Accelerating the Local Outlier Factor Algorithm on a GPU for Intrusion Detection Systems

Malak Alshawabkeh
Dept of Electrical and Computer Engineering
Northeastern University
Boston, MA
malshawa@ece.neu.edu

Byunghyun Jang
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA
bjang@ece.neu.edu

David Kaeli
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA
kaeli@ece.neu.edu

ABSTRACT

The Local Outlier Factor (LOF) is a very powerful anomaly detection method available in machine learning and classification. The algorithm defines the notion of local outlier in which the degree to which an object is outlying is dependent on the density of its local neighborhood, and each object can be assigned an LOF which represents the likelihood of that object being an outlier. Although this concept of a local outlier is a useful one, the computation of LOF values for every data object requires a large number of k-nearest neighbor queries – this overhead can limit the use of LOF due to the computational overhead involved.

Due to the growing popularity of Graphics Processing Units (GPU) in general-purpose computing domains, and equipped with a high-level programming language designed specifically for general-purpose applications (e.g., CUDA), we look to apply this parallel computing approach to accelerate LOF. In this paper we explore how to utilize a CUDA-based GPU implementation of the k-nearest neighbor algorithm to accelerate LOF classification. We achieve more than a 100X speedup over a multi-threaded dual-core CPU implementation. We also consider the impact of input data set size, the neighborhood size (i.e., the value of k) and the feature space dimension, and report on their impact on execution time.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance

Keywords

LOF, intrusion detection system, GPU, parallelization

GPGPU-3 March 14, 2010, Pittsburg, PA, USA

1. INTRODUCTION

The Local Outlier Factor (LOF) [3] algorithm is a powerful outlier detection technique that has been widely applied to anomaly detection and intrusion detection systems. LOF has been applied in a number of practical applications such as credit card fraud detection [5], product marketing [16], and wireless sensor network security [6].

The LOF algorithm utilizes the concept of a *local outlier* that captures the degree to which an object is an outlier based on the density of its local neighborhood. Each object can be assigned an LOF value which represents the likelihood of that object being an outlier. High LOF value are used to identify data objects that are potential outliers, whereas low LOF values indicate a normal data object.

1.1 Applying LOF to Intrusion Detection

Intrusion Detection Systems (IDSs) have become one of the fastest growing technologies within the security arena. As the ubiquity of computers and computer networks has grown, the threat posed and damage caused by cyber attacks has multiplied. Therefore, the need to develop robust and reliable IDSs is becoming increasingly important. There are two major intrusion detection techniques: signature-based detection and anomaly detection. Signature-based detection discovers attacks based on the patterns extracted from known intrusions. Such systems have low false positive rates, but cannot detect new attacks absent in the signature file. Anomaly detection, on the other hand, identifies attacks based on the deviations from the established profiles of normal activities, and the assumption is that attacks deviate from normal behavior. Most existing anomaly detection systems suffer from high false positive rates. This occurs primarily because previously unseen (yet legitimate) system behaviors are also recognized as anomalies, and hence flagged as potential intrusions.

To overcome this issue, there has been much interest in employing outlier detection methods for anomaly detection. Outlier detection methods aim to find the unusual activities that are different, dissimilar and inconsistent with respect to the remainder of the data set, based on some measure [2]. It has been shown that outlier detection techniques are effective in reducing the false positive rate with the ability to detect unseen attacks. LOF has been shown to perform

well in detecting abnormal behavior in a network Intrusion Detection System (IDS) [10]. LOF has been used to identify several novel and previously unseen intrusions in real network data that could not be detected using other state-of-the-art intrusion detection systems such as SNORT [14].

1.2 LOF Computational Overhead

Unfortunately, the LOF algorithm's time complexity is $O(n^2)$, where n is the data size. It is designed to compute the LOF for all objects in the data set, which results in a computationally intensive process, since it requires a large number of k-nearest neighbor queries. Because of this issue, designing efficient and reliable intrusion detection systems based on the LOF method is challenging.

Several methods have been proposed to reduce the computation time of the LOF method. The general idea of these approaches is to reduce the number of distances that need to be computed [1, 11]. However, these methods are very slow when working with high dimensional spaces (i.e., when the number of classification features is large).

Given the recent popularization of Graphics Processing Units (GPU), and the increased flexibility of the most recent generation of GPU hardware, and combined with high-level GPU programming languages such as CUDA, we would like to consider if LOF classification can be effectively accelerated using a GPU platform. The key to obtain good speedup with a GPU is to carefully map the data-parallel algorithm on to the available hundreds of processing cores. Many attempts have been made to use graphics processors for several traditional data mining techniques that include neural networks [15], support vector machine (SVM) [4] and recently for k-nearest neighbor (KNN) [7]. Since LOF shares a lot of similarities to KNN, this has motivated us to explore acceleration of LOF on a GPU platform.

In this paper we propose a CUDA implementation for the LOF algorithm to accelerate the processing throughput of an intrusion detection system. We compare runtimes of LOF run on an NVIDIA G200 class of GPUs. We compare this execution to runtimes on an X86 CPU. Our results show that we can accelerate LOF by more than 100X when using the GPU. We also study the impact of three different classification parameters that can impact the scalability of LOF GPU implementation in terms of performance: 1) the size of the input data set, 2) the number of neighbors, and 3) the feature space dimension. Our results show that changes to these parameters have only a small impact on the computation time on a GPU, whereas they have a much more dramatic effect on a CPU.

The rest of this paper is organized as follows. In Section 2 we describe the LOF algorithm considered in this work and discuss our implementation in CUDA. In Section 3, we describe our target intrusion detection application that has motivated this study. We present execution performance results in Section 4, and conclude the paper in Section 5.

2. THE LOCAL OUTLIER FACTOR (LOF) METHOD

2.1 Algorithm Overview

The main goal of the LOF method is to assign to each object a degree of being an outlier [3]. This degree is called the *local outlier factor* (LOF) of an object. It is *local* in the sense that the degree depends on how isolated the object is when compared to its surrounding local neighborhood. The concept of a local outlier is an important one since in many applications, different portions of a data set can exhibit very different characteristics, and it is more meaningful to decide on the possibility of an object being an outlier based on other objects in its neighborhood. In the LOF algorithm, the difference in density between a data object and its neighborhood is the degree of being an outlier, known as its local outlier factor. Intuitively, outliers are the data objects with high LOF values, whereas data objects with low LOF values are likely to be normal with respect to their neighborhood. Possessing a high LOF value is an indication of a low-density neighborhood, and hence, a higher potential of being an outlier. The computation of LOF values requires finding the k-nearest neighbors. To better understand the LOF method, we next provide a brief overview of the k-nearest neighbor (KNN) approach.

2.2 Review of the k-nearest neighbor method

The definition of the KNN search problem is: given a set S of reference points in a metric space M and a query point $q \in M$, find the the k nearest (closest) reference points in S to q. In many cases, M is taken to be d-dimensional space and the distance (closeness) is measured by either the Euclidean distance or Manhattan distance. Figure 1 shows an example of the KNN search problem with $k = 4$. A "brute force" approach is usually used to implement the KNN search. Following this approach, all distances between the query point q and all reference points in S are computed and then sorted in ascending order. The top k smallest reference points are then selected. This process should be repeated for all query points. The time complexity for this approach is $O(n^2)$.

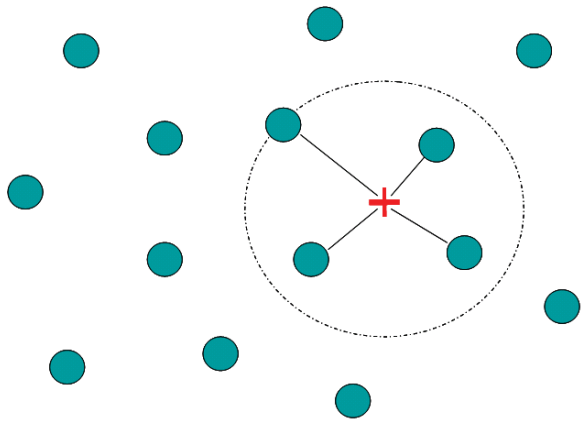

Figure 1: Example of KNN search problem, with $k = 4$.

2.3 The LOF algorithm

The algorithm for computing the LOF value of an object p in a data set D has several steps:

Let D be a data set, p, q, and o are some objects in D, and k be any positive integer. The distance function (for example Euclidean distance) $d(q,p)$ denotes the distance between objects p and q.

1. Computing (k-distance of p):
   For each data point p, find k-distance(p), the distance to its k-th nearest neighbor. Data points that lie within k-distance(p) to data point p are in its k-distance neighborhood.

   The k-distance of p is the distance $d(p,o)$ between p and o such that:

   (a) For at least k objects $o' \in D \setminus \{p\}$ it holds that $d(p,o') \leqslant d(p,o)$ and

   (b) For at most $k-1$ objects $o' \in D \setminus \{p\}$ it holds that $d(p,o') < d(p,o)$

   k-distance(p), provides a measure of the density around the object p, when k-distance of p is small that means the area around p is dense and vise versa.

2. Finding ($k-$distance neighborhood of p):
   The k-distance neighborhood of p contains every object whose distance for p is not greater than the k-distance.

   $$N_{k\_distance(p)}(p) = \{q \in D \setminus \{p\} \mid d(p,q) \leq k\_\text{distance}(p)\}$$

3. Computing the (reachability distance of p wrt object o):
   For each data point q in the k-distance neighborhood of p, define the reachability distance of p with respect to q as max{k-distance(q), $d(p,q)$}, where $d(p,q)$ is the distance between p and q. The reachability distance of object p with respect to object o is:

   $$reach\_dist_k(p,o) = max\{k\text{-distance}(o), d(p,o)\}$$

4. Computing (the local reachability density of p):
   The local reachability density of an object p is the inverse of the average reachability distance from the k-nearest neighbors of p:

   $$lrd_k(p) = \left[\frac{\sum_{o \in N_k(p)} reach-dist_k(p,o)}{|N_k(p)|}\right]^{-1}$$

   Essentially, the local reachability density of an object p is an estimate of the density at point p by analyzing the k-distance of the objects in $N_k(p)$. The local reachability density of p is just the reciprocal of the average distance between p and the objects in its k neighborhood. Based on local reachability density, the local outlier factor can be defined as follows.

5. Computing the local outlier factor of p:
   The local outlier factor is a ratio that determines whether or not an object is an outlier with respect to its neighborhood. $LOF(p)$ is the average of the ratios of the local reachability density of p and that of p's k-nearest neighbors.

   $$LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_k(p)|}$$

In order to illustrate the idea of the LOF approach, consider the simple two-dimensional data set shown in Figure 2. There is a much larger number of items in cluster $C1$ than in cluster $C2$, and the density of the cluster $C2$ is significantly higher than the density of cluster $C1$. With our notion of a "local" outlier, we wish to label both objects $p1$ and $p2$ as outliers. Due to the low density of the cluster $C1$, it is clear that for every item q inside the cluster $C1$, the distance between the item q and its nearest neighbor is greater than the distance between the item $p2$ and the nearest neighbor from the cluster $C2$, and so the item $p2$ will not be considered as outlier. Therefore, utilizing a simple nearest neighbor approach based on computing the distances fails to detect the outlier in these scenarios. However, the item $p1$ may be detected as an outlier using only the distances to the nearest neighbor. Alternatively, LOF is able to capture both outliers ($p1$ and $p2$) due to the fact that it considers the density around the points.

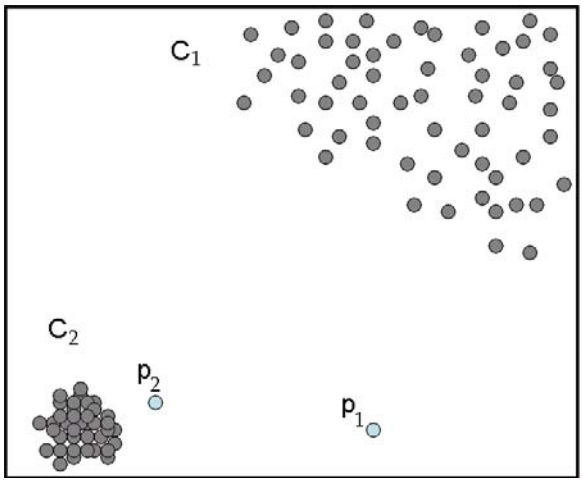

Figure 2: Example of LOF algorithm.

2.4 LOF CUDA implementation

The pseudocode of the LOF algorithm provided in Figure 3 shows that LOF exhibits a high degree of data parallelism. Given this property, LOF seems to be ideally suited for a GPU implementation. The compute-intensive step of the LOF algorithm lies in computing the reachability distances defined as: [$reach\_dist\_k(p,o) = max(k\_\text{distance}(o), d(p,o))$].

Computing reachability for p involves computing distances of all objects within p's neighborhood The runtime complexity of the algorithm is $O(n*$ time for a KNN query), where n is the size of the database D. For the KNN queries, if a brute force search method is used, the resulting computation has a complexity of $O(n^2)$ for LOF. However, since the computation of the distances between any pairs of points in a data set are independent, this can be fully parallelized on a GPU.

Our LOF CUDA implementation consists of multiple kernels, each of which is data dependent on the other; we could have integrated the multiple small kernels into one large kernel, though due to the sorting step of the algorithm, an in-

```
INPUT:   k (number of neighbors)
         D (a set of data points)

OUTPUT:  LOF values (a vector with local density factors)

ASSUME:  k_distance(D, p) – return a matrix that contains the
                            k_distances neighbors and their
                            k_distances
         reach_dist_k(p) – return the local reachability
                           density of each p ∈ D
BEGIN
  lof ⟵ NULL
  FOR each data point p
    KNNeighbors ⟵ k_distance (D, k)
            lrd ⟵ reach_dist_k (KNNeighbors, k)
    FOR each p in KNNeighbors
      templof[i] ⟵ sum (lrd[o∈N(p)])/lrd[i])/|N(p)|
            lof  ⟵ max(lof, templof)
  RETURN top(lof)
END
```

Figure 3: The LOF Algorithm

tegrated approach would be much less efficient. This series of kernels are invoked inside a loop iteration, with each loop iteration processing a subset of the input data set that fits nicely in GPU memory. We have shown in prior work that it is critical to perform a proper mapping of the data set to the GPU memory subsystem to obtain high performance [9].

To achieve high performance on a GPU using CUDA, we need to consider a number of optimization factors. During the course of our code development we applied two different classes of optimization techniques. First, we explored the best thread configuration to obtain the highest hardware utilization, based on the amount of shared memory used and the number of register used. Second, we employed different memory spaces available on the GPU to help improve memory bandwidth; we base our mapping decisions based on the data access pattern present in each computational kernel. These two optimization techniques have been shown to be extremely effective optimization techniques for CUDA programs [8, 12, 13]. In our case, we found that using shared memory for temporal data reuse and texture memory for more randomized memory access patterns together contributed significantly to the overall GPU speedup achieved. Proper memory mapping will increase the effective memory bandwidth and reduce the number of expensive off-chip memory accesses.

3. THE INTRUSION DETECTION APPLICATION

The motivating application for accelerating LOF is to construct an efficient intrusion detection system. In order to build an effective IDS using the LOF method, we need data to train our system. For this we employ a *profiling* approach that does not make assumptions about malicious behavior on the system being protected, other than that it is "outlier" from normal behavior. For example, in a production environment, new servers are often "stress tested" for many days or weeks with actual or realistic workloads before they are deployed. During such time, the behavior of the server can be profiled, and substantially different behavior encountered post-deployment can be flagged as potentially malicious.

Figure 4 demonstrates a high level design for the intrusion detection system framework. The input to the intrusion detection system is a stream of data extracted and transformed into features traces. These traces are then used for training LOF algorithm as well as for classification.

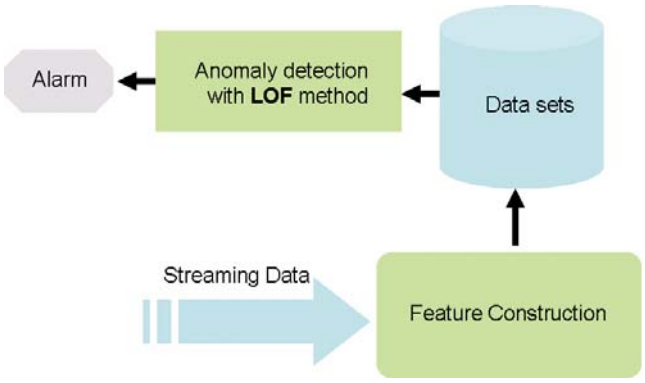

Figure 4: High level design of the intrusion detection system

4. EXPERIMENTS AND RESULTS

In this section, we report our experimental results, focusing on the scalability of our GPU implementation as compared to a multi-threaded CPU baseline. We also consider scalability in terms of three LOF design parameters: 1) input data size, 2) neighborhood size, and 3) space dimension (feature space). To demonstrate the scalability and speedups of our GPU implementation, we compare it with a multi-threaded CPU implementation developed in C (compiled with gcc 4.2). We run our GPU implementation on a NVIDIA GeForce GTX 285 GPU, using CUDA version 2.3. The CPU host system is equipped with an Intel Core 2 Duo running at 2.66 GHz with 2 GB main memory, running the Fedora 11 Linux operating system.

Tables 1 and 2, show the computation time in seconds for our two implementations LOF on a GPU and a CPU (LOF-CUDA and LOF-C, respectively). Here, k corresponds to the number of nearest neighbors, n corresponds to the number of reference and query points and d corresponds to the feature space dimension.

The main result of this paper is that by mapping the algorithm to a GPU, we can greatly reduce the execution time needed to compute LOF. According to Tables 1 and 2, our GPU execution (LOF-CUDA) is up to 100 times faster than LOF-C. For instance, with 3620 data points and with $k = 40$, the computation time is 24 seconds for LOF-C, whereas the execution is less than half a second for our LOF-CUDA implementation. Note that results in Table 1 are generated

k	LOF implementation	n=1810	n=3620	n=5430	n=10860
k=1	LOF-C	1	1	2	3
	LOF-CUDA	0.08	0.10	0.08	0.09
k=20	LOF-C	7	12	19	36
	LOF-CUDA	0.12	0.13	0.13	0.13
k=40	LOF-C	12	24	36	72
	LOF-CUDA	0.18	0.21	0.22	0.26
k=80	LOF-C	24	48	72	94
	LOF-CUDA	0.41	0.50	0.55	0.63
k=100	LOF-C	31	60	90	123
	LOF-CUDA	0.56	0.68	0.76	0.85

Table 1: Comparison of the computation time, with respect to the size of data set and the number of neighbors (all numbers in seconds)

.

d	LOF implementation	n=1810	n=3620	n=5430	n=10860
d=5	LOF-C	7	12	19	36
	LOF-CUDA	0.12	0.13	0.13	0.13
d=25	LOF-C	16	33	49	99
	LOF-CUDA	0.13	0.14	0.14	0.16
d=50	LOF-C	29	59	88	177
	LOF-CUDA	0.13	0.14	0.15	0.17
d=100	LOF-C	55	112	168	342
	LOF-CUDA	0.14	0.15	0.16	0.20
d=200	LOF-C	107	215	323	726
	LOF-CUDA	0.18	0.17	0.19	0.24

Table 2: Comparison of the computation time, with respect to the size of data set and the feature space dimension (all numbers in seconds)

.

with $d = 5$, and for Table 2 we set $k = 20$.

Next, we want to explore the sensitivity of our two different implementations to changes in input data size, size of neighborhood and space dimension. For our intrusion detection system, we will need to continually modify these parameters to produce the highest detection rate, while minimizing false positives. This may require us to run LOF for a range of values across each of these input parameters. Figure 5 shows the performance of the two implementations in terms of execution time in log-scale with respect to changing input data size. As the trends in the graph indicate, our GPU implementation scales very favorably as the input data size increases, making it very suitable for large-scale data processing.

In Figure 6 we plot the execution time vs. k (the size of the neighborhood). The computation time scales linearly with the size of k. The major difference between these implementations is the slope of the increase. Let us consider the case where $k = 20$, the slope of LOF-C is much steeper than the slope for the LOF-CUDA implementation.

The same trend can be seen in Figure 7, which shows the impact of changing the feature space dimension on execution time.In other words, our CPU implementation (LOF-C) is sensitive to data size, size of the neighborhood, and the space dimension in term of computation time. On the other hand, the impact of these factors on the performance of our GPU implementation (LOF-CUDA) is very small. Note that the y-axis for Figures 5, 6 and 7 uses a log scale.

Exploiting the fact that our GPU implementation is fairly insensitive to changes in the input parameters chosen is very attractive to our intrusion detection application. The process of analyzing and classifying data with LOF is generally limited when the size of data is large, or when the space dimension is high. In fact, LOF detection capabilities are very sensitive to the size of neighborhood. Our results show that with high values of k, classification accuracy is improved. However with our LOF CUDA implementation, we were able to construct an accurate and reliable intrusion detection system. The input to the intrusion detection system is a stream of data detected by light-weight profiling on the protected system. The data processed in a feature trace are used for both training the intrusion detection system, as well as for monitoring for malicious activity. Of course, near real-time performance is needed in order to detect any malicious activity before the system becomes fully compromised.

In Figure 8 we show an example of detecting a malware trace using the LOF method. Each + in this plot corresponds to a "malicious" (abnormal) data point and each ∘ corresponds to a "non-malicious" (normal) data point. The x-axis is time, and the y-axis is the LOF value (degree of outlying) assigned by LOF method. The vertical line represents the starting point of launching the malware on the system. For perfect detection, each + should have a high LOF value whereas each ∘ should have low LOF value. If the LOF-C implementation is used, it takes 54 seconds to detect an intrusion (much too long to be practical), while

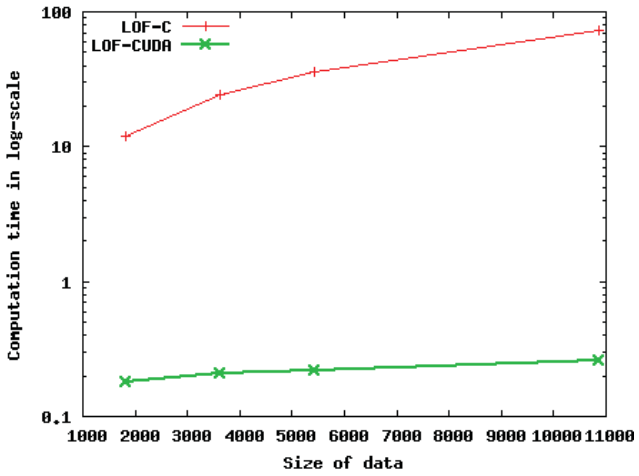

Figure 5: The impact of data set size on execution time. The y-axis is in log scale.

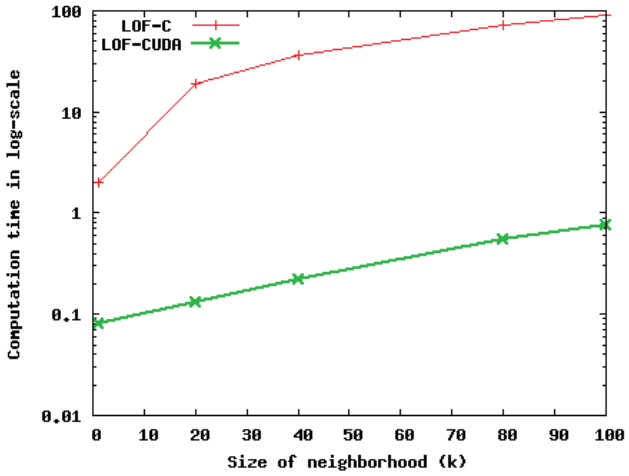

Figure 6: The impact of neighborhood size on execution time. The y-axis is in log scale.

with LOF-CUDA it only requires 0.13 seconds.

This considerable speedup we obtained exploits the highly data-parallel nature of the LOF method on a many-core GPU architecture. The desirable insensitivity observed from the experimental result benefits from the fact that GPU becomes more efficient as data size increases thanks to better memory latency hiding and hardware utilization. Note that the execution time of the CPU implementation follows the algorithm complexity as the input size increases. These two benefits make a GPU a very attractive platform for our intrusion detection application.

5. CONCLUSIONS

In this paper, we have developed a GPU-enabled CUDA implementation for the local outlier factor method. Our motivating application was to accelerate intrusion detection on a software security platform. This application requires

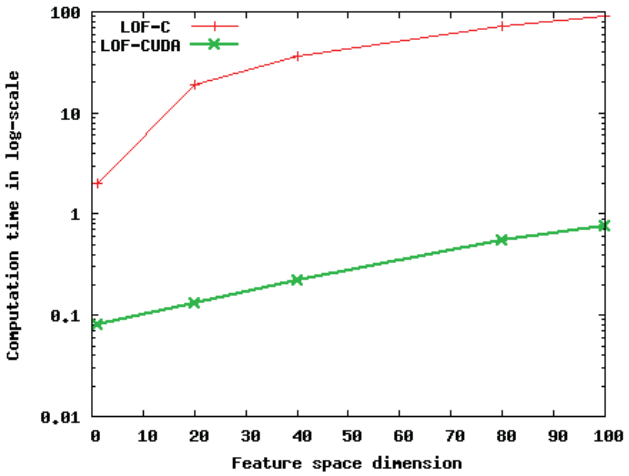

Figure 7: The impact of feature space dimension on execution time. The y-axis is in log scale.

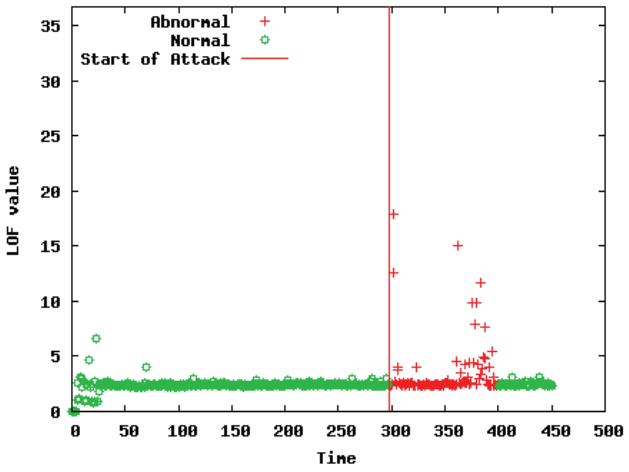

Figure 8: Example of detecting a malware using the LOF-CUDA implementation.

that we can perform both timely and accurate detection of anomalies. LOF is a very powerful outlier detection method. LOF defines the notion of local outlier in which the degree to which an object is an outlier is dependent on the density of its local neighborhood, and each object can be assigned an LOF which represents the likelihood of that object being an outlier. Although the concept of local outliers is a useful one, the computation of LOF values for every data object requires a large number of k-nearest neighbor queries and can be computationally expensive. Thus, it become a challenging issue when it comes to designing an efficient and reliable intrusion detection system. The popularization of the Graphics Processing Units (GPU), and the introduction of high-level GPU programming languages such as CUDA, have helped accelerate many algorithms that exhibit data-level parallelism.

In this paper we have presented a study comparing the execution time of LOF run on both a CPU platform and a

GPU platform. Besides gaining over a 100X speed-up, we observed that the size of the data set, the number of neighbors and the space dimension have only a small impact on the computation time when run on a GPU, versus the linear scaling experienced on a CPU.

In future work we will be exploring mapping additional machine learning applications to the GPU, and particularly focus on how to accelerate feature selection, which can be a compute intensive application. We also would like to consider how machine learning algorithms will scale on AMD GPUs, and will pursue future library development using OpenCL.

6. REFERENCES

[1] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[2] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons Chichester, 3rd edition, 1995.

[3] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *SIGMOD Conference*, pages 93–104, 2000.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 104–111, 2008.

[5] M.-C. Chen, R. Wang, and A.-P. Chen. An Empirical Study for the Detection of Corporate Financial Anomaly Using Outlier Mining Techniques. In *ICCIT '07: Proceedings of the 2007 International Conference on Convergence Information Technology*, pages 612–617, 2007.

[6] S. Ganeriwal, L. Balzano, and M. Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Transactions on Sensor Network*, 4(3):1–37, 2008.

[7] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *CVPR Workshop on Computer Vision on GPU (CVGPU)*, June 2008.

[8] B. Jang, D. Kaeli, S. Do, and H. Pien. Multi-GPU Implementation of Iterative Tomographic Reconstruction Algorithms. In *Biomedical Imaging: From Nano to Macro, 2009. ISBI 2009. 6th IEEE International Symposium on Biomedical Imaging*, Jun 2009.

[9] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli. Data transformations enabling loop vectorization on multithreaded data parallel architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 353–354, 2010.

[10] A. Lazarevic, L. Ertz, V. Kumar, A. Ozgur, and J. Srivastava. A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, 2003.

[11] E. Lozano and E. Acuna. Parallel Algorithms for Distance-Based and Density-Based Outliers. In *ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining*, pages 729–732, 2005.

[12] NVIDIA. CUDA Programming Guide 2.3, Jul 2009.

[13] NVIDIA. NVIDIA CUDA C Programming Best Practices Guide 2.3, Jul 2009.

[14] Snort. The SNORT Network IDS. www.snort.org.

[15] D. Steinkraus, I. Buck, and P. Simard. Using GPUs for Machine Learning Algorithms. In *Document Analysis and Recognition, Proceedings. Eighth International Conference*, 2005.

[16] J. Xi. Outlier Detection Algorithms in Data Mining. *Intelligent Information Technology Applications*, 1:94–97, 2008.