Fast Distance-based Outlier Detection using GPUs

Fernando Mussel, Carlos H. C. Teixeira, Wagner Meira Jr.

[1]Department of Computer Science – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{mussel,carlos,meira}@dcc.ufmg.br

Abstract. *Outlier detection is an important area of data mining with many practical applications, such as credit card and insurance fraud detection, network intrusion detection, etc. Distance-based detection methods, such as ORCA and DIODE, have stood out due to their parametric-free nature and good scalability on large and high dimensional datasets. In this paper we propose, a new parallel algorithm based on ORCA, which is designed to run efficiently on GPUs (Graphical Process Units). Then we discuss the main challenges pertaining its implementation and how we addressed them, in order to take full advantage of the GPU's parallel hardware. In our experimental analysis, we show that our algorithm not only is able to efficiently prune unnecessary distance computations, but can also achieve up to* $162X$ *speedup compared to state-of-the-art anomaly detection algorithms.*

1. Introduction

Outlier Detection in an important area of data mining, dedicated to find abnormal data records and patterns in datasets, i.e., records that present unusual or unexpected characteristics. These anomalies often carry useful information that can be employed in a wide range of practical applications, such as network intrusion detection, fraud discovery in credit card or insurance databases, as well as disease outbreak detection based on the analysis of patient data records, among several others.

However, there are several challenges associated with the outlier detection problem. First, the amount of data to be mined has grown in size (number of records), as well as in complexity (number of attributes). Second, many proposed methods for *outlier detection* require (1) expertise about the application domain (e.g. supervised methods, statistical approaches), and/or (2) they present high computational cost (e.g. unsupervised and non-parametric methods). Moreover, the notion of "normal" behavior is highly dependent on the application domain and can also change over time, i.e., time-series databases [Yankov et al. 2007, Gupta et al. 2012].

Over the last decades, distance-based methods for *outlier detection* have stood out due to their efficiency and good scalability on large and high dimensional databases. They usually do not demand any knowledge about the application domain and, therefore, are considered parameter-free approaches. The basic idea behind these methods is to project the dataset into a $\mathbb{R}^d$ space, such that data records become points and then to apply a *knn* search[1] to find the closest neighbours of a given point q. If q is very far from its closest points, then it is classified as an anomaly.

[1]Nearest-neighbour search

In principle, such methods have a $\mathcal{O}(M^2 \log p)$ worst-case, where M is the number of points in the dataset and p the number of outliers been detected, for they have to compute all pair-wise distances and then select the p largest in each row of the distance matrix. Hence a lot of research has been done to improve these methods in terms of running-time complexity, through the use of data partitioning, ordering and pruning rules [Ramaswamy et al. 2000, Orair et al. 2010, Ghoting et al. 2008, Bay and Schwabacher 2003]. One of the most important work was done in [Bay and Schwabacher 2003], where the authors propose ORCA , a new algorithm that achieves near linear running time complexity by randomizing the dataset and employing a simple pruning rule. In fact, the optimization introduced by this work was shown to be one of the most effective and, therefore, will be the basis of our research [Orair et al. 2010]. The ORCA algorithm will be discussed in further detail on section 2.

In this work, we explore another venue for improving the performance of outlier detection methods: the parallel capabilities of GPUs' SIMD-based architecture. In many research areas and in the industry, GPUs have been used for *General Purpose Computation* (GPGPU), leveraging their immense computational power and parallelism to accelerate workloads and algorithms. In data mining, for example, algorithms such as k-means and *knn* have been adapted to run on GPUs, attaining orders of magnitude speedups over their CPU implementation counter-parts [Li et al. 2010, Wasif and Narayanan 2011, Garcia et al. 2010, Ahmed Shamsul Arefin and Moscato 2012]. However, the uses of GPU in outlier detection have been surprisingly underexplored, limited to just brute-force implementations [Alshawabkeh et al. 2010], thus $\mathcal{O}(M^2 \log p)$ running time complexity, or by using approximate methods, with different definitions of outliers [Angiulli et al. 2013].

The core of this undergraduate research project is to investigate how to implement an efficient outlier detection algorithm on the GPU, with near linear running time complexity, by adapting the ORCA algorithm. Our main contributions are:

- We demonstrate that GPU brute force algorithms are not enough to accelerate the outlier detection problem and can be out-performed by highly optimized CPU implementations;
- We propose a GPU version of the ORCA algorithm, which efficiently applies the ANNS[2] pruning rule to avoid unnecessary distance computation;
- We show that our algorithm's pruning rule is as efficient as the CPU ORCA implementation, within a small constant factor;
- We show that our algorithm achieves up to $162X$ speedup over state-of-the-art, even while performing 6 times more computation.

This research paper is organized as follows: in section 2 we explain three important distance-based outlier detection algorithms, which will be referenced throughout the paper. In section 3 we propose a GPU outlier detection algorithm, based on ORCA , and discuss the major challenges and how they can be addressed. In section 4, we conduct an experimental analysis of our proposed algorithm, comparing its performance against the state-of-the-art methods. The related work are discussed in section 5. Finally, in section 6 we summarize our results and layout our future work.

[2]ORCA 's pruning rule. For more detail see section 2

2. Background

This section briefly discusses three important outlier detection algorithms, which will be referenced and used throughout this paper.

Algorithm 1: Orca-CPU algorithm

input : Dataset $\mathcal{D}$; **k**: # of neighbours to consider ; **p**: # of outliers to detect
output: $\mathcal{O}$, List of the p points with largest anomaly score, i.e. the outliers of $\mathcal{D}$
begin
  $\mathcal{O} \leftarrow$ min_heap() // Init. outlier list
  foreach $q_i \in \mathcal{D}$ **do**
    $dist\_vec \leftarrow$ max_heap() // Init. neighbour list
    // *knn* step
    foreach object r_j in $\mathcal{D}$ **do**
      heap_push ($dist\_vec$, ($d(q_i, r_j)$, r_j))
      if len($dist\_vec$) $> k$ **then**
        heap_pop ($dist\_vec$) // Keep the k closest neighbours
      // ANNS: Check if $d^k(q_i)$ is too low for q_i to be an anomaly
      if heap_head ($dist\_vec$)[0] $< d^k_{min}$ **then** break
    $d^k(q_i) \leftarrow$ heap_pop ($dist\_vec$)[0] // Get q_i's anomaly score
    heap_push ($\mathcal{O}$, ($d^k(q_i)$, q_i)) // Update the outlier heap
    if len($\mathcal{O}$) $> p$ **then** heap_pop($\mathcal{O}$) // Keep the p most abnormal points
  $\mathcal{O} \leftarrow$ sort_desc($\mathcal{O}$) // Sort outliers in descending order of score
  return $\mathcal{O}$

Brute-Force Such approach computes all the M^2 distance pairs. For each $q_i \in \mathcal{D}$, it finds q_i's nearest neighbours by computing the distance $d(q_i, r_j)$, $\forall r_j \in \mathcal{D}$. At the end of the *knn* search, the algorithm assigns $d^k(q_i)$ as the anomaly score of q_i, i.e the distance between q_i and its kth nearest neighbour. Finally, the algorithm classifies the p points with the highest anomaly scores as the outliers.

ORCA This algorithm is an improved version of the brute-force approach. As it performs the *knn* search of the outlier candidate q_i, it uses a heap to keep track of the k closest points to q_i, i.e. its neighbours. As ever closer points are added to such heap, $d^k(q_i)$, the distance between q_i and its k-th closest neighbour, decreases. It is important to understand, that until the *knn* search is completed, $d^k(q_i)$ is only an upper-bound of its actual value and, therefore, can be seen as an upper-bound to the anomaly score of q_i. Thus, if at any moment during the *knn* search, $d^k(q_i)$ becomes lower than d^k_{min}, the minimum anomaly score for an object to be considered an outlier, the search can be halted and q_i can be discarded as an anomaly. We refer to this optimization as *Approximate Nearest Neighbor Search* (ANNS) and its implementation on GPUs is the main challenge of our work. The algorithm 1 depicts, in greater detail, how ORCA works.

DIODE is an outlier detection framework, developed in [Orair et al. 2010], which implements 4 optimization methods on top of the ORCA algorithm, with the goal of increasing its pruning effectiveness. As a result, it performs as little distance computation

as possible, before discarding non-outliers. Note that 3 of these methods are dependent on a pre-processing phase for clustering data. We will not explain its optimization strategies due to space constraints, but we encourage anyone interested to read [Orair et al. 2010]. For the remainder of this paper, it suffices to know that this is one of the most efficient distance-based outlier detection algorithms in the literature. Thus, we will be using it in the experiment section, to assess how fast our GPU outlier detection algorithm is.

3. GPU-based Outlier Detection

In this section we start by arguing why a GPU brute-force approach is not enough to accelerate the outlier detection. Then we propose a GPU algorithm based on ORCA , which needs to compute much less distance pairs than a brute-force method; and discuss how to efficiently implement it on GPUs.

3.1. A Brief discussion on Brute-Force GPU Outlier Detection

As it was stated before, research on the use of GPUs to accelerate distance based outlier detection has been limited to brute-force approaches. These algorithms map nicely to GPUs due to the huge amount of independent work, i.e., M^2 distance computations which could be, in principle, performed in parallel – only limited by the amount of SIMD units on GPU's hardware. However, *state-of-the-art* outlier detection algorithms, such as ORCA and DIODE, may perform only a small fraction of the M^2 distance computations in order to find the p outliers. As a result, a GPU brute-force approach is not enough to accelerate the anomaly detection (see Table 1).

PARAMETERS		**BF-GPU**		**DIODE**	
k	p	Time (s)	Dist.	Time (s)	Dist.
4	0.05%	237.42	2250.00	21.13	0.12
128	0.05%	504.87	2250.00	46.58	0.15
16	0.1%	321.57	2250.00	35.50	0.17
16	0.6%	320.94	2250.00	188.75	0.88

Table 1. Brute Force GPU vs DIODE performance comparison, on the Kddcup dataset with $M = 1.5M$ points. *Time(s)* gives the average execution time in seconds, while *Dist* shows, in billions, the number of distance pairs calculated.

For instance, while processing the Kddcup dataset, the brute-force GPU algorithm (BF-GPU) is between 1.7 and 11.2 times slower than DIODE, specifically because it performs a lot more work. In other words, while DIODE only needs to calculate 0.15 billion distance pairs to find $p = 0.05\%$ of outliers, BF-GPU computes all the 2.25 trillion pairs, which corresponds to 15.000 times more computation. Therefore, we need to implement a GPU outlier detection algorithm that also prunes unnecessary distance computations. As such, we decided to implement a GPU version of the ORCA algorithm (Orca-GPU), since the Orca's ANNS pruning rule is the optimization that yields the most performance gains, as pointed out in [Orair et al. 2010].

3.2. ORCA-GPU: Algorithm Overview

The main challenge in implementing the ORCA algorithm on the GPU is how to efficiently map its pruning rule to the hardware, balancing the need to provide enough

Algorithm 2: Orca-GPU Algorithm

input : dataset $\mathcal{D}$; **k**: # of neighbours to consider ; **p**: # of outliers to detect; **r:** size of OCB; **s:** size of NCB
output: ($\mathcal{O}_{Id}$, $\mathcal{O}_{\mathcal{S}}$): List of outlier ids and scores
begin
  $(\mathcal{O}_{Id}, \mathcal{O}_{\mathcal{S}}) \leftarrow ([], [])$ `// Init. outlier id and score vectors`
  for $i \leftarrow 0$ **to** M, **step** r **do**
    $OCB \leftarrow$ load_pts $(\mathcal{D}, i, r)$ `// Load the next OCB`
    $pKnn \leftarrow []$ `// Init. Partial knn result matrix`
    for $j \leftarrow 0$ **to** M, **step** s **do**
      $NCB \leftarrow$ load_pts $(\mathcal{D}, j, s)$ `// Load the next NCB`
      $\rho^2 \leftarrow$ dist(OCB, NCB) `// Compute the pair-wise distances`
      $\rho^2 \leftarrow$ sort_asc (ρ^2) `// Sort neighbours in inc. order of distance`
      `// Update OCB's partial knn , with current NCB's results`
      $pKnn \leftarrow$ merge $(pKnn, \rho^2)$ `// Neighbour List`
      $S \leftarrow$ compute_score $(pKnn)$ `// Anomaly score`
      `// GPU-ANNS`
      $L \leftarrow$ map_pruning $(S, d^k_{min}, \&cand\_left)$ `// List of unpruned Out Cand`
      gather $(L, S, pKnn, OCB)$ `// Gather the data of the Out Cand left`
      if $cand\_left == 0$ **then break** `// CPU`
    if $cand\_left > 0$ **then**
      `// Update outlier list, with the Out Cand not pruned`
      $(\mathcal{O}_{Id}, \mathcal{O}_{\mathcal{S}}) \leftarrow$ update_outlist $(\mathcal{O}_{Id}, \mathcal{O}_{\mathcal{S}}, L, S, p)$
      $dkmin \leftarrow \mathcal{O}_{\mathcal{S}}[p]$ `// Get new` d^k_{min}
  return ($\mathcal{O}_{Id}$, $\mathcal{O}_{\mathcal{S}}$)

independent work to be processed in parallel, while still minimizing the amount of unnecessary distance computations performed.

GPUs are extremely parallel and optimized for throughput and, as such, they require a lot of concurrent work to be scheduled in order to keep all of its computation units busy. As a consequence, the detection algorithm needs to perform the *knn* search of multiple outlier candidates in parallel. Furthermore, the control flow of execution threads has to be the most similar possible, in order to minimize *branch divergence* (BD) – very costly on GPUs – and to ensure that the threads are performing *coalesced memory accesses* (CMA), i.e., adjacent threads reading from or writing to adjacent memory positions.

To work around the hardware constraints, our algorithm processes multiple outlier[3] and neighbour candidates[4] in groups, which we refer to as *outlier candidate batch* (OCB) and *neighbour candidate batch* (NCB). In other words, it performs the *knn* search of multiple outlier candidates (OCB), in parallel, comparing them to multiple neighbour candidates at a time (NCB). Algorithm 2 shows in more detail how it is done.
The GPU algorithm starts by loading the next OCB to be processed and begins their *knn* search, which can be divided in three main parts:

[3]Outlier Candidates are points $q_i \in \mathcal{D}$, that will have their anomaly score computed

[4]Neighbour candidates are points $r_j \in \mathcal{D}$ which will be compared to q_i, to check if they are nearest neighbours

1. **Point comparison:** In this step, the GPU computes distance matrix $\rho^2(OCB, NCB)$ and sorts each row i in ascending order, such that the first k distances of row i correspond to the distance between outlier candidate q_i and its k-nearest neighbours in NCB .
2. **Update Partial anomaly score:** Then, we obtain the lists of k points in $\mathcal{D}[0 \ldots j+s]$ that are the closest to the outlier candidates in OCB , i.e. the *partial knn result*. This is done by merging the *knn* result obtained from NCB , with the partial *knn* result that had been obtained so far. This updated partial *knn* result is stored in the matrix $pKnn$ and is used to update the upper-bound for the anomaly score of the outlier candidates in OCB .
3. **Pruning:** Finally, the algorithm uses the updated anomaly scores stored in vector S, to try to prune outlier candidates. For each $q_i \in OCB$, it checks if $S[i] < d^k_{min}$ to prune q_i. The pruned candidates have their data (i.e. attributes, anomaly scores, nearest neighbours) removed from the GPU buffers (i.e. OCB , S and $pKnn$)

For a given OCB , the algorithm will keep performing the *knn* searches until all the outlier candidates are pruned or all the NCB batches have been processed, i.e., we have searched for the nearest neighbours of OCB in the entire dataset. Once the processing of the current OCB is finished, the algorithm checks if there are any unpruned outlier candidates. If there are, then it updates the list of anomalies found so far, by inserting these outlier candidates and then sorting the list, using their anomaly scores as keys. Finally, the $d^k_{min}$is updated.

3.3. Trade-off: ANNS pruning efficiency $\times$ batching size

The rationale behind batching neighbour candidates is to provide a large enough group of points to be compared to the outlier candidates, such that the GPU is fully busy. In addition, by applying the ANNS rule only once per NCB batch, we ensure a high ratio between dense computation – distance calculations and sorting – and the amount of predicate evaluation and branching, which are expensive in the GPU but necessary to prune non-outliers. This minimizes the performance impact caused by the pruning on the detection execution time. Note that this approach is different from the one adopted by the CPU algorithm, which tries to apply the pruning rule after every neighbour candidate is compared to the outlier candidate q_i. As such, the Orca-CPU algorithm will perform less distance computation during the detection, since it will be able to prune non-outlier as soon as possible. Nevertheless, as we show in section 4, the Orca-GPU's pruning rule is still very effective at avoiding unnecessary computation.

Another important consideration is how to hide the cost of copying, from the GPU to the CPU, the number of outlier candidates left, so that the latter can either continue the *knn* searches or load a new OCB. We dealt with this issue in two ways: (1) we perform the copy asynchronously and overlap it with the executions of the kernel[5] *gather*; (2) we picked large enough OCB and NCB size, so that the communication cost becomes comparatively smaller, in relation to the computation cost.

4. Experiments

In this section we analyze the pruning efficiency and performance of our ORCA-GPU implementation, by comparing it against Orca-CPU and DIODE algorithms.

[5]Name given to functions which are executed on the GPU

4.1. Methodology

The ORCA-GPU algorithm was implemented using *OpenCL* and executed on an AMD R9290 with 4GB of memory and 2560 processors, clocked at 1.0 GHz. The CPU implementations were implemented in *C* language and compiled using gcc 4.8.3, with optimization level *-O3*. They were run on a system with a Xeon x3440 @ 2.53 GHz and with 16 GB of memory. We used three different datasets: (1) *Kddcup 1999*: Contains processed binary TCP data from a military computer network. We used a random sample from the original dataset, containing 1.5M points and 34 continuous attributes; (2) *COSMOS* [6]: dataset which contains data on the formation and evolution of galaxies, gathered from ground and space-based telescopes. We use a random sample, containing 1.4M points, with 32 continuous attributes ; (3) *2MASS* [7]: This is a Point Source Catalogue, from the All Sky Survey Mission, which contains data on 500 million stars and galaxies. We selected 1.4M points, each with 53 continuous attributes.

We will evaluate the performance of our algorithm, under different test configurations, by varying the number of neighbours, k, and the amount of outliers to be detected, p. Before running the GPU experiments, we determined empirically, that the best OCB and NCB sizes were 16384 and 8192, respectively.

4.2. Performance Comparison

Figures 1a and 1b show how much faster our GPU algorithm is than its CPU counter-part. When setting $p = 0.05\%$ and varying k, Orca-GPU achieved a maximum speedup of $117X$ for the *COSMOS* dataset and a minimum speedup of $58X$ for the *Kddcup* dataset. Moreover, as we increased p, the amount of outliers to be detected, the performance gap widen further, with the GPU achieving a maximum speedup of $319X$ for the *2MASS* dataset, reducing the detection time from 8700 seconds, to just 27 seconds.

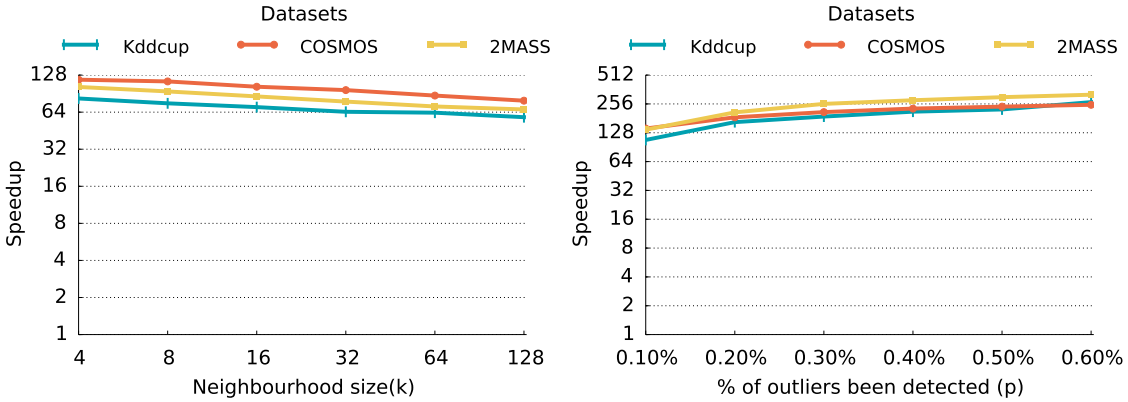
Figures 1c and 1d show that the speedups attained over DIODE are smaller and the reason is because the GPU is performing much more computation. As table 2 shows, Orca-GPU and Orca-CPU compute relatively the same amount of distance pairs, with the GPU computing at most 2.93 times more pairs. But relative to DIODE, the GPU performs between 24 and 418 times more computation. Nevertheless, our GPU algorithm still manages to offer considerable speedups over the state-of-the-art. When varying k for instance, the Orca-GPU is between $7X$ and $9X$ faster than DIODE for the *COSMOS* dataset, and up to $31X$ faster for the *2MASS* dataset. Moreover, as we increased p the speedups achieved were even bigger, with our algorithm being up to $8X$ faster on the *Kddcup* and up to $15.7X$ faster on the *COSMOS* dataset. The gains on the *2MASS* dataset are larger still. While DIODE performed the detection in 4439 seconds, with $p = 0.6\%$, the GPU took just 27.36 seconds, a $162X$ improvement.

4.3. Discussion on Orca-GPU Performance

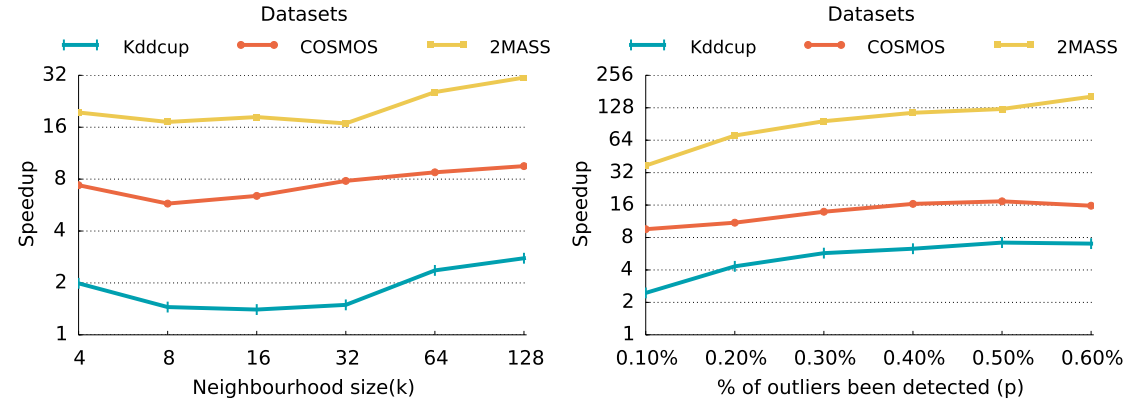
These experiments allow us to draw some important conclusions about the efficiency of our proposed algorithm. Firstly, as we predicted in section 3, the ANNS pruning rule on the GPU was not as effective as on the CPU, with the Orca-GPU computing between 1.81 to 2.93 times more distance pairs than its CPU equivalent. The main reason being, that

[6] http://irsa.ipac.caltech.edu/Missions/cosmos.html

[7] http://irsa.ipac.caltech.edu/Missions/2mass.html



(a) Speedup over Orca-CPU ; $p : 0.05\%$ **(b)** Speedup over Orca-CPU ; $k : 16$



(c) Speedup over DIODE; $p : 0.05\%$ **(d)** Speedup over DIODE ; $k : 16$

Figure 1. Speedups. In the first row, we report the speedup achieved by the Orca-GPU algorithm, over its cpu counter-part. In the second row, there are the speedups achieved over DIODE.

	k = 4						k = 128					
	Orca GPU		Orca CPU		DIODE		Orca GPU		Orca CPU		DIODE	
Dataset	Dist.	Ratio	Dist.	Ratio	Dist.	Ratio	Dist.	Ratio	Dist.	Ratio	Dist.	Ratio
Kddcup	49.0	1.00	16.8	2.91	0.1	418.88	40.4	1.00	18.1	2.24	0.1	272.24
COSMOS	51.5	1.00	23.9	2.16	0.5	104.94	55.9	1.00	30.8	1.81	0.7	78.70
2MASS	39.8	1.00	13.6	2.93	1.0	39.03	40.5	1.00	14.8	2.74	1.7	24.09

Table 2. Pair-wise distance calculations, for $p = 0.5\%$. *Dist.* reports, in billions, the number of pairs computed, whereas *Ratio*, shows the ratio between the amount of distance pairs calculated by Orca-GPU and the algorithm in question.

on the GPU the pruning rule is applied less frequently, once per NCB, in order to provide the GPU enough parallel work to fully utilize its hardware. This distance computation ratio might seem high, but it still allowed Orca-GPU to avoid computing up to 98% of all distance pairs on the Kddcup dataset, which is equivalent to 45 times less work than the brute-force approach would have performed.

In comparison to DIODE, our algorithm still had to perform up to 418 times more work, but this was expected since DIODE has other 4 optimizations in addition to the

ANNS pruning rule, which combined, greatly reduce the amount of distance computations required during the detection. However, because our GPU algorithm has a much higher distance computation throughput, it still was able to attain up to $162X$ speedup over the state-of-the-art algorithm, when detecting $p = 0.6\%$ of outliers.

These results are very promising and indicate that, if we are able to implement DIODE's optimization on the GPU, increasing its pruning efficiency, our algorithm could achieve minimum speedups of two orders of magnitude over the state-of-the-art.

5. Related Work

As it was mentioned in section 1, numerous research projects have been done into how to accelerate known data mining algorithms with GPUs. However, specifically in the area of outlier detection, development has been surprisingly limited.
In [Alshawabkeh et al. 2010], the authors implemented an anomaly detection algorithm for LOF outliers. It assigns to each object $q \in \mathcal{D}$, a *Local Outlier Factor*[8] based on how different is the density of q's neighbourhood and the density of the neighbourhood of each one of its k nearest neighbours. Their algorithm not only performed brute-force *knn* searches to find the neighbourhood of every point q, but it also stored the entire distance matrix on the GPU's memory, limiting its applicability to just very small datasets.

More recently, the authors of [Angiulli et al. 2013] implemented a family of GPU algorithms based on the approximate algorithm *SolvingSet* [Angiulli et al. 2006]. It computes the *knn* of every $q \in \mathcal{D}$ with respect to just a small object subset C_j, the outlier candidate set, instead of the whole dataset. This approximate *knn* search provides an upper-bound to the true anomaly score of q, which can then be used to discard the points as outliers. In their experiments, they show that the GPU implementations were able to achieve up to $45X$ speedup over their CPU SolvingSet code. However, it is important to note that they used an approximate algorithm to find outliers with a different definition than the ones that we use in this work.

6. Conclusion and Future Work

In this paper we proposed and implemented a GPU outlier detection algorithm based on Orca, which not only was much faster than its CPU equivalent, but also offered considerable speedups over the highly optimized DIODE algorithm, for all datasets tested. We discussed the challenges of implementing Orca's ANNS pruning rule on the GPU and showed that our proposed implementation was very effective at balancing two opposing requirements: provide enough independent work to be processed in parallel on the GPU and, at the same time, avoid as much unnecessary computation as possible. As a result, our implementation was up to $162X$ faster than DIODE.

For future work we intend to continue to improve our algorithm's performance, by implementing some of the optimizations that DIODE uses, in order to further reduce the amount of computation that the GPU has to perform. We expect that these changes will allow our algorithm to achieve much higher speedups over the state-of-the-art.

[8] LOF: an anomaly score.

References

Ahmed Shamsul Arefin, Carlos Riveros, R. B. and Moscato, P. (2012). Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus.

Alshawabkeh, M., Jang, B., and Kaeli, D. (2010). Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 104–110, New York, NY, USA. ACM.

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

Angiulli, F., Basta, S., and Pizzuti, C. (2006). Distance-based detection and prediction of outliers. *Knowledge and Data Engineering, IEEE Transactions on*, 18(2):145–160.

Bay, S. D. and Schwabacher, M. (2003). Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 29–38, New York, NY, USA. ACM.

Garcia, V., Debreuve, E., Nielsen, F., and Barlaud, M. (2010). K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3757–3760. IEEE.

Ghoting, A., Parthasarathy, S., and Otey, M. E. (2008). Fast mining of distance-based outliers in high-dimensional datasets. *Data Min. Knowl. Discov.*, 16(3):349–364.

Gupta, M., Gao, J., Sun, Y., and Han, J. (2012). Integrating community matching and outlier detection for mining evolutionary community outliers. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 859–867, New York, NY, USA. ACM.

Li, Y., Zhao, K., Chu, X., and Liu, J. (2010). Speeding up k-means algorithm by gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 115–122.

Orair, G. H., Teixeira, C. H. C., Meira, Jr., W., Wang, Y., and Parthasarathy, S. (2010). Distance-based outlier detection: consolidation and renewed bearing. *Proc. VLDB Endow.*, 3(1-2):1469–1480.

Ramaswamy, S., Rastogi, R., and Shim, K. (2000). Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec.*, 29(2):427–438.

Wasif, M. and Narayanan, P. (2011). Scalable clustering using multiple gpus. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10.

Yankov, D., Keogh, E., and Rebbapragada, U. (2007). Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 381–390.