A GPU Algorithm for Detecting Contextual Outliers in Multiple Concurrent Data Streams

Abinash Borah
School of Computer Science
University of Oklahoma
Norman, OK, USA
abinashborah@ou.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

Eleazar Leal
Department of Computer Science
University of Minnesota Duluth
Duluth, MN, USA
eleal@d.umn.edu

Egawati Panjei
School of Computer Science
University of Oklahoma
Norman, OK, USA
egawati.panjei@ou.edu

***Abstract*— A data stream is an infinite sequence of data points generated from a source continuously at a fast rate, which is characterized by the transiency of the data points, the temporal relationship among the data points, concept drift, and multi-dimensionality of data points. Outlier detection in data streams thus needs to deal with the characteristics of Big Data applications such as volume, velocity, and variety. The problem of detecting outliers in multiple concurrent data streams introduces additional challenges to the problem. In this paper, we propose a parallel outlier detection technique CODS to detect Contextual Outliers in multiple concurrent independent multi-dimensional Data Streams using a Graphics Processing Unit (GPU). The proposed algorithm addresses all the aforesaid characteristics of data streams. A set of experiments demonstrates reasonable outlier detection accuracy and scalability of CODS with the number of data streams.**

Keywords—Data Stream, Outlier Detection, Contextual Outlier, GPU

I. INTRODUCTION

A data point in a dataset that has a significantly different value compared to the other points in the dataset is called an outlier [1]. A contextual outlier is a data point whose value significantly deviates from the rest of the data points specified by the same context [1]. The context for a data point can be specified with one or more contextual attributes (s) such as time, space, etc. Outlier detection is the data mining task of identifying the outliers in a dataset by capturing their deviation from the expected behavior of the data points in the set. Outliers are unavoidable in any data acquirement process due to different factors [1]. Outlier detection can also be useful in finding previously unobserved interesting patterns in a dataset [2].

A wide range of applications demands outlier detection in data streams. A data stream is an infinite sequence of data points with explicit or implicit timestamps [3]. Examples of applications that require outlier detection in data streams are fault detection in an aircraft sensor [4], detection of erroneous sensor readings in environmental monitoring [5], network intrusion detection [6], etc.

While dealing with outlier detection in a data stream, the data volume to be examined is enormous and new data keep arriving and hence data cannot be stored entirely in memory for processing [7]. The rate at which real-life data streams are generated is high. Again, for data stream applications dealing with multiple data streams, different data streams may have different schemas [8]. Therefore, the task of mining outliers in data stream applications essentially deals with Big Data.

In addition to the above challenges pertaining to Big Data applications, outlier detection in data streams has to deal with issues such as the transiency of the data points, the temporal relation among data points, notion of infinity, concept drift, and multi-dimensionality [7], [9]. Applications such as network intrusion detection or detection of erroneous sensor readings [7] demand outlier detection in multiple concurrent data streams. In such contexts, the time constraints for outlier detection become even more stringent to deal with.

Despite these challenges, there are few works in the literature that have adopted parallel processing for outlier detection in data streams (refer to Section II). Graphics Processing Units (GPUs) are parallel co-processors that can support massive parallel computation with high instruction throughput. The efficacy of GPUs for outlier detection in data streams has been rarely explored. In this paper, we propose a parallel outlier detection algorithm for detecting **C**ontextual **O**utliers in **D**ata **S**treams (CODS) that detects contextual outliers in multiple concurrent independent multi-dimensional data streams using a GPU. The evaluation of the outlierness of a streaming data point in an appropriate context is important. For example, to find outliers in temperature readings generated by a sensor located at some place, a temperature reading should be compared with the other temperature readings in the temporal vicinity. Because an outlier temperature reading at noon can be an inlier when compared to the temperatures that are not in its temporal neighborhood, say temperatures of morning or night. In addition, CODS addresses all the above-mentioned characteristics of data streams.

The contributions of this paper are the following: 1) We propose an algorithm, CODS, for detecting contextual outliers in multiple concurrent independent multi-dimensional data streams using a GPU, which addresses the issues of data streams and GPUs; 2) We deal with the issue of scalability in terms of the number of data streams; 3) We evaluate the performance of CODS with a set of experiments using real-world and synthetic datasets.

The rest of this paper is organized as follows: Section II reviews necessary GPU concepts and related work; Section III formally defines the problem with the underlying assumptions; Section IV presents the proposed algorithm CODS; Section V discusses the results of the performance analysis; and finally, Section VI provides conclusions and future work.

II. Background and Related Work

In this section, we present the required background material on GPUs and discuss the related work.

GPUs are highly parallel co-processors installed in most computers for graphics rendering. It is possible to utilize the large-scale parallelism of GPUs for general purpose computing. GPUs are connected to the CPU through a relatively low throughput interface like PCIe bus. In this paper, we will use the terminologies of CUDA, an extension to C/C++ for GPU programming [10]. GPUs execute code functions called kernels [10], which are called from the CPU. Kernels launch a grid of simultaneously executing GPU threads, which are grouped into blocks. GPUs and CPUs have different memory spaces. This requires sending all input data through the PCIe bus before any processing can take place in the GPU and sending all output data from the GPU back to the CPU. GPUs have a hierarchical memory organization: threads have their own private registers; threads in a block can cooperate by using block-wide shared memory, and all threads across different blocks have access to the slower but much larger global memory.

To adeptly utilize the parallelism of GPUs, it is imperative to address the research issues of this architecture such as: 1) efficient use of the fast but limited amount of shared memory accessible to all the threads in a block; 2) global memory coalescing, which reduces the contention for the GPU's global memory by making consecutive threads access adjacent memory locations [11]; and 3) minimization of the amount of communication through the low throughput CPU-GPU interface.

Numerous works in the literature address the problem of outlier detection in data streams. Here we discuss related works relevant to the different aspects of our work.

A typical approach adopted for outlier detection in a data stream is the use of a count-based or time-based sliding window [12] to store a subset of the data points in the sliding window and evaluate the outlierness of the data points currently present in the window by capturing their deviation from the other data points currently in the window [13, 14, 15, 16, 17, 18, 19]. The outlierness of a data point is examined using a distance-based [13] or density-based approach [20] in these works within a temporal context specified by a window.

Density-based outlier detection in a data stream has been also explored in [21, 22, 23]. These works adopt the local outlier factor [20] for static data to data streams for measuring the local density of streaming data points from the average distance to its neighbors. The works in [7, 17, 24] are examples of other techniques that also use the density-based approach for outlier detection in a data stream.

A contextual anomaly detection technique for streaming sensor networks is proposed in [25] which uses the MapReduce model of parallel computing [26]. The work in [27] proposes a prediction-based contextual anomaly detection method for complex time series data.

Only a handful of the existing works performs outlier detection in multiple multi-dimensional data streams. In [28], a distance-based algorithm for outlier detection in multiple related data streams is proposed which processes a set of data points coming from multiple data streams together and detects the outliers among them based on their pair-wise distances. The two-phase algorithm in [29] first uses temporal correlation to identify outliers in an individual data stream and then uses the cross-correlations between the data streams to identify additional outliers. The work in [30] uses the LSHiForest data structure [31] to find outliers in multiple related multi-dimensional streams. These techniques, however, do not address the scalability issue in terms of the number of streams.

Parallel processing has been adopted for outlier detection in a data stream in [32, 33, 34]. These works implement a variety of parallel techniques for distance-based outlier detection on the Apache Flink platform [35]. [36] proposes a GPU-accelerated outlier detection algorithm for outlier detection in a multi-dimensional data stream using a kernel density estimation approach, while [37] proposes two different GPU algorithms for the problem using kernel density estimation [17] and local outlier factor [20] respectively. Both these works focus on exploiting parallelism for outlier detection in a single stream by maintaining summary statistics from the history data points of the stream. Therefore, the outlierness of a data point is evaluated in a global context, and not only in a local temporal context.

Although the problem of outlier detection in data streams has been extensively studied in the literature, there has been no effort towards finding contextual outliers from multiple concurrent data streams using parallel computing. In this work, we propose the first GPU based parallel algorithm that addresses the problem of contextual outlier detection in multiple concurrent independent multi-dimensional data streams. We also address the research issues of data stream outlier detection and GPU in this endeavor.

III. Problem Setting

We consider that there is a set of m concurrent data streams $S = \{S_1, S_2, \dots, S_m\}$. The i-th data point of data stream S_j is denoted by x_i^j. Each x_i^j is a $(d_j + 1)$ tuple of the form $< a_1, a_{2,\dots}\ a_{d_j}, t_i >$ such that $d_j \geq 1, \forall j$. Here, a_k's $(1 \leq k \leq d_j)$ are the attributes of the data points of stream j, d_j is the dimensionality of stream j and t_i is the associated timestamp of x_i^j (the time when x_i^j is received from its source for processing).

The m data streams have their arrival rates $R_1, R_2, \dots, R_m$; where, R_j is the arrival rate for data stream S_j and it defines how frequently a data point arrives for S_j.

We perform outlier detection using a time-based sliding window, i.e., a temporal context for outlier detection is specified by the time-based sliding window. We consider one time-based sliding window for each stream, i.e., m sliding windows are maintained for the m data streams.

The size of a sliding window, ω, determines that the window contains data points arriving from a stream in the last ω time units. ω is the same for all the m streams. Hence, depending on their arrival rates, different streams may have different numbers of data points in their respective sliding windows.

The slide size of a sliding window, σ, determines that when the window slides, the data points arriving from a stream in the last σ time units are added to the window removing the data points that had arrived in the oldest σ time units. σ is again the same for all the streams. Further, it is considered that the slide size σ is greater than the maximum of the arrival rates for the data streams and the window size ω is greater than the slide size σ.

IV. Proposed Algorithm

We now present our proposed GPU algorithm CODS for detecting contextual outliers within the settings formulated in the previous section.

A. Overview

CODS starts by transferring the data points arriving during the first ω time units from all the streams to the global memory of GPU (Lines 1-2 of Algorithm 1). The global memory array *S_d* is allocated in such a way that it can accommodate all the points that arrive in ω time units from all the streams in S in sequence, i.e., first the points in S_1, then the points in S_2, and so on. To overlap the data transfer and outlier detection kernel's execution (once the points in S_1 are transferred to the global memory of GPU, outlier detection is performed on those while the points in S_2 are still being transferred to the global memory of GPU, and so on), we make use of CUDA streams [10] and for this, a pinned memory [10] buffer is allocated in CPU to hold the data points from the streams. Before launching the CUDA operations, the data points from the sliding windows are copied to this pinned memory buffer. The kernel *Detect-Outliers* is then called (Lines 3-4) to find the outliers in each stream and the outliers among the points arriving during the latest σ time units are transferred to the CPU memory (Line 5). Subsequently, after every σ time units (Line 6), the data points from the streams received during the most recent σ time units are transferred to the global memory of GPU (Lines 7-8), outliers are detected in those (Lines 9-10) and detected outliers are transferred to the CPU memory (Line 11).

procedure CODS (S, R, ω, σ, r, Out)
// performs outlier detection in S using the values of A, ω, σ, r
Input: A set of m data streams $S = \{S_1, S_2, S_3, \ldots, S_m\}$, their corresponding arrival rates $R = \{R_1, R_2, R_3, \ldots, R_m\}$, sliding window size ω, slide size σ, neighbor distance r
Output: A set of outliers Out from S
1. Transfer the data points in the sliding windows of the streams from the
2. CPU memory to array *S_d* in GPU global memory after ω time units
3. Call kernel *Detect-Outliers* (*S_d*, r) which stores the detected outliers in
4. array O in GPU memory
5. Transfer the detected outliers O to CPU memory and store in set Out
6. **for each** slide of size σ
7. Transfer the data points from the streams received during the most
8. recent σ time units from CPU memory to array *S_d*
9. Call kernel *Detect-Outliers* (*S_d*, r) which stores the detected outliers
10. in array O in GPU memory
11. Transfer the detected outliers O to CPU memory and store in set Out
12. **end for**
13. **end procedure**

Algorithm 1. Pseudo-code of the CODS algorithm

We launch one block of GPU threads to perform outlier detection in the data points contained in the sliding window for each stream. To find the outliers among the data points that arrive from a stream during the latest σ time units, the kernel *Detect-Outliers* works in the following phases: (1) copying all the data points contained in the sliding window from the global memory to the shared memory of the thread block; (2) finding a set of vectors along which the outlierness of the data points is to be evaluated; (3) approximation of the neighbor density of the data points contained in the window along each of the vectors; (4) computation of an outlier score for each data point along each of the vectors based on the approximated neighbor densities of all the data points along the respective vector; and (5) deciding on the outlierness of the data points arriving in the latest slide from the computed outlier scores. We explain these tasks performed in the kernel *Detect-Outliers* (the pseudocode presented in Algorithm 2) in the remainder of this section.

B. Copy data points from global memory to shared memory

The threads in a block are assigned to copy the data points of the stream to be processed by the block from the global memory array *S_d* to the thread block's shared memory (Lines 3-4 in Algorithm 2) in such a way that consecutive threads copy consecutive attributes of the data points in *S_d*. This assignment of threads to the attribute values ensures that the global memory reads performed by the threads coalesce. It also ensures that shared memory bank conflicts [11] are minimized while performing the writes to the shared memory by all the threads in parallel.

C. Finding a set of vectors to evaluate outlierness

The idea of evaluating the outlierness of multidimensional data points by considering linear combinations of the original data dimensions is proposed in [7]. This work explains that the outlierness of the data points may not be revealed by all the linear combinations and hence it needs to be examined along different linear combinations. Instead of their idea of using an evolutionary algorithm for finding an optimal set of linear combinations of data dimensions, we propose to use a set of random vectors for the evaluation of outlierness. In this preliminary work, we choose the principal components computed from the covariance matrix of the data points contained in the window as the set of vectors for evaluating the outlierness of the data points.

After copying the data points of a stream to the shared memory, the thread block computes the covariance matrix in parallel (Line 5). Towards this, we first compute the mean vector of the data points in parallel by using a warp shuffle function for parallel reduction in CUDA [10] to sum up the values of each attribute of the data points. The mean vector is stored in the shared memory array after the data points so that it is accessible to all the threads. After this, all the threads subtract the mean vector from each data point in parallel. Once this centering of the data is done by subtracting the mean vector from each data point, the covariance matrix is computed by computing the variances and the covariances between the attributes by computing the dot product of the column vectors of the centered data. The computation of these dot products is again performed in parallel by all the threads in the block using parallel reduction with the help of a warp shuffle function. Again, we store the covariance matrix in the same shared memory array after the mean vector.

After computing the covariance matrix from the centered data points, the mean vector is added back to the data points to

restore their original values; this operation is performed in parallel by all the threads. All the threads in the block then work in parallel to compute the principal components by performing eigen decomposition of the covariance matrix (Line 6). We use the QL decomposition of a tridiagonal matrix implemented in [38] for the eigen decomposition of the covariance matrix. This

kernel *Detect-Outliers* (*S_d, r*)
// GPU kernel that performs outlier detection by copying the data points of streams from GPU global memory to shared memory of thread block
Input: GPU global memory array *S_d* containing the data points in the sliding windows, neighbor distance *r*
Output: A set of outliers O
1. $O \leftarrow \varphi$
2. **for each** data stream $S_j \in S$ **do in parallel** // thread blocks in parallel
3. Copy the current set of data points $X \subset S_j$ in parallel using all threads
4. in the block from the array *S_d* to the block's shared memory
5. Compute the covariance matrix C from X in parallel
6. Compute the set of principal components P from C in parallel
7. Copy the set of data points $X \subset S_j$ excluding the points received
8. during the least recent σ time units in parallel from the block's
9. shared memory to array *S_d*
10. **for each** principal component $p_i \in P$ **do** // $1 \leq i \leq d_j$
11. Compute the projected values of the data points in X along p_i **in**
12. **parallel**
13. **end for**
14. **for each** $p_i \in P$ **do**
15. **for** the data points $x \in X$ **do in parallel**
16. Compute data distribution $D(p_i)$ along p_i using the projected
17. values
18. Compute Neighbor-Density(x) using $D(p_i)$ and r
19. Compute mean μ of the Neighbor-Density(x) for $x \in X$
20. Compute Outlier-Score$(x) \leftarrow 1 -$ Neighbor-Density$(x)/\mu$
21. **end for**
22. **end for**
23. **for each** $p_i \in P$ **do**
24. Compute the standard deviation sd of Outlier-Score(x) along p_i
25. in parallel
26. **for each** $y \in X$ arriving in most recent σ time unit **do in parallel**
27. **if** Outlier-Score$(y) > 3 * sd$
28. $O \leftarrow O \cup \{y\}$
29. **end if**
30. **end for**
31. **end for**
32. **end for**

Algorithm 2. Pseudo-code of the kernel *Detect-Outliers*

implementation performs an in-place decomposition of the covariance matrix and replaces the covariance matrix with the principal components; hence reusing the same shared memory space to store the principal components. At this point, all the threads work in parallel to copy the current set of data points excluding the points received during the least recent σ time units (evicted from the sliding window) from the block's shared memory to the global memory array *S_d* (Lines 7-9) creating space for the new data points that will be transferred next from the CPU when the sliding window for the stream slides.

D. Neighbor density approximation for the data points

After obtaining the principal components, all the threads of the block now work in parallel to compute the projected values of the data points along each of the principal components (Lines 10-13). For this one data point is assigned to one thread for computing the projected values of that point along each of the principal components. We store the projected values of the data points along the principal components in the same shared memory space where the original attribute values are stored to avoid requiring additional space in shared memory. Since the number of data dimensions and number of principal components are equal, this perfectly meets the requirement to overwrite the original attribute values for a data point with the equal number of projected values, one along each principal component.

Once the projected values of the data points along each of the principal components are computed, we compute the distribution of the data points along each of the principal components (Lines 14-17). We create a fixed number of cells along one principal component for this purpose by dividing the range of projected values along that principal component into equal size intervals and count the number of data points that fall into each cell in parallel. To do that, the minimum and maximum of the projected values along the principal component are computed in parallel using a warp shuffle function and stored in the shared memory. To count the number of data points that fall into each cell in parallel, each thread maintains an array of size equal to the number of cells. Since the number of cells is specified in the compile-time, the threads can use their local registers to store this array [10] so long as the number of cells is within the limit of the number of registers a thread can have.

The threads then work in parallel to find their local counts in each cell, which is finally reduced through a series of warp shuffle functions to find the overall counts of data points in each cell along the principal component. Once the cell counts are obtained, the threads work in parallel to approximate the neighbor density (Line 18) of each of the data points along the principal component by summing up the counts in the cell where the data point lies and neighboring cells within the neighbor distance r. The neighbor density value along the principal component is stored in the shared memory array overwriting the projected value along the principal component. The above process is repeated for each of the principal components to approximate the neighbor densities of the data points along each of the principal components.

E. Computation of outlier score

The outlier scores for each of the data points x along each of the principal components are computed (Lines 19-20) from the approximated neighbor densities of all the data points along that principal component as: Outlier-Score$(x) \leftarrow 1 -$ Neighbor-Density$(x)/\mu$, where μ is the average of the neighbor densities of all the data points along that principal component. The outlier score computation is motivated from the Multi-granularity Deviation Factor proposed in [39]. Before computing the outlier score for the data points in parallel, μ is computed from the neighbor densities along the principal component using parallel reduction with warp shuffle. Here again, we reuse the shared memory space by overwriting the neighbor density values along the principal components with the outlier scores along the respective principal components.

F. Deciding the outlierness of data points

After obtaining the outlier scores for all the data points along each of the principal components, the threads in the block now work in parallel to decide the outlierness of the data points arriving in the most recent σ time units. We adopt the widely used three standard deviations test for outlier detection that

practically separates the rare events from the normal ones [40]. The threads first work in parallel to compute the standard deviation of the outlier scores of all the data points along a principal component (Lines 24-25) using parallel reduction with warp shuffle. Then the computed standard deviation is used to decide whether a data point received during the most recent σ time units is an outlier along that principal component (Lines 26-30). This is repeated for each of the principal components and a data point is classified as an outlier if it is found to be an outlier along any of the principal components.

V. Performance Analysis

In this section, we describe the experimental setup, datasets, values of the input parameters for CODS, performance metrics, and the experimental results obtained.

A. Experimental Setup

The experiments were carried out on an Ubuntu 14.04 workstation with two six-core Intel Xeon E5 2620v2 chips running at 2.1GHz and 64GB of DDR3 RAM; equipped with an Nvidia Quadro K5000 GPU. In our simulation model, there is one base station and multiple data sources, each producing a data stream. The data sources generate a data point every 5 milliseconds and send it to the base station. The simulation model is built using OpenMP multithreading and CUDA 10.

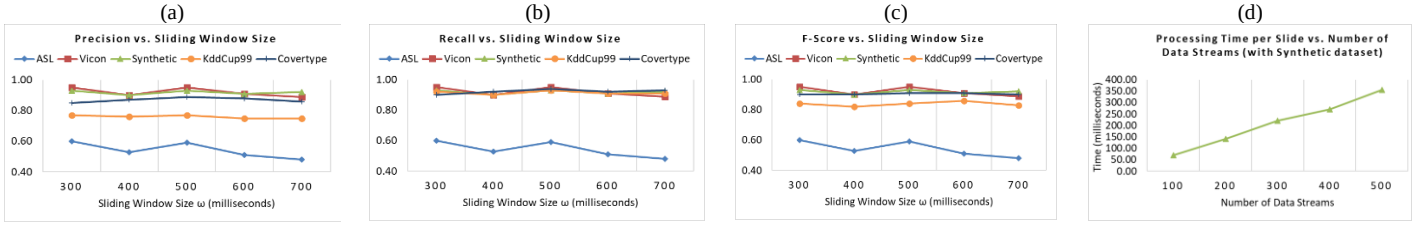

Fig. 1. Parameter Study for CODS

B. Datasets

We use four real datasets from the UCI machine learning repository: Australian sign language (150K records, 22 dimensions) [9], Vicon physical action (200K records, 27 dimensions) [9], KddCup99 (~3.8M records, 11 dimensions) [36] and Covertype (~498K records, 10 dimensions) [36] and a set of synthetic data streams (2.5M records, 50 dimensions).

C. Parameters

We use the values listed in Table I for the input parameters required for CODS with their default values indicated in bold. We use the small sizes of the sliding window as small window size produces better accuracy [41]. We use 64 GPU threads per block to process the small number of data points in a window.

TABLE I. Values For The Input Parameters

Parameter Name	Values
Sliding window size ω (milliseconds)	300, 400, **500**, 600, 700
Slide size σ (milliseconds)	**250**
Neighbor distance *r*	**25**
Number of Cells used along a principal component for density approximation	**100**
Number of data streams	**100**, 200, 300, 400, 500
GPU Threads per block	**64**

D. Performance Metrics

The outlier detection accuracy of CODS is evaluated in terms of Precision, Recall, and F-Score. A high precision value indicates low false positives, and a high recall value indicates that a high percentage of outliers are detected. The F-Score value gives an accuracy measure that considers both precision and recall. We also evaluate the average processing time (APT) in milliseconds required by CODS to process the newly arriving set of data points from all the streams in each slide.

E. Experimental Results

Table II shows the performance results of CODS obtained with the default values of the input parameters. The outlier data points in the Australian sign language dataset have similar values as the inliers in some dimensions. The projected values of the outliers along the principal components do not differ significantly from the inliers and thus do not cause the outliers to have low neighbor density. This results in the low accuracy of CODS on this dataset. For the other four datasets, the outliers and inliers have considerably different values along each dimension and CODS detects the outliers in these datasets with high accuracy. The average processing time taken by CODS to process the newly arriving set of data points from a hundred streams in each slide is short, between 15 to 70 milliseconds; this shows the effectiveness of adopting massive parallelism of GPUs to deal with large volumes of data.

TABLE II. Performance Results

Dataset	Precision	Recall	F-Score	APT (ms)
Australian sign language	0.59	0.67	0.63	33.5
Vicon physical action	0.95	0.99	0.97	37.8
KddCup99	0.77	0.93	0.84	16.7
Covertype	0.89	0.94	0.91	15.9
Synthetic dataset	0.93	0.96	0.94	70.3

We next discuss the impact of varying the sliding window size and the number of data streams on the performance of CODS.

1) Impact of the sliding window size:

The impact of varying the sliding window size on the datasets is shown in Fig. 1 (a) – (c). Although the impact of changing the sliding window size on the other datasets is not substantial due to the uniformly distributed outlier data points in these data streams, the impact is noticeable on ASL (the Australian sign language dataset). As the sliding window size increases, the accuracy measures for ASL degrade due to the non-uniformly distributed outliers and inliers in this dataset.

2) Impact of the number of data streams:

To study the impact of the number of concurrent data streams on the scalability of CODS, we vary the number of data streams using the Synthetic dataset. The steady increase in the average processing time per slide in Fig. 1(d) demonstrates the scalability of CODS with the number of data streams.

VI. Conclusions and Future Work

In this paper, we proposed a GPU algorithm CODS for detecting contextual outliers in multiple concurrent independent multi-dimensional data streams. The preliminary performance results show the efficacy of our approach for the fast processing of streaming data points from hundreds of streams. In future, we plan to develop GPU algorithms to detect outliers from multiple correlated data streams by exploring the correlations between the streams.

Acknowledgment

This work is supported in part by the National Science Foundation under Grant No. 1302439 and 1302423.

References

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," ACM Computing Surveys, vol. 41, no. 3, 2009.

[2] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Efficient and flexible algorithms for monitoring distance-based outliers over data streams," Inf. Systems, vol. 55, 2016.

[3] M. Stonebraker, U. Cetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," SIGMOD Rec., vol. 34, no. 4, Dec. 2005.

[4] S. Basu and M. Meckesheimer, "Automatic outlier detection for time series: an application to sensor data," Knowledge & Information Systems, vol. 11, no. 2, 2007.

[5] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," ser. SenSys, 2008.

[6] S. Babu, L. Subramanian, and J. Widom, "A data stream management system for network traffic management," in NRDM, 2001.

[7] S. Sadik, and L. Gruenwald, "Research issues in outlier detection for data streams," ACM SIGKDD Explorations Newsletter, vol. 15, no. 1, 2014.

[8] W. Wu and L. Gruenwald, "Research issues in mining multiple data streams," International Workshop on Novel Data, 2010.

[9] S. Sadik, L. Gruenwald, and E. Leal, "In Pursuit of Outliers in Multi-dimensional Data Streams," IEEE International Conf. on Big Data,2016.

[10] "CUDA C++ Programming Guide: Cuda Toolkit Documentation," NVIDIA Corporation, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Accessed September 28, 2021].

[11] "CUDA C++ Best Practices Guide," NVIDIA Corporation, 2021. [Online]. Available: https://docfs.nvidia.com/cuda/cuda-c-best-practicesguide/index.html. [Accessed September 28, 2021].

[12] L. Tran, L. Fan, and C. Shahabi, "Distance-based Outlier Detection in Data Streams," Proc. of VLDB Endowment. vol. 9, no. 12, 2016.

[13] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM, 2007.

[14] M. Kontaki, A. Gounaris, A. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams," IEEE 27th International Conf. on Data Engineering, 2011.

[15] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," IEEE 30th International Conference on Data Engineering, 2014.

[16] S. Yoon, J.G. Lee, and B.S. Lee, "NETS: Extremely Fast Outlier Detection from a Data Stream via Set-Based Processing," Proc. of VLDB Endowment. vol. 12, no. 11, 2019.

[17] X. Qin, L. Cao, E.A. Rundensteiner, and S. Madden, "Scalable Kernel Density Estimation-based Local Outlier Detection over Large Data Streams," Proceedings of the 22nd International Conf. on EDBT, 2019.

[18] S. Yoon, J.G. Lee, and B.S. Lee, "Ultrafast Local Outlier Detection from a Data Stream with Stationary Region Skipping," Proceedings of the 26th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, 2020.

[19] L. Tran, M.Y. Mun, and C. Shahabi, "Real-Time Distance-Based Outlier Detection in Data Streams," Proc. of VLDB Endow., vol. 14, no. 2, 2021.

[20] M. M. Breunig, H.P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," ACM International Conference on Management of Data (SIGMOD), 2000.

[21] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," Proc. CIDM, 2007.

[22] M. Salehi, C. Leckie, J. C. Bezdek, T. Vaithianathan, and X. Zhang, "Fast memory efficient local outlier detection in data streams," IEEE Trans. On Knowledge and Data Engineering, vol. 28, no. 12, 2016.

[23] G. S. Na, D. Kim, and H. Yu, "DILOF: Effective and Memory Efficient Local Outlier Detection in Data Streams," The 24th ACM SIGKDD International Conf. on Knowl. Disc. & Data Mining, 2018.

[24] S. Subramaniam et al., "Online outlier detection in sensor data using non-parametric models," Proc. of VLDB Endowment, 2006.

[25] M. A. Hayes and M. A. M. Capretz, "Contextual Anomaly Detection in Big Sensor Data," IEEE International Congress on Big Data, 2014.

[26] J. Dean, and S. Ghemawat, "MapReduce: Simplified data processing on large clusters,", Communications of the ACM, vol. 51, issue 1, 2008.

[27] K. Golmohammadi, and O.R. Zaiane, "Time Series Contextual Anomaly Detection for Detecting Market Manipulation in Stock Market," IEEE International Conf. on Data Science and Advanced Analytics, 2015.

[28] Ishida, K., and Kitagawa, H., "Detecting Current Outliers: Continuous Outlier Detection over Time-Series Data Streams," International Conference on Database and Expert Systems Applications, 2008.

[29] S. Sadik, L. Gruenwald, and E. Leal, "Wadjet: Finding Outliers in Multiple Multi-Dimensional Heterogeneous Data Streams,", IEEE 34th International Conference on Data Engineering, 2018.

[30] H. Sun et al., "Fast Anomaly Detection in Multiple Multi-Dimensional Data Streams," IEEE International Conference on Big Data, 2019.

[31] X. Zhang et al., "LSHiForest: A generic framework for fast tree isolation based ensemble anomaly analysis," IEEE International Conference on Data Engineering, 2017.

[32] T. Toliopoulos et al., "Parallel Continuous Outlier Mining in Streaming Data," IEEE 5th Int. Conf. on Data Science and Advanced Analytics, 2018.

[33] T. Toliopoulos et al., "Continuous outlier mining of streaming data in flink," Information Systems, vol. 93, 2020.

[34] T. Toliopoulos et al., "PROUD: PaRallel OUtlier Detection for streams," Proc. of the ACM SIGMOD Int. Conf. on Management of Data, 2020.

[35] https://flink.apache.org/ [Accessed September 28, 2021]

[36] C. HewaNadungodage, Y. Xia and J. J. Lee, "GPU-Accelerated Outlier Detection for Continuous Data Streams," IEEE International Parallel and Distributed Processing Symposium, 2016.

[37] K. Yu, W. Shi, and N. Santoro, "Designing a Streaming Algorithm for Outlier Detection in Data Mining – An Incremental Approach," Sensors (Basel), vol. 20, issue 5, 2020.

[38] W.H. Press et al., "Numerical Recipes in C: The Art of Scientific Computing," Second edition, Cambridge University Press, 1992.

[39] S. Papadimitriou et al., "Loci: Fast outlier detection using the local correlation," IEEE International Conference on Data Engineering, 2003.

[40] V. Barnett and T. Lewis, "Outliers in Statistical Data," John Wiley & Sons Inc, 1994.

[41] J. Clark, Z. Liu, and N. Japkowicz, "Adaptive Threshold for Outlier Detection on Data Streams," IEEE 5th Int. Conf. on DSAA, 2018.