DETECÇÃO DE EXCEÇÕES EM BASES DE DADOS MASSIVAS USANDO GPUS

FERNANDO AUGUSTO FREITAS DA SILVA DA NOVA MUSSEL

DETECÇÃO DE EXCEÇÕES EM BASES DE DADOS MASSIVAS USANDO GPUS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: WAGNER MEIRA JÚNIOR

Belo Horizonte

Março de 2017

FERNANDO AUGUSTO FREITAS DA SILVA DA NOVA MUSSEL

TOWARDS TERABYTE-SCALE OUTLIER DETECTION USING GPUS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Wagner Meira Júnior

Belo Horizonte

March 2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Mussel, Fernando Augusto Freitas da Silva da Nova

M989t Towards terabyte-scale outlier detection using GPUs / Fernando Augusto Freitas da Silva da Nova Mussel. — Belo Horizonte, 2017.
xxii, 98 f.:il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Wagner Meira Júnior.

1. Computação – Teses. 2. Mineração de dados (Computação) - Teses. 3. Detecção de anomalias (Computação) - Teses. I. Orientador. II. Título.

CDU 519.6*73(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Towards terabyte-scale outlier detection using GPUs

FERNANDO AUGUSTO FREITAS DA SILVA DA NOVA MUSSEL

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. WAGNER MEIRA JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ADRIANO ALONSO VELOSO
Departamento de Ciência da Computação - UFMG

PROF. GEORGE LUIZ MEDEIROS TEODORO
Departamento de Ciência da Computação - UnB

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de março de 2017.

Acknowledgments

These last two years have been an intense period of my life but they also allowed me to grow both professionally and personally. Now, I would like to take the time to express my gratitude towards those who supported me through this though period and helped me make today possible.

Thank you, Professor Wagner, for all the support, encouragement and guidance through my research. A major thank you to Carlos, whose assistance and experience was instrumental for my work. You spent countless hours discussing and troubleshooting with me, the issues that arose along the way. You deserve the highest praises for the help you gave me. I will be forever in your debt.

I would like to thank my family for supporting. My parents Wancleber and Iolanda, and my brother, Valner, always provided the encouragement and assistance through the years of study and self improvement. I also would to thank my good friend Giovander for the conversations and advices in times of need. My beloved dog, Liz, who always knew how to cheer me up, making tough days more bearable and the good ones even better. Finally, thank you Barbara. You listened, patiently, to me complaining about all sorts of problems I encountered during my research. You also took care of various minor issues and obligations I had during this time, allowing me to fully focus on my work. Your love and attention were paramount to help me go through the tough moments.

Thank you god for providing me with this learning experience and for putting such incredible people in my life, who helped me accomplish so much.

Finally, this work was partially supported by CNPq, CAPES, and FAPEMIG.

Resumo

Detecção de exceções é um importante método de mineração de dados, utilizado para encontrar registros inesperados em bases de dados. Essas anomalias comumente carregam informações úteis e podem ser utilizadas em diversas aplicações, tais como detecção de intrusões em rede, detecção de fraudes em bases de cartões de crédito e seguro, dentre outras.

Existem diversos desafios associados com detecção de exceções e o principal é o custo computacional. Muita pesquisa foi feita para melhorar a complexidade temporal de tais métodos, por meio de particionamento de dados, ordenação e regras de poda. Mesmo assim, o estado-da-arte só é capaz de detectar, em tempo hábil, uma pequena quantidade das top-n exceções. Recentemente, implementações para GPU foram propostas afim de contornar o custo computacional do problema. Os resultados obtidos foram promissores, porém, no melhor do nosso conhecimento, os algoritmos para GPU propostos até o momento estão restritos a processar bases de dados carregados na memória da GPU. Consequentemente, estes métodos têm aplicabilidade limitada pois não podem ser utilizados nos casos onde a GPU seria mais útil: bases de dados de larga escala.

O objetivo deste trabalho é utilizar GPUs para acelerar o processo de detecção de exceções em bases de dados de larga escala, residentes em disco. Dessa forma, desenvolvemos algoritmos e estratégias para minimizar a redução do *throughput* de computação causado por acessos ao disco. Este trabalho possui duas contribuições principais. Primeiro, nós desenvolvemos um conjunto de ferramentas e abstrações que facilitam a implementação algoritmos, para GPUs, de detecção de exceções em bases de dados armazenadas em disco. Entre tais abstrações temos uma nova estratégia de paralelização; algoritmos e kernels para operações essenciais à detecção de exceções; e um novo subsistema de I/O, capaz de reduzir o *overhead* de transferência de dados e permitir a execução concorrente de computação e I/O. Nossa segunda contribuição é um novo algoritmo, `DROIDg`, para a detecção de exceções, baseadas em distância, usando GPUs. Ele utiliza uma nova heurística de ordenação, a qual propusemos, que melhora

a eficiência de sua regra de poda, dessa forma reduzindo enormemente a quantidade de computação necessária para realizar a detecção.

Nossa análise experimental focou em determinar a aceleração que GPUs podem fornecer à detecção de exceções em bases de dados larga escala. Portanto, comparamos `DROIDg` contra alguns dos melhores algoritmos sequenciais *out-of-core* disponíveis na literatura: `Orca`, `Diskaware` e `Dolphin`. `DROIDg` alcançou speedups de 10X até 137X sob o melhor algoritmo para CPUs. Além disso, ele demonstrou escalabilidade consideravelmente maior com relação ao tamanho da base de dados e, também, do número de exceções sendo detectadas. Estes resultados demonstram que GPUs permitem realizar a detecção de exceções em escalas muito além do que, até mesmo, os algoritmos estado-da-arte para CPU são capazes.

Abstract

Outlier detection is an important data mining task for finding unusual data records in datasets. 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, among several others.

There are several challenges associated with the outlier detection problem and its computational cost is a major one. Significant research has been done to improve these methods' runtime complexity through the use of data partitioning, ordering and pruning rules. Though these advancements allow the outlier detection to be performed in near-linear time, they are not enough to enable processing large-scale datasets in a reasonable time. Even state-of-the-art methods are limited to processing small scale datasets and/or limited to find just a tiny fraction of the true top-n outliers. Recently, GPU-based implementations have emerged as an alternative to address the computational bottleneck. They have shown promising results but, to the best of our knowledge, all distance-based GPU algorithms currently available are designed for in-memory detection: they require the dataset to fit and be loaded into the GPU's memory. Consequently, their applicability is limited because they can not be used in scenarios where the GPU's computational power would the most useful: to process large scale datasets.

The goal of this work is to use GPUs to accelerate the outlier detection process in terabyte-scale, disk-resident datasets. To achieve it, we have to develop algorithms and strategies to overcome the massive reductions in the GPU's computation throughput caused by disk accesses. We made two main contributions in this work. First, we developed set of tools and abstractions for out-of-core distance-based outlier detection in GPUs, such as an effective parallelization strategy; algorithms and high-performance GPU kernels of essential operations for distance-based outlier detection; and an I/O subsystem that reduces data transfer overhead while allowing I/O and computation overlapping. The second main contributions is the development of a novel distance-based outlier detection algorithm for GPUs, `DROIDg`, capable of processing large scale

and disk-resident datasets in reasonable time. It leverages a new ranking heuristic, proposed by ourselves, to improve the efficiency of its pruning rule, thereby massively reducing the amount of computation required by the detection.

Our experimental analysis focused on assessing the performance benefits of using GPUs for outlier detection in large-scale datasets. Thus, we compared `DROIDg` against some of the best out-of-core outlier detection algorithms available for CPUs: `Orca`, `Diskaware` and `Dolphin`. `DROIDg` achieved speedups between 10X and 137X over the best sequential algorithm. Moreover, it displayed far superior scalability with regards to the dataset size and number of outliers being detected. These results showed that GPUs enable the outlier detection to be performed at scales far beyond what even state-of-the-art CPU algorithms are capable of.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Outlier Detection in an important data mining task for finding unusual data records and patterns in datasets. 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.

There are several challenges associated with the outlier detection problem. First, many of the proposed methods, such as supervised and statistical methods, require expertise on the application domain. Second, the notion of "normal" and "unusual" behavior can vary widely depending on the usage context and it can even change over time, such as in time-series datasets (Yankov et al. [2007]; Gupta et al. [2012]). Third, outlier detection is a computationally demanding task, specially unsupervised and non-parametric methods. With the ever increasing size (number of records) and complexity (number of features) of datasets, the computational cost of outlier detection becomes an ever greater issue to overcome.

There are a myriad of detection approaches, but distance-based methods have stood out for their simplicity and good scalability on large and high dimensional datasets. Moreover, they usually demand little knowledge about the application domain and, therefore, can be easily adapted and used in different contexts. The basic idea behind these methods is to project the dataset into an $\mathbb{R}^d$ space, such that data records become points. Then, Nearest Neighbor (KNN) searches are used to find the neighbors[1] of all dataset points p. If p is very far from its neighbors, then it is classified as an anomaly. In principle, these techniques have a $\mathcal{O}(N^2 \cdot \log n)$ worst-case runtime complexity, where N is the size of the dataset processed and n is the amount of outliers requested. Significant research has been done to improve these algorithms'

[1] Set of closest points to p. See Definition 4

runtime complexity, through the use of data partitioning, ordering and pruning rules (Knorr and Ng [1999]; Ramaswamy et al. [2000]; Bay and Schwabacher [2003]; Ghoting et al. [2008]; Orair et al. [2010]). They successfully reduced the average-case complexity to near-linear time, but it was not enough to allow processing terabyte-scale datasets in reasonable time. Outlier detection remains a challenge even for state-of-the-art algorithms, hence they are limited to processing smaller scale datasets. Additionally, the vast majority of methods in the literature are limited to find just a tiny fraction of the true top-n outliers of a dataset, typically less than 1000, which considerably limits their usefulness. For instance, imagine a scenario where outlier detection is used to clean a sensory data dataset. $n = 1000$ outliers equates to less than 0.00034% of the smallest dataset used in our experimental analysis (300 million data points) and characterizes an unreasonably small error rate in many practical applications. Therefore, even state-of-the-art outlier detection algorithms for CPUs would not be fit for removing noise from datasets of this magnitude, which can be considered relatively small for today standards. Therefore, addressing this computational challenge is paramount to improve the applicability of outlier detection techniques and is the focus of this work.

Recently Graphical Processing Units (GPUs) have been widely used by the scientific community to accelerate computational workloads, including outlier detection. GPUs are embarrassingly parallel, with thousands of processing elements connected to a high bandwidth memory hierarchy and capable of several TFlops[2] of computation throughput. However, to use this computation capability effectively, algorithms need to take into consideration the GPU's architectural constraints, such as its SIMD[3] design and relatively small memory buffer. The outlier detection algorithms for GPU currently available have successfully exploited the GPU's parallel hardware to address the computational bottleneck. But, to the best of our knowledge, all of them are only capable of in-memory detection, i.e. processing only datasets that are entirely loaded in the GPU's memory. Such constraint limits the applicability of these algorithms, since they can not be used in scenarios where the GPUs' computational power would be the most useful: to process large scale datasets.

Therefore, the goal of this work is to develop a new distance-based outlier detection algorithm for GPUs, that is capable of processing terabyte-scale, disk-resident, datasets. The algorithm will exploit the problem's parallel nature and leverage the GPU's architecture to significantly reduce the detection time. The goal is to push the boundaries and scale of outlier detection analyses that can be performed in reasonable time, both in terms of the size of the dataset processed, as well as the amount of top-n

[2] One trillion floating point operations per second.

[3] Single Instruction Multiple Data

outliers that can be detected. However, there are two main challenges to overcome. First, the cost of loading data into the GPU's memory is higher than simply loading data into RAM. Second, the penalties due to I/O bottleneck are much more severe for GPU algorithms than for their CPU counter-parts. That is the case because GPUs have far superior computation throughput than CPUs, thus the amount of computation potential wasted per unit of idle time is equally as large. To overcome these challenges and achieve our stated goals, our new GPU algorithm will have to: (i) achieve a good trade-off between reducing I/O and reducing computation; and (ii) extract enough parallelism from the problem to fully leverage the GPU's highly parallel hardware. Balancing these requirements is not a trivial task.

Our algorithm is designed to be used in *Extract, Transform and Load (ETL)* workloads, as a pre-processing step for detecting outliers. In other words, it is not suited for streaming applications but rather for use cases where the entire data is available beforehand. Despite this restriction, it is still widely applicable and can used in any practical application that requires the analysis of a large dataset.

1.1 Contributions

In this work, we remove a major barrier for wide-spread use of GPUs for outlier detection: dataset size. Our main contributions are the following:

- A new parallelization strategy that balances high computation throughput with the use of pruning rules to reduce the overall amount of computation required for outlier detection. Both absolute essential when dealing with large scale datasets.
- New core abstractions and I/O subsystem that significantly reduces data transfer latency and improves throughput by using two main optimization: (i) binary encoded datasets and (ii) asynchronous data transfers.
- A novel outlier detection algorithm for GPUs, capable of processing large scale, disk-resident, datasets. It uses a new outlier candidate[4] sorting heuristic, that allows the algorithm to better balance the reduction in computation and disk accesses. Consequently, our algorithm achieves better performance than competing methods in a wider range of datasets and detection configurations.
- Performance evaluation of our proposed algorithm, comparing it to both CPU and GPU algorithms. We showed that by using GPUs, the outlier detection process can be accelerated by up to 137X, when compared to state-of-the-art sequential

[4]Points likely of being outliers. See Definition 10

methods. Moreover, we showed that our algorithm has a more robust performance and is less sensitive to the quality of the initial classification threshold used. This makes it more flexible and, currently, the only viable option for processing very large datasets, using small k, where the classification threshold can not be estimated.

- In Appendix A we provide an in-depth explanation on how to implement, using *OpenCL*, the low-level parallel primitives and computation functions necessary to perform outlier detection.

1.2 Organization

This work has four more chapters and an appendix. Chapter 2 presents basic concepts pertaining to outlier detection and discusses the most commonly used detection strategies. Chapter 3 explains in more detail the fundamentals of distance-based outlier detection methods and the main algorithms available in the literature. Chapter 4 has four main parts. First, it discusses basic concepts about the GPU's architecture, execution and programming model. Then, it presents part of our GPU outlier detection framework, discussing the parallelization strategy, basic functions and I/O optimization. Next, it shows how to use such framework, to port to the GPU, well known out-of-core outlier detection algorithms. Finally, it presents our novel and more robust out-of-core algorithm for GPUs. Chapter 5 contains a thorough experimental analysis of both CPU and GPU algorithms discussed. First, it shows how much the GPUs are capable of accelerating the detection in large scale datasets. Second, it analyses the performance characteristics of the GPU algorithms implemented and shows that our algorithm is more robust. Appendix A discusses in detail how to efficiently implement the crucial operations for distance-based outlier detection: norm computation, distance computation and k-selection.

Chapter 2

Outlier detection techniques

This chapter provides a review of the main outlier detection techniques available, discussing their rationale, assumptions, advantages and disadvantages. Next, we define three terms that will be used extensively in work.

Definition 1 ***(Outliers)*** *Outliers, also referred to as anomalies, are data points and patterns that do not abide to a definition of "normal behavior".*

Note that the notion of *normal behavior* and the exact definition of an outlier are application dependent.

Definition 2 ***(Inliers)*** *Inliers are points that follow the expected/normal behavior of a dataset. Non-outliers.*

Definition 3 *(**Test point**) Synonym of **Test instance**. Point that will be classified by the algorithm as either an outlier or not.*

2.1 Statistical Methods

Statistical methods for outlier detection rely on the following basic assumption:

Assumption 1 *Normal instances occur in high probability regions of the stochastic model while anomalies occur in the low probability regions. (Chandola et al. [2009])*

These methods fit probability models to the data and use statistical inference tests to determine how likely it is that unseen data instances were generated or not by the fitted model. The unlikely ones are classified as outliers.

Statistical methods can be broadly divided into two types: Parametric and Non-parametric techniques. The *parametric* techniques assume the data have a specific underlying distribution model and they estimate the parameters of said model. These techniques usually work in one of two ways. In the first approach, each test instance is assigned an anomaly score inversely proportional to the probability of it being generated by the assumed model. Those instances with an anomaly score above a certain threshold are considered outliers. In the second approach, outliers are detected using hypothesis tests where the **null** hypothesis (H_0) is that instance p was generated by the assumed model. If H_0 is rejected, p is regarded as an outlier. For example, methods using variations of the *Grubb's* test (Chandola et al. [2009]) were used for outlier detection in high dimensional datasets (Aggarwal and Yu [2001]) and in graph structured data (Shekhar et al. [2001]).

Non-parametric statistical methods do not assume any knowledge of the data's underlying generative model and instead they build a model directly from the data themselves. For instance, a very common type of non-parametric detection method uses attribute-wise histograms to segregate the data. Given a data point p, each of its attributes is assigned an anomaly score equal to the inverse of the attribute's bin frequency. p's final anomaly score is computed as an aggregate of its per-attribute anomaly scores. This basic technique was used in many applications such as structural damage detection (Worden et al. [2000]); web-based attack detection (Kruegel and Vigna [2003]); network intrusion detection (Yamanishi et al. [2004]).

Statistical-based methods have limited applicability beyond low dimensional datasets. Their assumption that the data is generated by a specific distribution is rarely true for real multivariate datasets. Moreover, in the case of the non-parametric histogram-based methods, they fail to detect rare but important inter-attribute interactions (Chandola et al. [2009]).

2.2 Classification-based Methods

Classification-based methods can be used to detect outliers when the following assumption holds:

Assumption 2 *A model can be learnt to distinguish between normal and anomalous classes, given a feature space.*

Such methods divide the detection into two phases. During the training phase, the classifier chosen learns a model from the labeled training instances. Then, in the

test phase, the trained classifier is used to classify test instances as either normal or anomalies.

There are two different application scenarios for classification-based techniques. In an *one-class* classification setting, all training data are normal instances and the classifier learns a classification boundary around them. Test instances that fall outside such boundary are considered anomalies. For example, Manevitz and Yousef [2001] used an one-class SVM to classify documents, whereas Roth [2004] proposed an outlier detection method using one-class *Kernel Fisher Discriminants*. Alternatively, in a *multi-class* classification scenario, multiple normal classes exist and the classifier builds one model per class. During the test phase, the outliers are those instances that are not classified as normal by any of the classifiers.

Many different types of classifiers were proposed for use in outlier detection, such as *Neural networks* (Augusteijn and Folkert [2002]; Diaz and Hollmén [2002]); *SVMs* (Davy and Godsill [2002]; Heller et al. [2003]). These methods have two main disadvantages. First, their output consists of just the classification label, but many applications require anomaly scores for ranking and other purposes. Second, these methods need labeled datasets, which may be hard or impossible to obtain, specially if multiple normal classes exist.

2.3 Clustering-based Methods

Clustering-based methods rely mainly on the clustering of data instances for detection. Simpler methods rely solely on the instances' cluster assignment itself, whereas the more sophisticated ones use anomaly scores. These are assigned to data points and computed based on metrics pertaining to the point itself and the cluster that it belongs to.

These types of methods can be broadly divided into three types. The first type is the simplest and it detects anomalies as by products of the clustering operation. They work based on the following assumption:

Assumption 3 *Normal data instances belong to a cluster, while anomalies do not belong to any. (Chandola et al. [2009])*

These methods cluster the dataset using a clustering algorithm that does not force every data point to belong to a cluster, e.g., DBSCAN (Ester et al. [1996]) and SNN (Ertöz et al. [2003]); and the points assigned to no cluster are classified as outliers.

A second type of cluster-based methods make the following assumption

Assumption 4 *Normal data lie close to their cluster centroid, while anomalies are located in the outskirts of the cluster.*

For these methods, the distance from a point p and the centroid of its cluster is interpreted as p's anomaly score and those points with a score larger than a given threshold are considered outliers. The advantage these methods have over the previous type is the lack of restrictions on the clustering algorithm to be used. Moreover, their output is not binary, i.e., *Normal or Outlier*, as they may provide a rank of points according to their anomaly score. However, the main drawback of these methods is their inability to detect anomalies that form clusters themselves.

The third and last type of clustering-based methods is the most sophisticated and relies on the following assumption:

Assumption 5 *Normal data belongs to large and dense clusters, where as anomalies belong to small and sparse clusters.*

These algorithms can combine multiple metrics, e.g., size, density and distance from point p to the clusters' centroid, to compute the anomaly score of p. Consequently, the quality of the detection is superior to that of the other methods. Examples of such methods are the `FindCBLOF` algorithm (He et al. [2003]), which uses both distance to centroid and cluster size; and Pires and Santos-Pereira [2005] whose method's anomaly score considers both the size of the cluster and the *Mahalanobis* distance between the data point and every cluster found.

The main disadvantage of these techniques is that anomalies are just by products of the clustering itself. Thus, these methods depend on the clustering algorithm correctly capturing the cluster structure of the normal data, such that the outliers can be single out among the other points. Another issue with clustering-based outlier detection is the computational cost of finding the clusters, specially if quadratic algorithms are used.

2.4 Distance-based methods

As we explain later in Section 2.5, our GPU algorithms will use a distance-based outlier detection approach. Consequently, we will discuss this type of technique in much greater detail and the discussion will be divided into two parts. In this section we will provide an overview on this type of method, discussing the specific outlier definitions used, assumptions and rationale behind distance-based algorithms. Then, in Chapter 3 we provide a thorough literature review on the topic.

2.4.1 Notation

Before giving an overview on distance-based outlier detection techniques, we introduce essential notation (Table 2.1) and definitions that will be used throughout this work.

$\mathcal{D}$	dataset
N	number of points in the dataset
d	number of features in test instances
N_p	neighborhood of p
σ_p	anomaly score of test instance p
D_p^k	distance from p to its **current** k-th closest neighbor
D_M^k	classification threshold. Score of the top-n anomaly
D_*^k	optimal classification/pruning threshold.

Table 2.1: Notation

Definition 4 *(**Neighborhood**) Given a test point p, the neighborhood of p (N_p) is the set of k points that are the closest to it. Any point $q \in N_p$ is referred to as a **neighbor** of p.*

Definition 5 *(**Neighbor candidate**) Given a test point p, a neighbor candidate is any point that will be checked as a neighbor of p.*

Definition 6 *(**Neighbor comparison**) Given a test point p, a neighbor comparison is the set of operations performed during the* KNN *search to assess whether a neighbor candidate is an actual neighbor of p.*

Definition 7 *(**Anomaly Score** - σ) Anomaly score is a metric that quantifies how anomalous a test instance p is. It is usually computed based on features of p's neighborhood.*

Definition 8 *(**k-th neighbor distance** - D_p^k) Given a test point p, D_p^k is the distance between p and the k-th closest neighbor found so far.*

Definition 9 *(**Optimal classification threshold** - D_*^k) is the largest threshold that still allows the detection of n outliers. Given a dataset $\mathcal{D}$, k and n; D_*^k will be the anomaly score of the n-th top outlier in $\mathcal{D}$.*

Definition 10 *(**Outlier Candidate**) are points determined to have a higher likelihood of being outliers*

2.4.2 Overview

Distance-based methods, as the name suggests, require a distance/similarity metric between data instances, e.g., Euclidean distance for contiguous data and Matching coefficients for categorical data (Chandola et al. [2009]). Given a distance metric, these methods work according to the following assumption:

Assumption 6 *Normal data instances are located very closely to their neighbors, whereas anomalies are distant to even their closest neighbors.*

Broadly speaking, these methods work by computing the anomaly score of each test instance and then using a threshold to classify the instance as an outlier or not. To compute σ_p, most algorithms use some variation of the *k-nearest neighbor* (KNN) search to find p's *neighborhood*, i.e., the set of points closest to p. Then, the score is computed based on some characteristic of the neighborhood, such as the sum or the average (Chandola et al. [2009]) of the distances between p and its neighbors; but the most common metric used is the distance between p and its k-th closest neighbor in the **whole** dataset.

The process by which test instances are classified as outliers depends on the specific outlier definition being used. Next we will present the two most popular definitions from the literature.

Definition 11 *Outliers are the n points with the largest distance to their k-th closest neighbor.*

This definition allows algorithms to find the *top-n outliers* of a given dataset and the score of the n-th top anomaly, D_M^k, is used as the classification threshold. If $\sigma_p \geq D_M^k$, then p is an anomaly. Moreover, with this outlier definition the classification threshold may increase during the detection, as we explain in greater detail in the next chapter. Lastly, note that despite this definition specifically using D_p^k as the anomaly score, it can be easily generalized to use any other metric.
Another commonly used outlier definition was proposed by Knorr and Ng [1999].

Definition 12 *p is an outlier if it has fewer than k points within a distance R*

One possible interpretation of this definition is as follows: the anomaly score of p is the number of points within the hypersphere of radius R centered in p; and the classification threshold is k. However, in this work we favor a different interpretation. Instead, the anomaly score metric is D_p^k and the classification threshold has a value of R, fixed throughout the detection. The interpretation is valid because if p is an outlier, it has

fewer than k points closer than R, which is the case iif D_p^k is larger than the threshold R. Lastly, the parameter R must be provided by the user. In the next chapter we will discuss how to find such parameter, as well as, the advantages and disadvantages of using each of these outlier definitions for our application context.

Advantages

Distance-based techniques are advantageous for several reasons. First, they are unsupervised and thus very useful when labeled data is not available for some reason or when it is hard to model anomalous behavior. Second, these methods can be easily adapted to different application domains, only requiring tailoring the distance metric to each context. Lastly, depending on the distance metric chosen, these methods can scale very well for high dimensional data.

2.5 Discussion: Why use distance-based methods

All of the outlier detection approaches described have important shortcomings that need to be addressed. But, apart from distance-based methods, all other approaches' flaws are difficult to overcome in practice. For instance, statistical methods are only applicable to datasets with very few attributes and generated from a single underlying data distribution, which is not the case for real-world datasets. Classification-based methods are supervised, thus requiring labeled data which might be hard or impossible to obtain. Finally, clustering-based methods have an extremely costly training phase, requiring multiple passes over the data and thus not being suitable for disk-resident datasets. Moreover, these types of methods are not designed for outlier detection.

Distance-based methods also have their flaws such as their, often, simplistic score metric, distance computation cost and worst-case quadratic runtime complexity. However, they can be easily addressed. Moreover, a distance-based approach offers unique advantages for implementing out-of-core outlier detection algorithms for GPUs. First, it was already shown that euclidean distances can be computed incredibly efficiently on GPUs (Garcia et al. [2008]). Second, these methods have great potential for parallelization, e.g., the independent KNN searches (Section 4.2). Third, they can perform the detection while performing only two passes over the dataset (Section 3.4), thus making them well suited for out-of-core execution. Therefore, we decided to use a distance-based approach to implement our GPU algorithm.

Chapter 3

Distance-based outlier detection

In this chapter we provide a more thorough review of distance-based outlier detection techniques. We will start by explaining the *canonical* distance-based outlier detection algorithm, and then we will discuss various optimization strategies proposed to improve its performance. Next, we review the best outlier detection algorithms in the literature, analyzing their optimizations and shortcomings. Finally, we will discuss methods used for producing good initial classification thresholds with the goal of accelerating the detection process.

3.1 Canonical outlier detection algorithm

The canonical algorithm is the basis of the majority of the algorithms that will be discussed in this section. It uses Definition 11 for outliers and D_p^k as the anomaly score metric. Algorithm 1 shows in detail how it works. The algorithm has a nested-loop design and it uses the `MinHeap` $\mathcal{O}$ to save the top-n anomalies found so far. In the outer loop, it selects the reference point p whose anomaly score needs to be computed. In the inner loop, the algorithm performs p's KNN search. Its distance to the other points in the dataset is computed and a `MaxHeap` is used to store the k smallest ones, i.e., the distances between p and its neighbors (Line 7). After every neighbor comparison, the algorithm applies the ANNS rule (Line 9) and discards p if its score became smaller than the classification threshold. The rationale behind the ANNS will be explained in the next section. Finally, If p is not pruned by the end of its KNN search, it is added to $\mathcal{O}$ as a top-n outlier (Line 14).

The canonical algorithm has many shortcomings. First, it has a worst-case quadratic runtime complexity on the size of the dataset, $\mathcal{O}(N^2 \cdot N \log k)$. Though the ANNS improves the average case, the algorithm is still unsuitable for large datasets.

Algorithm 1: Canonical algorithm for distance-based outlier detection

```
1  Function CanonicalOutlierDetection (𝒟, k, p)
2      𝒪 ← MinHeap ()
3      D_M^k ← 0
4      forall p ∈ 𝒟 do
5          N_p ← MaxHeap ()
           // Run KNN search of q
6          forall q ∈ 𝒟 do
               // Keep only the k smallest distances
7              N_p.PushReplace (δ_{p,q}, k)
8              D_p^k ← N_p.top() // p's anomaly score
               // ANNS rule
9              if D_p^k < D_M^k then
10                 break
11             end
12         end
13         if D_p^k ≥ D_M^k then
               // Keep only the top-n probable outliers
14             𝒪.PushReplace ({D_p^k, p}, n)
15         end
16     end
17     return 𝒪
```

Moreover, it would also perform poorly in an out-of-core scenario, since it performs N passes through the dataset, one per point. Nevertheless, in the next sections we will discuss optimization strategies that build upon the canonical algorithm and help address both of its drawbacks.

3.2 Optimizations strategies

In this section we will build a framework for discussing optimizations to distance-based outlier detection algorithms. The discussion will focus on the optimizations' strategy, insights and rationale, rather than on their implementation. The strategies will be categorized as either pruning or ranking ones and will be assigned acronyms. This framework will simplify the analysis of optimizations in general, by decoupling their overall strategy from their specific implementation. The framework will be used extensively throughout this work to explain outlier detection algorithms from the literature, as well as the GPU algorithms we implemented (Chapter 4).

3.2.1 Approximate Nearest Neighbor Search (ANNS)

ANNS is a simple yet powerful pruning rule introduced by Ramaswamy et al. [2000], that allows the detection algorithm to stop the KNN search of an instance p as soon as it finds out that p can not be an outlier. This pruning rule is capable of drastically reducing the amount of distance computation required for detection and thus it is used by the majority of distance-based outlier detection algorithms in the literature. Due to its importance, we elected to explain it separately from the other optimization strategies.

Detection Insights

Before explaining how the ANNS works, we need to understand two phenomenon that occur during the outlier detection with the **canonical algorithm**.

- **D_p^k is an upper-bound of σ_p:** During the KNN search of instance p, the detection algorithm uses a *MaxHeap* to keep track of the k closest neighbors of p processed so far. As closer points are found and added to p's neighborhood, the distance between p and its k-th closest neighbor, D_p^k, decreases. σ_p will be equal to D_p^k at the end of the KNN search thus, before the search ends, D_p^k represents the maximum possible anomaly score p can have.

- **D_M^k is a lower-bound of D_*^k:** A similar argument can be used to explain this statement. The detection algorithm uses a *MinHeap* to keep track of the top-n outliers found so far. As the detection progresses, if a new point has an anomaly score equal to or larger than the score of the current n-th outlier[1] then: (i) this point will be added to the set of outliers; (ii) the current n-th anomaly will be discarded and (iii) D_M^k will either remain the same or increase. At the end of the detection, D_M^k will be equal to D_*^k.

The ANNS' rationale is simple. If during the KNN search of p, D_p^k becomes smaller than the current D_M^k, then it is impossible for p to be an outlier. Therefore, its KNN search can be terminated and the algorithm can start analyzing the next data instance. The ANNS' efficiency may be expressed by how much computation it prevented or, alternatively, how soon into p's KNN search p was pruned. This efficiency depends on two factors. First, on how small D_p^k is, i.e. how many close neighbors were found. Second, on how close D_M^k is to its optimal value: D_*^k.

[1] Equal to or larger than D_M^k

As we discuss next, the majority of the optimizations in the literature focus in improving the ANNS efficiency and do so by speeding up the convergence rate of either D_M^k or D_p^k.

3.2.2 Pruning

Pruning strategies are widely used to accelerate the detection. They allow the algorithm to employ domain knowledge to prune the detection search space, e.g. the ANNS leverages the D_p^k and D_M^k insights outlined previously to prune the KNN search space. Next we will discuss two classes of pruning strategies, one for reducing the number of points to be classified and the other to reduce the number of neighbors to be processed during the KNN search. We should stress that all pruning strategies mentioned here will require data partitioning.

Pruning Partition during Search of Neighbors (PPSN)

The main idea of these strategies is to avoid, during the KNN search, processing partitions too far away from point p to contain any of p's closest neighbors. During the KNN search, before starting to look for neighbors in partition P, the algorithm computes the minimal distance between p and any point in partition P, i.e. *MinDist*. If $D_p^k < MinDist$, P is too far away to contain any neighbors of p, thus the entire partition can be pruned.

Pruning Partition during Search for Outliers

The goal of this pruning strategy is to prune entire partitions that can not have outliers. There are slightly different implementations of this strategy in the literature (Ramaswamy et al. [2000]; Orair et al. [2010]), but in general they work very similarly. Given a partition P with more than k instances, if the maximum distance, *MaxDist*, between any of its two points is smaller than D_M^k, then it is impossible that P contains any outliers. It is easy to see why this statement is true:

$$\forall p \in P,\ D_p^k \leq MaxDist \leq D_M^k \tag{3.1}$$

3.2.3 Ranking

The goal of ranking optimizations is to accelerate either the increase of D_M^k or decrease of D_p^k, by changing the evaluation order of dataset points.

Ranking Objects Candidates for Neighbors (ROCN)

This strategy changes the evaluation order of neighbors in the KNN search of p, such that the closest ones are examined first. The goal is to decrease D_p^k faster and allow the ANNS to be more effective and halt p's KNN search earlier. Algorithms that have a partitioning pre-processing step can use ROCN in two levels. At the *inter-partition* level, the algorithm sorts the partitions according to their distance to p. The KNN search starts at p's partition and then continues to the next closest partition until the ANNS prunes p or until the end of the dataset. The rationale is that closer partitions are more likely to contain the nearest neighbors of p. A less common and more complex way to use ROCN is at the *intra-partition* level, which involves changing the evaluation order of points within a given partition.

Ranking Objects Candidates for Outlier (ROCO)

The goal of this type of ranking is to make D_M^k grow faster, by classifying first points likely to have large anomaly scores. In an ideal scenario, the first n points classified would be the actual top-n outliers. The D_M^k would converge to its real value only after n classifications, thus making the ANNS rule extremely effective. However, even if just a mix of outliers and points with large anomaly scores are classified first, it is enough to raise D_M^k sufficiently to prune the majority of inliers very quickly.

3.3 In-Memory Algorithms

In this section we provide a review of the main distance-based outlier detection algorithms from the literature, which are designed to process datasets stored in memory.

3.3.1 Ramasamy

Ramaswamy et al. [2000] were the first to use the definition of anomaly score and outliers used in the canonical algorithm: D_p^k the distance to the k-th closest neighbor of p. They proposed a two-phase, partition-based algorithm that used all three pruning strategies discussed: ANNS, PPSO and PPSN. The first phase partitions the data using the clustering algorithm BIRCH and builds a list of the *candidate partitions*, i.e., partitions that may have outliers. To identify these partitions, the algorithm needs to compute two types of bounds: (i) for every cluster C, compute the lower and upper-bound of D_p^k that the points in C could have and (ii) the lower-bound for the pruning threshold D_M^k. Any partition that has D_p^k upper-bound smaller than

the D_M^k lower-bound can not possibly have an outlier and thus is not included in the candidate list (PPSO). In the second phase, the algorithm classifies the points belonging to the candidate partitions. To classify point p, the algorithm performs p's KNN search starting from its partition, C, and only examines points in partitions close enough to C to have neighbors of p (PPSN). More specifically, only partitions whose distance to C is smaller than C's lower-bound for D_p^k.

3.3.2 RBRP

RBRP (Ghoting et al. [2008]) uses the canonical definitions of outlier and anomaly score and its focus is on improving the detection performance by applying the ROCN optimization in two different levels: inside and between data partitions. The algorithm has two phases. In the first, it uses a divisive hierarchical algorithm to partition the data into bins. Then, for each bin B, it reorders B's points by finding the bin's *Principal Component (PC)* and then projecting the points onto it. In the second phase, RBRP detects the anomalies. The KNN search of every point p starts with the point next to it in its bin's PC. Once the search reaches the end of the bin, it wraps around to the beginning of the bin and continues until all points were compared to p. The ordering imposed by the PC characterizes the use of ROCN inside the partition. If the ANNS is not able to prune p inside its own bin, then the KNN search continues to the next closest bin; ROCN between partitions. At the end of second phase, the n points with the largest anomaly scores are the outliers.

3.3.3 DIODE

DIODE (Orair et al. [2010]) is one of the most optimized in-memory outlier detection algorithm available and it implements all five optimization strategies discussed. It divides the detection process into two phases. The first phase is very similar to RBRP's first phase. It uses a divisive hierarchical clustering algorithm to partition the data and, additionally, it computes bounds on the D_p^k values for each partition. Lastly, DIODE uses a novel ROCO heuristic to rank partitions according to their likelihood of containing outliers. The likelihood is approximated by the inverse of the density[2] of a partition P. The intuition is that anomalies occur in less dense regions/partitions.

In the second phase, DIODE classifies the points similarly to RBRP. The detection starts by the points more likely to be outliers, thanks to the ROCO optimization. In the classification of p, DIODE starts p's KNN search by examining points in the same

[2] $\frac{|P|}{R(P)}$, where $R(P)$ is MBR diagonal length of P

partition P (ROCN). If p is not pruned before all the points in P are examined, the search continues to the closest partition P' not used yet, again another example of ROCN. Before switching to C', `DIODE` applies the PPSN optimization: if the minimum distance between p and C''s centroid is larger than D_p^k, than C' can not contain any neighbors of p and is pruned. p's KNN search continues until either p is pruned or it reaches the end of the dataset. Once `DIODE` classifies all the points of a particular partition and has to move to the next one (P_N), it applies a similar version of Ramasamy algorithm's PPSO optimization: if P_N's D_p^k upper-bound is smaller than D_M^k, then P_N can not contain outliers and is thus pruned.

3.4 Out-of-Core Algorithms

In this section we will discuss in greater detail three of the main out-of-core outlier detection algorithms in the literature, two of which will serve as baselines in the experiments (Chapter 5).

3.4.1 `Orca`

`Orca` (Bay and Schwabacher [2003]) was one of the first out-of-core outlier detection algorithms. It is heavily based on the canonical algorithm, sharing the nested loop design, anomaly score definition (D_p^k) and outlier definition (top-n outliers). However, it improves upon the canonical algorithm by introducing two new optimizations: (i) a non-deterministic version of ROCN, which greatly improves the pruning efficiency and (ii) dividing the dataset into batches of b points and classifying them concurrently, in order to reduce the overall number of dataset passes.

Algorithm 2 shows in detail how `Orca` works. Unlike the canonical algorithm, `Orca`'s outer loop (Line 4) selects the next batch, B, of points to be classified, rather than just one point. Moreover, the second and third loops (Lines 7 and 8) perform the KNN search of every point inside B concurrently: every neighbor candidate q loaded from the dataset is compared to all test points in B, before loading the next neighbor candidate from disk. Inside the third loop, the distance between q and p is computed, D_p^k is updated (Line 9) and the ANNS rule is applied (Line 10). Any points left in B after its KNN search ends are considered outliers. They are used to update both the MinHeap ($\mathcal{O}$) containing the top-n outliers found so far and the D_M^k. `Orca` continues this process until there are no more batches left to be classified.

The optimizations introduced by `Orca` were extremely effective at improving the detection performance. First, their ROCN implementation relies on the shuffling of the

Algorithm 2: Orca algorithm

```
1  Function Orca (D, k, p)
2      D_M^k ← 0
3      O ← MinHeap()
4      while D.HasTestBatchLeft () do
5          B ← NextTestBatch (D)
6          N[p] ← MaxHeap () ∀p ∈ B
           // Run KNN search of points in B
7          forall q ∈ D do
8              forall p ∈ B | p ≠ q do
                   // Kepp only the k smallest distances
9                  N[p].PushReplace (δ_{p,q}, k)
10                 if N[p].size == k ∧ N[p].top () < D_M^k then
11                     B.Remove (p) // Prune p
12                 end
13             end
14         end
           // Keep only the top n probable outliers
15         if ! B.Empty() then D_M^k ← O.SaveOutliers (B, N)
16     end
17     return O
```

dataset to induce a non-deterministic ranking of the neighbor candidates, accelerating the convergence of D_p^k and improving the pruning. The authors showed that this ranking strategy, in conjunction with the ANNS rule, was able to reduce the average case runtime complexity of the detection from quadratic (canonical algorithm) to near-linear time, for various real datasets. The second optimization proposed, *batching*, was successful at reducing the number of dataset traversals. It allows the KNN search of all points in the batch to be run concurrently, reducing the upper-bound of the number of dataset passes from N to at most N/b.

3.4.2 `Diskaware`

Yankov et al. [2008] proposed an algorithm, which we will refer to as `Diskaware`, for detecting unusual time series in large scale datasets. It uses the Definition 12 for outliers and, as a result, it requires the user to supply the classification threshold, which will not change during the detection. Lastly, the algorithm only uses $k = 1$.

`Diskaware`'s design is very different from the canonical algorithm's and its derivatives. It has two phases. In the first phase, the goal is to quickly identify outlier candidates and segregate them from the inliers. The candidates are stored in a set Q and are the only points that will actually undergo classification. The second phase will classify

Algorithm 3: `Diskaware`- Phase 1

```
1  Function CandidateSelection (D, k, D_M^k)
2      Q ← ∅
3      forall q ∈ D do
4          isCandidate = true
5          forall p ∈ Q do
6              if δ_{p,q} < D_M^k then
                   // Neither p nor q can be outliers
7                  Q.Remove (p)
8                  isCandidate = false
9              end
10         end
11         if isCandidate then
12             Q.Add (q)
13         end
14     end
15     return Q
```

the candidates found. It runs the KNN search for every point in Q concurrently, thus requiring only one dataset traversal. In total, the algorithm only makes two passes in the data, one for each phase.

Algorithm 3 shows how the first phase works. It starts by including just one point in Q. Then, for every point $q \in \mathcal{D}$ it checks whether q has a neighbor p from Q that is closer than D_M^k (Line 6). If yes, then neither p nor q can be outliers (ANNS) and the algorithm removes p from the set of candidates. On the other hand, if q has no neighbors from Q, q is added to Q as an outlier candidate. This method is equivalent to perform a "restricted KNN search" that only uses points in Q as neighbors candidates. The authors showed that, given a good D_M^k value, this method produces few false positives, which makes Q small and the KNN searches quick.

The second phase (Algorithm 4) classifies all the points in Q concurrently: for every neighbor candidate q loaded from disk, the algorithm compares q to every point $p \in Q$ before loading the next neighbor (Lines 5 and 6). If q is too close to p, then p can not be an outlier (ANNS). After the KNN searches end, it is known that the points remaining in Q do not have any neighbors closer than D_M^k. Therefore, they are classified as outliers.

3.4.3 `Dolphin`

The algorithm `Dolphin` (Angiulli and Fassetti [2009]) is conceptually very similar to `Diskaware`. It uses the same definitions of anomaly score and outliers but without

Algorithm 4: `Diskaware`- Phase 2

```
1  Function CandidateRefinement (D, Q, k, D_M^k)
       // Initialize the candidates' anomaly score
2      forall p ∈ Q do
3          p.σ = ∞
4      end
5      forall q ∈ D do
6          forall p ∈ Q do
7              if p ≠ q and δ_{p,q} < D_M^k then
8                  Q.Remove (p)
9              else
                   // Update p's anomaly score
10                 p.σ ← min(p.σ, δ_{p,q})
11             end
12         end
13     end
14     return Q
```

fixing $k = 1$. Consequently, it also requires the user to supply an initial classification threshold. Moreover, `Dolphin` also divides the detection process into two phases: (i) phase one identifies outlier candidates by using "restricted KNN searches" and (ii) phase two classifies the candidates identified, running all their KNN searches concurrently. However, `Dolphin` also introduces two important improvements over `Diskaware`: (i) it uses ROCN and PPSN optimizations to speedup the KNN searches and (ii) it keeps some inliers in the group of outlier candidates to improve the efficiency of the ANNS.

`Dolphin`'s first phase is more sophisticated than `Diskaware`'s. Rather than simply building a set of outlier candidates, it builds an index structure that summarizes the dataset points read so far. The index provides approximate range queries that, given a point p and range R, return the *superset* of p's neighbors in the index[3]. These queries are the basis of the KNN searches performed during the first phase and employ the ROCN and PPSN optimizations to speedup the queries. In addition to keep candidates in the index, `Dolphin` also keeps a small fraction $P_{inliers}$ of the inliers identified. The authors argue that, because inliers occur in dense regions, they are more likely to be neighbors of other inliers. Therefore, by having a small fraction of them in the index, it increases the chances that future KNN searches will find enough neighbors of other inliers to prune them, thus improving the efficiency of the ANNS. Finally, at the end of the first phase, the inliers kept in the index are discarded and only the actual outlier candidates remain for the next phase.

The second phase (Algorithm 5) classifies the candidates found. Similarly to

[3]The neighbors of p are the points in the index at a distance closer than R from p

Algorithm 5: `Dolphin`- Phase 2

```
1  Function CandidateRefinement (D, k, p)
2      forall q ∈ D do
3          NN = Index.RangeQuery (q, R)
4          forall p ∈ NN do
               // Remove the false positives from the neighbor superset
5              forall r ∈ NN do
6                  if δ_{p,q,≤} D_M^k then
                       // q and p are neighbors
7                      p.nn++
8                      if p.nn == k then
                           // Too many neighbors.Inlier
9                          Index.Remove(p)
10                     end
11                 end
12             end
13         end
14     end
15     return Index
```

`Diskaware`, it also performs the KNN search of all the candidates concurrently to ensure only one dataset traversal is needed. For every point q read from the dataset, the algorithm finds the superset N of q's neighbors in the index (Line 3). Then, for every actual neighbor p of q (Line 6), the algorithm increments p's count of neighbors by one (Line 7). Any actual neighbor p whose neighbor count goes past k must be an inlier and thus is pruned from Index (ANNS).

`Dolphin` offers two main advantages over `Diskaware`. Firstly, it uses an index structure that employs the ROCN and PPSN optimizations to speedup the KNN searches. Secondly, by keeping some inliers in the index, the algorithm is able to improve the ANNS efficiency, thus reducing the number of false positives that need to be classified during phase 2.

3.5 D_M^k estimation methods

Among the detection algorithms discussed in this chapter, only `Diskaware` and `Dolphin` require an initial threshold larger than 0. However, all the algorithms can use an initial threshold and their performance improves significantly the closer D_M^k is to D_*^k. Therefore, in this section we will present and analyze two methods for estimating classification thresholds for a particular detection instance, i.e. given a dataset D and values for k and n.

3.5.1 Diskaware's estimation method

Yankov et al. [2008] proposed a three step method for estimating the classification threshold to use to detect the top-n outliers of a dataset. First, a uniformly random sample S' is chosen. Then, a fast, memory-based, outlier detection algorithm is used to detect the top-n anomalies within the sample. Lastly, the threshold estimate is set to the score of the n-th top outlier.

There are several issues with this method but mainly, it uses an unproven, *ad-hoc* approach. First, the authors do not provide any formal way of choosing the sample size. For their experiments, they use a sample of size 10^3 for datasets with $N < 10^6$ and for larger datasets they use samples of size 10^4. Second, there is no statistical guarantee on quality or correctness[4] of the estimates produced. Third, if an invalid estimate is generated, the authors propose an *ad-hoc* approach for guessing new estimates based on the the previous ones. This process could take as many as three attempts and with each invalid estimate, another detection run is made. Finally, the method was only tested with $k = 1$ and $n = 10$. Given the lack of proof in the correctness of this method, we do not believe it would work for more meaningful values of k and n.

3.5.2 Dolphin's estimation method

Angiulli and Fassetti [2009] proposed a new estimation method that generalizes `Diskaware`'s method, allowing the use of any k and n value. First, let's introduce some notation:

- η is the size of sample S
- $\rho = \frac{n}{N}$ the percentage of outliers to be detected
- $\varrho = \frac{k}{N}$, the percentage of neighbors to be considered during the KNN search
- $n_s = \rho \cdot \eta$, number of outliers to be detected in S
- $k_s = \varrho \cdot \eta$, number of neighbors to be considered during estimation
- σ_p^S, p's anomaly score using only neighbor candidates from a sample S

The method works as follows. First, it selects an uniformly random sample S of size η. Second, it runs a fast, in-memory, outlier detection algorithm, e.g. `Orca`, to detect the top-n_s outliers in S, considering k_s neighbors during the KNN search. Then, the estimate will be equal to the anomaly score of the n_s-th outlier found: σ_p^S.

[4] The threshold estimate is smaller than D_*^k to allow at least n outliers to be detected

This estimation method has a major issue. The required sample size, in some cases, might be too big to fit in the GPU's memory. For the described method to work properly, $k_s \geq 1$. The minimum sample size for which this restriction is satisfied is given by:

$$\eta_{min} = \frac{1}{\varrho} \tag{3.2}$$

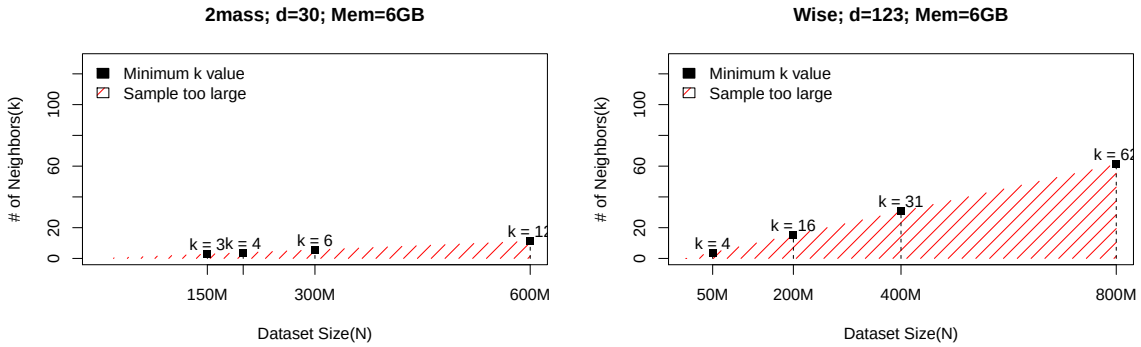
So, the smaller ϱ is, i.e. small k and large N, the larger the sample has to be. This restriction is a big issue for our outlier detection scenario. First, the values of k used by most of the papers in the literature is in the range $[1, 128]$, with most of them using $k \leq 10$. Secondly, our work targets outlier detection in large scale datasets, i.e. large N. Both of these factors contribute to large samples. Finally, the GPU is memory constrained, thus severely limiting the size of the samples that can be used.

Since processing large datasets using GPUs is the core proposal of this work, in order to still use `Dolphin`'s estimation method, we are limited to using larger k values. Figure 3.1 shows the minimum k value that can be used for various dataset sizes, such that S fits in the memory of the GPU we used during the experiments. This solution is far from ideal because in the case `WISE` dataset, about half of the range of common k values can not be used ($k_{min} = 62$).

3.5.3 Practicallity of D_M^k estimation methods

In summary, `Dolphin`'s estimation method can produce D_M^k values very close to D_*^k, allowing for very efficient ANNS usage and enabling very fast detection times, specially for `Dolphin`. However, for our use case scenario, very large scale datasets, the estimation requires equally large samples. Thus, its applicability is limited. This highlights a major flaw in both `Diskaware` and `Dolphin` algorithms, which will be discussed further in the next section: They depend on a very good initial D_M^k value to perform well, but it is not always possible to produce such D_*^k estimate. **Therefore, we state that a very important characteristic for a versitile out-of-core outlier detection algorithms is to be able to perform well even when the initial D_M^k available is very poor (i.e., far from D_*^k) or even inexistent.**

Due to the quality of the estimates generated by this method, we use it extensively in our experimental analysis (Chapter 5), at the cost fo being limited in the values of N and k we could use. Therefore, in our experiments we also analyse the alternatives a user can choose from if a particular detection configuration does not allow for the use of this estimation method.



(a) Dataset with $d = 30$ float attributes

(b) Dataset with $d = 123$ attributes

Figure 3.1: k_{min} values for different **samples of the datasets** `2MASS` and `WISE` which are used in the experiments.

Algorithm	# db traversals	D_M^k Required	D_M^k Robust
`Orca`	$\mathcal{O}(N/b)$	N	N*
`Diskaware`	2	Y	N
`Dolphin`	2	Y	N
`DROIDg`	$\mathcal{O}(1)$	N	Y

Table 3.1: Overview on the strengths and weaknesses of three out-of-core CPU algorithms presented in this section. Lastly, `DROIDg`, our GPU algorithm proposed in Section 4.6, has all the strengths of the other algorithms but none of their drawbacks.

3.6 A comparison between the out-of-core outlier detection algorithms

The out-of-core algorithms presented in this chapter use different outlier definitions and strategies to efficiently detect outliers in disk-resident datasets. In this section we will compare their detection strategies on three main aspects and how these affect performance:

1. **Number of dataset traversals**: When datasets are disk-resident it is paramount to avoid traversing the dataset in order to reduce disk access.

2. **Requires initial D_M^k value**: As discussed in **??** it is not always possible to estimate a good initial D_M^k. Thus, algorithms that require an initial D_M^k value are less versitile.

3. **Robustness on quality of initial D_M^k**: Even in scenarios where it is possible

to estimate the D_M^k, there are certain factors that could reduce the quality of the estimate, e.g., dataset with uniform data distribution. Algorithms robust to the quality of the D_M^k offer good performance on a wider range of scenarios and thus are more versitile.

Table 3.1 provides a summary of the comparison.

The `Orca` algorithm uses the Definition 11 of outliers and, as a result, it can iteratively improve the D_M^k during the detection by finding points with higher anomaly scores and refining the list of top-n outliers. This approach has two main advantages. First, `Orca` does not need an initial D_M^k estimate and can start the detection with $D_M^k = 0$. Second and more importantly, its performance is robust to the quality of the initial D_M^k value/estimate, specifically because it can be improved during the detection. However, `Orca`'s iterative detection strategy is also the cause of its main disadvantage for out-of-core execution: excessive number of dataset traversals. Even though most batches, B, of points being classified will be quickly and fully pruned by the ANNS, if at least one point in B has an anomaly score larger than the current D_M^k, such point will not be pruned and `Orca` will have to traverse the entire dataset. In a random dataset, this is expected to happen with $\mathcal{O}(N/b)$ batches and as our experiments show (Chapter 5), this leads to prohibitively large I/O cost.

`Diskaware` is much better suited than `Orca` for out-of-core execution, because its two phase design addresses `Orca`'s main drawback: the excessive number of dataset passes. However, because it uses a fixed D_M^k, its pruning efficiency and thus performance are entirely dependent on the quality of the initial D_M^k value. A low initial D_M^k will mean the ANNS will perform poorly during the entire detection process. The worst the D_M^k is, the closer `Diskaware` will perform to a brute-force algorithm. During phase 1, a large number of false positive outlier candidates will be saved (Section 3.4.2). Every new test point being analysed will have to be compared to an ever increasing number of outlier candidates. Whereas in phase 2, every single saved candidate will have to be classified, causing an imense amount of distance-pair computations due to and the poor efficiency of the ANNS. In summary, `Diskaware`'s performance is not robust to bad initial D_M^k values, i.e., values far from D_*^k.

Lastly, `Dolphin` is very similar to `Diskaware` with regards to its overall design, thus it shares the same fundamental advantages and disadvantages. But, due to its KNN search optimizations, namely *ROCN* and *PPSN*, `Dolphin` will be significantly faster than `Diskaware` under ideal conditions: when a good initial classification threshold is available. However, it still has the same limited aplicability to only scenarios where a good initial D_M^k is available.

All the algorithms discussed here have drawbacks that can lead to significant decrease in performance on an out-of-core execution scenario. In Section 4.6 we propose a new algorithm called `DROIDg` that combines `Orca`'s D_M^k improvement strategy, which has a higher I/O cost, to make it more robust to poor initial thresholds; with `Diskaware` and `Dolphin`'s two phase design, to keep the amount of dataset traversals extremely low. The result is a more robust algorithm that provides a better balance between the reduction of computation and I/O costs.

Chapter 4

GPU outlier detection

As we briefly discussed during the introduction of this work (Chapter 1), the applicability of outlier detection analysis is still very limited when dealing with terabyte-scale datasets. Detection times are prohibitively high when detecting a reasonable percentage of anomalies, even in relatively small datasets and using state-of-the-art methods. There are two main reasons. First, the cost of detection increases more than linearly with the size of the dataset (Chapter 3). Second, the amount of distance-pair computation required increases massively with n due to a loss in the ANNS' efficiency (Section 5.3). As n gets larger, D_*^k gets closer to the score of inliers, thus making it harder to prune test points quickly or even to prune them at all. When combining large datasets with reasonable values of n, the problem quickly becomes intractable using conventional CPU algorithms. Consequently, the vast majority of papers in the literature limits their experimental analysis to tiny datasets of a few million points and/or use small values of n, typically less than 1000 (Knorr and Ng [1999]; Ramaswamy et al. [2000]; Bay and Schwabacher [2003]; Angiulli et al. [2006]; Ghoting et al. [2008]; Yankov et al. [2008]; Orair et al. [2010]; Angiulli and Fassetti [2009]).

Why large N and large n are important

Enabling detection of meaningful percentages of outliers (n) in large scale datasets is very relevant. First, the amount of data being currently generated and stored is, and will continue, to increase rapidly. Faster and more efficient anomaly detection methods are needed to process these large datasets. Second, only detecting tiny percentages of outliers limits the usefulness of outlier detection analysis. For instance, consider the case where one wishes to remove noise from a sensory data dataset with 300M points, the same size as the smallest dataset used in our experiments (Chapter 5). Being

able to only detect the top-1000 outliers would not be enough to clean the data in many cases, because $n = 1000$ would represent just 0.00034% of the dataset size, an unreasonably small error rate for many practical applications.

Consider another example. We have a dataset where the expected anomaly rate is unknown, thus an exploratory analysis has to be conducted in order to determine true amount of anomalies in the data. In this case, it would be necessary to compute as many true anomaly scores as possible to plot the top-n anomaly score curve and study its behavior in order to find a cutoff score D_*^k, above which every point is an anomaly. Again, if the methods available can only find $n = 1000$ true scores in a reasonable time, it is not feasible to perform such analysis for any significantly sized dataset because n would represent a vastly insignificant percentage of points and there would be too few score to perform the analysis. Therefore, having methods that can perform outlier detection in large scale datasets with large value of n is essential to making outlier detection analysis widely applicable.

Our goal

The main goal of this work is to make viable the use of outlier detection analysis in ETL[1] workloads dealing with terabyte-scale datasets. We aim to achieve it by drastically reducing detection times through the use of GPUs. These chips are extremely parallel, with thousands of processing elements, and are great for accelerating parallel workloads with high arithmetic intensity. As we further discuss in Section 4.2, outlier detection is such a workload.

However, to be able to accelerate outlier detection in terabyte-scale datasets using GPUs, there are two important challenges that need to be overcome. Firstly, GPUs are highly specialized and optimized to run linear, with high arithmetic intensity, SIMD code. Anything that deviates from this, greatly reduces the maximum attainable computation throughput, thus imposing great constraints in the **design and implementation** of high performance algorithms. For instance, the implementation of the ANNS discussed so far is extremely inefficient to run in GPUs. The second challenge pertains to the size of datasets. We are targeting processing terabyte-scale data, but GPUs have very limited memory buffers, usually less than 12GB. Thus, we will have to develop an out-of-core GPU algorithm which, to the best of our knowledge, has never been done before. Efficiently processing datasets in this fashion with GPUs is more challenging than with CPUs, because the I/O penalty and I/O bottlenecks are far greater to the

[1] *Extract, Load and Transform.* The entire dataset is available beforehand, unlike the case with streaming applications

former, as we will discuss in Section 4.4.

Chapter summary

In this chapter we discuss the design aspects that address the two aforementioned challenges and thus allow the development of outlier detection algorithms that effectively exploit the GPU's computation power. First, we discuss preliminary concepts about GPU architectures, their programming models and *OpenCL*. This is essential to understand the design decisions made and the algorithms and optimization implemented (Appendix A). Second, we propose a parallelization strategy that eases the implementation of efficient distance-based outlier detection algorithms for the GPU. Third, we discuss the challenges of developing out-of-core algorithms for GPU and provide a detailed design of a I/O subsystem that efficiently performs asynchronous data streaming to the GPU and enables overlapping computation and I/O. Fourth, we introduce `DROIDg`, a novel GPU algorithm that combines `Orca`'s D_M^k improvement strategy with `Dolphin`'s two phase algorithmic design for reduced I/O. It has robust performance even when using an initial threshold of 0, thus addressing `Dolphin`'s main shortcoming. Lastly, we present other outlier detection algorithms for GPUs available in the literature.

The discussion in this chapter is focused more in fundamental design decisions and higher-level algorithms/strategies to enable high-performance out-of-core outlier detection using GPUs. Therefore, all the algorithms shown in this chapter are **host-side** code (see next section). The functions that have the prefix '*Gpu*' are still host-side but they do setup work for the execution of computation **Kernels**, i.e., functions that run on the GPU. For an in-depth look on the design and implementation, using *OpenCL*, of the kernels required by the algorithms shown in this chapter, see Appendix A.

Finally, it is important to highlight that our proposed algorithm, as well as all outlier detection algorithms discussed in the past chapters, are suited only for ETL workloads. Despite preventing the use of these algorithms in streaming applications, ETL workloads are still incredibly popular and important.

4.1 Background

4.1.1 GPU Architecture

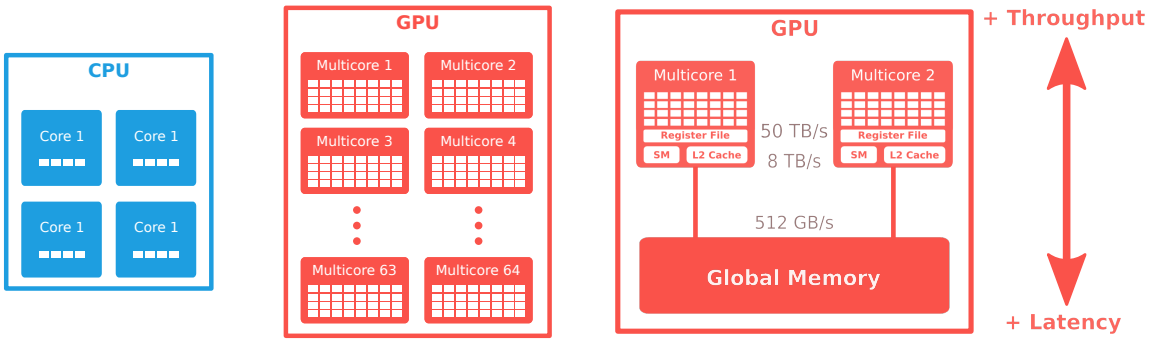
GPUs were originally designed for graphic processing workloads. Today, they are fully programmable and are used in various fields that require massive computational

power. Their main advantage over traditional CPUs is their embarrassingly parallel hardware, with thousands of compute cores and capable of achieving several TFlops of computation throughput. The architectural overview we provide in this section will use the GTX 980 Ti GPU and the NVIDIA's Maxwell micro-architecture as a case-study, but the basic concepts outlined here are applicable to other GPU models, even from other vendors.

GPUs are massively parallel, with thousands of *Processing Elements (PEs)* for performing arithmetic operations. To efficiently manage the PEs and allow them to co-operate, the GPUs have a highly organized and hierarchical architecture. For instance, the GTX 980 Ti has 16 *Compute Units (CU)* called Stream Multiprocessors (SMMs), each with 128 PEs, called *CUDA cores*, resulting in 2048 cores in total. Within each SMM, the PEs are organized as four 32-wide *Single Instruction Multiple Data (SIMD)* units, which are capable of executing instructions over vector data. The execution model of the SMMs is called Single Instruction Multiple Threads (SIMT) and it uses SIMD units to emulate multithreading on GPUs. Rather than exposing to programmers only fixed-length vector instructions (SIMD execution model), in the SIMT model the programmer expresses the parallelism as scalar operations over contiguous sections of memory (arbitrarily large vectors) and the hardware automatically generates the set of SIMD instructions required. This provides more flexibility to the programmer.

In SIMT, *warps*, i.e., groups of 32 threads, execute in lock-step. This is essential for simplifying the execution control and thread scheduling across thousands of available cores in the GPU. The downside, however, is expensive *thread divergence*. Whenever threads take different execution paths in a branch, all 32 threads have to execute all diverging paths, ergo massively wasting GPU computing cycles.

To feed enough data to the thousands of compute cores, the SMXs are connected to a high bandwidth memory hierarchy, with 2 main memory regions. *Global Memory* is the largest region, with sizes between 4 and 12 GB. However, it is also the slowest, with latency upward of 400 ns and a bandwidth of a few hundred GB/s. The second region is the *Shared Memory*. It is visible to all threads running in the same SMX and is often used for coordinating computation between threads. This memory space is much smaller than global memory, around 64KB per SMX, but it is much closer to the PEs and offers one order of magnitude lower latency and higher bandwidth. Therefore, making sure to use shared memory instead of global memory whenever possible is one of the optimization that yields the most performance benefits.



(a) CPU vs GPU computing cores (b) GPU memory architecture

Figure 4.1: Overview on the architecture characteristics of GPUs

4.1.2 *OpenCL*

OpenCL (Open Computing Language) is an open standard for general purpose parallel programming of many-core architectures[2]. It provides a framework for parallel computing which includes, but not limited to, a cross-platform language for implementing parallel algorithms and an API for managing their execution. To better explain the concepts behind *OpenCL* and how *OpenCL* programs execute, we will discuss three of the abstraction models introduced by the standard: *OpenCL*'s *Platform, Memory and Execution Models*.

4.1.2.1 Platform Model

An *OpenCL*-capable system has a *Host device* connected to one or more *OpenCL devices*. *OpenCL* programs run in both types of devices and, like the platform, can be divided into host-side and device-side kernel code. The device-side code consists of parallel functions, called *Kernels*, required for performing the program's computation; and they are written using *OpenCL* C, a cross-platform parallel programming language. On the other hand, the host-side code, as the name implies, runs on the system's host and manages the execution of the parallel work. It uses the *OpenCL*'s framework API to perform *host-to-device* and *device-to-device* data transfers. Additionally, the API allows the host to schedule the execution of kernels across the available devices, allowing it to map and to coordinate how the computation should be performed.

4.1.2.2 Execution Model

Command Queues are abstractions introduced by the *OpenCL* execution model to allow the host to submit API calls to *OpenCL* devices. Each queue is associated with only

[2]CPUs, GPUs, FPGAs, etc

one device and it may be of two types: (i) **Out-of-order**, which allows commands to be executed in any particular order, as soon as the required resource is available; (ii) **in-order** that enforces commands to be executed in FIFO.

To run a kernel, the host code submits to a command queue the kernel, along with *Memory Objects*, to serve as arguments; and two *NDRange* objects to describe the execution's *Index Space* and specify the amount of parallelism to be used. The kernel will be run by threads called *work-items*, which are mapped to the device's processing elements, e.g., CUDA cores. *OpenCL* allows *work-items* to cooperate by organizing them on work-groups, which are mapped to the device's compute units, e.g., SMMs. Only *work-items* in the same work-group may cooperate through the use of synchronization functions, such as barriers and memory fences, as well as collective computation functions: reduction, broadcast, prefix sum and predicate evaluation.

The index space specified in the kernel execution maps the problem domain to the execution domain, by specifying the number and addressing scheme of *work-items*. For example, when performing the following matrix multiplication: $C_{mxn} = A_{mxp} \cdot B_{pxn}$, the host code can map the multiplication into a 2D index space, where each *work-item* is identified by two ids, one per dimension, and computes one element of the output matrix C. There will be a total of $m \cdot n$ *work-items*, with m sets of *work-items* in the first dimension, one per row, and n sets of *work-items* in the second dimension, one per column. In this configuration, *work-item* $w_{i,j}$[3] will compute the output element $c_{i,j}$. The ability to change the mapping of *work-items* and their grouping gives the programmer freedom on how to expose the parallelism in the problem at hand.

4.1.2.3 Memory Model

Memory subsystems vary greatly between different computing platforms. In order to achieve code portability, *OpenCL* defines an abstract memory model that programmers can code for, and that hardware vendors can map to their actual memory architectures. In *OpenCL*, memory is divided into 4 different memory spaces: **Global, Constant, Local,** and **Private**. Global memory is visible by all compute units in a device and it is usually the address space with the biggest capacity, but it also has the highest latency. All the data transferred between host and *OpenCL* devices must be either read from or written to global memory.
Constant memory is a sub-region of global memory dedicated to store data which does not change during the execution of a work-item. It is the host's responsibility to setup memory objects in this address space, and, in some hardware architectures, using con-

[3]Id i in first dimension and id j in the second dimension

stant memory instead of global memory may yield some performance improvements. Local memory is an address space restricted to individual compute units, i.e., only work-items in the same work-group can share data using **Local** variables and arrays. It is usually implemented as an on-chip memory e.g., in GPUs, and therefore it often provides much smaller latency and much higher bandwidth than global memory. However, this address space is much smaller, with current high-end GPUs having between 16KB and 48KB of local memory per compute unit. Finally, private memory is a memory restricted to each work-item. This is the default address space for kernel arguments and variables and only supports static allocation. This memory space is often mapped to registers, which may yield very low latency access, but it can hold very little data and private arrays and spilled registers are usually mapped to global memory. Therefore, it should be used with care to avoid registers spillage, which could decrease kernel performance.

4.2 Extracting Parallelism from the Problem

To leverage the GPU's computational power, algorithms need to decompose the problem into dense and independent computations, as to provide enough parallel work to keep the thousands of GPU PEs busy. Our parallelization strategy extracts parallelism from the problem within and between KNN searches.

The first level of parallelization occurs inside the KNN search itself. To compute the anomaly score of an instance p and classify it, the algorithm needs to find p's k closest neighbors by (*potentially*) comparing p to all points in the dataset. And, since these comparisons are independent, they can be performed in parallel. Therefore, rather than comparing p to one point at a time, p is compared in parallel to a whole batch of neighbor candidates at a time, which we refer to as *Neighbor Candidate Batch (NCB)*. Consequently, p's KNN search is performed in iterations. p is compared to one NCB at a time, until either it is pruned or until it is compared to all points in the dataset, i.e., no more NCBs left. The same reasoning can be applied for the second level of parallelization. To perform the detection in the dataset, all points need to be classified. But, since the KNN searches are independent, they may also be done in parallel. Therefore, we propose to divide the outlier detection into KNN *iterations*. In each iteration, a batch B of test points is compared to an entire NCB, in parallel. For brevity, hereafter, we may also refer to a generic test batch B as a *Test Point Batch (TPB)*.

4.3 KNN iteration

Using the parallelization strategy outlined previously, the classification process of each TPB can be broken-down into a series of KNN iterations. Given a TPB B, each iteration executes a "step" in B's KNN search [4] with the goal of improving D_p^k for every point $p \in B$, by refining the their set of closest neighbors using neighbor candidates from a given NCB.

KNN iterations are at the core of all GPU outlier detection algorithms that we implemented. In this section, we explain in greater detail which operations are needed by the KNN iteration and we will show the host-side algorithm that controls the computation. All functions that start with the prefix *Gpu* do setup work for computation kernels to run on the GPU. For an in-depth look on how to implement the GPU kernels needed by the KNN iteration, see Appendix A.

4.3.1 KNN iteration algorithm

Algorithm 6 shows in detail the operations performed during a KNN iteration. Given a TPB B with b test points and a NCB with c neighbor candidates; the iteration starts with the computation of all pair-wise distances between the points in B and NCB. This results in an $b \times c$ distance matrix (Δ), where row i contains the distances between the i-th test point and all points in the NCB. Then, each row of Δ is sorted in ascending order such that the first k elements in each row will correspond to the test points' closest neighbors in the NCB. Next, this $b \times k$ sub-matrix is used to update[5] the *partial* KNN *result* (`pKnn`), i.e., improve the current set of nearest neighbors of B by including neighbors closer still, from the NCB. Consequently, tighter anomaly score upper-bounds (vector S) may be computed for the points in B.

The last part of the iteration uses the updated anomaly scores to try to prune test points. The pruning method is divided into two parts: (i) mapping, where the GPU records the index of the test points which can still not be pruned by the ANNS; and (ii) gather, in which the points not pruned are repacked in B, such that the pruned ones are overwritten and thus effectively erased.

It should be noted that the KNN iteration is actually a two-step process. The first part is shown in Algorithm 6 and runs asynchronously on the GPU. In the second part, the host needs to decide what to do next based on the iteration's result. There are three possible outcomes:

[4] KNN search of a TPB should be understood as the KNN search of every point within the batch

[5] The update process has three steps: (i) "concatenate" the matrices; (ii) sort their rows in ascending order ; (iii) truncate the rows, keeping in `pKnn` only the k closest neighbors

Algorithm 6: Algorithm for KNN iteration

```
1 Function GpuKnnIteration (B, NCB, k, D_M^k)
2     Δ ← GpuComputeDistPair (B, NCB)
3     Δ ← GpuKSortDistances (Δ, k)
      // Update outlier candidates' closest neighbors
4     B.pKnn ← GpuUpdateNearestNeighbors (B.pKnn, Δ)
5     B.S ← GpuUpdateAnomalyScore (B.pKnn)
6     ev ← GpuPruning (B, D_M^k, k)
7     return ev
```

1. B **is empty**: The ANNS pruned all the test instances. The host must end B's KNN search.
2. B **is not empty and there are still NCBs to use**: The host will continue the search
3. B **is not empty and there are no NCBs left**: The KNN search ended but there are still test instances in B. The appropriate action will depend on the meaning of the points remaining, which will be different depending on algorithm and context. Those points could be either outliers or outlier candidates.

4.3.2 Implementing the ANNS

Effective pruning is essential for fast outlier detection, but, if not implemented properly, it can impose a significant overhead. Once the ANNS is applied, the GPU needs to inform the CPU how many points there are left in the TPB and wait for further instructions. However, due to the high latency of communicating through the PCI-E bus, the GPU idles for a significant amount of time, thus decreasing the overall computation throughput of the algorithm.

We reduce the pruning cost of in two ways. First, it is implemented using the *stream compaction* primitive, which is explained in Appendix A.4. Second, pruning is used less frequently, in order to amortize the resulting idle time over large amounts of computation. The KNN iteration design presented divides the detection into large blocks of computation, which can effectively use the GPU's hardware. Pruning is then applied only at the end of each iteration, i.e., once per NCB rather than once per neighbor candidate. Moreover, the ratio between pruning idle time and time spent on computation can be regulated by choosing appropriate sizes for the TPBs and NCBs.

The downside of using the ANNS less frequently is reducing the pruning efficiency. The CPU algorithms, in general, apply the ANNS after every neighbor comparison, allowing them to prune test points as soon as they are proven inliers. This is equivalent of having an NCB of size 1. However, with our KNN iteration design, the ANNS will be only applied once all neighbors in the NCB are compared to the test points, resulting in more computation performed overall. Nevertheless, as we show in the experiments (Chapter 5), our approach provides a good balance between reducing pruning overhead and allowing the ANNS to still be effective at avoiding unnecessary computation.

4.4 I/O subsystem for out-of-core GPU outlier detection

Streaming data from disk to the GPU is extremely costly and thus posed a significant barrier to the development of out-of-core outlier detection algorithms. In this section, we discuss the challenges of out-of-core execution in general; why it is specially costly to GPU algorithms; and how we addressed these challenges.

4.4.1 The cost of out-of-core execution on GPUs

Out-of-core algorithms incur two main performance penalties when streaming data from disk. First, the latency of fulfilling the data request. This is the time between requesting a chunk of the dataset and it being available in RAM. It can be affected by many factors such as disk latency, disk bandwidth and the degree of post-processing that has to be applied to the chunk once it reaches memory, for instance, parsing the data from text format into binary. The second performance penalty is idle time: not performing any computation while waiting for data to reach RAM. These overheads affect both CPU and GPU algorithms but they are far more severe for the latter. Sending data to the GPU requires one extra data transfer, from RAM to the the GPU's global memory, further increasing I/O latency and GPU idling time. Additionally, GPUs have up to two orders of magnitude greater computation throughput than CPUs, meaning the amount of computation not performed per unit of idle time is equally larger. This puts an upper-bound on the computation throughput that can be achieved with the GPU, negating the benefit of using it over CPUs. Therefore, minimizing I/O overhead and idle time is paramount to achieve any appreciable reduction in computation time with GPUs.

4.4.2 Important design decisions

The primary concern when developing out-of-core algorithms should be to minimize disk accesses altogether, by designing outlier detection algorithms that perform as few dataset passes as possible, e.g., `Diskaware` and `Dolphin`. Only after the appropriate algorithmic design is chosen, one should focus on addressing the I/O overhead. To implement the GPU algorithms presented in Section 4.5, we used a common low-level I/O subsystem that implements optimizations to reduce the I/O costs outlined previously. The I/O subsystem implemented has three major design decisions: (i) Bulk I/O, (ii) Binary datasets and (iii) Asynchronous I/O. Each of these will be explained in more detail bellow.

Bulk I/O

Hard drives have a significant access latency cost due to their mechanical nature. This cost can be mainly broken into: (i) Seek time, the time taken by the head assembly to move to a given position and (ii) Rotational Latency, the time it takes for the target sector to rotate under the read head. Therefore, performing many small data reads, e.g., reading one NCB at a time, is considerably less efficient than performing a single large read. For example, the hard drive used in the experiments (**??**) has an average data read rate of 156MB/s, an average seek time of 8.5ms and an average rotational latency of 4.2ms (Seagate [2011]). To transfer a single NCB with 8192 points[6], each with 30 attributes[7], it would take an average of 18ms, of which 68%, 12ms, is just latency. However, if we transfer 20 NCBs at a time, as was done in the experiments, the latency overhead can be reduced to just 7% of the I/O time. This example perfectly highlights the importance of bulk I/O for improving data transfer efficiency and reducing overall I/O time.

To implement bulk data transfers, our I/O subsystem uses a LRU cache on the GPU's global memory. The dataset is logically split into chunks, several point batches in size, and each will be mapped to a *cache page*. When a NCB is requested, for instance, the cache will first determine which chunk it belongs to and whether it is already in cache. If that is the case, the request is fulfilled immediately, otherwise, the target dataset chunk is loaded from disk into the GPU. Any further requests for data within this chunk will hit the cache. Because the data access pattern of outlier detection algorithms is sequential (Section 3.4), all the chunks' point batches will be requested at least once. Thus, the latency cost of reading data from disk is paid only once and

[6]NCB size used in the experiments
[7]NCB of the `2MASS` dataset

spread over all the point batches within a given chunk. This is crucial optimization to reduce overall I/O cost, increasing the effective data transfer rate of the disk (in points per second) and allowing the GPU to achieve higher computation throughput rates.

Binary datasets

Our I/O subsystem only supports datasets in binary format. This encoding is more compact than text and in the datasets used in our experiments (**??**), for instances, it reduced dataset sizes by around 60%. As a result, the effective streaming throughput of the disk, in terms of data points per second, is increased by the same proportion and, in turn, significantly reducing data transfer times from disk to RAM. Additionally, since the data is already in binary format once it reaches RAM, it does not need to be parsed and can be sent immediately to the GPU. This results in a massive reduction on the latency between requesting a data point batch and it being available on the GPU. In summary, using datasets in binary format reduces point batch transfer times through compression and eliminates the need for parsing, thus further reducing the time data takes to reach the GPU.

Asynchronous I/O

Asynchronous I/O is implemented to allow overlapping data streaming with both work scheduling and computation on the GPU. The main thread can make an asynchronous request for a TPB, for instance, and then continue execution and enqueue all work to be performed on that TPB, while the data transfer is taking place. The time taken to enqueue and schedule work on the GPU[8] will be overlapped by the I/O time, allowing the computation to be executed immediately after the TPB reaches the GPU. As a result, GPU idle time due to work scheduling is considerably reduced. Additionally, data prefetch can be used to overlap I/O with computation. The main thread can enqueue work to be performed on data already on the GPU's global memory and then prefetch the next TPBs and NCBs, such that the GPU will remain busy while I/O is taking place. As mentioned before, the algorithm's data access is sequential, thus the pre-fetching can be used through the entire detection process, massively reducing GPU idle time, thus reducing processing times.

[8] There is a significant latency between sending commands to the GPU and those commands being executed, due to PCI-E latency, enqueuing and scheduling costs. For more information Khronos Group [2017]

4.4.3 I/O subsystem architecture

The I/O subsystem can be divided into two layers. The first one is the GPU layer that offers fundamental abstractions to the outlier detection algorithms, such as data points, TPBs and NCBs. The layer bellow is the Storage layer and it is responsible for loading data from disk and then uploading it to specific buffers on the GPU, in an asynchronous manner.

4.4.3.1 GPU Layer

The main component of the GPU layer is the cache discussed previously, which keeps the most recently used dataset chunks in the GPU's global memory. The cache receives, from the GPU layers above, requests for specific point batches, e.g., an NCB, to be copied to a given GPU buffer supplied with the request. The cache then determines which dataset chunk contains the NCB and checks if the target chunk is already in cache. If that is the case, the cache will enqueue, using the *OpenCL* API, an asynchronous copy of the requested NCB. The API will return a **dependency event object** whose state is tied to the the status of the copy. This event is then returned to the caller, which can then use it as dependency lock when enqueuing computation to be made on the NCB. The *OpenCL* runtime will ensure the dependency is met, only allowing the computation to be performed once the copy completed.

On the other hand, if the dataset chunk containing the requested NCB is not in cache, the cache will have to request the Storage layer to upload it to the GPU. In this scenario, the cache fulfills the request in two steps. First, it asks the Storage layer to fetch the target dataset chunk and upload it to a specific cache page, selected through the LRU policy. The cache also creates and sends to the Storage layer a **dependency event** to be tied to the status of the upload. In the second step, the cache enqueues an asynchronous copy of the requested NCB, using the aforementioned event as a dependency. The *OpenCL* API will return an event object for the NCB copy, which will be then returned to the caller. As mentioned before, the caller will be able to use this event as a dependency when enqueuing computation to be done on the NCB.

This chain of dependencies, which is established in both scenarios between dataset chunk upload, batch copy and computation; is enforced by the *OpenCL* runtime and is the critical enabler of asynchronous I/O for both device and host-side code.

4.4.3.2 Storage Layer

The storage layer implements asynchronous host-side I/O by using a consumer-producer architecture, which somewhat decouples the GPU computation rate from the disks data streaming rate allowing the overlapping of I/O and computation.[9]

This layer is divided into three main components: *DataSource*, a *Work queue* and *ReadWorkers*. The *DataSource* component receives, from the GPU layer, requests for specific dataset chunks to be uploaded to a target cache page (i.e., GPU buffer). Then, it builds a *ReadRequest* object containing all the necessary information to read the chunk off the disk, asynchronously: chunk offset, chunk size, host-side buffer, callback function and request id. The *ReadWorkers* are the entities that perform the data reads, consuming *ReadRequest* objects from the *Work queue*, and are executed in separate threads. Once a read is finished, the *ReadWorkers* calls the callback function passing the request id from the *ReadRequest* to notify the *DataSource*. The *DataSource* will then enqueue an asynchronous copy of the just-read dataset chunk to the cache page originally supplied. The **dependency event** originally received by the Storage layer will be passed with the API call, so that the *OpenCL* runtime will tie the event's state to the copy status, ensuring that the previously discussed dependency chain works[10]. After the callback is finished, the *ReadWorker* thread can return to consuming the *Work queue*. Please note that the work performed during the callback is minimal, thus the I/O thread spends the overwhelming majority of its time running the *ReadWorker*.

4.5 Distance-based outlier detection algorithms for GPUs

In this section we will describe how to port `Orca` and `Diskaware` to the GPU, using the KNN iteration design proposed. Additionally, we will introduce our GPU algorithm that is based on `Diskaware` but offers significantly better performance when a poor initial D_M^k is available. Lastly, `Dolphin` was not ported due to its index, which is hard to implement in GPUs and it is still an open problem.

[9] The maximum computation throughput is still limited by two factors: arithmetic intensity and disk bandwidth. The overlapping just allows the algorithm to get closer to the theoretical peak compute rate.

[10] Dataset chunk upload, point batch copy and computation

Algorithm 7: Orca-GPU Algorithm

```
1  Function OutlierDetection (D, k, p)
2      O ← []
3      while D.HasTestBatchLeft () do
4          B ← NextTestBatch(D)
           // Run KNN search of B
5          while ! B.FinishedKnn () do
6              NCB ← NextNeighborBatch (D, B)
               // Dispatch all GPU work
7              GpuKnnIteration (B, NCB)
8              if !FinishLastKnnIteration (B, O) then Break
9          end
10     end
11     return O

12 Function FinishLastKnnIteration (B, O)
       // Ensure GPU finished the iteration
13     WaitLastKnnIteration (B)
14     if B.Empty () then
           // Batch was fully pruned
15         return False
16     else if B.FinishedKnn () then
           // New outliers found
17         D^k_M ← O.GpuSaveOutliers (B)
18         return False
19     else
20         return True // Continue processing B
21     end
```

4.5.1 Orca-GPU

A GPU version of the `Orca` algorithm (Algorithm 7) has a nested-loop design. The outermost loop builds TPBs for classification with test points from the dataset, whereas the inner loop executes the KNN search of the TPB. After requesting the GPU to load the next NCB (Line 5), the host will schedule the KNN iteration to be executed and wait until the GPU uploads the number of points left in the TPB after the pruning (Line 13). If the TPB was fully pruned, its KNN search is halted (Line 8) and the host schedules the classification of a new one. Otherwise, the host schedule the next iteration of the TPB's KNN search (Line 5).

For some TPBs, their KNN search will finish but they will still have points not pruned (Line 17). These remaining points will be classified as *outliers*, in a process similar to the `pKnn` update: (i) Insert points in the list; (ii) Sort them in descending order according to anomaly score; (iii) Truncate the list, keeping only the top-n anomalies. Lastly, D_M^k is updated using the anomaly score of the current n-th outlier. After all

TPBs are classified, the detection algorithm returns the top-n outliers as the outliers of the dataset (Line 11).

4.5.2 `Diskaware-GPU`

Before porting `Diskaware` to the GPU, two pre-requisites need to be met. First, develop a data structure for storing outlier candidates, which is a key aspect of the algorithm. Second, the KNN iteration procedure presented needs to be extended to also refine the nearest neighbor list (`pKnn`) of the NCBs used, i.e., batches of outlier candidates in this case.

4.5.2.1 Storage of Candidates

The container for storing outlier candidates has a set of requirements that need to be met. First, it needs to be able to not only store the candidate points themselves, but also associated data needed for the KNN iteration such as: (i) the points' norm; (ii) `pKnn` and (iii) anomaly score. Second, the container needs to provide efficient read, update and removal operations. The read is necessary because every KNN iteration will need to read candidates from the container to server as neighbors. The removal is required because candidates can be pruned during the iteration due to a D_p^k reduction. Lastly, the update operation will be needed for updating the `pKnn` and score of the candidates that were not pruned.

The candidate container was implemented using a *Structure of Arrays (SoA)* layout (Oster [2008]) to ensure coalesced data access. The candidate data is divided into four *parallel arrays*, one per data being stored: features, norm, `pKnn` and score. For efficiency reasons the arrays are implemented as circular buffers. Before a KNN iteration, the outlier candidates that will be part of the NCB are popped from the front of the container, along with their associated data. Once the iteration ends and the pruning is applied, the remaining candidates are re-inserted at the back of the container. This procedure handles both the candidate **removal** operation due to pruning and the `pKnn` and score **update** operation due to the refinement of the closest neighbors list. Since the buffers are circular, both removing from the front and appending at the back have $O(1)$ running time complexity.

Algorithm 8: `Diskaware-GPU`-KNN iteration

```
1  Function GpuDiskawareKnnIteration (B, C, k, D_M^k)
2      Δ_TPB ← GpuComputeDistPair (B, C.pts)
3      Δ_NCB ← GpuTransposeMat (Δ_TPB)
       // Run the KNN iteration of each point batch
4      KnnIteration (B, C, Δ_TPB, k, D_M^k)
5      ev ← KnnIteration (C, B, Δ_NCB, k, D_M^k)
6      return ev
7  Function KnnIteration (B, C, Δ, k, D_M^k)
8      Δ ← GpuKSortDistances (Δ, k)
       // Update outlier candidates' closest neighbors
9      B.pKnn ← GpuUpdateNearestNeighbors (B.pKnn, Δ)
10     B.S ← GpuUpdateAnomalyScore (B.pKnn)
       // Decide which candidates to prune
11     ev ← GpuPruning (B, D_M^k, k)
12     return ev
```

4.5.2.2 knn iteration

During `Diskaware-GPU`'s **first phase**, its KNN iteration (Algorithm 8) refines the `pKnn` of points in the TPB as well as those in the NCB[11]. It first computes the distance matrix for the TPB's KNN iteration (Line 2) and then transposes the matrix (Line 3) to obtain the distances for the NCB's KNN iteration. This optimization avoids computing the same set of distances twice. Next, an overloaded version of the KNN iteration algorithm, which receives the distance matrix instead of computing it, is used to run both TPB and the NCB's KNN iterations.

4.5.2.3 The algorithm

Phase 1 `Diskaware-GPU`'s first phase (Algorithm 9) traverses the dataset once and runs a KNN search for every TPB, using only the outlier candidates saved as the neighbor candidates (Lines 9 and 11). After every KNN iteration execution, the host decides what to do next based on the result obtained (Line 13) and, finally, the neighbor candidates that were not pruned are re-inserted into Q with their updated `pKnn` and scores (Line 14). After the KNN search of a TPB ends and before the moving to process the next one, the algorithm ensures that the outlier candidate storage is not empty. If it is, the algorithm simply inserts a TPB into the storage (Line 5), such that the next one o be processed has neighbor candidates to be compared to.

[11] In this case, the points in the NCB are sourced from the set of saved candidates

Algorithm 9: `Diskaware-GPU-` Phase 1

```
1  Function DiskawarePhase1 (𝒟, 𝒪, D_M^k, k)
2      Q ← ∅
3      while 𝒟.HasTestBatchesLeft () do
4          if Q.Empty () then
               // Save some points as candidates to use them as neighbors
5              Q.Savebatch (NextTestBatch(𝒟))
6              continue
7          end
8          B ← NextTestBatch (𝒟)
9          SetKnnSearchLength (B, Q.size)
           // Run B's KNN search
10         while cont do
11             C ← Q.GetNextBatch ()
12             GpuDiskawareKnnIteration (B, C)
13             cont ← FinishLastKnnIteration (B, 𝒪)
               // Save the candidates remaining with updated pKnn and score
14             Q.SaveBatch (C)
15         end
16     end
17     return Q
```

Phase 2 The second phase (Algorithm 10) is straightforward and uses the regular KNN iteration implementation (Algorithm 6), which sources neighbor candidates from the dataset itself. The phase starts by building a list of TPBs, containing all the candidates that were saved during the previous phase. Then, it runs the KNN search of all the TPBs in lock-step, such that each NCB is loaded from disk only once. If the KNN search any particular TPB finishes, it is removed from the list of active TPBs and the execution continues until there is none left (Line 4).

4.6 `DROIDg` (Disk-Resident, Outller Detection using GPUs)

The outlier detection algorithms presented in Section 3.4 and their GPU counterparts, which we proposed in Section 4.5, all have one of these two major drawbacks: either (i) they perform way too many dataset traversals or (ii) their performance is entirely dependent on the quality of the initial D_M^k value, thus severely diminishing the applicability of these algorithms as we discussed in Sections 3.5 and 3.6. Therefore, in this section we introduce `DROIDg`, a new out-of-core outlier detection algorithm for GPUs. It is high-performance and requires only a small and constant number of dataset traversals. Additionally, it is very versatile, capable of providing fast detection times

Algorithm 10: `Diskaware-GPU-` Phase 2

```
1  Function DiskawarePhase2 (D, Q, O, D_M^k, k)
2      tpbs ← ∅
       // Build a list of TPBs with all candidates
3      while ! Q.Empty () do
4          tpbs.Append(Q.GetNextBatch())
5      end
6      ClassifyBatches (tpbs, D, O, D_M^k, k)
7  Function ClassifyBatches (tpbs, D, O, D_M^k, k)
8      while ! tpbs.Empty() do
           // Run one, regular, KNN iteration per TPB
9          forall B ∈ tpbs do
10             NCB ← NextNeighborBatch (D, B)
11             GpuKnnIteration (B, NCB)
12             if !FinishLastKnnIteration (B, O) then tpbs.Remove (B)
13         end
14     end
```

regardless of the quality of the initial D_M^k, even if it is equal to 0.

Our algorithm is based on the `Diskaware`'s two phase design but, similarly to `Orca`, it is able to improve the D_M^k during the detection. In other words, it trades extra I/O cost, in the form of a small number of extra dataset traversals to classify more outlier candidates, which improves the D_M^k and massively reduces the computation costs. Thus it achieves a better balance between reducing I/O and computation costs than any other algorithm discussed in this work.

Combining these approaches is not trivial and to understand why, let's first re-examine the effects of a poor initial D_M^k on `Diskaware` (Section 3.6). Under such scenario, the amount of candidates saved in Q grows significantly, resulting in larger runtime for both detection phases. In the first phase, there is a considerable increase in the TPB's KNN search cost because all the outlier candidates are used as neighbor candidates. In the second phase, a much greater number of full KNN searches will be performed, one per candidate.

To achieve fast detection times when using a bad initial D_M^k, our algorithm will have to avoid the runtime increases mentioned above while the D_M^k is being increased. This process is slow because the ΔD_M^k per dataset pass is small. While this process takes place, our algorithm is subject to the same side-effects of a large Q as `Diskaware`, since the former is based on the latter. Therefore, in addition to enabling the D_M^k improvement in `Diskaware`'s two phase outlier detection design, our algorithm must address the following issues:

1. Expedite the D_M^k improvement process to maintain the number of dataset passes low and minimize the overhead caused by a large Q.
2. Decouple the cost of the KNN search of TPBs from the number of outlier candidates saved.
3. Lower the cost of classifying a large set of candidates. This will be relevant for D_M^k improvement and the detection's second phase.

Next we explain how the key components of the algorithm work and how the issues above are solved.

4.6.1 Phase 1 - knn search

Our algorithm's first phase uses a different KNN search algorithm from `Diskaware`, which makes the search's cost independent from the size of Q while limiting the increase in false positives among the candidates. This is accomplished by: (i) limiting the number of candidates used per KNN search and (ii) using `Orca`'s ROCN strategy for improving the pruning of test instances. The search is divided into two stages.

Stage 1 The goal of this stage is to decrease the outlier candidates' D_p^k and prune those that can be proved inliers. It works similarly to `Diskaware`'s KNN search, except it only compares the TPB to the first TNCBs from Q, instead of all of them. Note that the NCBs will be built with outlier candidates selected in a Round-Robin fashion. This is the case because the points are removed from the front of Q and, those not pruned, are re-inserted at the back[12]. This ensure that eventually all candidates in Q will their anomaly score upper-bounds improved. However, there is a downside to this approach. By limiting the the number of candidates compared to each TPB, the algorithm also slows down the D_p^k increase of the candidates, thus decreasing their pruning rate. The second stage of the KNN search mitigates this side-effect.

Stage 2 To counter the slower pruning rate of the outlier candidates, this stage seeks to reduce the amount of points saved as candidates in the first place. This can be achieved by employing `Orca`'s ROCN strategy to improve the pruning of test points (TPB). After the first KNN search stage, the points remaining in the TPB are compared to a random sample of the dataset. However, this stage needs to be implemented carefully to avoid comparing points to the same neighbor candidates twice, which would lead to incorrect results. Our ROCN implementation takes advantage of the

[12]See the description of the outlier candidate storage in Section 4.5.2.1

Algorithm 11: Second stage of modified KNN search

```
1  Function KnnStage2 (D, B, Q, O, D^k_M, k)
2      SetKnnSearchLength (B, S)
3      SetKnnSearchStart (B, B.start + tpbsize)
4      while ! cont do
5          NCB ← NextNeighborBatch (D, B)
6          GpuKnnIteration (B, NCB)
7          cont = FinishLastKnnIteration (B, O)
8      end
       // Put new candidates in a stagging buffer
9      StageBatch(R, Q, B)
10 Function StageBatch (R, Q, B)
11     if R.size ≥ S then
           // Batch at the front of R completed stagging.  Remove it
12         Q.SaveBatch (R.PopBatch())
13     end
14     R.PushBatch (B)
```

dataset randomization. The random sample used is comprised of the STPBs following the TPB being currently processed. In this case, the double comparison issue can only happen between newly added candidates and the points in its random sample, i.e., the next STPBs. So, to prevent it, new candidates are place at a staging buffer R during the next S KNN searches and not used for building NCBs.

Algorithm 11 shows in detail how this stage of the KNN search works. Lines 2 and 3 set the KNN search to use only SNCBs and to use get the NCBs starting from the next TPB. The search proceeds normally, using the original KNN iteration implementation and, at the end, the points remaining in B are added to the staging buffer R. While adding the new batch, the algorithm checks whether there are already S batches in R (Line 11). If so, the one at the front completed its staging and can be inserted into the candidate buffer Q.

4.6.2 Efficient outlier candidate classification

Classifying a large number of outlier candidates is costly, since it requires a full KNN search per candidate. But it is also wasteful. The majority of those candidates are inliers that could be easily pruned had a better threshold been available. Therefore, we introduce an algorithm for efficiently identifying and discarding those inliers. It prioritizes the classification of candidates likely to have large σ_p, thus achieving larger ΔD^k_M per execution. Consequently, it can use the ANNS to discard a large amount of candidates cheaply. This makes this algorithm invaluable for both improving the D^k_M and the performance of the detection's second phase.

Algorithm 12: Efficiently reducing the size of Q

```
1  Function PruneCandidates (D, Q, O, D_M^k, k)
2  |  DkminImprovement (D, Q, O, D_M^k, k)
   |  // Use new/higher D_M^k to discard as many inlier as possible from Q
3  |  PruneInliers (Q, D_M^k)

4  Function DkminImprovement (D, Q, O, D_M^k, k)
5  |  GpuSortCandidatesByDk (Q)
6  |  tpbs ← []
7  |  for i = 0 to H do
8  |  |  tbps.Append (Q.GetNextBatch())
9  |  end
10 |  ClassifyBatches (tpbs, D, O, D_M^k, k)
```

The outlier candidate classification algorithm has two steps: (i) improve the D_M^k and (ii) prune candidates using the new threshold. It increases the D_M^k as much as possible by classifying a small subset of Q containing the candidates more likely to have larger anomaly scores. Such subset is chosen using a novel ROCO heuristic based on the following assumption:

Assumption 7 *Candidates with larger D_p^k are more likely to have higher anomaly scores*

The algorithm (Algorithm 12) starts by sorting the candidates in Q in descending order, according to their D_p^k. Then, it builds H TPBs using the candidates with higher D_p^k and classifies them using `Diskaware`'s second phase batch classification algorithm (`ClassifyBatches` - Algorithm 10). The newly classified outliers help refine the set of top-n anomaly thus increasing the D_M^k. Finally, the ANNS is used to prune the candidates remaining in Q using the new D_M^k (Line 3). It should be noted that newer candidates tend to have larger D_p^k because they were used in less KNN iterations. However, as the experiments show (Section 5.6.1), this heuristic is still very effective.

4.6.3 The algorithm

In this section, we discuss how our algorithm incorporates the two changes discussed previously: (i) the new KNN search algorithm and (ii) candidate pruning method.

Phase 1 Our algorithm's first phase (Algorithm 13) is very similar to `Diskaware`'s, but it incorporates two improvements. First, it uses the two stage KNN algorithm we discussed previously, which keeps the search cost low, regardless of the number of candidates saved. The downside is the reduced pruning rate of outlier candidates,

Algorithm 13: DROIDg

```
1  Function DROIDgPhase1 (D, O, D_M^k, k)
2      Q ← ∅
3      while D.HasTestBatchesLeft () do
4          if Q.Empty () then
               // Save some points as candidates to use them as neighbors
5              Q.Savebatch (NextTestBatch(D))
6              continue
7          else if Q.IsAlmostFull() then
               // Free space in the candidate buffer
8              PruneCandidates (D, Q, O, D_M^k, k)
9              continue
10         B ← NextTestBatch (D)
           // Run B's KNN search
11         KnnStage1 (B, Q, O, D_M^k, k)
12         KnnStage2 (B, D, O, D_M^k, k)
13     end
14     return Q
15 Function DROIDgPhase2 (D, Q, O, D_M^k, k)
16     tpbs ← ∅
17     PruneCandidates (D, Q, O, D_M^k, k)
18     while ! Q.Empty() do
19         tpbs.Append (Q.GetNextBatch ())
20     end
21     ClassifyBatches (tpbs, D, O, D_M^k, k)
```

which is only partially offset by the KNN search's second stage. As a result, Q can become full during the detection on a large enough dataset. The second improvement addresses this issue. If Q is close to get full, the candidate pruning algorithm will be executed to efficiently free space in Q and allow the first phase to continue. Moreover, with the improved D_M^k, the ANNS will become more efficient, further slowing down the growth of Q and significantly reducing the chances of the buffer becoming full again.

Phase 2 The second phase of our algorithm can efficiently classify a large number of candidates, even when the classification threshold is poor. It starts by using the candidate pruning algorithm (Algorithm 12) to efficiently discard the majority of inliers among the candidates in Q. Then, only the fewer remaining candidates will undergo classification (Algorithm 13).

It should be noted that even if the basic premise of a poor threshold does not hold, our algorithm still has a performance similar to `Diskaware`'s, as the experiments show (Section 5.4). During the first phase, there will be no extra dataset traversals, since ANNS will be very efficient and Q will not become full. Moreover, the cost of the second

stage of the KNN search will be minor. The second phase of our algorithm can be slightly longer. If the number of candidates saved is bigger than H, the algorithm will perform an extra dataset traversal to, unnecessarily, improve the D_M^k. But, again, the extra cost will be minor. On the other hand, if the initial threshold is not good or is even 0, our algorithm will perform considerably better than `Diskaware`. Our new KNN search algorithm for the detection's first phase allows the algorithm to handle a large number of candidates without degrading performance. This is critical. The proposed outlier candidate classification algorithm is equally as important. By leveraging our novel ROCO heuristic, it substantially increases the ΔD_M^k per dataset pass. Consequently, the overall number of traversals is kept low and the amount of candidates pruned per traversal is increased. As the experiments show (Section 5.6.2), the latter is crucial to curb the cost of the detection's second phase.

4.7 Related work - GPU algorithms

In this section we review three of the main outlier detection algorithms for GPU from the literature. However, none of them is suitable for our usage context: disk-resident, large scale datasets. Hence, why they will not be used in our experimental analysis. They are included here for sake of completeness.

4.7.1 LOFCUDA

Alshawabkeh et al. [2010] were one of the first to propose the use GPUs for outlier detection. Their method is based on the LOF algorithm. It assigns to each point p a *Local Outlier Factor*[13], which quantifies how different is the density of the region around p in comparison to the surroundings of its k closest neighbors. The points with the largest LOF scores are considered the outliers. Though LOF score produces high quality outlier detections, their algorithm is unsuitable for processing large scale, disk-resident, datasets. First, their algorithm has a quadratic runtime complexity on N. Second, their implementation performs the KNN search of all the dataset points concurrently, thus it has to store a $N \times N$ distance matrix in the GPU memory. This makes its space complexity also quadratic on N, severely limiting the size of datasets that can be processed. For instance, for $N = 65$K points, their algorithm would need more than 15GB of video memory, which is considerably more memory than most consumer GPUs have available nowadays.

[13] LOF: an anomaly score.

4.7.2 GPU-SS

More recently, Angiulli et al. [2016] implemented a family of GPU algorithms based on the approximate algorithm *SolvingSet* (Angiulli et al. [2006]). The *SolvingSet* is also a distance-based detection algorithm and uses D_p^k as the anomaly score metric. Moreover, points may have one of three labels: (i) *candidates* are the ones with the highest likelihood of been outliers but have not been classified yet; (ii) *inactive*, for the ones already classified as non-outliers; (ii) *active* are the remainder of the points, with D_p^k larger than the classification threshold, but not large enough to qualify them as candidates.

The GPU algorithm proposed, like the original *SolvingSet*, performs multiple passes through the dataset. At iteration j, it compares the current set of candidates, C_j, to all points in the dataset, regardless of label. At the end the iteration, σ_p of the candidates will have been computed[14] and D_p^k of the active points will have been improved. Candidates whose anomaly score is above D_M^k are classified as outliers and their scores are used to update the threshold. The remainder of the candidates are classified as non-outliers and are removed from C_j. Similarly, active points whose D_p^k fell bellow the pruning threshold are labeled inactive. Finally, the next candidate set C_{j+1} is built by selecting the m active points with the highest D_p^k. This process is repeated until there are no active points left.

As their experiments show, this iterative approach is incredibly successful at reducing the amount of computation necessary for detection, reducing the number of point comparisons to below 3% for most of the dataset. Consequently, they reported that their GPU implementation achieved a 45X speedup over its CPU counter-part. However, its necessary to highlight some shortcomings of their study. Their experiments were conducted using only small datasets with the largest one comprising just 1.6M 3d points. Moreover, whistle their algorithm is effective at reducing computation, it requires multiple passes through the dataset, making their solution ill suited for our desired out-of-core execution scenario.

4.7.3 Stream GPU outlier detection

HewaNadungodage et al. [2016] proposed a GPU algorithm detecting anomalies in data streams. The anomaly score metric used was the inverse of the density around a point and it was computed using density estimation kernels. Their method splits the input stream into non-overlapping batches of points and by combining summary

[14] They were compared to all points in the dataset

statistics from batches already processed with data form the current window, it is capable of classifying data more accurately than GPU_SS, while performing just one pass through the dataset.

To generate the summary statistics, it partitions each of the d data dimensions into k bins with a user-specified width of δ, totaling k^d bins. The points of the stream are mapped to one bin, according to their attribute values, and for each bin B_j the method stores: (i) C_j, the number of points in it so far and (ii) the aggregate mean value of all the points mapped to B_j. Moreover, their method also computes global statistics such as the mean (μ) and standard deviation (Σ) of the whole dataset. These statistics are then used to estimate the density $p_s(X)$ around the point X considering all the data seen so far. To compute the anomaly score of the point X, their method combines, through a weighted sum, $p_s(X)$ and the density around X considering just the points in the current window.

The authors compared the GPU and CPU implementations of their algorithm. They used two datasets Kddcup and Covertype, with 3.8M and 405k points respectively and their CPU versions was executed using 16 threads. They showed that the GPU version was up to 20X faster than the CPU counter-part, for both datasets.

This algorithm only requires a small amount of data to be in memory to perform the detection: (i) the summary statistics and (ii) one batch of points. Therefore, it can process arbitrarily large datasets. Furthermore, its ability to perform the detection while requiring only one pass through the data makes it well suited for processing disk-resident dataset and a competing method to our GPU algorithm. However, the authors' study and algorithm has important shortcomings. First, the authors only reported their algorithm's performance on very small datasets. Second, the algorithm took between 16 and 32 seconds to process batches of just 60.000 points on low dimensionality datasets and despite using a very powerful GPU: GTX Titan. This indicates that while their method is very efficient with regards to I/O, it is much more computationally intensive than distance-based methods. Unfortunately, we did not have access to their code and were unable to include their algorithm in our experimental analysis to test our hypothesis.

4.8 Summary

In this chapter we presented an entire set of algorithms and abstractions designed for high-performance outlier detection in GPUs. The proposed parallelizations strategy balances high computation throughput with pruning for much better overall perfor-

mance. We also talked in length on how to reduce the performance impact caused by disk accesses, through the use of improved data encoding and asynchronous I/O. Finally, we presented a new and robust distance-based outlier detection algorithm that offers good performance regardless of the quality of the initial D_M^k. The robustness is accomplished through two main improvements over `Diskaware-GPU`. First, a modified KNN search algorithm that still refines the set of saved candidates but that can handle an arbitrarily large Q without degrading performance. Second, our algorithm uses a clever candidate classification method to speedup the D_M^k improvement during the search and to significantly reduce the amount of candidates that must be classified during the second detection phase. Our novel ROCO heuristic is the crucial enabler of this classification method. In the next chapter we will compare our algorithm against `Diskaware` under varying D_M^k quality scenarios, and will also thoroughly analyze the impact of the both improvements implemented in our algorithm.

Chapter 5

Experimental Evaluation

In this section, we analyze the performance the GPU outlier detection algorithms in two parts. First, we compare `DROIDg` to three out-of-core algorithms for CPUs, using scalability tests for the parameters N, n and k. The goal is to assess by how much GPUs can accelerate the outlier detection process. In the second part of the experiments, we compare the GPU algorithms among themselves and analyze how much their performance is influenced by the quality of the initial D_M^k.

5.1 Datasets

Two datasets were used for the experimental analysis. The first[1] was produced by the *Two Micron All Sky Survey* (`2MASS`) mission (Skrutskie et al. [2006]), which used ground-based telescopes to survey the sky on three near-infrared wavelengths, from the year of 1997 to 2001. The specific dataset used, hereafter `2MASS`, is a *Point Source Catalogue* (PSC) containing 127 position and photometry data features on more than 470 million objects. The second dataset used is the *Source Catalogue* obtained from the ALLWISE data release[2] of the *Wide-field Infrared Survey Explorer* (`WISE`) mission (Mainzer et al. [2011]). It used an unmanned satellite, equipped with an infrared-sensitive telescope, to survey the entire celestial sphere in four bands of the infrared spectrum, during the years of 2010 and 2011. The dataset itself, hereafter `WISE`, contains 298 astrometry and photometry data features on more than 747 million objects.

[1] http://www.ipac.caltech.edu/2mass/releases/allsky/
[2] http://irsa.ipac.caltech.edu/data/download/wise-allwise/

	d	N
2MASS	30	303M
WISE	123	678M

Table 5.1: Source datasets after the pre-processing procedure

5.1.1 Pre-processing

Neither of the datasets were fit for our use in their original form since, among other things, they contained non-numeric features as well as missing values. Therefore, both datasets underwent the following pre-processing procedure.

1. Discard non-numeric features
2. Remove all the features with more than 5% of missing values
3. Discard any data instance with missing fields
4. Normalize the dataset using the Z-score normalization and truncate the features in the value range $[-5, 5]$
5. Shuffle the dataset and save it a binary format

This procedure generated two **source** datasets and their basic characteristics are summarized in Table 5.1.

5.1.2 Dataset samples used

The experiments conducted were designed to highlight the GPU's superior performance for outlier detection. Because speedups can potentially be as large as two orders of magnitude, using the full source datasets would lead to unreasonably long runtimes for the CPU algorithms in some tests. Therefore, multiple random samples, of increasing sizes, were created from each dataset and used accordingly during the experiments. Furthermore, since the goal is to test the out-of-core performance of the algorithms, the smallest sample produced has to be larger than the memory available in the test system (16GB). This ensures that dataset traversals will be expensive and impact performance, since no dataset can be loaded entirely into the OS' I/O cache. Tables 5.2 and 5.3 summarize the samples generated.

Sample	N (10^6)	Text (GB)	Binary (GB)
150M	150	43	16.7
200M	200	56	22.4
300M	300	85	33.5

Table 5.2: `2MASS` samples

Sample	N (10^6)	Text (GB)	Binary (GB)
50M	50	57	23
100M	100	114	46
200M	200	227	92
400M	400	456	183
678M	678	776	311

Table 5.3: `WISE` samples

5.2 Methodology

All experiments were run in a single machine, with a Xenon E3-1241 @ 3.5 GHz processor, 16GB of DDR3 and a GTX 980 Ti with 6 GB of VRAM. The datasets were stored in a Seagate Barracuda hard drive, model ST3000DM001, with 156MB/s average data rate and a maximum read seek latency of 8.5 ms(Seagate [2011]). The operating system used was Ubuntu 14.04 LTS and the programs were compiled using GCC 4.8.4 with the `-O3` flag. flag.

It should be noted that the `swap` area of the testing machine had to be disabled because it was causing performance inconsistencies. During the execution of the GPU algorithms, there was little free memory in the system because of the OS' I/O cache and the amount of memory used by the algorithm. We believe this led the virtual memory to evict pages from RAM to the `swap`, thus reducing the disk bandwidth available and impacting the algorithms' performance. By disabling the `swap`, the results' consistency was much improved between test runs.

All the experiments, except when explicitly stated otherwise, use an initial threshold estimate produced by `Dolphin`'s estimation method (Section 3.5.2). Five estimates are produced per test run and the median value is selected for use. Lastly, before every run, the OS' I/O cache is emptied to avoid having parts of the dataset already in RAM.

5.2.1 Algorithms tested

In total, five algorithms were tested and compared to `DROIDg`: four CPU algorithms and one GPU algorithm. Our selection was restricted to the best out-of-core algorithms in the literature, to better understand `DROIDg`'s advantages in an out-of-core scenario when compared to what is already available.

Even though the algorithms chosen may use different approaches for detection and even have different outlier definitions, all of them can be, and **were**, configured to return the same set of outliers. Additionally, there is no ground-truth for the datasets chosen nor it is our intention to determine the quality of the set of outliers returned by

these algorithms. There is already extensive research done on distance-based techniques which shows that they can achieve good detection quality in wide range of applications. As a result, in these experiments we focus solely on the performance aspect of the algorithms and do conduct any analysis on the quality of the set of outliers returned.

5.2.1.1 CPU algorithms

We had access to and used four out-of-core algorithms for CPU as baselines in the experiments, namely: `Orca`, `Diskaware`, `Dolphin` and `DROIDc` the sequential version of our proposed algorithm. In order to better assess the algorithms' performance, we instrumented the first two algorithms to retrieve the number of distance computations performed, number of disk accesses made and I/O time. This was not done with `Dolphin` because we only had access to its binary, which was provided by the first author of the original paper (Angiulli and Fassetti [2009]). Consequently, we lacked proper instrumentation to explain some of the results involving `Dolphin` and had to put forth some hypothesis instead. Lastly, the original `Diskaware` algorithm does not allow the user to change the value of k to be used, which is set to 1. We did not remove this restriction for the CPU algorithm, thus in all the results reported for `Diskaware`, k was equal to 1.

5.2.1.2 GPU algorithms

From all the outlier detection for GPUs discussed in this work, only two were used in the experiments: `DROIDg` and `Diskaware-GPU`. `Orca-GPU`, despite been implemented, was not included in the tests because it suffers from the same issue as the algorithm from which it is based: I/O bottleneck (Section 5.3). Additionally, none of the algorithms in Section 4.7 were used. The first two, LOFCUDA and GPU-SS, were not used simply because they were not designed for out-of-core execution. Thus, they would do considerably more dataset passes than `Orca-GPU` and be even slower. Conversely, the third algorithm was designed for processing data streams and, thus, was able to perform the detection with one pass. Therefore, it would be interesting to include it in our experiments. However, despite reaching out to the authors, we were not able to get access to the its source code for testing.

Parameters Used `DROIDg` and `Diskaware-GPU` were both implemented using our framework, thus they share most of their parameters. All of the parameter values were chosen empirically for giving the best overall performance for our GPU. The batches TPB and NCB were set to 4096 and 8192 points in size, respectively. The batch size of

the GPU's cache was also set to 8192 points and each cache page was 30 batches big, such that the page's I/O latency overhead was less than 10% of the total transfer time (Section 4.4.2). Q was set to a size of 800K for `DROIDg` and 1.6M for `Diskaware-GPU`. The larger Q for the latter algorithm is to prevent the buffer from becoming full during the experiments, since `Diskaware-GPU` is not able to empty Q like `DROIDg` does. `DROIDg` also had some extra parameters. From its new KNN search algorithm, both T and S were set to 10 NCBs. From its outlier classification method, parameter H, the number TPBs with outlier candidates to be classified, was set to 3.

5.3 Parameter Scalability

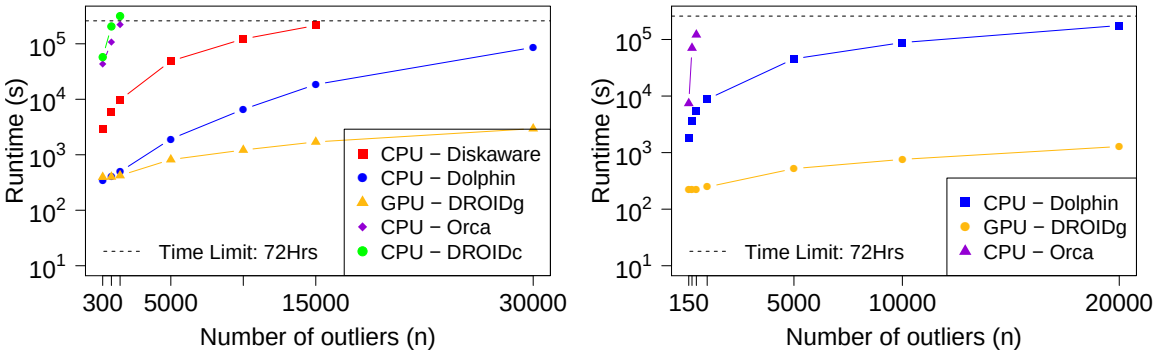
The goal of this section is to determine how effective GPUs are at accelerating the detection process. We compared `DROIDg` against four out-of-core outlier detection algorithms for cpu: `Orca`, `Diskaware`, `Dolphin` and `DROIDc`. Three sets of tests were conducted, each testing the algorithms' scalability with regards to the main detection parameters: (i) dataset size (N); number of neighbors (k); and (iii) number of outliers to be detected (n). Each test was allowed to run for up 72 hours and in case it did not finish, the execution was halted. The halted executions are not shown.

5.3.1 Number of outliers (n)

Good scalability on the amount of outliers being detected is essential in outlier detection algorithms. In most cases, the precise number of outliers in a dataset is unknown and an exploratory analysis has to be conducted. A user will need to analyze the top-n anomaly score curve to determine the number of outliers. A larger n gives a more complete picture of the score distribution in question and allows the number of true anomalies to be gauged with greater confidence.

This experiment assesses the algorithms' scalability with regards to n. The tests were performed using the smallest samples of the two datasets: 150M points for `2MASS` and 50M points for `WISE`. Moreover, the parameter n varied between 0.0002% and 0.04% of the dataset size.

For the `2MASS` dataset, `DROIDg` achieved a maximum speedup of 29X over `Dolphin`, the second fastest algorithm. The speedup achieved increased with n (Figure 5.2a), showing that `DROIDg` has better scalability. For instance, when going from $n = 300$ to 30K, `Dolphin`'s runtime increased over 250X while `DROIDg`'s increased only 7X. In these tests, `DROIDg` did not improve the D_M^k during the detection, thus the superior scalability is mainly due to the GPU. For small n values, the hardware was



(a) Runtime for 150M `2MASS`, $k = 8$

(b) Runtime for 50M `WISE`, $k = 8$

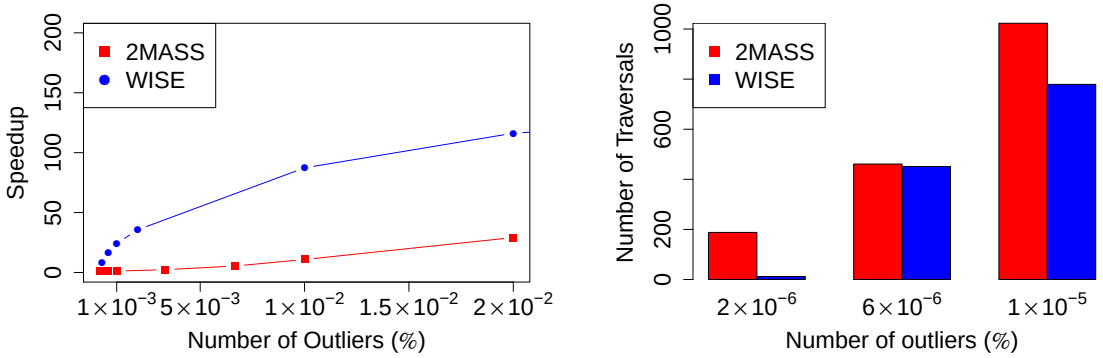
Figure 5.1: Runtime, in seconds, for the n variation test, with y-axis is in log scale. `DROIDc` is not shown for dataset `WISE` because it did not manage to finish any of the test runs within the 72 hour limit

underutilized. But, as the detection cost increased with larger n, `DROIDg` was able to leverage more of the GPU's parallel computing power.

The performance gap between CPU and GPU algorithms was even more pronounced for the `WISE` dataset. `DROIDg` achieved a maximum speedup of 137X (Figure 5.2a), allowing a staggering reduction in detection time for $n = 20$K, from 49 hours to only 21 minutes. The GPU algorithm showed, again, better scalability. Increasing n from 150 to 20K raised `DROIDg`'s detection time by less than 6X, whereas `Dolphin`'s increased 96X. The reason for the larger performance gap in this dataset can not be determined precisely, since we lack proper instrumentation from `Dolphin`. However, we believe the gap can be explained by the higher dimensionality of `WISE`. With four times as many features, the cost of distance computation is increased considerably. But, `DROIDg` is able to scale better due to its highly efficient and parallel distance computation implementation (Appendix A.2).

`Diskaware`, `Orca` and `DROIDc` performed poorly due to the computation cost and, in the case of `Orca`, also I/O cost. As mentioned before, `Orca`'s main flaw is its propensity to perform a large amount of dataset traversals during detection (Section 3.6). The experiments performed confirm this assertion. While all other algorithms performed only two traversals for all n values, `Orca`'s number of dataset passes, and thus I/O time, increased sharply with n (Figure 5.2b), e.g., over 1000 passes for `2MASS` and $n = 1500$. Consequently, it was I/O bound in all test runs and its detection time surpassed the 72 hour limit for very small n values in both datasets.

These results highlight the acceleration GPUs can provide to outlier detection, but the performance gains shown are even more impressive when considering the per-



(a) Speedup achieved by `DROIDg` over `Dolphin`

(b) Number of dataset traversals performed by `Orca`

Figure 5.2: Summary of results of the n variation test.

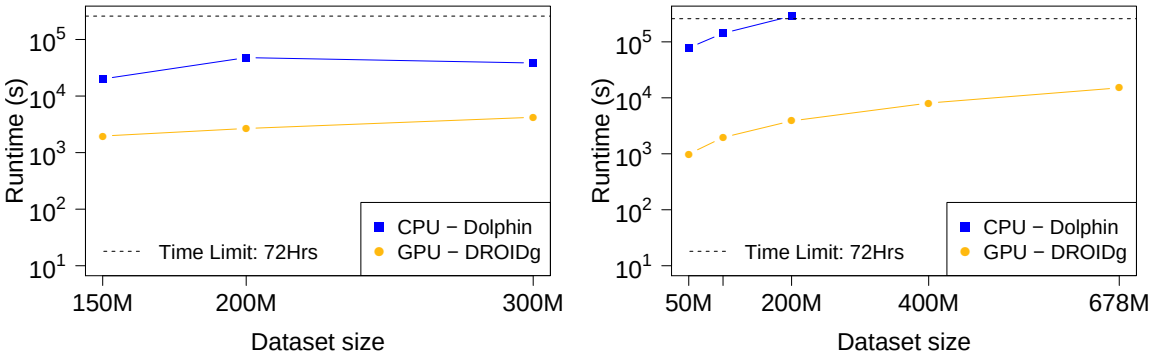
centage of outliers detected. The values of n used represent a tiny percentage of the total number of points in the dataset samples processed, and these samples were themselves a small fraction of the size of their respective source datasets (50% and 7%). Considering `DROIDg`'s superior scalability, had the full datasets being used, the performance delta between CPU and GPU algorithms would have been even larger.

5.3.1.1 `DROIDc`'s performance

As fig. 5.1 showed, `DROIDc` was the slowest algorithm tested. In order to make `DROIDg` both I/O efficient and extremely parallel, certain trade-offs were made, namely batching (NCB and TPBs) and deferring probable outlier candidate classification (set $\mathcal{Q}$ Section 4.6.2). Both of these optimization methods have the downside of increasing significantly the amount of distance computations that have to be performed during outlier detection. For `DROIDg` the trade-off of more computation for better /io efficiency and scalability more than pays off. But, for `DROIDc` that is not the case. Without the GPU's superior parallelism, it incurs the downsides of the optimization methods with none of their benefits. In summary, the optimizations employed in `DROIDg` make sense only for algorithms running on GPUs.

5.3.2 Dataset size (N)

This experiment tests the algorithms' scalability on the size of the dataset processed, thus all dataset samples created were used (Tables 5.2 and 5.3). The choice of k value was limited by the D_M^k estimation method. Since the same k was used for all dataset sizes, it had to be chosen such that even for the largest dataset the sample for D_M^k estimation would fit into the GPU's memory (Section 3.5.2). k was set to 12 for the



(a) Runtime for `2MASS`, $k = 12$, $n = 15$K (b) Runtime for `WISE`, $k = 80$, $n = 5$K

Figure 5.3: Runtime, in seconds, for the dataset size variation test, with the y-axis in log scale. Neither `Orca`, `Diskaware` nor `DROIDc` completed any test runs within 72 hours.

`2MASS` samples and to 80 for the `WISE` samples. Similarly, the value of n had to be chosen carefully due to its large impact on the detection time. For the `2MASS` datasets, n was set to 15K. But for `WISE`, n had to be set to just $5K$, otherwise not even `Dolphin` would have been able to process, within the time limit, samples larger than 50M points. In other words, n was set to such a small value, 0.01% of the **smallest** sample, to allow a performance comparison to be made against `DROIDg`.

The only CPU algorithm able to perform the tests within the time limit was `Dolphin`. For the `2MASS` datasets, `DROIDg` achieved a 9X speedup for the 300M sample. But it showed a slightly worse scalability, with its detection time increasing 2.1X when increasing the number of points from 150M to 300M, versus a 1.9X increase for `Dolphin`. Similar to the n variation test, `DROIDg`'s performance improvement was considerably better for the `WISE` datasets, likely for the same reasons. Since `Dolphin` could not finish detection for the two largest samples, `DROIDg`'s maximum speedup was 79X for 200M points. The scalability of both algorithms was similar with detection times increasing around 4X when increasing the dataset size from 50M to 200M. performance improvements were achieved for n smaller than 0.01% of the dataset size. Had we used larger and more meaningful values, the speedup would be between two and three orders of magnitude. This perfectly depicts how GPUs can enable outlier detection to be performed at a scale that is far beyond the capability of even state-of-the-art CPU algorithms.

5.3.3 Number of neighbors (k)

For this experiment we varied the parameter k from 8 to 128. This range was chosen because it spans the values most commonly used by the outlier detection research in the literature. The other two important parameters, i.e. dataset size and n, were set as to allow at least `Dolphin` to complete the experiments. Thus, the smallest sample of each dataset was used and n was set to just 0.01% of the dataset size.

`DROIDg` was, again, up to two orders of magnitude faster than `Dolphin` but the latter showed slightly better scalability. That is because `Dolphin` leveraged its index to perform less neighbor comparisons during the KNN search, whereas `DROIDg` used a brute-force approach, which is better suited for the GPU. Therefore, as k increased, the cost of the KNN search increased faster for `DROIDg`, thus decreasing its speedup (Figure 5.4c). For `2MASS`, the speedup decreased 45% from a maximum of 11X for $k = 8$, to 6X for $k = 128$. For `WISE`, the results were substantially better. The speedup decrease was only 16%, from a maximum of 87X for $k = 8$, to 73X for $k = 128$.

This experiment showed a small downside of outlier detection algorithms for GPUs in general: the lack of an indexed KNN search hurts their scalability with regards to k. However, this is a minor issue. First, because the superior GPU computation throughput offsets some of the extra cost. Second, `DROIDg` was still significantly faster for the range of k values commonly used.

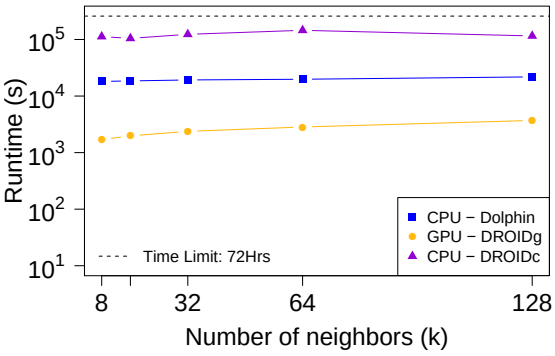
5.3.4 Summary: CPU vs GPU algorithms

These experiments perfectly highlight how GPUs can push the boundaries of the kind of outlier detection analysis that can be made in a reasonable time. `DROIDg` was up to two orders of magnitude faster, despite the experiments being conducted in the best case scenario for the CPUs:

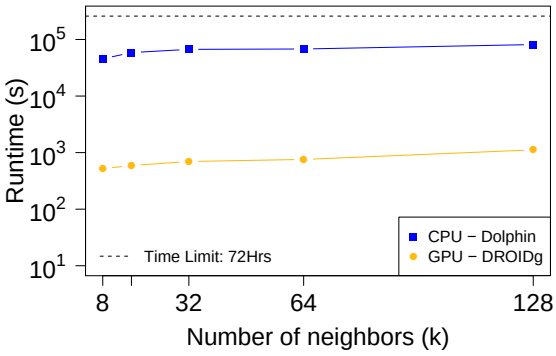
1. Only small n values were used: up to 0.04% of the smallest dataset sample
2. More importantly, all the tests were run using excellent initial D_M^k values: more than 90% of their optimal value (Section 5.5).

If any of the points above were not true, the computational cost of the detection would be considerably higher. In this scenario, the `DROIDg`'s speedup would be even larger because the GPU would be able to scale better than the CPUs, due to its superior computation throughput.

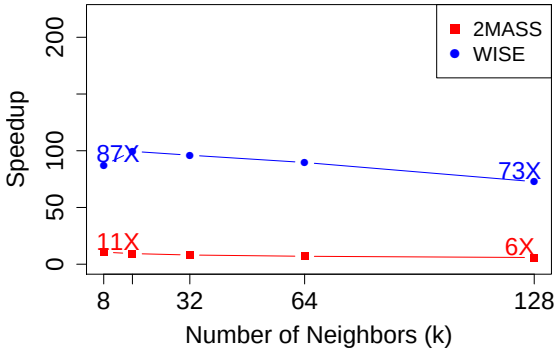
Given the magnitude of the speedups achieved, using GPUs is also considerably more cost effective for large scale outlier detection. Assuming `Dolphin` had a perfect



(a) Runtime for 150M `2MASS`, $n = 15$K



(b) Runtime for 50M `WISE`, $n = 5$K



(c) `DROIDg` speedup

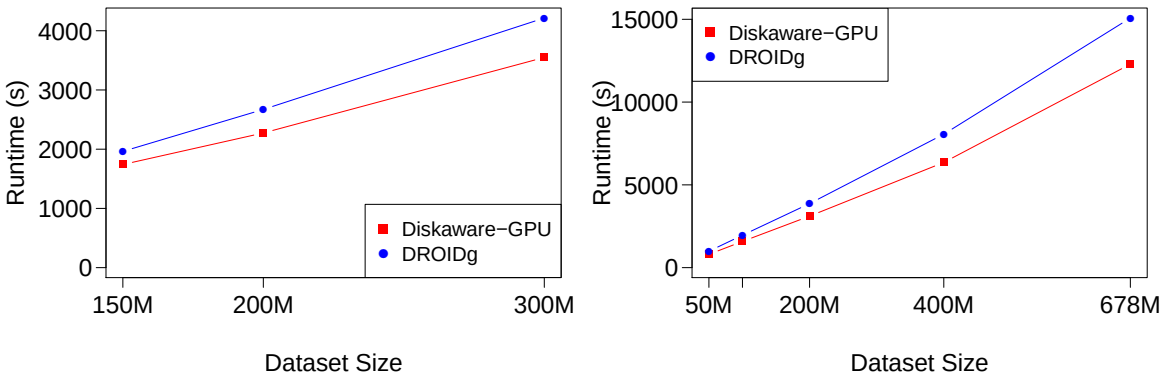
Figure 5.4: Results for the scalability test of the parameter k. The figures in the first row show the runtime, in seconds, of the algorithms with the y-axis in log scale. The third figure shows the speedup achieved by `DROIDg` in the experiment. Neither `Orca` nor `Diskaware` completed any test runs within 72 hours.

scaling, which is highly unlikely, it would need up to 137 threads to the beat `DROIDg`'s best results. Moreover, assuming the best thread/$ CPU on the market was being used, the `Ryzen` 1700 CPU; it would take 9 CPUs to achieve 137 threads. In processors alone, it would cost $2700 (Amazon [2017]), more than seven times the cost of the GTX 980 GPU used in the experiments. In summary, using GPUs for outlier detection is not only considerably more cost effective, but it also enables the detection to be performed at scales far beyond the capabilities of even state-of-the-art CPU algorithms.

5.4 GPU algorithm analysis

Now that we have shown how much potential the GPU has for accelerating the outlier detection process, we will compare two of the out-of-core algorithms for GPU that we implemented: `Diskaware-GPU` and `DROIDg`. `Orca-GPU` was not be included in the

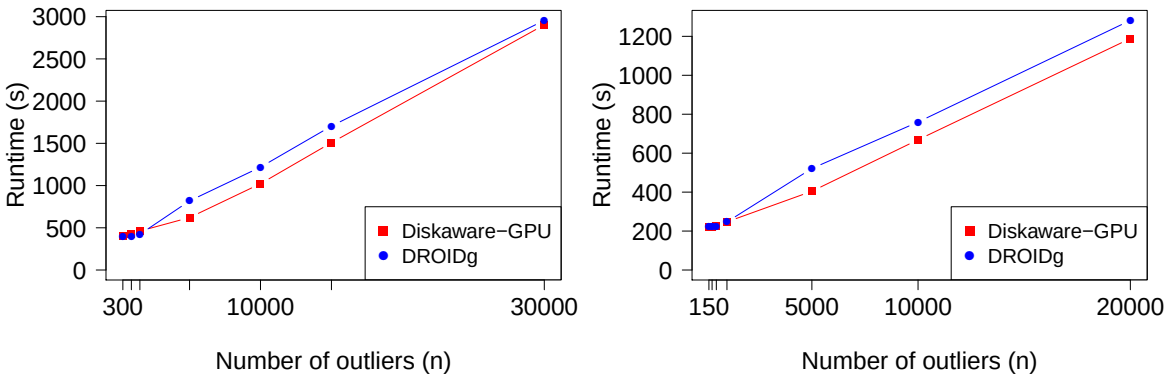
analysis because it is not be able to compete with the other algorithms. It is based on `Orca` and, as the previous section show, it is heavily I/O bottlenecked.



(a) Runtime for `2MASS`, $k = 12$, $n = 15$K (b) Runtime for `WISE`, $k = 80$, $n = 5$K

Figure 5.5: GPU algorithms' scalability on dataset size

For the first set of tests (Figure 5.5), we evaluated the algorithm's scalability on the dataset size. We ran the detection on the same samples of `2MASS` and `WISE` used in Section 5.3.2. `Diskaware-GPU` and `DROIDg` scale linearly with the dataset size and also have similar detection times, with `Diskaware-GPU` being less than 20% faster. For the dataset `WISE` (Figure 5.5b), the results were similar. Therefore, despite the changes implemented in `DROIDg` being designed to improve performance under a poor initial threshold scenario, they also work well when a good threshold is available.



(a) Runtime for 150M `2MASS`, $k = 12$ (b) Runtime for 50M `WISE`, $k = 80$

Figure 5.6: GPU algorithms' scalability on the number of outliers to be detected

In the second set of tests (Figure 5.6) we evaluate the algorithms' scalability on the number of outliers to be detected. We used the same dataset samples and n values as in Section 5.3.1. The results are similar to the ones in the previous test: `Diskaware` is less than 10% faster than `DROIDg` for all n values.

	Estimate %
900	95.4
1500	96.0
5000	96.3
10000	97.6
15000	97.8

Table 5.4: 150M `2MASS`

	Estimate %
500	99.2
1000	99.0
5000	98.7
10000	98.3
20000	98.4

Table 5.5: 50M `WISE`

Table 5.6: Quality of the D_M^k estimates used for each test run of the n variation test of both datasets. The column on the right shows which % of D_*^k these estimates are.

`Diskaware-GPU` and `DROIDg` performed so similarly because the initial thresholds used were close to their optimal value, as Table 5.6 shows. However, as discussed in Section 3.5.2, it is not always possible to get estimates this good. For detections in large scale datasets using small k, the samples required by `Dolphin`'s estimation method are too big. For instance, we had to use $k > 62$ in tests that included the `WISE` 678M dataset in order for the estimation sample to fit in the GPU's memory. Therefore, the next section will evaluate the algorithms in scenarios where `Dolphin`'s estimation method can not be used and, thus, the D_M^k quality is far inferior.

5.5 Poor initial D_M^k

When faced with a situation where the sample required for D_M^k estimation is too large (small k and large N), the user has three alternatives. It could use `Dolphin`'s estimation method anyway but limit the sample size to the maximum that fits the GPU's memory. Alternatively, it could use a different estimation approach which does not have the same sample size and k limitation issues; but which will certainly offer worse quality estimates. Lastly, it could perform the detection using an initial D_M^k of 0, if the algorithm supports it. Next, we will analyze each one of these scenarios.

5.5.1 Alternative 1 - Using `Dolphin` estimation method with limited sample size

Using a sample size smaller than η is not advised. It will produce an overestimate, resulting in an outlier recall rate smaller than 1. Let M be the amount of memory available to the GPU and consider that each point has d attributes of type `float`. The

maximum sample size that can be used is given by:

$$\eta_{max} = \frac{M}{4 \cdot d} \tag{5.1}$$

When using a sample of size $\eta_{max} < \eta$, the necessary k_s to perform the estimation is smaller than one. From the k_s definition (Section 3.5.2):

$$\begin{aligned} k_s &= \varrho \cdot \eta_{max} \quad \text{and} \quad \varrho \cdot \eta = 1 \\ k_s &= \varrho \cdot \eta_{max} < \varrho \cdot \eta \\ k_s &= \varrho \cdot \eta_{max} < 1 \\ k_s &< 1 \end{aligned} \tag{5.2}$$

Since it is impossible to use $k_s < 1$, it has to be fixed at 1. Consequently, σ^S of the points in the sample will not be decreased enough by the k_s "correction method" (Section 3.5.2). Considering p the top-n_s anomaly in the sample, it will be very likely that $\sigma_p^S > D_*^k$. By Definition 9, less than n outliers will be detected.

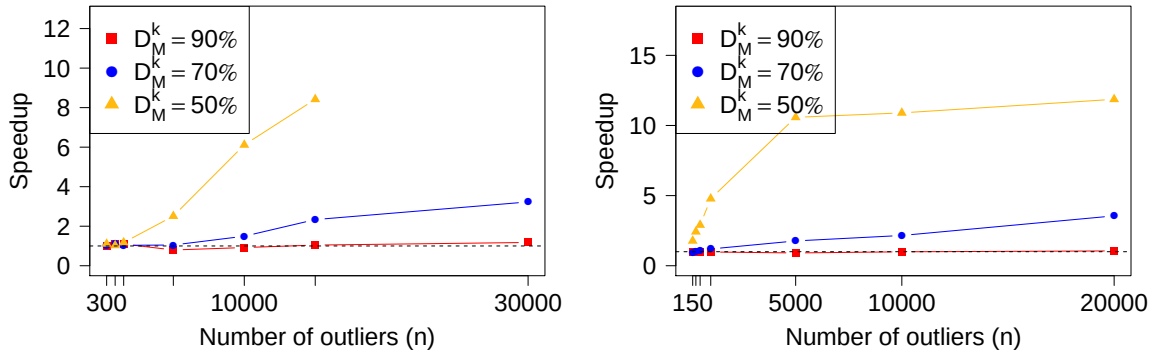
To illustrate the outlier recall issue, we re-ran the dataset size variation test for `WISE`, with $k = 8$ and using estimation samples of at most η_{max} points. For the `WISE` dataset, $\eta_{max} \approx$ 9M. As Table 5.7 shows, for all datasets where $\eta > \eta_{max}$ both the k_s and, consequently, the recall are smaller than 1. Additionally, the recall is directly proportional to k_s. Such trend agrees with our analysis: the closer k_s is to 0, the greater the overestimation of D_M^k. Therefore, the approach of using the estimation method with samples smaller than η is inadequate, since it is incapable of finding the desired percentage of outliers.

	η	k_s	$\bar{\rho}$
50M	7M	1.3	1.00
100M	14M	0.7	0.77
200M	29M	0.3	0.48
400M	57M	0.2	0.23

Table 5.7: Recall of outliers for `WISE` N variation test with $k = 8$ and $n =$ 5K. The estimates used for detection were generated using $k_s = 1$ and the maximum sample possible: $\eta_{max} \approx 9M$. η gives the required sample size; k_s is the value obtained by using η_{max}; $\bar{\rho}$ is outlier recall rate.

5.5.2 Alternative 2 - Using different estimation methods

The preferred approach is to use a different method for estimating D_M^k. Since, `Dolphin`'s method uses σ^S to increase the estimate as much as possible, it is certain that any other method will produce worst estimates. Under this scenario, algorithms that are robust to the quality of the initial D_M^k are more desirable. Therefore, in the following experiments we analyze their robustness to this parameter.



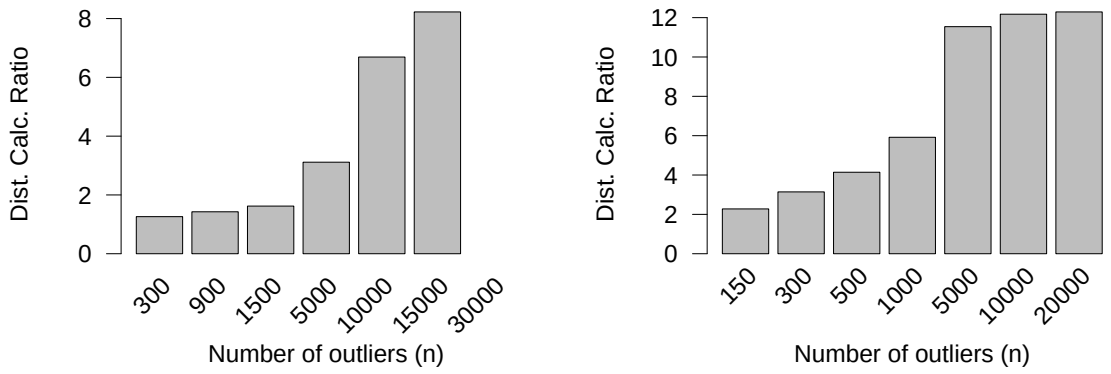
(a) 150M `2MASS` (b) 50M `WISE`

Figure 5.7: `DROIDg` speedup over `Diskaware-GPU`

We re-ran the scalability test for parameter n three times with progressively worse thresholds. For each value of n, the initial threshold will be set to 90%, 70% and 50% of D_*^k. Figure 5.7 shows that `DROIDg` is more robust to poor initial thresholds than `Diskaware-GPU`. For $D_M^k < 70\%$ of the D_*^k, `Diskaware-GPU`'s performance suffers considerably in both datasets. For instance, in the `WISE` dataset `DROIDg` was up to 12X faster with $D_M^k = 50\%$ and $n = 20$K. Moreover, in the `2MASS` dataset, it was up to 8X faster for $n = 15$K. It should be noted however, that the trend in Figure 5.7a suggests that `DROIDg` would be even faster for $n = 30$K. However, this could not be confirmed because `Diskaware-GPU` was not able to finish the detection for such value of n. It saved too many candidates in Q (more than $1.6M$) and ran out of memory.

The fundamental reason for `DROIDg`'s much faster detection times is that it performs considerably less distance computations than `Diskaware-GPU`. For instance, in the `2MASS` dataset with $D_M^k = 50\%$ and $n = 15$K (Figure 5.8a), it performed 8.2X less computation. Whereas for $n = 20$K in the `WISE` dataset `DROIDg` performed 12X less computation. It is interesting to note the correlation between the distance computation ratio and the speedup achieved by `DROIDg`.

In Section 5.6 we will perform a thorough analysis of both phases of `DROIDg`, to assess how each improvement proposed and implemented contributes to the algorithm's



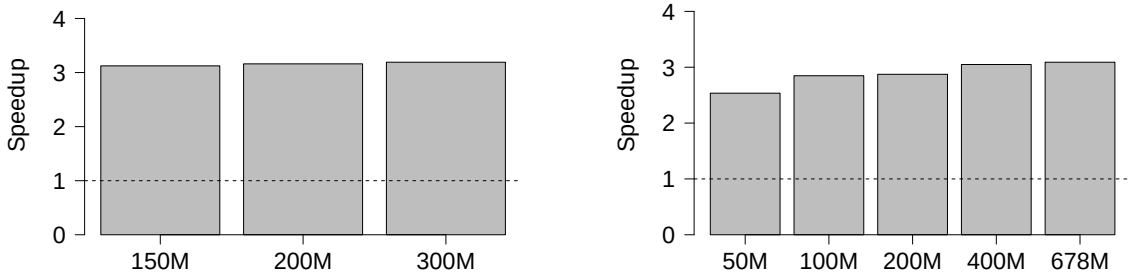
(a) 150M `2MASS`, $D_M^k = 50\%$ (b) 50M `WISE`, $D_M^k = 50\%$

Figure 5.8: Ratio between the number of distance-pairs computed by `DROIDg` and `Diskaware-GPU`

robustness to poor initial thresholds.

5.5.3 Alternative 3 - Using no initial threshold

The third alternative is to start the detection with $D_M^k = 0$ but, in this scenario, `DROIDg` is the only algorithm the can perform the detection reasonably fast. `Diskaware-GPU` is not capable of improving the D_M^k, thus it would use $D_M^k = 0$ throughout the whole detection. Its second phase would be equivalent to a brute-force detection, since all points would be considered outlier candidates. Additionally, its first phase would perform half as much computation as the second phase. In other words, `Diskaware-GPU` would be too slow for any reasonably large dataset. Therefore, in this section will focus in testing just `DROIDg`'s performance.



(a) `2MASS`, $k = 12$, $n = 15$K (b) `WISE`, $k = 80$, $n = 5$K

Figure 5.9: Speedup achieved by using a D_M^k produced by `Dolphin`'s estimation method versus starting the detection with a threshold of 0

We re-ran the dataset size variation test for both 2MASS and WISE datasets, and compared DROIDg's performance with and without an initial D_M^k value. Figure 5.9 shows that the performance decrease is relatively indifferent to $\mathcal{D}$ size. For the 2MASS dataset, DROIDg was about 3X slower when using no initial threshold, whereas the slowdown was about 10X for WISE dataset. For WISE, the slowdown for the 50M was considerably smaller than for the other ones, probably due to the small dataset size. Despite the substantial performance penalty, the longest detection, WISE 400M, only took 6.5 hours. For comparison, Dolphin, despite using an estimate close to its optimal value, was not able to find $n = 5$K outliers in the 50M dataset within the 72 hours time limit. This experiment shows that DROIDg can offer excellent performance even without any initial threshold.

5.5.4 Summary: robustness towards D_M^k quality

As Section 3.5.2 shows, there is a sizable range of detection parameters for which Dolphin's estimation method can not be used. The larger the dataset to be processed, the larger k_{min} is and, eventually, it would be completely outside of the range of most commonly used k values. This is a real issue for our usage context, because we specifically deal with large scale datasets. There are two viable alternatives to increasing k, but both require the detection algorithms to use initial D_M^k values much worse than those provided by Dolphin's method. Hence, there is a real need for algorithms that can offer good performance regardless of the quality of the initial D_M^k used, such as DROIDg. As the experiments in this section showed, under this common scenario, DROIDg was up to 12X faster than Diskaware-GPU. Furthermore, it was even able to perform the detection reasonably fast while starting with $D_M^k = 0$.

5.6 Analysis of DROIDg's performance

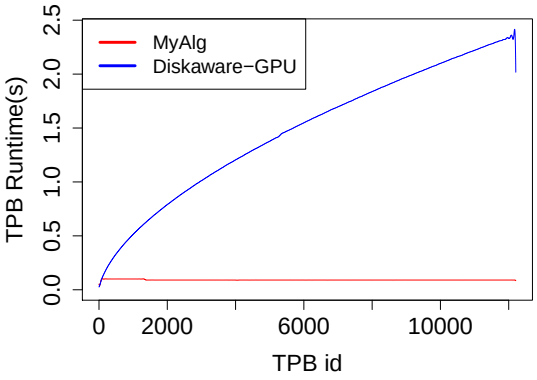
In this section we will perform a more in-depth analysis of DROIDg's performance when a bad initial D_M^k is used. The goal is to assess the impact of each individual improvement implemented in the algorithm and how they contribute to DROIDg's robustness to poor initial thresholds. The analysis will be made using the data from the n variation test for the WISE 50M sample, with $D_M^k = 50\%$, whose results were shown in Figures 5.7 and 5.8.

	Avg. Cand. Save	Avg. Cand. Pruning	$\overline{Q_\Delta}$
DROIDg- Initial	933	337	596
DROIDg- After Trav. 1	561	407	154
DROIDg- After Trav. 2	**350**	320	**30**
Diskaware-GPU	2751	**2645**	106

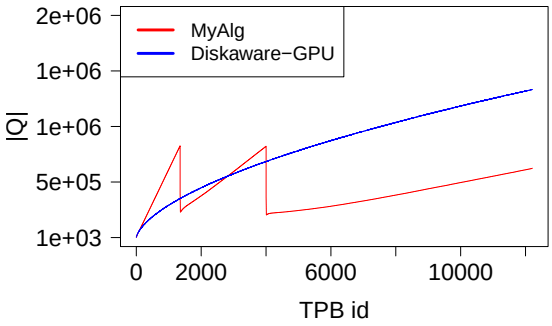
Table 5.8: Average number of candidates saved and pruned per TPB for DROIDg and Diskaware-GPU algorithms. For DROIDg, the averages are shown for each D_M^k value used during the detection. The last column shows the average increase in Q per TPB.

5.6.1 DROIDg- Phase 1

The two improvements made to DROIDg's first phase were: (i) new KNN search that scales better with $|Q|$ and (ii) D_M^k improvement. We will analyze their impact individually.



(a) Growth of the outlier candidate buffer Q during detection. 50M WISE, $D_M^k = 50\%$



(b) Growth of the outlier candidate buffer Q during detection. 50M WISE, $D_M^k = 50\%$

Figure 5.10: Ratio between the number of distance-pairs computed by DROIDg and Diskaware-GPU

Figure 5.10a shows how the cost of processing TPBs evolves during the detection, as Q increases. We can clearly see a substantial increase in processing time for Diskaware-GPU, 92X, whereas DROIDg's TPB's processing time increases mildly, 2.5X. This massive improvement is due to DROIDg's KNN search limiting the amount of outlier candidates used as neighbor candidates per search. However, this optimization has two side-effects: (i) more test instances saved as candidates per TPB; and (ii) less candidates pruned per TPB processed. The first issue is addressed by DROIDg's KNN search second stage. But the second issue is so severe that Q grows rapidly and becomes full during the detection. Table 5.8 shows that, before any traversals, DROIDg saved on average 933 new candidates per TPB, whereas Diskaware-GPU saved 2751 new candi-

	Pts. Proc	$\|Q\|$	$\Delta_{\|Q\|}$	D_M^k	D_M^k'	t (s)	t_P (s)
Pass 1	40960	809681	370892	3.39	4.26	458	4576
Pass 2	16384	805736	423120	4.26	5.16	188	4870

Table 5.9: Summary of the traversals made by `DROIDg` during phase 1. *Pts. Proc* is the number of candidates classified in the traversal; $|Q|$ is the size of Q; $\Delta_{|Q|}$ is the number of points pruned using the new D_M^k; D_M^k' is the new value of the threshold, after the traversal; t is the traversal duration; and t_P is the projected duration of the traversal if the points pruned were actually classified.

dates. This 3X reduction is due to the ROCN optimization in the KNN search second stage. However, it was not enough to offset the even higher decrease in the average amount of candidates pruned per TPB: from 2645 for `Diskaware-GPU`, to only 337 for `DROIDg`. This lead to, on average, 600 new outlier candidates per TPB processed, causing Q to become full twice during `DROIDg`'s first phase, which is represented by the spikes in Figure 5.10b.

Outlier candidate classification method

To address the issue of Q becoming full, `DROIDg` uses the outlier candidate classification method (Section 4.6.2) to both: (i) free space in Q, to allow the detection to proceed and (ii) improve the D_M^k, to improve the ANNS and slowdown Q's growth. Q became full twice during the first phase, so two traversals were performed, leading to a 52% increase in the D_M^k (Table 5.9). As a result, the ANNS efficiency increased dramatically, reducing the average amount of candidates saved per TPB to 350 and Q's growth to just 30 points per TPB (Table 5.8). In other words, `DROIDg`'s Q buffer growth rate became 3X smaller than `Diskaware-GPU`'s after the second traversal. The slower growth rate of Q after each traversal can be seen in Figure 5.10b as the smaller line slopes after each spike.

The outlier candidate classification method was also able to, cost effectively, free space in Q. The first traversal took 458 seconds to classify 40K candidates[3]. But, due to the D_M^k improvement, an additional 370K candidates were pruned, thus halving the number of points in Q (Table 5.9). If `DROIDg` were to classify all these 410K candidates, it would have taken over 4500 seconds[4]. The second traversal performed even better.

[3]The traversal's purpose is to improve the D_M^k. So, the first traversal needs to process enough candidates such that at the end, there are more than n candidates being considered outliers, so that the D_M^k can be updated

[4]Assuming that the cost of classifying the pruned outlier candidates would be the same as the cost of classification of the 40K points

	Pts. Proc	$\|Q\|$	$\Delta_{\|Q\|}$	D_M^k	D_M^k'	t (s)	t_P (s)
Pass 1	16384	611913	372636	5.16	6.19	189	7058
Pass 2	222893	222893	0	6.19	6.77	716	716

Table 5.10: Summary of the traversals made by DROIDg during phase 2

	Pts. Proc	$\|Q\|$	$\Delta_{\|Q\|}$	D_M^k	D_M^k'	t (s)	t_P (s)
Pass 1	1305414	1305414	0	3.39	3.39	14134	14134

Table 5.11: Summary of the single traversal made by Diskaware-GPU's second phase

The algorithm took just 188 seconds to both improve the D_M^k over 20% and discard more than 420K outlier candidates.

5.6.2 DROIDg- Phase 2

The outlier candidate pruning method is equally as important during the second phase, for it allows the classification of a large volume of outlier candidates to be done efficiently. For instance, Table 5.10 shows that at the beginning of the second phase, there were 600K outlier candidates saved. Regularly[5] classifying all of them, even with the improved D_M^k, would have taken almost 8K seconds[6]. But, by using the outlier candidate pruning method we proposed, they were all classified in less than 1K seconds (Table 5.10). The classification of these points was done in two dataset passes. In the first, the algorithm classified 16K candidates, improving the D_M^k by 20%. With the higher threshold, it was able to prune 64% of the candidates in Q. In the second pass, due to the higher D_M^k, the remaining 200K candidates were classified rather quickly, in just 716 seconds.

For sake of comparison, Table 5.11 summarizes Diskaware-GPU's second phase for the same experiment. It took over 14K seconds to classify 1.3M outlier candidates concurrently, using the original D_M^k. DROIDg on the other hand, when considering both of its phases, classified more than 1.46M candidates in just 1500 seconds; 9X faster than Diskaware-GPU.

[5] One full KNN search per candidate

[6] Sum of the projected times, column t_P, in Table 5.10

Chapter 6

Conclusion

Outlier detection is an important data mining task, with a wide-range of practical applications. There are several challenges associate with this task, but the main one is its computational cost. A significant amount of research was done on how to accelerate the detection process, thus current state-of-the-art methods are able to perform it in near-linear time on the average case. However, these improvements are still not enough to allow large scale datasets to be processed in reasonable time. Therefore, to finally address the computational cost issue, the goal of this work was to develop a new outlier detection algorithm for GPUs to accelerate the processing of terabyte-scale, disk-resident, datasets. But, to allow the GPU to efficiently process datasets in disk, we had to overcome two challenges: (i) the extra latency of transferring data to the GPU's memory; and (ii) the much more severe I/O bottleneck penalty incurred by GPUs.

In this work we developed a set of algorithms, kernels and abstractions, designed for out-of-core distance-based outlier detection, that allowed us to overcome the aforementioned challenges. Among our contributions there is a new parallelization strategy for anomaly detection algorithms; high-performance algorithms and GPU kernels for essential operations required by distance-based methods; and a new I/O sub-system that greatly reduces data transfers' overhead and their impact on the GPU's computation throughput. By leveraging all of these separate contributions, we were able to develop a novel outlier detection algorithm for GPUs, with two key advantages over the existing methods: (i) It is capable of processing disk-resident datasets; and (ii) It is a robust algorithm, with good performance regardless of the quality of the initial D_M^k used. It is based on `Diskaware` but it has two crucial improvements to achieve its robust performance:

1. The KNN search cost was made independent of $|Q|$, without significantly increasing the amount of false positives among the candidates saved. This prevents the algorithm's performance from degrading when Q becomes large, e.g., when D_M^k is too low, a major issue for `Diskaware`.
2. An effective method to both improve the D_M^k and prune saved candidates. It relies on our novel ROCO heuristic that identifies and prioritizes the classification of candidates likely to have large anomaly scores. As a result, it improves the D_M^k much quicker, decreases the growth of Q and massively reduces the amount of work performed during the detection.

To better analyze the performance of both CPU and GPU algorithms, we divided the experimental analysis into two main parts. First, we compared the performance of `DROIDg` against three state-of-the-art outlier detection algorithms for CPUs and also against its sequential counterpart `DROIDc`. The goal was to assess by how much the use of GPUs could accelerate the outlier detection in large scale datasets. Our analysis used three scalability experiments, one for each of the main parameters of the outlier detection problem: dataset size (N), number of outliers to be detected (n) and number of neighbors to be considered (k). The main results of these experiments were:

- We showed that `DROIDg` was between one and two orders of magnitude faster than the best sequential algorithm tested, `Dolphin`. These results were achieved despite the experiment conditions representing the best case scenario for CPU algorithms: relatively small N and n values were used; and, more importantly, the initial D_M^k were above 90% of their optimal value.
- The experiments also showed that `DROIDg` had far superior scalability for both N and n parameters, specially when processing the, high dimensionality, `WISE` samples. This indicates that if larger datasets were used and/or more of the top-n outliers were requested, the performance advantage provided by the GPU would be even larger. Therefore, we concluded that GPUs allow GPUs the outlier detection to be performed at scales far beyond of what current state-of-the-art algorithms for CPUs are capable of.
- We showed that the optimization methods employed in `DROIDg` were specifically chosen for GPU execution and actually degrade performance when used on sequential algorithms, such as `DROIDc`
- We also showed that using GPUs is also considerably more cost-effective. Assuming `Dolphin` can achieve perfect scaling, it would cost **at least** 7X the price

of the GPU used in the experiments, to buy enough processors to achieve the number of CPU threads needed to beat DROIDg's best results.

The second part of the experiments we compared two GPU algorithms, DROIDg and Diskaware-GPU, both implemented using our proposed framework. The goal was to determine how their performance was affected by the quality of the initial threshold used. The main results were:

- DROIDg showed similar performance to Diskaware-GPU when using initial thresholds close to their optimal value, despite not being optimized for such scenario. It was at most 20% slower than Diskaware-GPU for the N scalability test and less than 10% slower in the n scalability test.
- We showed that the range of outlier detection configurations for which Dolphin's estimation method can not be used, is not uncommon. Therefore, we concluded that there is a real need for algorithms capable of offering good performance, despite bad initial D_M^k values.
- When testing the algorithms' performance with bad thresholds, we used progressively worse D_M^k: 90%, 70% and 50% of their optimal value. DROIDg was up to 12X faster than Diskaware-GPU. We showed that this superior performance was due to two factors: (i) KNN search cost independent of the amount of candidates saved; and (ii) ability to improve the D_M^k during the detection with a few extra dataset passes, thereby massively decreasing the amount of computation performed.

6.1 Future Work

We believe there are at least three research ideas that extend this work and are worth pursuing. They are:

1. Choose a different algorithm for selecting the k smallest distances. Since this research stated and we chose the TBiS, at least two papers (Komarov et al. [2014]; Tang et al. [2015]) have been published proposing new and, possibly, more efficient algorithms for this task. Since, the sorting was the bottleneck for DROIDg in all the experiments, choosing a better algorithm and/or optimizing the k-Selection kernel could yield substantial performance improvements.
2. Despite the GPU's massive computation throughput, DROIDg was substantially faster when using a good initial D_M^k. Therefore, we believe it is worth while to

develop and estimation method that does not have the same sample size limitations that `Dolphin`'s estimation method has. This would allow even analysis in massive datasets with common k values, i.e., small, to be accelerated by a good initial threshold.

3. Given that a close to optimal D_M^k is available, the best optimization strategy to further improve the ANNS' pruning efficiency is to increase the convergence of D_p^k. This can be achieved by accelerating the KNN search through indexing structures, akin to the one used by `Dolphin`. We feel this is the most interesting extension of our work, because it not only would yield a significant performance improvements, but also because indexed KNN searches on GPUs have never been done before.

Bibliography

Aggarwal, C. C. and Yu, P. S. (2001). Outlier detection for high dimensional data. In *ACM Sigmod Record*, volume 30, pages 37--46. ACM.

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*, pages 104--110. ACM.

Amazon (2017). Ryzen 1700 cpu. `https://www.amazon.com/AMD-YD1700BBAEBOX-Processor-Wraith-Cooler/dp/B06WP5YCX6/ref=sr_1_1?s=pc&ie=UTF8&qid=1489081328&sr=1-1&keywords=1700`. Accessed: 2017-03-09.

Angiulli, F., Basta, S., Lodi, S., and Sartori, C. (2016). Gpu strategies for distance-based outlier detection. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3256–3268. ISSN 1045-9219.

Angiulli, F., Basta, S., and Pizzuti, C. (2006). Distance-based detection and prediction of outliers. *IEEE transactions on knowledge and data engineering*, 18(2):145--160.

Angiulli, F. and Fassetti, F. (2009). Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(1):4.

Augusteijn, M. and Folkert, B. (2002). Neural network classification and novelty detection. *International Journal of Remote Sensing*, 23(14):2891--2902.

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*, pages 29--38. ACM.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15.

Davy, M. and Godsill, S. (2002). Detection of abrupt spectral changes using support vector machines. an application to audio signal segmentation. In *ICASSP*, volume 2, pages 1313--1316.

Diaz, I. and Hollmén, J. (2002). Residual generation and visualization for understanding novel process conditions. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2070--2075. IEEE.

Ertöz, L., Steinbach, M., and Kumar, V. (2003). Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *SDM*, pages 47--58. SIAM.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226--231.

Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1--6. IEEE.

Ghoting, A., Parthasarathy, S., and Otey, M. E. (2008). Fast mining of distance-based outliers in high-dimensional datasets. *Data Mining and Knowledge Discovery*, 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.

He, Z., Xu, X., and Deng, S. (2003). Discovering cluster-based local outliers. *Pattern Recognition Letters*, 24(9):1641--1650.

Heller, K. A., Svore, K. M., Keromytis, A. D., and Stolfo, S. J. (2003). One class support vector machines for detecting anomalous windows registry accesses. In *Proc. of the workshop on Data Mining for Computer Security*, volume 9.

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

Khronos Group (2017). *The OpenCL Specification, Version: 2.2*.

Knorr, E. M. and Ng, R. T. (1999). Finding intensional knowledge of distance-based outliers. In *VLDB*, volume 99, pages 211--222.

Komarov, I., Dashti, A., and D'Souza, R. M. (2014). Fast k-nng construction with gpu-based quick multi-select.

Kruegel, C. and Vigna, G. (2003). Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251--261. ACM.

Mainzer, A., Bauer, J., Grav, T., Masiero, J., Cutri, R., Dailey, J., Eisenhardt, P., McMillan, R., Wright, E., Walker, R., et al. (2011). Preliminary results from neowise: An enhancement to the wide-field infrared survey explorer for solar system science. *The Astrophysical Journal*, 731(1):53.

Manevitz, L. M. and Yousef, M. (2001). One-class svms for document classification. *Journal of Machine Learning Research*, 2(Dec):139--154.

NVidia (2008). Chapter 39. parallel prefix sum (scan) with cuda. `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html`. Accessed: 2017-03-07.

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

Oster, B. (2008). Advanced cuda, optimizing to get 20x performance. `https://www.nvidia.com/content/cudazone/download/Advanced_CUDA_Training_NVISION08.pdf`. Accessed: 2017-03-07.

Pires, A. and Santos-Pereira, C. (2005). Using clustering and robust estimators to detect outliers in multivariate data. In *Proceedings of the international conference on robust statistics*.

Ramaswamy, S., Rastogi, R., and Shim, K. (2000). Efficient algorithms for mining outliers from large data sets. In *ACM SIGMOD Record*, volume 29, pages 427--438. ACM.

Roth, V. (2004). Outlier detection with one-class kernel fisher discriminants. In *Advances in Neural Information Processing Systems*, pages 1169--1176.

Seagate (2011). Seagate barracuda hard drive data sheet. https://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf.

Shekhar, S., Lu, C.-T., and Zhang, P. (2001). Detecting graph-based spatial outliers: algorithms and applications (a summary of results). In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 371--376. ACM.

Skrutskie, M., Cutri, R., Stiening, R., Weinberg, M., Schneider, S., Carpenter, J., Beichman, C., Capps, R., Chester, T., Elias, J., et al. (2006). The two micron all sky survey (2mass). *The Astronomical Journal*, 131(2):1163.

Tang, X., Huang, Z., Eyers, D., Mills, S., and Guo, M. (2015). Efficient selection algorithm for fast k-nn search on gpus. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 397--406. IEEE.

Worden, K., Manson, G., and Fieller, N. (2000). Damage detection using outlier analysis. *Journal of Sound and Vibration*, 229(3):647--667.

Yamanishi, K., Takeuchi, J.-i., Williams, G., and Milne, P. (2004). On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery*, 8(3):275--300. ISSN 1573-756X.

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. ISSN 1550-4786.

Yankov, D., Keogh, E., and Rebbapragada, U. (2008). Disk aware discord discovery: finding unusual time series in terabyte sized datasets. *Knowledge and Information Systems*, 17(2):241--262.

Appendix A

Implementation Details

In this appendix we will getting into much greater detail on the GPU-centric design decisions and algorithms discussed in Chapter 4 and showing how we implemented them using *OpenCL*. First, we present the data layout we used in the dataset and in the GPU memory and discuss why. Second, we show how to implement efficient distance computation using clBLAS to perform matrix multiplication and custom kernels to perform the rest of the necessary steps. Third, we discuss how we efficiently implemented k-selection part of the KNN search on the GPU by using sorting network to perform a partial sort rather than a full sort of the distance matrix rows. Fourth, we show how the pruning can be implemented to minimize thread divergence by using a primitive called *stream compaction*.

A.1 Point storage layout

Matrices can be stored in two different ways in memory: Row-major and Column-major. They differ on which dimension is stored in adjacent memory positions. While column-major matrices store their columns continuously in memory, row-major matrices store their rows in adjacent positions. In this work, all matrices and point batches will be row-major.

Now consider a set A of b points with d attributes. These points can also be stored in one of two ways in a matrix. Using a *row-wise* storage layout, each row of the matrix would contain a point. So, to store A, it would required a matrix with b rows and d columns. On the other hand, in a *column-wise* layout every point is stored in one column. A matrix that stores A would have d rows and b columns. Each layout has its advantages and the choice between them depends on the application requirements,

e.g. access pattern and hardware-related memory constraints. Next we explain which storage layout is used for different parts of the application.

Dataset storage layout The dataset is stored in disk using a row-wise layout. This choice is guided by the application's access pattern and by the fact that dataset must remain disk-resident throught the exection. Our application reads sequential bacthes of points from the dataset and a row-major layout translates these accesses into sequential reads from the disk. This is extremelly important to maximize the applicaion's performance, since the dataset is never loaded entirely into memory and sequential disk accesses provide the best read throughput.

Storage layout in point batches Because points are stored row-wise in the dataset, they are initially read into memory with the same layout. But, internally, the points are not necessarily used nor stored row-wise. While TPBs store points row-wise, NCBs store them column-wise. The different choice in layouts is determined by how frequent they need to be created and how they are used in distance computations. But, the storage of these batches also dictates how points are stored internally. Since a new NCB needs to be created for every KNN iteration, it is more effecient to store chunks of the dataset internally column-wise. Once a chunk of points is read from disk, the whole chunk is transposed once, which is more efficient than multiple small transpositions. To create a new NCB, the application simply needs to copy the desired points to a new buffer. To create a new TPB, it needs to transpose the requested points and then copy them. Since TPBs are not created as frequently, the penalty of tranposing the points is significanly less.

A.2 Distance computation

The algorithm used for computing the distance pairs is based on the work developed by Garcia et al. [2008]. The authors showed that given two batch of points B and NCB, all the distance pairs could be computed using matrix operations in the following manner:

$$\Delta = N_B + N_{NCB} - 2 \cdot B^T \cdot NCB \tag{A.1}$$

Considering that B and NCB have b and c points, respectively, N_B and N_{NCB} are both $b \times c$ matrices containing the norms from the points in B and NCB respectively. In N_B, all elements in row i contain the norm of the i-th point in B. Similarly, all elements

in column j of N_{NCB} contain the norm of the j-th point in NCB. The arrangement of these norm matrices mimicks the storage layout of the associated point batches.

As the authors show in their work, there are a few advantages to computing distances on GPUs using matrix operations. First, the problem in matricial form exposes more parallelism to the GPU, allowing better utilization of the hardware. In Garcia et al. [2008], this resulted in up to 4X performance improvement over competing methods. Second, a matricial problem formulation allows the use of highly optimized linear algebra libraries, such as clBLAS or CUBLAS for CUDA. These libraries have low-level, hardware-dependent, optimizations to maximize computation throughput.

A.2.1 Implementation

A.2.1.1 Matrix multiplication

To perform the matrix multiplication to compute $-2 \cdot B^T \cdot NCB$ we leverage the clBLAS library. We use the *clblasSgemm* function, passing B and NCB as the matrices containing the points and specify their memory storage as row-major. Note that though B appears transposed in Equation (A.1), it is already transposed because its points are stored row-wise. Therefore, clBLAS should not transpose eiher of the input matrices and the the actual multiplication performed is $-2 \cdot B \cdot NCB$.

A.2.1.2 Norm computation

The norm computation kernel implemented (Algorithm 14) receives as input a column-wise batch of points and outputs a vector ($vNorm$) with the points' norm. It uses a 1-d index space and each work-item is responsible for computing one norm. Each work-item will iterate d times in a for-loop (Line 5) to load its assigned point's attributes, square them, and acumulate the sum. Once the norm is computed, each work-item writes its norm to the output vector in global memory (Line 9).

The choice of a column-wise storage layout allows the implementation of simple kernel, that at the same time:

- Maximizes memory bandwidth utilization through coalesced global memory loads and memory writes. During the i-th iteration, the work-items will access adjacent memory positions to load the i-th row of the input batch matrix.
- Increases the level of parallelism, since there are as many work-items as points and the amount of points in a batch is usually larger than d. While it would also be possible to achieve a good amount of parallelism using a row-wise layout, the kernel would have to considerably more complex.

Algorithm 14: Kernel for computing the norm of points

```
1 Function ComputeNorm (B, ptNum, d)
2     gid ← get_global_id(0)
3     ptId ← gid
4     sum ← 0
5     for i = 0 to d do
6         attr ← pts [ptId · ptNum + i]
7         sum + = attr · attr
8     end
9     vNorm[gid] ← sum
```

The bandwidth optimization is specially important because this kernel is memory-bound. It only performs 1 flop[1] per global memory access, which is an extremelly low ratio for modern GPUs.

A.2.1.3 Norm Addition

The norm matrices N_B and N_{NCB} are mostly comprised of redundant elements and, as a result, waste too much memory. Considering TPBs and NCBs of size 4096 and 8192, these matrices would require 128MB each. This is a problem when there are multiple TPBs being processed concurrently, as it is the case with `DROIDg`. A better approach is to represent them as vectors and use special kernels to appropriately add the norms to matrix $-2 \cdot B \cdot NCB$.

From now on, we will consider N_B and N_{NCB} as vectors of size b and c respectively. The addition of each vector is done differently. N_{NCB} needs to be added to each row of matrix $-2 \cdot B \cdot NCB$, whereas N_B needs to be added to each column. However, adding these vectors with dedicated kernels is inneficient due to the low *arithmetic intensity*: only one *flop* per 3 global memory accesses. Consequently, the performance of the additions would be memory-bound. Instead `DROIDg` performs the additions together with other important operations. N_{NCB} is added by the sorting kernel, just before the sorting happens, since both operate on a row basis. Even though the actual distances were not computed yet (N_B has not being added), $N_{NCB} - 2 \cdot B \cdot NCB$ already has the same ordering as the actual distance matrix. N_B, on the other hand, is added to the sorted matrix by the kernel that computes the anomaly score. The fact that each matrix row is sorted is irrelevant, since all elements in a given row will be added the same norm.

[1] Floating-point operation

A.3 Sorting

Once the distance matrix[2] is computed, the next step in the KNN iteration is to: (i) find the distance between the test points in B and their k closest neighbors; (ii) place these distances, sorted, at the beginning of each row of Δ. The naive solution is to sort every row of the matrix in ascending order, but this is inneficient. Firstly, we are only interested in the k smallest values and sorting the whole row just wastes computation cycles. Secondly, the working set of the sorting kernel would be the whole row, too big to fit in shared memory. Consequently, for a significant portion of the kernel's execution the data would reside in global memory, thus reducing the kernel's arithmetic intensity and performance.

Our solution uses the *Truncated Bitonic Sort* (`TBiS`) algorithm to perform partial sorts in each row of Δ. This approach has some advantages. First, `TBiS` is based on the *Bitonic Sort* algorithm. Therefore, it is extremelly parallel and is able to use up to $n/2$ work-items to sort n elements. Additionally, each of its comparison and swap operations are data independent, thus they can be executed in parallel without causing branch divergence on the GPU. Second, `TBiS` uses truncation during the merge step of the `Bitonic Sort` to discard unnecessary parts of the partial solution. This has two benefits: (i) reduce the complexity of the algorithm, since it does not sort the whole input; (ii) keep the algorithm's working set small enough to allow the extensive use of the shared memory; it improves performance considerably by eliminating stalls caused by global memory accesses and increasing the PEs utilization.

The explanation of how `TBiS` is employed in our algorithm is divided into three parts. Fist, we explain how the `Bitonic Sort` algorithm works. Then, we explain `TBiS` iself. Finally, we discuss how `TBiS`' kernel was implemented.

A.3.1 `Bitonic Sort`

Before explaining the algorithm, we need to explain what *Bitonic sequences* and k-**Bitonic Networks** are. Bitonic sequences are sequences of numbers where each half is sorted in opposing directions, e.g. $(1, 3, 10, 0)$. Additionally, by definition, every pair of numbers is a bitonic sequence. k-Bitonic Networks are sorting networks that are only able to sort bitonic sequences. They receive a bitonic input of length k and output a sorted sequence of the same length.

To sort an n element input, the `Bitonic Sort` algorithm needs to first convert the input into a set of bitonic sequences of length 2. This can be done by simply viewing the

[2] As we saw before, the actual distances are not computed until the very end of the KNN iteration. However, we will refer to $\Delta = -2 \cdot B \cdot NCB$ as the distance matrix from here onwards

input as separate 2 element sequences, which by definition are bitonic. Then, through $\log_2 n$ stages, it uses k-Bitonic Networks to build larger bitonic sequences, until the entire input is sorted.

At stage 0, the input is comprised of $n/2$ bitonic sequences and each sequence is sorted by a 2-Bitonic Network. Adjacent networks will sort their inputs in opposing directions and the stage's output will be $n/2$ sorted sequences of length 2. Alternatively, one can view pairs of adjacent sequences together (opposing directions) and interpret the stage's output as $n/4$ bitonic sequences of length 4. In stage 1, $n/4$ 4-bitonic networks will be used, each receiving as input (and sorting) one of the bitonic sequences from the previous stage. As a result, this stage's output will be bitonic sequences of length 8, which are going to serve as inputs for the next stage. The last stage will be stage $i = \log_2(n) - 1$. It will receive one bitonic sequence of length $n = 2^{i+1}$ and will convert it into a sorted sequence, using just one n-bitonic network. The algorithm's time complexity is: $\mathcal{O}(n \log^2 n)$.

A.3.2 `TBiS`

`TBiS` has two phases. In the first, the algorithm splits input $\mathcal{I}$ into n/k segments of length k and uses `Bitonic Sort` to sort each segment independently and in parallel. Since the sorting is independent, the working set is just the segment being sorted, thus it has size k and is small enough to fit in shared memory. In the second phase (the "conquer phase"), the sorted k-element segments are merged pair-wise, using one $2k$-Bitonic network per segment pair. This is possible because adjacent segments are sorted in opposing directions and together they form a bitonic sequence. Therefore, the $2k$-Bitonic network will sort the segment pair, which is equivalent to merging them. After each merge, the resulting merged segment is truncated and only the k smallest elements are kept. Depending on the sorting order, these could be either the first or last k elements.

The truncation is essential for `TBiS`' performance. Without it, the size of the merged sequences would increase exponentially, doubling after each merge step. Consequently, the working set (merged sequences) size would rapidly become too large and the algorithm would have to resort to global memory. However, by using truncation after each merge, `TBiS` ensures that the working set size does not grow beyond $2k$ elements, allowing it to use shared memory during its entire execution.

The algorithm's time complexity can be derived as follows. The bitonic sort algorithm is used to sort each of segments, with a total complexity of $\mathcal{O}(n \log^2 k)$.

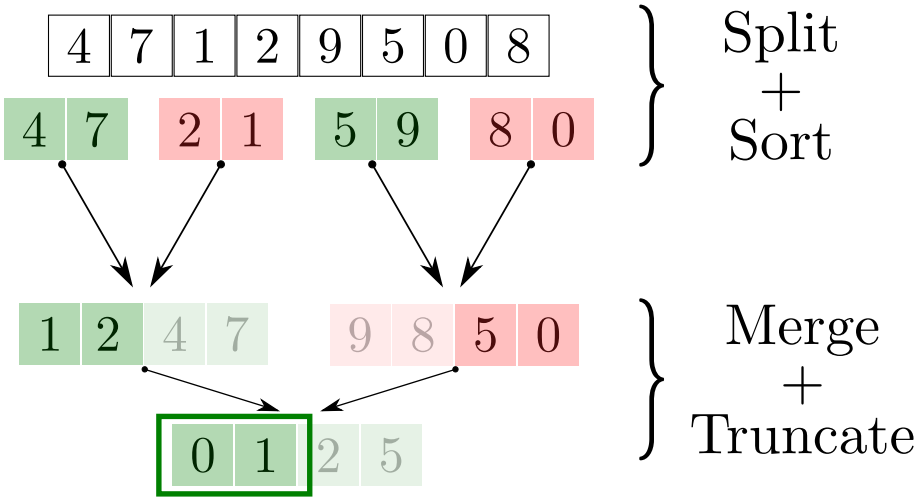


Figure A.1: `TBiS` splits the input into segments of length $k = 2$ and sorts them with `Bitonic Sort`. They are merged pair-wise and, after each merge, truncation is applied: the half with the largest elements is discarded.

Moreover, $\frac{n}{k} - 1$ merges [3] are performed during the merge phase, each using one $2k$-bitonic network, with a total complexity of $\mathcal{O}(n \log k)$. Therefore, the total complexity of the `TBiS` algorithm is $\mathcal{O}(n \log^2 k)$.

A.3.3 Implementation

The modified `TBiS` kernel (Algorithm 15) uses a 1-d index space. Work-groups have a fixed size of 1024 and one work-group is assigned to each row of the matrix.

The kernel receives as input three parameters: (i) the distance matrix Δ; (ii) N_{NCB}, containing the norm of the neighbor candidates used in the iteration; (iii) the amount of smallest elements to be selected from each row (k). The work-items start by loading the NCB norm (Line 6) that needs to be added to the elements of their assigned row (*rowId*). Then, each work-group will collectively load 2048 elements from their row into lbuff (Line 7), a buffer located in shared memory. Next, the work-groups will sort segments of $2 \cdot k$ elements, such that adjacent segments are sorted in inverse order (Line 9). Each sorted segment is truncated. The smallest k elements of each segment are written in the first half of lbuff and its second half is considered empty.

The next part of the kernel will process the elements from column 2048 ownwards. In each iteration of the for-loop (Line 11), 1024 elements are read from global memory, added to the row's specific NCB norm and written into the "empty" half of lbuff.

[3] The merge tree has $\log_2 \frac{n}{k}$ levels and level i has 2^i merges. So, the total number of merges is $\sum_{i=0}^{\log_2 (\frac{n}{k})-1} 2^i = \frac{n}{k} - 1$

Algorithm 15: `TBiS`

```
1  Function TBiSMat (Δ, N_NCB, k)
2      lid ← get_local_id(0)
3      locs ← get_local_size(0)
4      rowId ← get_group_id(0)
5      nextElem ← lid
       // Norm to be added to elements of this row
6      norm ← N_NCB [rowId]
       // Each work-item loads two distances into shared memory
7      lbuff [lid] ← Δ [rowId · width + nextElem] + norm
8      lbuff [lid + locs] = ← Δ [rowId · width + nextElem + locs] + norm
       // Sort segments of 2 · k elements
9      BitonicSort (lbuff, 0, 2 · k)
10     Truncate (lbuff, k)
11     for (; nextElem < width; nextElem += locs) do
12         lbuff [locs + lid] ← Δ [rowId · width + nextElem] + norm
13         BitonicSort (lbuff, locs, k)
14         BitonicMerge (lbuff, 0, 2 · k)
15         Truncate (lbuff, k)
16     end
17     for (segNum ← locs; segNum > 1; segNum »=1) do
18         BitonicMerge (lbuff, 2 · k)
19         Truncate (lbuff, k)
20     end
21     if lid < k then
22         Δ [rowId · width + lid] ← lbuff [lid]
23     end
```

Then, the newly loaded elements are sorted in segments of length k and all the bitonic sequences in lbuff are merged and truncated. This process is repeated until there are no more elements to be read.

The last part of the kernel performs the final selection of the k smallest elements of the row (Line 17). At this moment, lbuff has $1024/2 \cdot k$ bitonic sequences in its first half. With every iteration of the for-loop, the bitonic sequences are sorted and then truncated, until only the k smallest elements remain. Finally, these distances are written back to Δ in global memory.

A.4 Pruning

At the end of a KNN iteration, the anomaly score upper-bound of all test points in B have been updated and the `DROIDg` needs to apply the pruning operation. The pruning

needs to remove from B all the points proved to be inliers, i.e. whose score has fallen bellow D_M^k.

To implement the pruning we used an important primitive called *stream compaction* (NVidia [2008]). It allows filtering a heterogeneous buffer to produce a homogeneous one, containing just the desired elements. In our case these elements would be the ones not pruned. This primitive has three steps:

- **Map**: Maps which elements should be kept and which should be discarded. It generates a binary buffer A, such that $A[i] = 1$ indicates that the i-th input element was selected.
- **Scan**: Computes the output index of the selected elements. It generates a buffer I contaning the result of an exclusive scan on A. If the i-th element was selected, then $I[i]$ is its output index.
- **Scatter**: Uses A and I to scatters the selected input elements into their respective positions in the output buffer

For the pruning to be implemented properly, each buffer associated to the test points needs to be scattered. So, in a general way, our pruning algorithm works in the following manner. First, it performs the *map* and *scan* steps of the stream compaction. Second, it performs the scatter step once for every associated data: point attributes, ids, norm, score and `pKnn`.

A.4.1 Map

Map is the simplest of the three steps. It receives as input the buffer S containing the anomaly score of the test points and the D_M^k and applies the following predicate:

$$A[i] = \begin{cases} 1, & \text{If } S[i] < D_M^k \\ 0, & \text{otherwise} \end{cases} \tag{A.2}$$

The kernel will be launched using a 1-d index space with $|S|$ work-items, such that each work-item is responsible to apply Equation (A.2) to their respective score.

A.4.2 Scan

Once A is computed, the pruning algorithm computes the output index of the selected elements using an *exclusive* scan operation. Because the scan operation is memory intensive, to implement it reasonably efficiently on the GPU, the kernel(s) should use

Algorithm 16: Mapping inliers and non-inliers

```
1 Function PruningMap (S, D_M^k, A)
2     gid ← get_global_id(0)
3     res = (S[gid] < D_M^k) ? 0 : 1
4     A[gid] = res
```

as much of shared memory as possible. Therefore, our scan implementation is divided into three parts:

- **Local Scan** Performs the scan on segments of A using exclusively the shared memory. Each work-group loads two elements from A per work-item. Then an implementation of the `Blelloch` (ref) algorithm is used to perform the scan on shared memory. At the end, two outputs are generated. The results of the local scan are written to I. Additionally, work-group i writes to $T[i]$ the last element of the exclusive scan it performed.
- **Top-level Scan**: Performs an exclusive scan of T, using one work-group.
- **Uniform Update**: Obtains the final scan result of A. Add $T[i]$ to all the elements in the local scan result of work-group i. The, output is written to I.

It should be noted that this imposes a maximum size on the input buffer A of: $2 \cdot l^2$, where l is the work-group size. Considering the maximum size possible on current GPUs, 1024, the maximum size of A is over 2M elements. This upper-limit can be raised in two way. Top-level Scan and Uniform Update can be repeated recursively, or Top-level Scan can be made to use multiple work-groups and perform the scan on global memory. However, for the purpose of this work, the 2M elements size limit is enough.

A.4.2.1 The `Blelloch` algorithm

The `Blelloch` algorithm was chosen because of its great parallel efficiency. It has two phases: Reduction and Downsweep. The reduction phase performs a staggared parallel reduction of A with the sum operation, in $\log_2 |A|$ steps. At step i, only $|A|/2^{i+1}$ work-items are used and each work-item uses two operands with an offset $\Delta = 2^i$ positions. The operand indexes can be computed as follows:

$$lId = \Delta \cdot (2 \cdot lid + 1) - 1 \tag{A.3}$$

$$rId = \Delta \cdot (2 \cdot lid + 2) - 1 \tag{A.4}$$

Algorithm 17: The Reduction function of the `Blelloch` algorithm

```
1  Function Reduction (A, n, lbuff)
2      gid ← get_global_id(0)
3      lid ← get_local_id(0)
4      lbuff [2· lid] = A[2· gid]
5      lbuff [2· lid + 1] = A[2· gid + 1]
6      lval = lbuff [2· lid]
7      m ← n/2
8      for Δ = 1; Δ < n; Δ <<= 1 do
9          rId ← Δ · (2 · lid + 2) − 1
10         if lid < m then
11             res ← lVal + lbuff [rId]
12             lbuff [rId] = res
13         end
14         lVal = res
15         barrier()
           // Halve the # of active work-items
16         m >>= 1
17     end
```

The second phase, Down-sweep, is a mirror image of the Reduction, thus the number of steps is the same and the operand indexes can also be computed using Equation (A.4). There are two differences, however: (i) for step i, the offset is $\Delta = |A|/2^{i+1}$; (ii) the phase uses the sweep operator instead of the sum. This operator takes two inputs and produces two outputs. The left output is a copy of the right operand and should be written to the left operands position. The right output is the sum of the inputs and should be written to the right output's position.

A.4.2.2 The scan algorithm

To scan A, the three kernels shown in Algorithm 19 need to be called. `ScanLocal` is called using a 1-d index space, with $\frac{|A|}{2}$ work-items. We pass as input five parameters:

1. A: the binary map to be scanned. It is located in global memory
2. I: global memory buffer to store the result of the local scan. It must have the same size as A
3. T: global memory buffer to store the last element of each local scan result. Its length should be equal to the number of work-groups used.
4. n: number of elements to be scanned

Algorithm 18: The Down-Sweep function of the `Blelloch` algorithm

```
1  Function Downsweep (I, n, lbuff)
2      gid ← get_global_id(0)
3      lid ← get_local_id(0)
4      if lid == 0 then
5          lbuff [2· get_local_size(0) - 1] = 0
6      end
7      lVal ← lbuff [n/2 - 1]
8      for Δ = n/2; Δ >= 1; Δ >>= 1 do
9          lId ← Δ · (2 · lid + 1) − 1
10         rId ← Δ · (2 · lid + 2) − 1
11         if lid < Δ then
12             rVal ← lbuff [rId]
13             res ← lVal + rVal
14             lbuff [lId] ← rVal
15             lbuff [rId] ← res
16         end
17         lVal = rVal
18         barrier()
19     end
       // Write local scan result to I
20     I[2· gid] ← lbuff [2· lid]
21     I[2· gid + 1] ← lbuff [2· lid + 1]
```

5. lbuff: where the scan will be done. It resides in shared memory and has a size equal to twice the size of the work-groups used.

The other two kernels have roughly the same list of parameters and are also called using a 1-d index space. `TopLevelScan` is run using just one work-group with as many work-items as work-groups that were used to run `ScanLocal`. Finally, `UniformUpdate` is called with $|A|$ work-items.

A.4.3 Scatter

The scatter primitive is very simple. It uses a binary map (A) and its exclusive scan (I) to place the desired elements (E) at the beginning of the output buffer (O) and the undesired ones at the back. However, for our application, the undesired elements, i.e. pruned, are of no interest and thus are ignored.

For our use case, there needs to be two scatter implementations: (i) 1d scatter for the buffers storing the ids, norms and point scores; (ii) 2d scatter for the matrices `pKnn` and the points themselves. The 1d scatter is fairly simple. It is launched using a 1d index space, with $|A|$ work-items. If $A[i] == 1$, then work-item i copies the input element E[i] to its output position $O[I[i]]$. The 2d scatter works similarly but it also

Algorithm 19: The three kernels for performing the scan operation

```
1  Kernel ScanLocal (A, I, T, n, lbuff)
2      Reduction (A, n, lbuff)
3      Downsweep (I, n, lbuff)
       // Write the result of each work-group local scan
4      wgId ← get_group_id(0)
5      wgSize ← get_local_size(0)
6      T [wgId] ← lbuff [2· wgSize · (wgId + 1) - 1]

7  Kernel TopLevelScan (A, I, T, n, lbuff)
8      Reduction (T, n, lbuff)
9      Downsweep (T, n, lbuff)

10 Kernel UniformUpdate (I, T)
11     wgId ← get_group_id(0)
12     tSum ← T[wgId]
13     I [gid] ← I [gid] + tSum
```

needs to take into consideration the matrices' row-wise storage layout. It is launched using a 2d index space, with *width* work-items in the first dimension (x) and *height* = $|A|$ elements in the second dimenion (y). If $A[j] == 1$, then the j-th row of the input matrix will be copied to the output. All work-items with $gidy = j$ will copy their assigned element inside the row, indicated by $gidx = i$, and write it to its output position: $O[I[j] \cdot \text{width} + i]$.

Algorithm 20: Scatter Kernel

```
1  Kernel Scatter (A, I, E, O)
2      gid ← get_global_id(0)
3      outId ← I [gid]
4      if A [gid] == 1 then
           // Element selected
5          O [outId] = E [gid]
6      end

7  Kernel Scatter2D (A, I, E, O, width)
8      gidx ← get_global_id(0)
9      gidy ← get_global_id(1)
10     colId ← gidx
11     rowId ← I[gidy]
12     if A [gidy] == 1 then
           // Element selected
13         O [rowId * width + colId] = E [gidy * width + gidx]
14     end
```

A.4.4 The pruning algorithm

Using all the functions and kernels discussed, we implemented the pruning algorithm shown in Algorithm 21. It starts by creating the temporary buffers A and I, with size equal to the number of test points in B. Then, it builds the binary map A of the points not pruned and computes the output indexes of these points, I, by performing an exclusive scan of A. At this point, the host will perform an asynchronous copy of the number of points not pruned [4] and saving it to B. *ev* is an event object that will allow the host to when the copy finished. Finally, the scatter of all the data associated with the TPB is performed.

Algorithm 21: The pruning algorithm

```
1  Function Pruning (B, D_M^k, k)
2      A ← Buffer(B.ptNum)
3      I ← Buffer(B.ptNum)
4      PruningMap(B.S, D_M^k, A)
5      Scan(A, I)
       // Get the number of points not pruned
6      ev ← GetPtNumLeft(A, I, B)
       // Scatter all the data associated with the TPB
7      Scatter2D(A, I, B.pts, B.pts, B.d)
8      Scatter2D(A, I, B.pKnn, B.pKnn, 2k̇)
9      Scatter(A, I, B.Norm, B.Norm)
10     Scatter(A, I, B.S, B.S)
11     return ev
```

It should be noted that there are many performance optmizations that could be made to the kernels presented in this section. For instance: (i) Map and Scan could be joined to avoid a full read/write of the A buffer; (ii) LocalScan could be made to perform the scan on private memory, before resorting to shared memory. This would considerably improve the utilization of the PEs; (iii) Scatter2D could use vector copies to read from and write to global memory, in order to improve memory bandwidth utilization. However, these optimizations are unnecessary for our application, because the Pruning, **implemented as shown here**, of out algorithm. Thus, the resulting performance gains from these optimizations would minor, but the kernels would have become considerably more complex and harder to maintain.

[4] This can be computed by summing the last elements of A and I