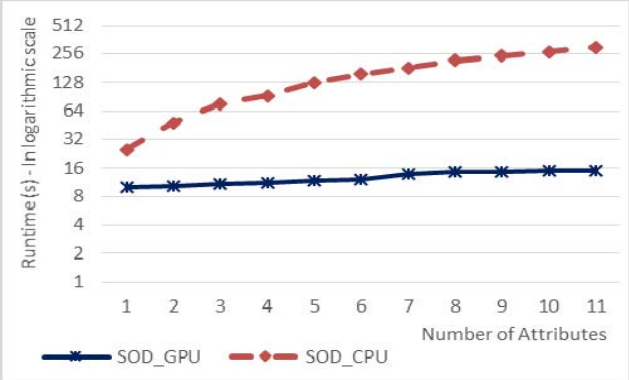

Figure 11. Runtime w.r.t. number of dimensions on KddCup99 dataset.

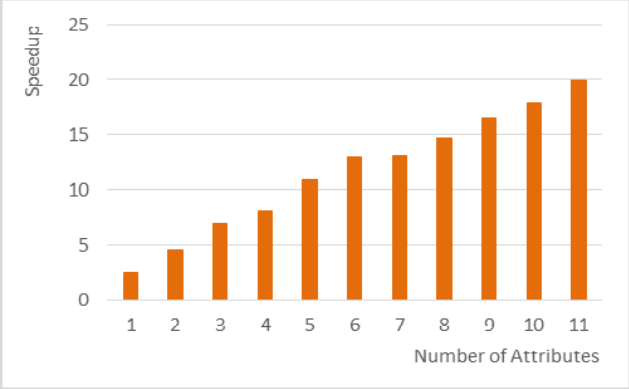

Figure 12. Speedup w.r.t. number of attributes on KddCup99 dataset.

In contrast to the static data, when processing streaming data, input data rate is an important factor to be considered. It is crucial that the algorithm should be able to keep up with the input data rate to avoid backlogs. To observe how the proposed SOD_GPU algorithm will scale with the input data rate, we increased the input rate from 200 tuples/sec to 1000 tuples/sec and compared the runtimes of the SOD_GPU and SOD_CPU algorithms.

Fig. 13 and Fig. 14 present the runtime comparison of the two implementations at different data rates for the Covertype dataset and the KddCup99 dataset, respectively. The number of attributes is set to the highest value available for each dataset, and the batch size is set to 60,000.

It is visible that the runtime of the SOD_GPU algorithm scales well with the input data rate. For example, when the data rate is 1000 tuples/sec and data is processed in batches of 60,000 tuples, algorithm has 60 seconds (60,000/1000) to complete processing the current batch of data while the next batch of data accumulates in the input stream. If processing exceeds 60s, it will cause a backlog of input data and algorithm will not be able to keep up with the input stream. It can be seen from Fig. 13 that the SOD_GPU algorithm only took around 30s and well within the processing time limit; whereas, the SOD_CPU algorithm took more than 10 mins and ten times slower than the required time limit.

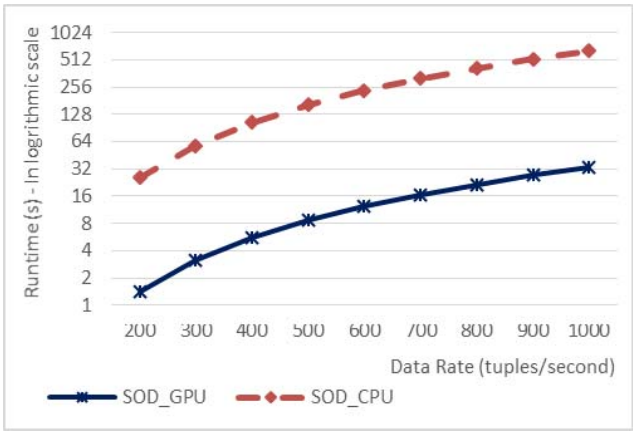

Figure 13. Runtime w.r.t. input data rate on Covertype dataset.

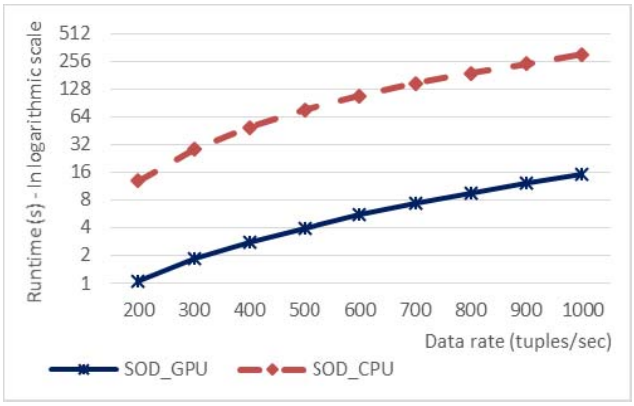

Figure 14. Runtime w.r.t. input data rate on KddCup99 dataset.

Fig. 15 depicts the performance gain achieved by the SOD_GPU algorithm compared to the SOD_CPU algorithm, at different data rates, for the Covertype dataset.

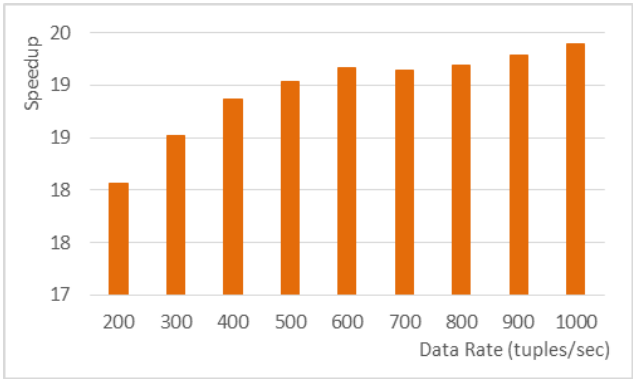

Figure 15. Speedup w.r.t. input data rate on Covertype dataset.

It is observed that the proposed GPU implementation achieved up to 20X speedup compared to the multi-core CPU implementation, and speedup increases with the increasing data rate of the input stream. Experiments on the KddCup99 dataset has demonstrated a similar behavior, but

the graph is omitted due to space limitations. Therefore, it is visible that the proposed SOD_GPU algorithm scales well with input data rate and is capable of processing streaming data with high input rates in a timely manner.

In this paper, we have conducted our experiments using a single GPU hosted by a single CPU process. However, it is possible to extend this algorithm to use multiple GPUs hosted by a single CPU or even to use a GPU-grid. As the outlier factor computation for data points is independent from each other, this task can be distributed across multiple GPUs. Once all the outlier factors are computed, results can be gathered in one GPU to mark the outlier points. Therefore, the proposed algorithm can be easily scaled to handle much higher data input rates.

V. Conclusion

In this paper we presented a GPU-assisted outlier detection algorithm for continuous data streams. We used binned kernel density estimation to detect outliers in a cumulative fashion, and used parallel processing power of GPUs to accelerate the computation. The proposed algorithm processes streaming data in batches, only using a single pass over the dataset, yet considering the effect of the previous data points to detect outliers. Experiments show that the proposed method is capable of detecting outliers in streaming data in a timely manner without compromising the detection accuracy. Also it was observed that the proposed SOD_GPU algorithm achieved up to 20X speedup compared to the respective multi-core CPU implementation, and it was visible that speedup increases with the increasing number of attributes in the data and the increasing input rate of the data stream. Therefore, we can conclude that the proposed SOD-GPU algorithm is an efficient method for detecting outliers in high-dimensional, high-speed data streams.

Acknowledgements

This work was supported by the IT R&D program of MSIP/KEIT, Korea. [10041709, Development of Key Technologies for Big Data Analysis and Management based on Next Generation Memory].

References

[1] D. Steinkraus, I. Buck, and P. Simard, "Using GPUs for machine learning algorithms," Proc. IEEE International Conference on Document Analysis and Recognition, 2005, pp. 1115–1120.

[2] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2008, pp. 1–6.

[3] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A parallel implementation of k-means clustering on GPUs," Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2008, pp. 340–345.

[4] K. Kato and T. Hosino, "Solving k-Nearest Neighbor Problem on Multiple Graphics Processors," Proc. IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 769-773.

[5] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," Proc. ACM Conference on Information and Knowledge Management, 2007, pp. 811–820.

[6] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams," Proc. IEEE International Conference on Data Engineering (ICDE), 2011, pp. 135–146.

[7] S. Sadik and L. Gruenwald, "Online outlier detection for data streams," Proc. ACM Symposium on International Database Engineering and Applications, 2011, pp. 88-96.

[8] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," Proc. Computational Intelligence and Data Mining (CIDM), 2007, pp. 504–515.

[9] K. Yamanishi, J.-I. Takeuchi, G. Williams, and P. Milne, "On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms," Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data mining, 2000, pp. 320–324.

[10] D.-l. Curiae, O. Banias, F. Dragan, C. Volosencu, and O. Dranga, "Malicious node detection in wireless sensor networks using an auto regression technique," Proc. International Conference on Networking and Services (ICNS), 2007, pp. 83–83.

[11] K. Yamanishi and J.-i. Takeuchi, "A unifying framework for detecting outliers and change points from non-stationary time series data," Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data mining, 2002, pp. 676–681.

[12] M. Elahi, K. Li, W. Nisar, X. Lv, and H. Wang, "Efficient clustering-based outlier detection algorithm for dynamic data stream," Proc. International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2008, pp. 298–304.

[13] P. Dhaliwal, M. Bhatia, and P. Bansal, "A cluster-based approach for outlier detection in dynamic data streams (korm: k-median outlier miner)," Journal of Computing, vol 2, Issue 2, 2010, pp. 2151-9617.

[14] Y. Yu, S. Guo, S. Lan, and T. Ban, "Anomaly intrusion detection for evolving data stream based on semi-supervised learning," Advances in Neuro-Information Processing, LNCS, vol 5506, 2009, pp. 571–578.

[15] M. Alshawabkeh, B. Jang, and D. Kaeli, "Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems," Proc. ACM Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 104–110.

[16] F. Azmandian, A. Yilmazer, J. G. Dy, J. A. Aslam, and D. R. Kaeli, "GPU-accelerated feature selection for outlier detection using the local kernel density ratio," Proc. IEEE International Conference on Data Mining (ICDM), 2012, pp. 51–60.

[17] D. W. Scott, Multivariate density estimation: theory, practice, and visualization, Wiley, 2009, vol. 383.

[18] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Fast outlier detection using a GPU," Proc. International Conference on High Performance Computing and Simulation (HPCS), 2013, pp. 143-150.

[19] T. Matsumoto and E. Hung, "Accelerating Outlier Detection with Uncertain Data Using Graphics Processors," LNCS, vol. 7302, 2012, pp. 169–180.

[20] F. Angiulli, S. Basta, and C. Pizzuti, "Distance-based detection and prediction of outliers," TKDE, vol. 18, no. 2, 2006, pp. 145–160.