Distrib Parallel Databases manuscript No.
(will be inserted by the editor)

Parallel Outlier Detection on Uncertain Data for GPUs

Takazumi Matsumoto · Edward Hung · Man Lung Yiu

the date of receipt and acceptance should be inserted later

Abstract Outlier detection, also known as anomaly detection, is a common data mining task in identifying data points that are outside expected patterns in a given dataset. It has useful applications such as network intrusion, system faults, and fraudulent activity. In addition, real world data are uncertain in nature and they may be represented as uncertain data.

In this paper, we propose an improved parallel algorithm for outlier detection on uncertain data using density sampling and develop an implementation running on both GPUs and multi-core CPUs, using the OpenCL framework. Our main focus is on GPUs, as they are a cost effective massively parallel floating point processor that is suitable for many data mining applications. Our implementation exploits some key features in GPUs, and is significantly different from a traditional CPU implementation.

We first present an improved uncertain outlier detection algorithm. Then, we demonstrate two parallel micro-clustering implementations. The performance and detection quality comparisons demonstrate the benefits of the improved algorithm and parallel implementation on GPUs.

Keywords GPU · outlier detection · parallel processing · uncertain data

1 Introduction

In this paper, we propose an efficient parallel outlier detection system for uncertain data designed to run at high speed using graphics processing units (GPUs). This speed is useful for applications that require interactive response times even while processing large amounts of data, such as in intrusion detection systems [6].

GPUs have become popular as they can provide substantial parallel computing resources at a low cost. A GPU can provide considerably more computational power

T. Matsumoto · E. Hung · M. L. Yiu
The Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong
E-mail: {cstmatsumoto,csehung,csmlyiu}@comp.polyu.edu.hk

than a CPU of similar price (for instance, a NVIDIA GTX Titan GPU delivers approximately 6 to 24 times as many GFLOPS as a comparably priced hexacore Intel Core i7 4960X CPU). This general purpose computing using graphics processors (GPGPU) practice has been successfully applied to computationally intensive tasks other than traditional applications in 3D graphics, such as in scientific modeling, video encoding and cryptography [33]. Data mining tasks are also of interest as they are typically highly data parallel, making them strong candidates for parallelization on GPUs.

Outlier detection is a widely employed fundamental data mining technique in which unusual events are extracted from a dataset. It has applications in network intrusion detection [39], fraud detection [10] and system fault detection [27].

There has also been increased interest in mining uncertain data, driven by the quantities of data recorded [5]. Real world data, such as from temperature sensor networks or location-based services [42] [44], often contain value uncertainty as well as possibly erroneous and/or missing values. Some statistical techniques such as privacy-preserving data mining may deliberately add uncertainty into data [13]. A traditional outlier detection method would discard this uncertainty information as modeling uncertainty adds additional complexity. However it has been shown in [3] that using uncertainty information can result in improved outlier detection quality.

In this paper we adopt a representative method for outlier detection on uncertain data that is based on the simple yet effective density sampling approach [4]. Modeling uncertainty introduces extra complexity to outlier detection [42] as uncertain objects are no longer represented by a single point, but rather can be represented as a probabilistic object that is sampled. The outlier detection process consists of two steps:

1. micro-clustering to compress the data, followed by
2. a density sampling based outlier detection.

We implement these two steps for parallel processing on GPUs using the OpenCL framework [21]. Using GPUs not only provides a high degree of parallelism required for micro-clustering compression on large datasets, but also hardware acceleration for mathematical functions used in uncertain outlier detection. Another benefit of the OpenCL framework is that it allows the same code to run on multi-core CPUs in parallel.

1.1 Contributions

This paper proposes an improved algorithm leveraging GPU acceleration for outlier detection with uncertain data. It enhances the density sampling approach described in [4], with micro-cluster compression being used on the GPU to maintain a representation of large datasets in the limited GPU video memory.

Our preliminary work [31] demonstrated acceleration of the basic density sampling algorithm using GPUs. The issue of scaling with the number of objects however was left open, which led to super-linear scaling similar to that seen in Fig. 14(a).

This paper extends it substantially with an improved approach that offers the following:

- An improved uncertain outlier detection algorithm that scales significantly better with dimensionality while matching the outlier detection quality of [4] (Sect. 4.2)
- Parallel micro-clustering for compression of the dataset allows for operation on significantly larger datasets compared to [6] and [31] (Sect. 4.3)
- More detailed discussion of parallel programming optimizations for GPUs, with performance comparisons applied to our implementation (Sect. 4.4)
- Additional experimental comparisons with both synthetic and real-world data demonstrate the benefits of the improved algorithm and optimization techniques (Sects. 5.3 – 5.5)

1.2 Outline

The rest of the paper is organized as follows. Sect. 2 considers related work mainly in the field of outlier detection with uncertain data as well as other GPU accelerated outlier detection techniques. Sect. 3 defines the problem of outlier detection and uncertain data and gives a summary of the original baseline algorithm as background. Sect. 4 presents our algorithm, along with details on parallelization and optimization using the OpenCL framework for GPU programming. Sect. 5 describes the testing methodology and experimental results demonstrating improved performance and outlier detection quality using our algorithm. Finally, Sect. 6 summarizes the key contributions and concludes this paper with future work directions.

2 Related work

2.1 Outlier detection

Intuitively, an outlier is a data point that is far apart from other data points in a given dataset. We first introduce outlier detection techniques on certain data, and then discuss outlier detection techniques on uncertain data.

2.1.1 Outlier detection on certain data

Outlier detection techniques can be broadly categorized into two main methodologies: *distance-based* approaches [22] [23] [38] and *density-based* approaches such as *Local Outlier Factor* (LOF) [11].

The classic k-NN derived approach as described in [22] uses two parameters, n and v. For any p in dataset $\mathcal{D}$ and a positive value of v, p is an outlier with respect to n and v if $|N_v(p)| < n$, where the v-neighborhood of p is defined as $N_v(p) = \{o : distance(p, o) \leq v, o \neq p, o \in \mathcal{D}\}$. A variant of this approach was proposed in [38], which uses the concept of k-distance (the distance to p's k-nearest neighbors) to assert the objects with the highest k-distances as outliers. [38] also uses partitioning clustering of the dataset and performs pruning to ignore partitions that do not contain outliers.

LOF [11] uses a similar concept to DBSCAN [14] by using reachability in a local neighborhood around each point. k-distance is used to define the k-neighborhood,

following from the previous definition of a v-neighborhood, is $N_k(p) = \{o : distance(p,o) \leq k\text{-}distance(o), o \neq p, o \in \mathcal{D}\}$. To smooth out variations in neighbor distance, *reachability distance* is defined as $reach\text{-}dist_k(p,o) = max(k\text{-}distance(o), distance(p,o))$.

2.1.2 Outlier detection on uncertain data

A considerable portion of data in the real world contains some degree of uncertainty [3], due to factors such as limitations in measuring equipment, partial responses or interpolation [5] [13]. An example of this could be a remote sensor network or location-based tracking system [42]. There have been several different clustering algorithms that have been adapted for uncertain data, including UK-means [12] which is an extension of k-means using expected distance, as well as $\mathcal{F}$DBSCAN [24] and $\mathcal{F}$OPTICS [25] which use the probability that objects lie in proximity to each other to estimate reachability.

Two common ways of modeling uncertainty are *tuple-uncertainty* which stores single probability values together with data objects [42], or *attribute-uncertainty* which instead uses probability density functions (*pdf*s) to describe uncertainty for data objects [4].

Using a *pdf* such as the Gaussian distribution, which offer a convenient closed form representation, eliminates the need to store discrete samples. In addition, the density sampling approach works in conjunction with the sampling of the *pdf* to determine uncertainty. The method described in this paper uses a Gaussian distribution for uncertainty modeling, in [4], it is shown that by considering uncertainty in outlier detection the detection quality can be improved compared to a more traditional deterministic approach.

Since it is impractical to determine η-probabilities directly, a sampling approach of the *pdf*s is used. This is one significant concern with regard to handing uncertain data — the number of expensive calculations (density or distance) is multiplied by the number of samples, negatively impacting performance. A large amount of works [38] [34] [20] have focused on increasing efficiency by reducing the number of candidates. In this paper, we utilize a *micro-clustering* technique [4] to reduce the number of data objects by creating clusters of data objects, and substituting the cluster centroids rather than the raw data to effectively compress the dataset. This has the added benefit of allowing very large datasets to fit in the limited dedicated GPU memory.

Our proposed algorithm is based on the density sampling algorithm described in [4]. Each record (data object) has a number of attributes (dimensions), and each dimension has a *pdf*, and the objective is to calculate the probability each object lies in an area with data density of at least η (expressed as the η-probability of a data object). If the η-probability of that object is less than δ, it is considered an outlier.

We note that the uncertain outlier detection algorithm described in [4] exhibits super-linear scaling with respect to dimensionality. It has also been noted in [8] that high dimensionality is a key issue when scaling up. Our algorithm in Sect. 4 is able to improve performance scaling as well as improve detection quality compared to the baseline algorithm.

2.2 Parallel and GPU-accelerated data mining

Data mining applications such as outlier detection are good candidates for parallelization [16] [18] as a large amount of data is processed by a small number of routines. Several outlier detection algorithms have been parallelized for acceleration with GPUs — notable works include [6], which accelerates the LOF algorithm by parallelizing the computationally expensive k-nearest neighbor method on a GPU, and the feature selection method described in [8] which also uses GPUs to accelerate kNN. The GPU component of [41] focuses on two aspects: using parallel memory accesses on the GPU to quickly manage data structures, and decomposing the computationally intensive covariance sum calculation for parallelization. In [9], the computation for training probabilistic neural networks is simple, however the performance gain can still be significant from standard parallelization approaches such as loop unrolling.

In the aforementioned work the GPU has been primarily used to accelerate a portion of an existing algorithm; in contrast, our algorithm runs entirely on the GPU (leaving the CPU to manage I/O). By extensively using GPU functions (including native functions), we achieve higher performance compared to multi-threading a limited number of functions. In addition, there are two distinct tasks: micro-clustering compression using a variant of k-means, and outlier detection by density sampling, which could take advantage of the portability of OpenCL to be scheduled flexibly on a combination of CPU and GPU.

GPUs have also been successfully used in other areas of data mining such as clustering [16], collaborative filtering [43], sequence alignment [29] and similarity search [26]. These tasks are 'data parallel', and are also well suited for execution on a massively parallel GPU [6]. Unlike conventional parallel processing computers that have many complex CPU cores, a modern GPU consists of a large number of simple processors that can execute floating point arithmetic and logic. Thus, GPUs can execute many threads together and are usually tied to high bandwidth onboard RAM. The advantage of GPU-based systems over a traditional parallel computer such as the Intel Itanium 2 system used in [28] is in cost — individual graphics boards are relatively inexpensive, and can also be scaled up.

Parallel processing on GPUs falls between two traditional forms of parallel processing, Single Instruction Multiple Data (SIMD) and Simultaneous Multi-threading (SMT). SMT is a very general model of parallelism in which a program runs multiple copies of a routine in threads that can run concurrently to achieve parallelism. SIMD on the other hand uses operations that work on fixed length vectors, so each instruction affects multiple values simultaneously. Most modern processors, as well as some dedicated accelerators such as Intel Xeon Phi use both multi-threading and SIMD in achieving peak floating point performance. As the algorithm in this paper does not explicitly use vectors, any SIMD optimization would depend on the compiler. If the platform supports OpenCL, such as Xeon Phi [19], porting the implementation should be straightforward, although optimization for accelerators such as Xeon Phi are different to that for GPUs.

More specialized computation devices such as Field Programmable Gate Arrays (FPGAs) allow algorithms to be implemented more efficiently than standard SMT parallelism by removing constraints such as data types and threading overhead. Gen-

erally this requires more specialized implementation knowledge. Nevertheless,there exist development tools [7] that can translate an OpenCL implementation into a FPGA implementation.

[35] refers to the GPU method of parallelism as Single Instruction Multiple Threads (SIMT). In general, this can be considered a restricted form of SMT which can both support SIMD-like operation on vectors as well as scalar parallelism with a large number of threads. However unlike a processor with a dedicated wide vector unit, most modern GPU designs employ a large number of narrower SIMD units suitable for graphics computation that can also run scalar code without significant loss of performance. Unlike true SMT processors however, GPUs have limitations on threading and memory access in order to achieve optimal parallel operation. For instance, extensive random memory accesses will degrade performance due to serialization of memory requests [2]. In such a case, an operation such as scatter-gather [17] may be useful for optimizing a bulk memory access step in an algorithm.

Two popular programming frameworks for GPGPU are (i) C for CUDA, a proprietary solution developed by NVIDIA Corporation [36], and (ii) OpenCL, an open standard managed by The Khronos Group [21] and backed by multiple companies including Intel, AMD, NVIDIA and Apple. In both CUDA and OpenCL, work is split from the host (i.e. the CPU) to *kernels* that execute on a computing device (e.g. GPUs). Kernels contain computationally intensive tasks, while the host is tasked with managing the other computing devices. A single kernel can be executed in parallel by many worker threads on a GPU. In this paper, we develop our implementation in OpenCL as the framework has been implemented by vendors for many GPUs, CPUs and other accelerators such as Xeon Phi and Altera FPGAs. We utilize this portability to demonstrate the performance improvement from parallel execution on CPUs to GPUs in Sect. 5.3.

3 Problem formulation and background

Sect. 3.1 presents the problem of outlier detection. Sect. 3.2 describes the outlier detection method on uncertain data in [4] as a background to our work. Table 1 listing the symbols used throughout this paper.

3.1 Problem formulation

We adopt the density-based definition of uncertain outliers in [4], which will be described in Sect. 3.2.1. We make three fundamental assumptions about the data:

- The overall size of the set of outliers is assumed to be a small proportion of the overall dataset, distributed differently to other data objects
- There may be several clusters of non-outlier objects, with varying distributions
- Outlier and non-outlier objects may overlap

The first condition follows directly from the intuitive definition of outliers. The remaining conditions model several possibilities when dealing with real world data that

Notation	Meaning
$\mathcal{D}$	The dataset
$\mathcal{O}$	The set of outliers
$\mathcal{C}$	The set of subspaces
n	Number of objects in $\mathcal{D}$
d_i	i-th object in $\mathcal{D}$
r	Number of dimensions
h_i^j	The probability density function of the jth dimension of d_i
$\psi_j(d_i)$	The standard deviation of the jth dimension of d_i
F_i^j	The inverse cumulative distribution function of h_i^j
η	The data density for a non-outlier object
δ	The threshold probability for an object to be considered an outlier
a	The dimensionality of the subspace
A	The current subspace
b	The maximum dimensionality of the subspace (subspace window size)
s	The number of samples taken in each *pdf*
q	The number of microclusters

Table 1: Table of symbols used throughout this paper

does not completely fit within outlier definitions. In particular, the last condition notes that outliers may overlap with non-outliers, often referred to as Bayesian error.

More formally, given a dataset $\mathcal{D}$ containing n uncertain data objects, let each object d_i have r dimensions. For each object, each dimension has a *pdf*, for a total of nr *pdf*s. $\mathcal{D}$ is normalized to the range [0,1].

The *pdf* for object d_i along dimension j is denoted by $h_i^j(\cdot)$ and the standard deviation of $h_i^j(\cdot)$ is denoted $\psi_j(d_i)$.

3.2 Background: Outlier detection on uncertain data

This section describes the outlier detection process on uncertain data in [4], which employs a density-based outlier detection algorithm (in Sect. 3.2.1) and a micro-clustering algorithm (in Sect. 3.2.2). In [4], this is implemented as a standard serial program that run on CPUs.

3.2.1 Density-based outlier detection

Definition 1 **(η-probability [4])** Let a subspace $S = \{1, 2, \cdots, a\}$, where $a \leq r$. The η-probability of object d_i is the probability that d_i lies in a subspace S with overall data density of at least η.

Specifically, it is defined as the following integral [4]:

$$p_i = \int_{G(x_1,\ldots,x_a)\geq\eta} \prod_{j=1}^{a} h_i^j(x_j)\, dx_j \tag{1}$$

where $G(x_1, \ldots, x_a)$ represents the overall probability density function on all coordinates in the given a-dimensional subspace. Note that it is assumed that each dimension is independent with its own probability distribution.

However, as p_i is difficult to calculate precisely as in Eq. 1, it can be estimated using a density sampling algorithm. Thus the density function h' given a uniformly random value u in $[0, 1]$ and uncertainty values $\psi(\cdot)$ is defined as the error-based density function [4]:

$$h'(u, \psi(d_i)) = \frac{1}{n} \sum_{k=1}^{n} \frac{1}{\sqrt{2\pi}(w + \psi(d_k))} e^{-\frac{(u-\overline{d_i})^2}{2(w^2+\psi(d_k)^2)}} \tag{2}$$

Note that $\overline{d_i}$ is the mean value of d_i and w is a smoothing factor determined using the Silverman approximation rule as $1.06n^{-0.2}\sigma$ [4].

The full sampling algorithm is given in Algorithm 1.

Algorithm 1 ESTIMATEPROBABILITY (Dataset $\mathcal{D}_A$, object index i, data density threshold η, dimensionality r, number of samples s)

Let $F_i^j(\cdot)$ be the inverse cumulative distribution function of *pdf* $h_i^j(\cdot)$
$success = 0$
for s sample iterations **do**
    $density_i = 0$
    for each dimension $j \in$ subspace A **do**
        $y =$ a uniform random value in $[0, 1]$
        $X_j =$ a sample point $F_i^j(y)$
        for each $d_k \in \mathcal{D}_A$ **do**
            $density_i = density_i + h_k^j(X_j)$
    $density_i = density_i / (|\mathcal{D}| \times r)$
    if $density_i > \eta$ **then**
        $success = success + 1$
Return $success/s$ ▷ estimated outlier probability

Definition 2 **((δ, η)-outlier [4])** An object d_i is said to be a (δ, η)-outlier if the η-probability of d_i in some subspace is less than δ.

Algorithm 2 DENSITYOUTLIER (Dataset $\mathcal{D}$, data density threshold η, outlier probability threshold δ, dimensionality r, number of samples s)

$\mathcal{O} = \emptyset$
$a = 1$
while $|\mathcal{O}| < |\mathcal{D}|$ and $a \leq r$ **do**
    Let $\mathcal{E}_a$ be all a dimensional subspaces
    for each subspace $A \in \mathcal{E}_a$ **do**
        $\mathcal{D}_A = \mathcal{D} - \mathcal{O}$ in subspace A
        for each $d_i \in \mathcal{D}_A$ **do**
            $p =$ ESTIMATEPROBABILITY($\mathcal{D}_A, i, \eta, r, s$)
            if $p < \delta$ **then**
                Add d_i to $\mathcal{O}$
    $a = a + 1$
Return $\mathcal{O}$ ▷ Outlier set $\mathcal{O}$

3.2.2 Micro-cluster compression

It requires high cost to compute overall data density at each sampled point. Thus, the overall complexity of the outlier detection step is high which leads to issues when attempting to deal with large datasets. To alleviate this, the number of objects n can be reduced by using 'micro-clustering' [4], effectively compressing many uncertain objects into a single representative uncertain object. This process of clustering (both terms used interchangeably in this paper) is as follows:

Each cluster $\mathcal{C}$ is defined by a $3r + 1$ tuple $(\overline{CF2^x}(\mathcal{C}), \overline{EF2^x}(\mathcal{C}), \overline{CF1^x}(\mathcal{C}), n(\mathcal{C}))$.

- $\overline{CF2^x}(\mathcal{C})$ is a r-dimensional vector, with each entry p containing the sum of squares of mean values of the member objects' p-th dimension.
- $\overline{EF2^x}(\mathcal{C})$ is a r-dimensional vector, with each entry containing the sum of squares of errors of the member objects' p-th dimension.
- $\overline{CF1^x}(\mathcal{C})$ is a r-dimensional vector, with each entry p containing the sum of mean values of the member objects' p-th dimension.
- $n(\mathcal{C})$ is the number of data objects in the cluster.

A parameter q defines the number of centroids that are randomly selected from the dataset. Each object is then assigned to the closest centroid as determined by Euclidean distance between the centroid and the data object. No new clusters are created after initialization, with the centroids recalculated and the clustering process repeated until a threshold is reached, as in k-means clustering. Since this is an approximation of the underlying data, the maximum number of clustering iterations is set to a moderate amount. Note that unlike in typical k-means clustering the goal is not to find the closest match to the underlying data distribution, but to group several close objects into a single object (i.e. a micro-cluster).

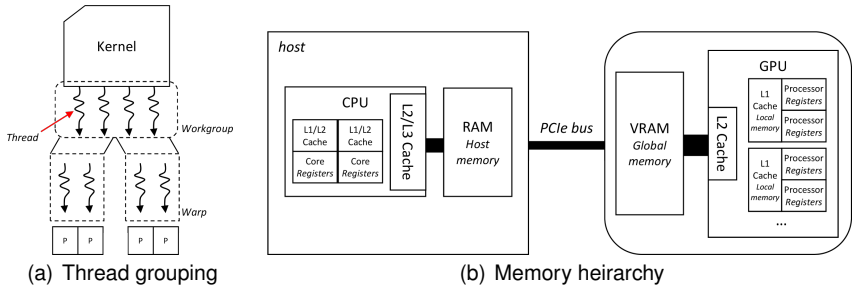
4 Parallel outlier detection

In this section, we first briefly discuss GPU architecture and the terminology used (Sect. 4.1). We then present our improved algorithms for parallel density-based outlier detections with subspace windows (Sect. 4.2) and parallel micro-clustering (Sect. 4.3). The optimizations employed for parallel processing on GPUs is discussed in Sect. 4.4. We also analyze the time complexity of our algorithms in Sect. 4.5.

4.1 Background: GPU architecture

A *kernel* is a function or subroutine which runs on the target computing device (i.e. a GPU) and is invoked from the host (CPU). Thus the first step is to determine how to divide the problem into kernels. This paper uses the *parfor* keyword to describe the parallel nature of kernels in pseudo-code. A *parfor* block represents a for-loop where all iterations are executed by corresponding threads in parallel.

Kernels are executed in instances called *threads*. While there is no practical limit to the number of threads that can be spawned, for GPUs there are specific thread



(a) Thread grouping (b) Memory heirarchy

Fig. 1: GPU threading and memory models

groupings that impose some limits on threading. The first is that threads are grouped in *workgroups* (also known as thread-blocks). In this paper, a nested *parfor* block fits under the workgroup/thread structure of OpenCL.

Within a workgroup, thread synchronization can occur. Synchronization is necessary to ensure all the threads have completed memory operations prior to the results being used across threads. The common 'reduce' operation has one thread gathering the results from all the threads in the workgroup to store in global memory, and requires synchronization for the results to be valid.

In OpenCL, synchronization is achieved using *barriers* [21]. There are two parts to a barrier, one is ensuring all threads in a workgroup reach the barrier before continuing, and the other is to ensure all memory operations are complete before continuing. A *local barrier* waits until all operations on shared (local) memory are complete, while a *global barrier* waits until all operations on global memory are complete. In addition a *workqueue barrier* can be placed between kernel executions to ensure the kernels are executed in order, ensuring that data dependency is met.

In addition, as shown in Fig. 1(a) threads are also grouped into smaller groups called warps (or wavefronts) that execute in lockstep on the hardware. This can result in added running time when threads in a warp encounter branch divergence, as the branch must be replayed for all threads in the warp. The maximum number of threads in a workgroup and the size of a warp is defined by the hardware used.

Another important consideration is the memory access pattern on GPUs. As shown in Fig 1(b), there are several levels of caching and memory. The largest and slowest memory usable by GPUs (excluding secondary storage) is host memory. However, since host memory has a different address space and requires copy operations over the relatively slow PCIe interconnect, GPU operations are almost exclusively done using the GPU's onboard memory. This memory is broken down into the largest and slowest global memory, the fast cache-backed but limited capacity shared memory, and the per-thread registers. The memory location also determines which threads can share data — global memory can be used by all threads, shared memory is local to the workgroup, and registers are private to each thread.

More discussion on optimization for GPUs is given in Sect. 4.4.

4.2 Density-based outlier detection on uncertain data with subspace window

The approach taken in this paper generalizes the density sampling outlier detection as described previously, with parameters that allow it to work in four distinct methods:

- an exhaustive 'roll-up' method as described in [4],
- a restricted roll-up method that is a simpler variant of the above,
- an intuitive global averaging method, and
- our proposed 'independent' method.

Fig. 2 shows a simple example with 5 objects $(d_1, \cdots, d_5)$ in 2 dimensions, measured using 3 samples each. At each sampling point the overall density is measured. For example, the samples in d_4 lie in a high-density area while the samples in d_5 lie in a low-density area.

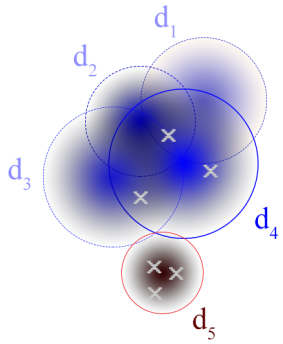


Fig. 2: Example sampling of uncertain objects

In Algorithm 3, we introduce parameters a and b to control the subspaces that are explored. s denotes the number of samples, fixed at 800 by default as in [4]. The remaining parameters are as mentioned previously.

By sampling an object at multiple random points, calculating the overall data density at each sampled point, and counting how many of those sampled points exceed η, the probability that object lies in a subspace with data density of at least η (i.e. η-probability) can be estimated. Finally, object d_i is defined as a (δ, η)-outlier if the η-probability of d_i in any subspace is less than a user-defined parameter δ, that is, the object has a low probability of lying in a region of high data density.

Computationally, the most expensive functions are the calculation of the inverse cumulative distribution function (inverse *cdf*) $F_i^j(\cdot)$, and the *pdf* $h_i^j(\cdot)$. Due to the lack of a closed form representation of the inverse *cdf*, it is computed numerically with rational minimax approximations as in [1].

Through a combination of values for parameters a and b it is possible for Algorithm 3 to behave in four distinct methods as shown in Table 2. A simple example with 4 objects $(d_1, \cdots, d_4)$ and 2 dimensions is shown in Fig. 3, with the possible subspaces labeled. The subspaces considered and the number of object-dimensions considered is also listed in Table 2 as an indication of the computational complexity

Algorithm 3 DENSITYOUTLIERW (Dataset $\mathcal{D}$, data density threshold η, outlier probability threshold δ, initial subspace size a, maximum subspace size b, dimensionality r, number of samples s)

$\mathcal{O} = \emptyset$
while $|\mathcal{O}| < |\mathcal{D}|$ and $a \leq b$ or ($a \leq r$ if $b = 0$) **do**
  Let $\mathcal{E}_a$ be all a dimensional subspaces or a-th dimension if $b = 0$
  for each subspace $A \in \mathcal{E}_a$ **do**
    $\mathcal{D}_A = \mathcal{D} - \mathcal{O}$ in subspace A
    for each $d_i \in \mathcal{D}_A$ **do**
      $p =$ ESTIMATEPROBABILITY($\mathcal{D}_A, i, \eta, a, s$)
      if $p < \delta$ **then**
        Add d_i to $\mathcal{O}$
  $a = a + 1$
Return $\mathcal{O}$ ▷ Outlier set

and sampling strategy. In this example, it is apparent that the outlier is not visible in subspace I, while it is visible in subspaces II and III. Thus as shown in Table 2, the roll-up approach has a higher computational complexity to achieve the same result. In addition, the larger subspace III reduces the effectiveness of sampling, as samples are spread over a larger area. This can negatively affect outlier detection quality as demonstrated in Sect. 5.

The original outlier detection algorithm as described in [4] uses a roll-up method to explore the data space. This is an iterative approach that starts with a set of all possible one dimensional subspaces and with each iteration explores all higher dimensional subspaces. Each subspace is tested for outliers and any objects determined to be outliers are removed from further consideration in other subspaces. It is assumed that outliers will have lower density in some subspaces than others, and conversely a non-outlier in this definition has high density in all tested subspaces. This mode is emulated by setting $a = 1$ and $b = r$. However, a disadvantage of this method is that with increasing numbers of dimensions, the computational workload increases exponentially with the number of possible subspaces. This inefficiency makes it unsustainable when dealing with progressively higher dimensional datasets. In this paper we label this method 'exhaustive roll-up'.

Our proposed method tests data density in each dimension independently by setting $a = 1$ and $b = 1$. This limits the subspaces considered to a single dimension, similar to the first iteration of roll-up. The rationale is similar to the roll-up method — outliers may be obvious in certain dimensions than others. However since averaging the data density in larger subspaces may dilute the density difference, each dimension is tested independently. As in the roll-up method, if an object is found to be an outlier in any given dimension, it is marked as an outlier and not considered in other dimensions. To reduce the risk of error or bias caused by a particular ordering of dimensions, the dimensions are randomized prior to density sampling.

For performance comparison there are two other methods, a global averaging method and a restricted roll-up method.

The averaging method uses the average data density over all dimensions, requiring only a single pass over the entire data space. This mode is achieved by setting $a = r$ and $b = r$, and is equivalent to the final iteration of the roll-up method. Sim-

ilar to the independent method this is a linear time complexity method as there is only a single pass over the entire dataset, but since only outliers that are in an area of low density in the entire data space can be found the outlier detection ability is significantly reduced.

The restricted roll-up method is different to the others as there is a fixed window of subspaces considered, so for each iteration only the first subspace is checked for outliers. This is achieved by setting $a = 1$ and $b = 0$, with $b = 0$ being a special case in Algorithm 3. This gives a quadratic performance curve with respect to dimensionality, falling between the linear independent and averaging methods and the exponential exhaustive roll-up method. In this paper if the specific variant of roll-up is not mentioned, this restricted variant is used to streamline testing.

Algorithm 4 presents a parallel version of Algorithm 3. At a high level, this is achieved by unrolling the innermost loop and assigning each object a thread. Note that objects mentioned here are the result of micro-clustering, so the number of objects is the number of clusters q rather than n. Thus, the number of threads in this step is equal to the number of clusters. Sect. 4.4 discusses in more detail the implementation differences between Algorithms 3 and 4. Sect. 5 demonstrates that our proposed method runs in a time comparable to the simple global averaging method, with the detection quality that matches the computationally expensive exhaustive roll-up method.

Algorithm 4 PARALLELDENSITYOUTLIERW (Dataset $\mathcal{D}$, data density threshold η, outlier probability threshold δ, initial subspace size a, maximum subspace size b, dimensionality r, number of samples s)

```
1: O = ∅
2: parfor each object i ∈ D do
3:     while |O| < |D| and a ≤ r or (a ≤ r if b = 0) do
4:         Let E_a be all a dimensional subspaces or a-th dimension if b = 0
5:         for each subspace A ∈ E_a do
6:             D_A = D − O in subspace A
7:             p = ESTIMATEPROBABILITY(D_A, i, η, a, s)
8:             if p < δ then
9:                 Add d_i to O
10:        a = a + 1
11: end parfor
12: Return O                                              ▷ Outlier set
```

Algorithm mode	Parameter a	Parameter b	Subspaces used	Total subspace size
Independent	1	1	I + II	$2 \times 4 = 8$
Exhaustive Roll-up	1	r	I + II + III	$2^2 \times 4 = 16$
Restricted Roll-up	1	0	I + III	$((2 \times 3)/2) \times 4 = 12$
Global Averaging	r	r	III	$2 \times 4 = 8$

Table 2: Parameter settings for different algorithm modes. Note that $r = 2$ in Fig. 3

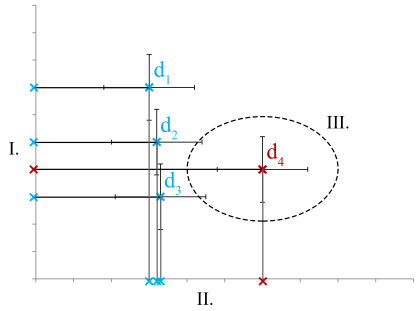


Fig. 3: Different sampling subspaces of each algorithm

4.3 Parallel micro-clustering

When using micro-clustering for compression, clusters are used in place of the data objects. True error Δ is calculated by averaging the members, with bias equal to the distance from the cluster centroid and variance equal to the original error. This is defined for a given dimension j as:

$$\Delta_j(\mathcal{C}) = \sum_{i=1}^{r} \frac{bias_j(\overline{X}, \mathcal{C})^2 + \psi_j(\overline{X})^2}{r} \tag{3}$$

Eq. 3 can be rearranged to use the stored cluster tuple as follows:

$$\Delta_j(\mathcal{C}) = \frac{CF2_j^x(\mathcal{C})}{r} - \frac{CF1_j^x(\mathcal{C})^2}{r^2} + \frac{EF2_j^x(\mathcal{C})}{r} \tag{4}$$

The advantage of this approach is that it does not require significant modification of the algorithm, as $\Delta(\mathcal{C})$ and $n(\mathcal{C})$ can be substituted into Eq. 2 as follows:

$$\overline{f^Q}(p, \Delta(\mathcal{C})) = \frac{1}{n} \sum_{i=1}^{q} n(\mathcal{C}_i) \frac{1}{\sqrt{2\pi}(h + \Delta(\mathcal{C}_i))} e^{-\frac{(p-\overline{X_i})^2}{2(h^2+\Delta(\mathcal{C}_i))^2}} \tag{5}$$

For parallelization, we consider two approaches. Algorithm 5 describes a straightforward approach that creates a thread for each object, with each thread looping over each of the dimensions. However, since the number of objects is generally very large, the dataset is partitioned among a fixed number of threads to control resource usage and memory access patterns. Note that the function to find the closest centroid is a loop over the q centroids.

The advantage of this approach is that it is simple and self-contained, and is more work efficient when the number of dimensions is low since more objects are processed concurrently.

To reduce overhead there is no heuristic to determine when to stop iterating, thus the performance is directly impacted by the preset number of maximum iterations. Sect. 5 also shows the performance-quality trade-off of adjusting the clustering parameters.

The alternative ‘B’ approach described in Algorithm 6 makes two key changes from the earlier ‘A’ pattern. The first is that the kernels are now parallel in dimensions as well as objects, enabling greater parallelization. The second change is that

Algorithm 5 PARALLELMICROCLUSTERINGA (Dataset $\mathcal{D}$, Clusters $\mathcal{C}$, dataset size n, number of clustering iterations $iter$)

Let $blocksize$ be the number of threads in workgroup
Let an object stripe be a set of $n/blocksize$ objects
parfor each thread up (1 to $blocksize$) **do**
  Let m be a float variable (register)
  repeat
    $CF1 = \emptyset$
    $CF2 = \emptyset$
    $EF2 = \emptyset$
    $m = \infty$
    for each object i in stripe **do**
      for each cluster $j \in \mathcal{C}$ **do**
        $m = \min(\text{DISTANCE}(\mathcal{D}, i, \mathcal{C}, j), m)$
      Add i to C_m
      Update $CF1^i_m, CF2^i_m, EF2^{i,m}$
    Global barrier
    for each cluster j (1 to q) **do**
      Reduce $CF1^i_j, CF2^i_j, EF2^i_j$ to $CF1_j, CF2_j, EF2_j$
      Update C_j
    Global barrier
  until repeated for $iter$ iterations
end parfor
Return $\mathcal{C}$ ▷ Cluster centroids, uncertainties and mapping

synchronization between the clustering and the updating sub-steps are removed from the kernels and are instead handled by the host, that is, synchronization occurs automatically when the kernel completes.

This approach has the advantage of greater parallelism that results in improved performance with higher dimensionality datasets, at the cost of fewer objects being processed concurrently.

4.4 Optimizations for GPU implementation

Previously we have described the design of our parallel algorithms (Algorithms 4 to 6). In this subsection we discuss some key points of the parallel GPU implementations. Note that while the CPU executes the same code using the OpenCL framework, most of the discussed optimizations (i.e. native functions, memory access patterns, local memory) are dependent on the architecture of the GPU and provide no benefit for CPU execution.

For illustrative purposes, small benchmarks that measure only the GPU processing time are also conducted on the single distribution synthetic dataset ‘one’ (see Sect. 5.1), with default parameters as stated in Table 3 except where noted.

4.4.1 Native functions

Within each kernel, optimizations including the use of single precision floating point values and special hardware accelerated mathematical functions (*native* functions)

Algorithm 6 PARALLELMICROCLUSTERINGB (Dataset $\mathcal{D}$, Clusters $\mathcal{C}$, dimensionality r, number of clustering iterations $iter$)

repeat
  parfor each object $x \in \mathcal{D}$ **do** ▷ Kernel 1: Create micro-clusters
    Let $\mathcal{ML}[1,\ldots,r]$, $\mathcal{DL}[1,\ldots,r]$ and $\mathcal{CL}[1,\ldots,r]$ be temporary arrays (shared memory)
    parfor each dimension j (1 to r) **do**
      $\mathcal{DL}[j] = x_j \in \mathcal{D}$
      $\mathcal{ML}[j] = \infty$
      for each cluster $y \in \mathcal{C}$ **do**
        $\mathcal{CL}[j] = y_j \in \mathcal{C}$
        $\mathcal{ML}[j]$ =min($(\mathcal{DL}[j] - \mathcal{CL}[j])^2$,$\mathcal{ML}[j]$)
      Local barrier
      $m =$ reduce from $\mathcal{ML}$
      Add i to C_m
    end parfor
  end parfor ▷ End Kernel 1
  Workqueue barrier
  parfor each cluster $y \in \mathcal{C}$ **do** ▷ Kernel 2: Update micro-clusters
    parfor each dimension j (1 to r) **do**
      Let $cf1$, $cf2$, $ef2$ be float variables (register)
      for each object $i \in \mathcal{D}$ **do** Update $cf1, cf2, ef2$
      Update C_y^j using $cf1, cf2, ef2$
    end parfor
  end parfor ▷ End Kernel 2
  Workqueue barrier
until repeated for $iter$ iterations
Return $\mathcal{C}$

have been used. Single precision floating point values are used extensively in graphics, and thus GPUs are optimized for many single precision functions. Fig. 4 demonstrates a significant speedup from applying these native functions to the calculation of inverse *cdf* and *pdf* as illustrated in Fig. 4.

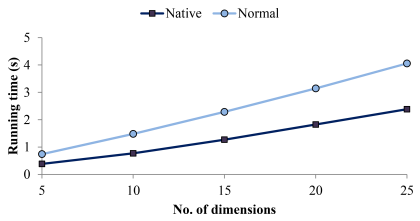


Fig. 4: Running time of Algorithm 1 using native and standard C++ math functions

4.4.2 Space complexity and memory access patterns

The primary data structures used are arrays of single precision floats of 4 bytes each. Initially, the two major arrays for the object value and uncertainty are of size $4nr$ bytes each. In addition, there are arrays for cluster centroids and uncertainty (size $4qr$ bytes), a mapping between object-cluster and the outlier list ($4n$ bytes, assuming

integers are 4 bytes each), along with $CF1$, $CF2$ and $EF2$ as described in Sect. 3.2.2. Once micro-clustering is complete, the full value and uncertainty arrays are no longer needed, and can be replaced with the per-cluster arrays of size $4qr$ instead.

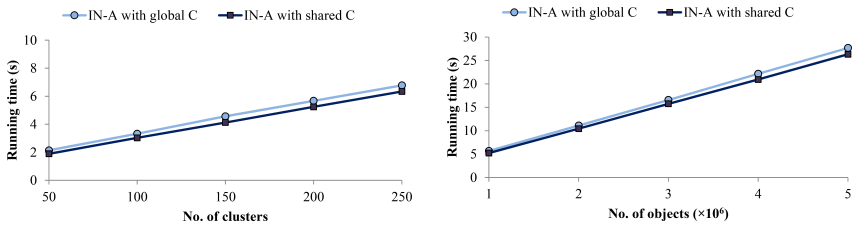
It is recommended practice to store frequently used data into the faster, cache-backed shared memory, however from the above it is noted that for many datasets the size of the main data structures in the micro-clustering step are too large to fit into the limited shared memory (for the GT440 this is 48KB). In addition, the main data arrays are not reused since the algorithm only takes a single pass over the dataset.

In Algorithm 5 global memory is used to synchronize across all work threads, with shared memory available for caching the frequently accessed cluster centroids array. The key considerations are the full dataset $\mathcal{D}$ and the micro-clustering $\mathcal{C}$, and if they fit in the GPU buffers.

The full dataset is allocated a buffer in the GPU's global memory, however the dataset itself can be partitioned and streamed in from host memory without any notable issues. More important is the micro-clustering $\mathcal{C}$. Our algorithm expects $\mathcal{C}$ to be present in GPU VRAM as it is read and updated frequently. There are two possibilities for the micro-clustering buffer: it can be placed in shared (local) memory for a performance boost or global memory.

Fig. 5 demonstrates a minor improvement in total running time when the cluster centroids array is present in local memory rather than global memory. However, this applies when q and r are relatively small.

Note that in Algorithm 6 a different approach is used — since synchronization has moved to the host, local memory is used to cache each object and reduce the number of global memory accesses required. This allows a thread to operate on each dimension of each object, greatly increasing parallelization. Since there is no global memory implementation of Algorithm 6, it is not tested here. Fig. 9 includes a comparison between the two approaches to micro-clustering.



(a) Local memory with no. of clusters (b) Local memory with no. of objects

Fig. 5: Running time of Algorithm 5 with the use of shared memory to cache $\mathcal{C}$

Since global memory is the only large memory space on the GPU where all threads can access data, it is important to consider the importance of caching and optimal memory access patterns on a GPU compared to a CPU. Fig. 6 shows a simple example with four objects (A, B, C, D) with four dimensions (1, 2, 3, 4), with the assumption that the CPU is working in serial and the cache holds two elements (highlighted left) and the GPU has four worker threads in parallel and a read block

of four (highlighted right). Memory requests are shown by the arrows. In the case shown on the left in Fig. 6, in the case of the CPU the next memory request for the second dimension of object A is already in cache and can be satisfied without waiting on main memory.

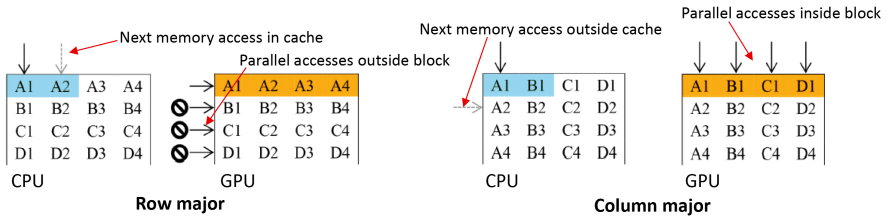


Fig. 6: Row major and column major memory layouts for the CPU and GPU

Note that however the access patterns for Algorithm 5 and Algorithm 6 are different. Algorithm 5 is parallel on objects, while Algorithm 6 are mainly parallel on dimensions.

4.4.3 Threading

In this paper, our development system had a NVIDIA GeForce GT440 GPU with 96 processors (each containing a arithmetic logic unit and floating point unit). Fig. 7 demonstrates performance scaling of Algorithm 5 with the number of GPU threads. Although the number of physical processors is less than the maximum number of threads, GPUs rapidly switch between groups of threads to reduce the impact of memory access latency.

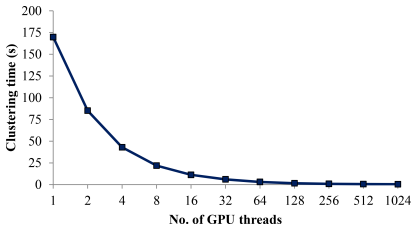


Fig. 7: Running time of Algorithm 5 with a varying number of threads

Note that in the outlier detection step each microcluster is assigned a thread and all threads access the compressed dataset concurrently, thus the number of threads is always equal to the number of clusters.

Since each data object is assigned one worker thread, each worker is responsible for density sampling of that object's space. This essentially unrolls the most computationally expensive loop in Algorithm 3, as shown in Algorithm 4. Note that $blocksize$ is limited by the maximum workgroup size, which in this case is 1024.

In addition, since threads in a warp execute the same code path simultaneously, branching code reduces execution efficiency by forcing execution to occur multiple times. In the typical outlier detection case, the limiting factor in execution time is generally in the micro-clustering step, which has to pass over the entire dataset, while the branching code that deals with outliers operates on the much smaller compressed dataset.

In addition to Algorithm 4, a number of other functions have been implemented in OpenCL including the uniform pseudo random number generator (PRNG), and the calculation of *pdf* and *cdf*. For portability and simplicity, the basic PRNG *xorshift* [30] has been used.

4.5 Time complexity analysis

To estimate the theoretical performance, we consider the two main parts of the algorithm, the initial micro-clustering and the outlier detection itself.

The micro-clustering process requires by design at least one pass over the entire dataset to assign each object into clusters, with several iterations for refinement. Given large values of n and/or r, this makes micro-clustering the more expensive operation than the outlier detection itself. As mentioned previously, we propose two parallel algorithms for micro-clustering. The more straightforward Algorithm 5 has a time complexity of approximately $\mathbf{O}(nrq/workers)$ per iteration, which is the product of the number of objects in the dataset n, the dimensionality of the dataset r and the number of clusters q divided by the number of worker threads running concurrently. The second implementation, which uses Algorithm 6 has time complexity of $\mathbf{O}(nq/workgroups)$, with workgroups referring to the number of workgroups that can be executing concurrently.

The key difference between these two implementations is the degree of parallelism and how much concurrency can be achieved on the hardware. Since the simpler Algorithm 5 does not parallelize the dimension-based calculation, more objects can be processed concurrently compared to Algorithm 6 that do assign a thread to each dimension. Since there are fewer concurrent workgroups than worker threads, the preferred algorithm depends on the number of objects and dimensions of the dataset used.

The parallel outlier detection process uses a worker thread for each micro-cluster, which samples the overall data density s times. The key difference between the methods for outlier detection are in the subspaces tested: in the exhaustive roll-up case there are $2^r - 1$ possible subspaces, giving an exponential complexity of $\mathbf{O}((2^r-1)qs)$. The restricted roll-up has a quadratic complexity of $\mathbf{O}((r(r+1)/2)qs)$ as it adds dimensions to subspaces, while the independent and averaging methods have linear complexity of $\mathbf{O}(rqs)$.

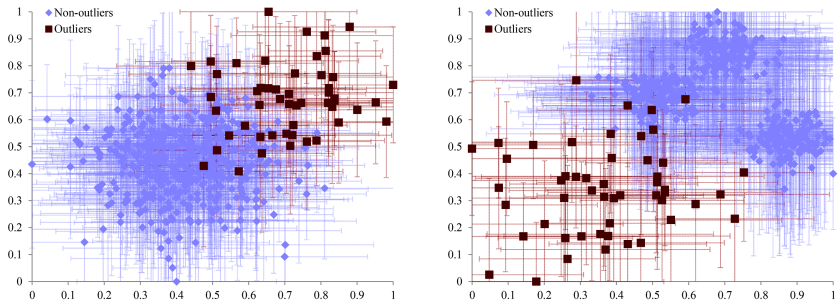
With the higher complexity methods, as dimensionality increases the outlier detection process quickly becomes the dominant factor in running time, while with the lower complexity methods

5 Experimental Evaluation

The two primary criteria for evaluation are: (i) detection quality, which can be measured with metrics such as precision and recall [4] [40], and (ii) performance, typically observed with running time comparisons [6]. In this paper we consider two factors: the difference in quality and performance between variants of the density sampling algorithm, and the difference in performance between platforms.

The methods being tested are restricted roll-up (a simplified version of [4]), basic global averaging, and the proposed 'independent' method (using each dimension independently for density calculation), along with performance testing on LOF (GPU implementation based on [6][1]) and our preliminary implementation [31]. To streamline testing, we do not test performance with the exhaustive roll-up method as its performance is orders of magnitude worse than the three previously mentioned methods. In addition, comparison between the three tested platforms (CPU, CPU with OpenCL parallelization, and GPU with OpenCL parallelization) is restricted to the independent method only, as the detection quality of the methods is the same regardless of platform.

5.1 Experimental setting



(a) One distribution dataset (b) Three distribution dataset

Fig. 8: Example synthetic datasets with 500 objects (450 normal objects, 50 outliers) and 2 dimensions

To test performance and quality, tests were conducted on synthetically generated data for performance and quality testing, as well as real datasets from the UCI Machine Learning Repository. For these real datasets, uncertainty is generated by calculating the standard deviation of each dimension and then multiplying it with a uniformly random number and an uncertainty factor.

The model used for generating the synthetic data is based on [11]. The 'one' dataset is a standard single distribution (similar to the example shown in Fig. 8(a)),

[1] We implement it ourselves as the source code of [6] is not publicly available

with random numbers generated following a Gaussian distribution with a mean value of 0.0 and a standard deviation of 1.0. The outliers are generated similarly, but an offset value is added to the mean (e.g. in Fig. 8(a), the offset value of the uncertain objects is 1.0), then the entire dataset, including outliers, is normalized to $[0.0, 1.0]$. The uncertainty information is also generated as described previously. The number of outliers is 5% of the total number of data objects unless otherwise noted.

The 'three' dataset is similar to the example shown in Fig. 8(b) except that the non-outlier data objects are arranged in three tight distributions of standard deviation 0.5, with mean values 1.5, 3.0 and 4.5. The outliers in this case are distributed with a mean value of 0.0 and a standard deviation of 1.0. Again, the entire dataset is normalized to $[0.0, 1.0]$.

The algorithm parameters η and δ control the overall sensitivity to outliers. Other important parameters include the number of clusters q and the number of iterations of clustering. The selection of appropriate parameters is a challenge in any algorithm; in Sect. 5 graphs of precision-recall are provided with varying values for several parameters. The parameters tested, the ranges and the default values are shown in Table 3.

Parameter	Meaning	Range
η	The minimum density for a non-outlier object	$0.3, 0.4, \mathbf{0.5}, 0.6, 0.7$
δ	The threshold probability for an object to be classed as an outlier	$0.3, 0.4, \mathbf{0.5}, 0.6, 0.7$
n	Number of objects	$\mathbf{100000}, \cdots, 6000000$
r	Number of dimensions	$5, \mathbf{10}, \cdots, 30$
q	The number of microclusters	$50, \cdots, \mathbf{200}, \cdots, 800$
$iterations$	The number of iterations to run the micro-clustering kernel	$1, \cdots, \mathbf{4}, \cdots, 16$
Offset	The relative difference between outlier and non-outlier objects (synthetic data)	$1.5, \mathbf{2.0}, 2.5$
Outlier dimensions	The number of dimensions the outlier is significant in (synthetic data)	$20\%, 50\%, \mathbf{100}\%$
Outlier ratio	The relative amount of outliers in the data set (synthetic data)	$1\%, \mathbf{5}\%, 25\%$
Uncertainty multiplier	The amount to adjust underlying uncertainty by	$1.0, 1.5, \mathbf{2.0}, 2.5, 3.0$

Table 3: Parameters adjusted in testing, along with the range tested. The default value is in **bold**

Experimental testing was conducted on a PC running Microsoft Windows Vista SP2 with an Intel Core 2 Duo E8200 dual core CPU and an NVIDIA GeForce GT440 GPU (with 96 processors). The serial and host code was compiled using Microsoft Visual Studio 2010. The OpenCL code was compiled and run using NVIDIA CUDA Toolkit 4.2 and driver 301.32 for the GPU, and AMD APP 2.6 for the CPU. The serial implementation uses standard C++ math functions, while the OpenCL implementation uses native math functions. Both implementations use single precision floating point variables.

5.2 Kernel efficiency

To determine the efficiency of the parallel implementation, the two key factors to consider are instruction or compute throughput and memory bandwidth. In Table 4 we compare the theoretical maximum of the hardware used to the measured throughput and bandwidth of the two kernels. The running time of global memory access and floating point computation in isolation [32] is also given.

	Theoretical max.	Micro-clustering (A)	Outlier detection
Global memory access time (s)	–	0.384	0.00123
Floating point computation time (s)	–	0.0251	0.0129
GFLOPS	311	33.6	183
Bandwidth (GB/s)	51.2	12.2	3.49

Table 4: Theoretical performance values for the GPU as well as empirically determined values achieved by the micro-clustering and outlier detection kernels

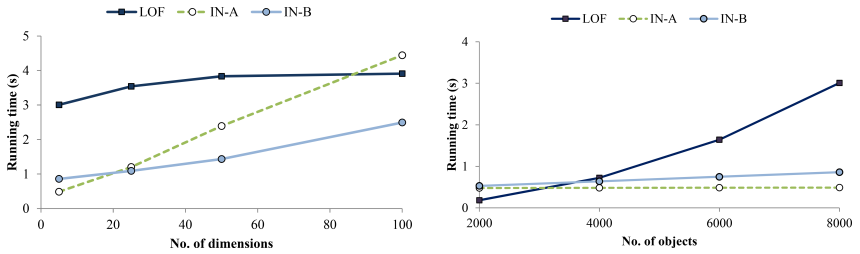
From the running times it is clear that the micro-clustering step is memory-intensive while the outlier detection step is compute-intensive. This is as expected since the micro-clustering process is of relatively low arithmetic intensity but operates on the entire dataset. The outlier detection step operates on the compressed representation of the data and thus does not require much memory bandwidth, however has many floating point operations including GPU efficient single precision FMAD (floating point multiply-add).

Two common metrics for GPU performance are: (i) throughput, measured in GFLOPS (billions of floating point operations per second) and (ii) bandwidth, measured in GB/s. The theoretical maximum performance specified by the vendor can only be achieved under ideal circumstances, by solely using single precision FMAD and sequential coalesced memory access patterns without control flow. In practice, the intensity of floating point operations varies significantly. To estimate GFLOPS, it is required to count the relevant instructions generated by the compiler. Since the version of the compiler and the settings used can have a significant impact on the number of FLOPS, this number serves only as an approximate indication of relative performance levels in this particular test setting.

With respect to the throughput of the outlier detection step, consideration should be given to the non-exclusive use of FMAD instructions and the effect of branch divergence (measured in this case at 13.6%) that occurred when dealing with outliers. It is also evident that while micro-clustering is memory intensive, there is a considerable amount of potential memory bandwidth remaining. One limiting factor is that while reading the data objects occurs sequentially and can be coalesced effectively, updating the cluster information is inevitably random access. Another is that at the end of the clustering iteration, in most cases a reduced number of threads will be active per warp. In such a case, loop unrolling may increase efficiency.

5.3 Performance on synthetic datasets

First, we evaluate the performance of the LOF algorithm compared to our proposed algorithm. As we were not able to obtain the original CUDA implementation of LOF, we have re-implemented LOF in OpenCL following [6]. As such, there is no support for uncertainty modeling. For this test LOF is compared against the independent algorithm with two different implementations of micro-clustering, labeled 'IN-A' (Algorithm 5) and 'IN-B' (Algorithm 6) respectively. This test is carried out on the 'one' dataset with dataset size and dimensionality within the parameters used in [6].



(a) Varying no. of dimensions vs. running time

(b) Varying no. of objects vs. running time

Fig. 9: Running time of LOF compared to implementations of 'independent'

While the serial implementation of LOF has a high complexity, LOF is known to scale well with dimensionality. Fig. 9(a) shows the LOF algorithm scales better with dimensionality than either of the IN implementations, particularly IN-A, although it is possible to switch to IN-B in the case of data of higher dimensionality. However as shown in Fig. 9(b), for large datasets of moderate dimensionality IN-A is consistently faster. LOF in particular has relatively high linear complexity, although it is an improvement from the serial complexity of $\mathbf{O}(n^2)$. Note that to streamline testing, for the remainder of this paper the 'A' implementation is used for our algorithm.

For reference, our preliminary work demonstrated a basic implementation of Algorithm 4 without micro-clustering (labeled 'DO' in Fig. 10). Although our test parameters have changed since our earlier work, the performance difference between our new implementation and the preliminary implementation is significant.

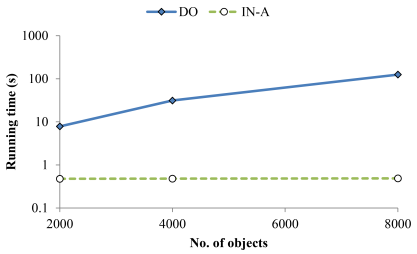
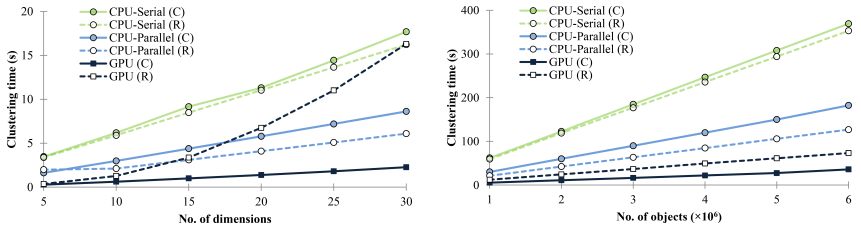


Fig. 10: Running time of our preliminary implementation compared to this implementation

Next we demonstrate Algorithm 5 on different platforms (parallel on the GPU, parallel on the CPU and serial on the CPU), to determine the impact of data array orientation on memory access times.

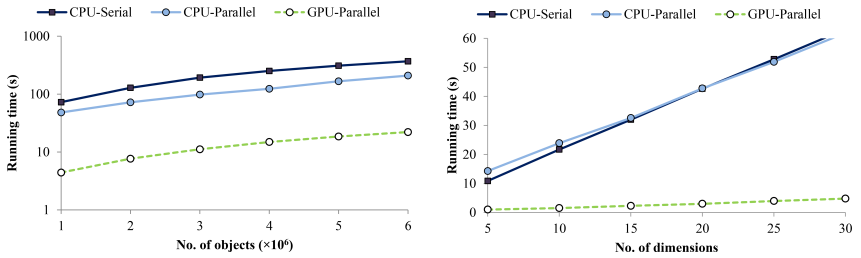


(a) Memory layout with no. dimensions (b) Memory layout with no. objects

Fig. 11: Running time comparison between (R)ow-major and (C)olumn-major memory layout

With respect to dimensionality in Fig. 11(a), it is apparent that there is a significant improvement in the GPU performance scaling (from super-linear to linear), allowing the GPU to be fastest in all tests. There is also a moderate negative impact on the parallel CPU implementation and a minor negative impact on the serial implementation, possibly due to caching effects. The performance differences are less dramatic when considering the number of objects in Fig. 11(b) — scaling modes remain consistent. To streamline testing, subsequent performance tests are conducted solely using the column-major memory arrangement.

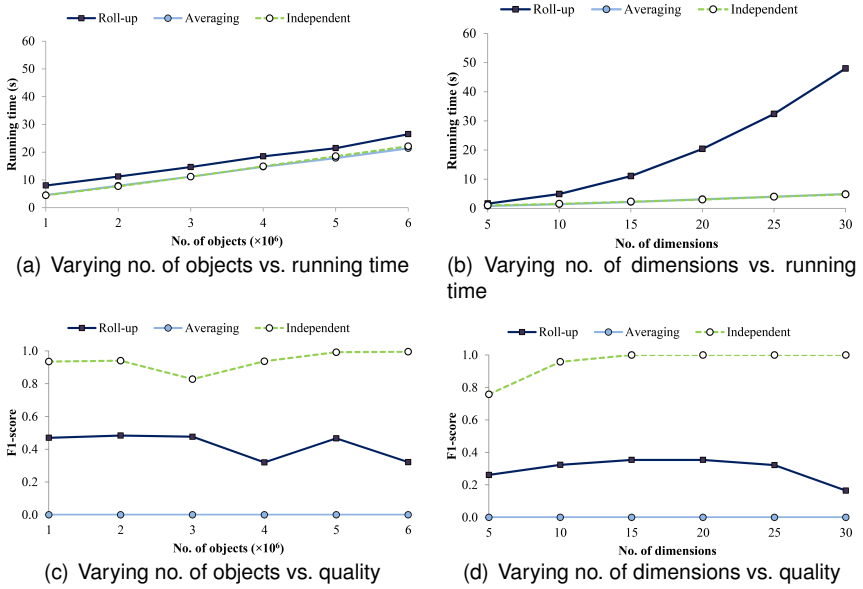
Next is a performance comparison of the different methods on the GPU, adjusting the number of dimensions and objects and measuring the total running time of the system.



(a) Varying no. of objects vs. running time (b) Varying no. of dimensions vs. running time

Fig. 12: Running time of ‘independent’ on different platforms

Fig. 12(a) shows a performance comparison between the implementations of the proposed method on three platforms: GPU-Parallel, CPU-Parallel and CPU-Serial. As expected, the GPU implementation is significantly faster (between $8\times$ and $16\times$ faster) in both clustering and outlier detection at all times compared to the CPU implementations. Fig. 12(b) shows the GPU implementation perform even better rela-



(a) Varying no. of objects vs. running time

(b) Varying no. of dimensions vs. running time

(c) Varying no. of objects vs. quality

(d) Varying no. of dimensions vs. quality

Fig. 13: Running time and quality comparison of the three tested algorithms

tive to the other implementations; in fact the CPU-Parallel implementation performed slightly below the baseline serial implementation, however as noted previously CPU-Parallel is negatively affected by the GPU optimized memory arrangement. Even after compensating for the memory arrangement the CPU-Parallel implementation remains significantly slower than the GPU implementation however. Note that for subsequent tests, we restrict testing to the GPU implementations.

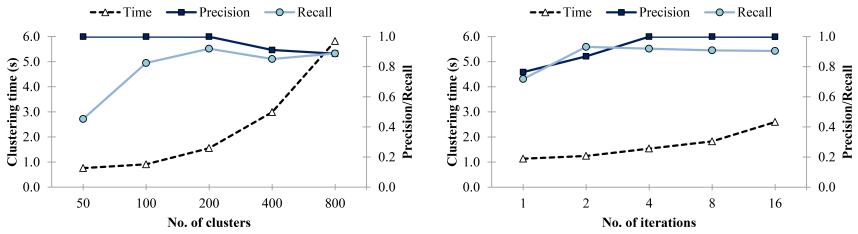
Figs. 13(a) and 13(b) show a performance comparison between the three methods (independent, restricted roll-up and global averaging). Fig. 13(a) shows the expected linear scaling with object count, since the clustering is the limiting step. Our proposed method achieves very similar performance to global averaging, while restricted roll-up is slightly slower.

Fig. 13(b) shows a different outcome, with the roll-up method exhibiting super-linear scaling that results in long processing times even with a GPU. 'independent' on the other hand exhibits significantly better linear scaling with dimensionality. As noted previously, with large numbers of objects clustering is the performance bottleneck in the system. As a quick test of quality, Figs. 13(c) and 13(d) demonstrate that despite having a performance profile similar to simple global averaging, 'independent' has the best detection quality (using the same parameters as the performance test). Further quality testing is conducted in Sect. 5.4.

Note that while the maximum size of the dataset tested in this paper and in [4] (6 million objects and 10 dimensions) fits in the global memory of the tested GPU, it is clear that larger datasets may not fit as conveniently. This is not an issue as long as $\mathcal{C}$ can be stored in memory, as the dataset can be easily partitioned and the initial centroids are randomly chosen from the entire dataset. Since relatively few clusters

are required for reasonable representation of the data (see Fig. 14(b)), it is likely $\mathcal{C}$ would fit in global memory.

The final set of performance tests focus on the trade-off between performance and quality resulting from adjusting the number of clusters and clustering iterations. Since the quality of the outlier detection depends on the quality of the clustering and yet the clustering step can be a performance bottleneck, it is important to strike a balance between quality and speed.



(a) Run time & quality vs. no. of clusters (b) Run time & quality vs. no. of iterations

Fig. 14: Precision, recall and clustering times of 'independent'

Looking at performance in Fig. 14(a), the number of clusters q has a significant impact on running time, with super-linear scaling that was also observed in our earlier work [31]). With regards to quality however, there is a clear peak in quality before plateauing. Varying the number of centroid recalculation iterations as shown in Fig. 14(b) has a lesser impact on performance and quality, although again there is a clear point where additional iterations do not improve quality. Note that quality results using the 'three' dataset is not shown here since its underlying distribution lends itself to clustering rather easily.

5.4 Quality on synthetic datasets

As noted previously, performance testing has been conducted on synthetically generated 'one' and 'three' datasets based on datasets demonstrated in [6]. The 'three' dataset is a more stereotypical outlier detection scenario where there are high density clusters of valid objects surrounded by a low density cloud of outliers. The 'one' dataset, despite having fewer distributions, is a more complicated scenario since there is significantly more overlap between outliers and non-outliers, with outliers and non-outliers having similar distribution characteristics and with a smaller difference in density.

To test quality with synthetic data, the outliers in the datasets are manipulated to determine the influence of the data on the outlier detection ability of the three methods. F1-score (harmonic mean of precision and recall) is used as an aggregate indication of detection quality. In the case an empty outlier set is returned (i.e. a failure to detect any outliers) a zero F1-score is recorded.

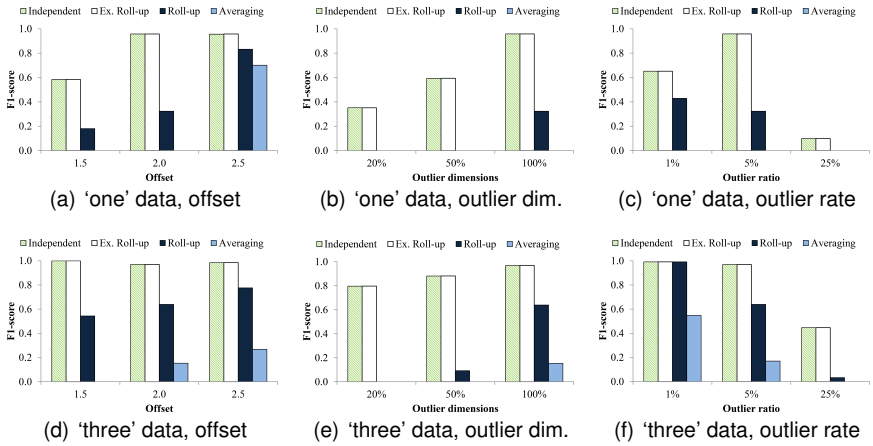


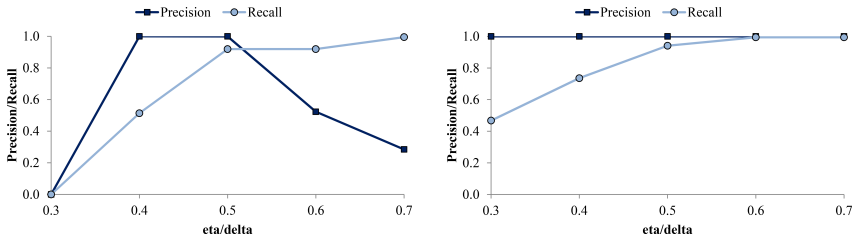
Fig. 15: F1 scores of the tested algorithms with varying offset, outlier dimensionality and outlier rate

Figs. 15(a) and 15(d) demonstrate the effect of overlap between outliers and non-outliers, by adjusting the mean difference between them. It is apparent that the proposed independent method is capable of matching the detection quality of the exhaustive roll-up method and is significantly better than the restricted roll-up method or the simple global averaging method at all offsets, but particularly at lower offsets (i.e. higher overlap). The global averaging method is particularly poor at dealing with the 'three' dataset with localized areas of high density.

Figs. 15(b) and 15(e) demonstrate the effect of the number of outlier dimensions, that is, changing the number of dimensions the outlier is visible in from 20% to all dimensions. Again, the proposed independent method matches the expensive exhaustive roll-up method and is significantly better than the either restricted roll-up or global averaging.

Figs. 15(c) and 15(f) demonstrate the effect of changing the ratio of outliers to non-outliers, from few outliers (1%) to a significant portion (25%). It is apparent that all the methods decline in detection ability as the outlier rate increases — it is harder to learn the difference between outlier and non-outlier objects the less exceptional the outliers are. The exception to the pattern is the 1% case in Fig. 15(c), where the independent and exhaustive roll-up methods have an elevated false positive rate due to the greater influence of uncertainty in lower dimensional subspaces.

The algorithm parameters η and δ, as defined in Sect. 3.2.1, control the algorithm's sensitivity to outliers. η defines the minimum density for non-outlier objects, while δ defines the minimum ratio of samples that are required for non-outlier objects. In this paper however, η and δ are kept equal to streamline testing. The 'independent' method is used in these tests. Figs. 16(a) and 16(b) show the effect of varying η and δ on the 'one' and 'three' datasets. In the 'one' dataset there is a clear peak in precision and recall before precision is lost (more false positives). The 'three' dataset is shown here to have easily determined outliers, as false positives are not an issue at all tested parameter values.



(a) 'one' dataset (b) 'three' dataset

Fig. 16: Precision and recall of 'independent' with varying η and δ

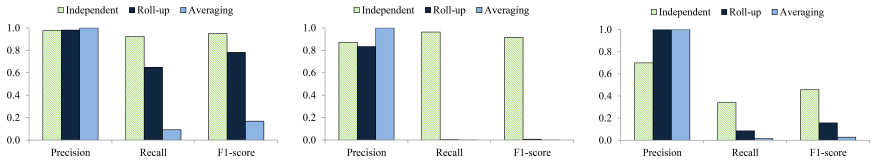
	Independent	Exhaustive Roll-up	Restricted Roll-up	Global Averaging
Average running time (s)	1.533	482.5	4.888	1.436

Table 5: Average running times of tested methods in this section

For reference, Table 5 shows the average total running time of all the tested methods, including exhaustive roll-up, on the GPU. Since the micro-clustering time is constant, the difference is due to the number of subspaces explored by each method.

5.5 Quality and performance with real datasets

We also test the detection quality and performance of our algorithm with three commonly used datasets from the UCI Machine Learning Repository: landsat, shuttle and cancer. Since these datasets are not specifically intended for outlier detection, the problem is changed slightly to finding a single class [8] [15] [37]. Landsat has a relatively few 4435 objects with 36 dimensions each (10.8% target class), while shuttle has 36752 objects with only 9 dimensions each (7.2% target class). The cancer dataset is an atypical dataset with 569 objects with 30 dimensions, but with a 37% target class. Note that for the shuttle dataset, one class of objects were removed to reduce the 'outlier' rate, however the cancer dataset was left unedited.



(a) Algorithms on the 'landsat' dataset (b) Algorithms on the 'shuttle' dataset (c) Algorithms on the 'cancer' dataset

Fig. 17: Comparison of algorithms on UCI ML datasets

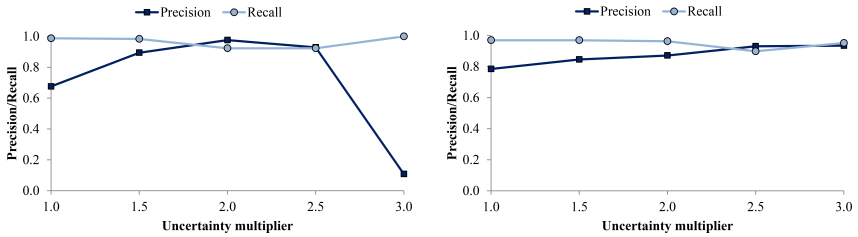
With respect to outlier detection quality with constant parameters, Figs. 17(a) to 17(c) demonstrate that the independent method performed the best overall — although all the algorithms had high precision, our proposed method had the highest

Time (s)	Independent	Roll-up	Averaging
landsat	4.670	68.17	4.669
shuttle	1.178	3.951	1.130
cancer	3.786	46.82	3.775

Table 6: Average running times of tested methods on UCI ML datasets

recall with both a large number of features and a small number of features. The exception is with the cancer dataset where the high outlier rate reduced the effectiveness of the algorithms significantly.

In terms of performance, as shown in Table 6, independent and averaging are similar in running time, however independent offers significantly higher outlier detection quality. Roll-up is significantly slower and scales particularly poorly with higher dimensionality datasets, even in the simpler restricted version.



(a) Uncertainty on the 'landsat' dataset (b) Uncertainty on the 'shuttle' dataset

Fig. 18: Precision and recall of 'independent' with varying uncertainty multipliers

Finally we look at the effect of uncertainty on outlier detection quality. Note that uncertainty for these certain datasets is generated using the standard deviation of each dimension, multiplied by a uniformly distributed random number and a multiplier (default 2.0), so the amount of uncertainty depends on both the underlying standard deviation and the chosen uncertainty multiplier. The default tested sensitivity parameters η and δ are set for a moderate amount of uncertainty. We do not look at the cancer dataset here due to the unusual outlier rate mentioned previously.

The landsat dataset has a moderate base uncertainty of 0.29, and Fig. 18(a) shows that the tested settings are a good compromise. Quality drops off at the extreme settings as the multiplier has a larger effect on the higher base uncertainty.

The shuttle dataset has a low base uncertainty of 0.063, and Fig. 18(b) shows an upward trend in quality as the uncertainty multiplier is increased. This suggests that the tested sensitivity parameters are too high for this dataset.

6 Conclusion

In this paper we demonstrated an improved method for outlier detection on uncertain data, implemented using the OpenCL framework for parallel execution on GPUs. Our proposed method addresses the baseline roll-up method's poor performance with

high dimensionality data, while matching the detection quality of the computationally expensive roll-up method. With the test platform used in this paper, the pure GPU implementation also significantly outperformed the CPU implementations in parallel and serial, in both micro-clustering and outlier detection components.

One bottleneck identified in testing was memory access and the efficient use of available memory bandwidth. Data mining is typically a data intensive application, and the high bandwidth memory on the GPU can be well suited to this application. While reads from global memory are arranged in an efficient fashion, writing cluster information depends on random access and would benefit from an enhanced data structure, utilizing operations such as scatter-gather.

Another potential issue is buffer size restrictions on the maximum size of the dataset. While it is reasonable to partition the dataset and complete the micro-clustering in this fashion, overall system efficiency could be improved by leveraging a heterogeneous computing approach that would enable both a pipeline of operations (multiple kernels executing simultaneously on different devices) and reduce overhead from transfers over the PCIe bus between the host and GPU.

One possible area for improvement is the way uncertainty is modeled. In this work a Gaussian distribution of randomness is assumed, however there are two key concerns with this model: in many cases a distribution can be significantly skewed, and a random model does not take into account any underlying patterns in the data. In future work we propose to investigate learning the uncertainty model, for instance from time series data.

Another key area is selection of subspaces for testing. Although micro-clustering reduces the search space considerably, the dimensionality of the space is not affected. Although we include implementations to deal with either large or high dimensional datasets. feature selection and dimensionality reduction could increase efficiency considerably when faced with very large and high dimensional datasets.

Beyond outlier detection, the micro-clustering compression approach is a useful compressed representation of a large dataset that can be applied to other data mining tasks such as classification and similarity search. The GPU optimization strategies used are also broadly applicable to any data parallel processing system. With the growth in heterogeneous computing systems that integrate both CPU and GPU-like features, this massively parallel methodology will continue to be pursued going forward.

References

1. Acklam, P.J.: An algorithm for computing the inverse normal cumulative distribution function. Tech. rep. (2003)
2. Advanced Micro Devices, Inc: AMD accelerated parallel processing opencl programming guide
3. Aggarwal, C.C. (ed.): Managing and Mining Uncertain Data. Springer, New York, NY (2009)
4. Aggarwal, C.C., Yu, P.S.: Outlier detection with uncertain data. In: Proceedings of the SIAM International Conference on Data Mining 2008 (2008)
5. Aggarwal, C.C., Yu, P.S.: A survey of uncertain data algorithms and applications. IEEE TKDE **21(5)**, 609–623 (2009)
6. Alshawabkeh, M., Jang, B., Kaeli, D.: 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 (2010)

7. Altera Corporation: Altera SDK for OpenCL programming guide (2014)
8. Azmandian, F., Yilmazer, A., Dy, J.G., Aslam, J.A., Kaeli, D.R.: GPU-accelerated feature selection for outlier detection using the local kernel density ratio. In: Proceedings of the 12th IEEE ICDM (2012)
9. Bastke, S., Deml, M., Schmidt, S.: Combining statistical network data, probabilistic neural networks and the computational power of GPUs for anomaly detection in computer networks. In: 1st Workshop on Intelligent Security (Security and Artificial Intelligence) (2009)
10. Bolton, R.J., Hand, D.J.: Statistical fraud detection: A review. Statistical Science **17(3)**, 235–255 (2002)
11. Breunig, M.M., Kriegel, H.P., Ng, R.T., Sander, J.: LOF: identifying density-based local outliers. In: Proceedings of SIGMOD 2000 (2000)
12. Chau, M., Cheng, R., Kao, B., Ng, J.: Uncertain data mining: An example in clustering location data. In: Proceedings of the 10th PAKDD (2006)
13. Denoeux, T.: Maximum likelihood estimation from uncertain data in the belief function framework. IEEE TKDE **25(1)**, 119–130 (2013)
14. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd Int. Conf. on Knowledge Discovery and Data Mining (1996)
15. F. Angiulli, S.B., Pizzuti, C.: Distance-based detection of outliers. IEEE TKDE **18(2)**, 145–160 (2006)
16. Fang, W., Lau, K.K., Lu, M., Xiao, X., Lam, C.K., Yang, P.Y., He, B., Luo, Q., Sander, P.V., Yang, K.: Parallel data mining on graphics processors. Tech. rep., Hong Kong University of Science and Technology (2008)
17. He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: Proceedings of the ACM/IEEE Conference on Supercomputing (2007)
18. Hung, E., Cheung, D.W.: Parallel mining of outliers in large database. Distributed and Parallel Databases **12(1)**, 5–26 (2002)
19. Intel Corporation: Intel SDK for OpenCL applications XE 2013 R2 – optimization guide (2013)
20. Kao, B., Lee, S.D., Cheung, D.W., Ho, W.S., Chan, K.F.: Clustering uncertain data using voronoi diagrams. In: Proceedings of the 8th IEEE ICDM (2008)
21. Khronos Group: OpenCL. `http://www.khronos.org/opencl/` (2011). Last accessed 9-Oct-2012
22. Knorr, E.M., Ng, R.T.: Algorithms for mining distance-based outliers in large datasets. In: Proceedings of VLDB 1998 (1998)
23. Knorr, E.M., Ng, R.T.: Finding intensional knowledge of distance-based outliers. In: Proceedings of VLDB 1999, pp. 211–222 (1999)
24. Kriegel, H.P., Pfeifle, M.: Density-based clustering of uncertain data. In: Proceedings of the 11th ACM SIGKDD (2005)
25. Kriegel, H.P., Pfeifle, M.: Heirarchical density-based clustering of uncertain data. In: Proceedings of the 5th IEEE ICDM (2005)
26. Krulis, M., Skopal, T., Lokoc, J., Beecks, C.: Combining CPU and GPU architectures for fast similarity search. Distributed and Parallel Databases **30**, 179–207 (2012)
27. Lan, Z., Zheng, Z., Li, Y.: Toward automated anomaly identification in large-scale systems. IEEE TPDS **21(2)**, 174–187 (2010)
28. Lozano, E., Acuna, E.: Parallel algorithms for distance-based and density-based outliers. In: Proceedings of the 5th IEEE ICDM (2005)
29. Lu, M., Tan, Y., Bai, G., Luo, Q.: High-performance short sequence alignment with GPU acceleration. Distributed and Parallel Databases **30**, 385–399 (2012)
30. Marsaglia, G.: Xorshift RNGs. Journal of Statistical Software **8(14)**, 1–6 (2003)
31. Matsumoto, T., Hung, E.: Accelerating outlier detection with uncertain data using graphics processors. In: Advances in Knowledge Discovery and Data Mining, vol. LNCS 7302, pp. 169–180 (2012)
32. Micikevicius, P.: Analysis-driven optimization. In: GPU Technology Conference 2010 (2010)
33. Murakami, T., Kasahara, R., Saito, T.: An implementation and its evaluation of password cracking tool parallelized on GPGPU. In: Proceedings of the 2010 International Symposium on Communications and Information Technologies (2010)
34. Ngai, W.K., Kao, B., Chui, C.K., Cheng, R., Chau, M., Yip, K.Y.: Efficient clustering of uncertain data. In: Proceedings of the 6th IEEE ICDM (2006)
35. NVIDIA Corporation: NVIDIA's next generation CUDA compute architecture: Fermi (2009)
36. NVIDIA Corporation: CUDA. `http://www.nvidia.com/object/cuda_home_new.html` (2011). Last accessed 9-Oct-2012

37. Reif, M., Goldstein, M., Stahl, A.: Anomaly detection by combining decision trees and parametric densities. In: 19th International Conference on Pattern Recognition 2008 (2008)
38. S. Ramaswamy, R.R., Shim, K.: Efficient algorithms for mining outliers from large data sets. In: Proceedings of the 2000 ACM SIGMOD (2000)
39. Sequeria, K., Zaki, M.: ADMIT: Anomaly-based data mining for intrusions. In: Proceedings of the 8th ACM SIGKDD (2002)
40. Tang, J., Chen, Z., Fu, A.W., Cheung, D.W.: Capabilities of outlier detection schemes in large datasets, framework and methodologies. Knowledge and Information Systems **11(1)**, 45–84 (2006)
41. Tarabalka, Y., Haavardsholm, T.V., Kaasen, I., Skauli, T.: Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing. Journal of Real-Time Image Processing **4(3)**, 287–300 (2009)
42. Wang, L., Cheung, D.W.L., Cheng, R., Lee, S.D., Yang, X.S.: Efficient mining of frequent item sets on large uncertain databases. IEEE TKDE **24(12)**, 2170–2183 (2012)
43. Zhanchun, G., Yuying, L.: Improving the collaborative filtering recommender system by using GPU. In: Proceedings of 2012 Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery (2012)
44. Zhang, Y., Lin, X., Tao, Y., Zhang, W., Wang, H.: Efficient computation of range aggregates against uncertain location-based queries. IEEE TKDE **24(7)**, 1244–1258 (2012)