Accelerating Outlier Detection with Uncertain Data Using Graphics Processors

Takazumi Matsumoto and Edward Hung

Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong
{cstmatsumoto,csehung}@comp.polyu.edu.hk

Abstract. Outlier detection (also known as anomaly detection) is a common data mining task in which data points that lie outside expected patterns in a given dataset are identified. This is useful in areas such as fault detection, intrusion detection and in pre-processing before further analysis. There are many approaches already in use for outlier detection, typically adapting other existing data mining techniques such as cluster analysis, neural networks and classification methods such as Support Vector Machines. However, in many cases data from sources such as sensor networks can be better represented with an uncertain model. Detecting outliers with uncertain data involves far more computation as each data object is usually represented by a number of probability density functions (*pdf*s).

In this paper, we demonstrate an implementation of outlier detection with uncertain objects based on an existing density sampling method that we have parallelized using the cross-platform OpenCL framework. While the density sampling method is a well understood and relatively straightforward outlier detection technique, its application to uncertain data results in a much higher computational workload. Our optimized implementation uses an inexpensive GPU (Graphics Processing Unit) to greatly reduce the running time. This improvement in performance may be leveraged when attempting to detect outliers with uncertain data in time sensitive situations such as when responding to sensor failure or network intrusion.

1 Introduction

In recent years there has been increased interest in mining uncertain data [1]. A significant amount of data collected, such as from temperature sensors, contain some degree of uncertainty, as well as possibly erroneous and/or missing values. Some statistical techniques such as privacy-preserving data mining may deliberately add uncertainty to data. In addition, with the proliferation of affordable, capacious storage solutions and high speed networks, the quantity of data collected has increased dramatically. To quickly deal with the large quantities of mostly uninteresting data, outlier detection is a useful technique that can be used to detect interesting events outside of typical patterns. However, uncertainty adds greatly to the complexity of finding outliers as uncertain objects are not represented by a single point, but rather a probabilistic object (i.e. the point could be anywhere in the given space with some probability). This increase in complexity leads to the problem of reduced scalability of algorithms to larger amounts of data.

Within a similar time frame, multi-core processors and most recently general purpose computing using graphics processors (GPGPU) have become popular, cost effective approaches to provide high performance parallel computing resources. Modern GPUs are massively parallel floating point processors attached to dedicated high speed memory – for a fraction of the cost of traditional highly parallel processing computers. Programming frameworks such as NVIDIA CUDA and OpenCL now allow for programs to take advantage of this previously underutilized parallel processing potential in ordinary PCs and accelerate computationally intensive tasks beyond typical applications in 3D graphics, seeing use in scientific, professional and home applications (such as video encoding).

Our contributions in this paper are a modified density sampling algorithm and implementations for fast parallel outlier detection with uncertain data using this parallel processing resource. Our implementation has been optimized for the features and restrictions of the OpenCL framework and current GPUs.

This paper is organized as follows: Sect. 2 briefly covers related work in the field of outlier detection with uncertain data as well as other GPU accelerated outlier detection techniques. Sect. 3 describes our modified algorithm used in this paper for outlier detection with uncertain data. Sect. 4 details our implementation of the algorithm, with attention to key points in parallelization and optimization using the OpenCL framework. Sect. 5 describes our testing methodology and demonstrates the effectiveness of our OpenCL-based approach in greatly improving performance using both GPU and CPU hardware. Sect. 6 summarizes our contributions and concludes this paper.

2 Related Work

Outlier detection is a well established and commonly used technique for detecting data points that lie outside of expected patterns. The prototypical approach to outlier detection is as a by-product of clustering algorithms [2]. In a clustering context, an algorithm such as DBSCAN [3] will exclude data points that are not close (given an appropriate metric such as distance or density) to other objects. Later, more approaches were proposed for outlier detection, such as Local Outlier Factor (based on k-nearest-neighbors), Support Vector Machines, and neural networks.

Data mining applications such as outlier detection are also candidates for parallelization to reduce running time [4] as in typical cases there is a large amount of data that is processed by a small number of routines, possibly in real-time or interactively (for example, in an intrusion detection system). These tasks are said to be 'data parallel', and such tasks are well suited for execution on a GPU [2]. Unlike conventional parallel processing computers that have many complex CPU cores, a modern GPU consists of a large number of simple 'stream processors' that are individually capable of only a few operations. However, the ability to pack many stream processors into the same space as a single CPU core gives GPUs a large advantage in parallelism. A similarly parallel traditional CPU-based system would be significantly more costly and complex.

The two most popular programming frameworks for GPGPU are C for CUDA, a proprietary solution developed by NVIDIA Corporation, and OpenCL, an open standard backed by multiple companies including Intel, AMD, NVIDIA and Apple. With both

CUDA and OpenCL, work is split from the host (i.e. the CPU) to kernels that execute on a computing device (typically GPUs). Kernels contain the 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.

Several outlier detection algorithms [2] [5] [6] [7] have been adapted for acceleration with GPUs using CUDA and have seen significant reductions in running times (e.g. a hundred fold improvement in [2]). In this paper, we opt to use OpenCL, as it provides a high degree of portability between different manufacturers of GPUs, as well as the ability to execute the same parallel code on a CPU for comparison.

While there is already a large body of work in the area of accelerating outlier detection on regular (certain) data, often in real world cases data collected has some degree of uncertainty or error [1], for instance, a network of temperature sensors monitoring a greenhouse. Moreover, some statistical techniques such as forecasting and privacy preserving data mining will naturally be uncertain. These uncertainties can be represented by a number of common probability density functions (e.g. Gaussian distribution), which offer a convenient closed form representation. However, sampling a *pdf* for outlier detection using a typical distance or density based approach will result in greatly increased running time due to the computational load from calculations from all the sampled points (e.g. LOF has a complexity of $O(n^2)$ [2], where n would in this case be the total number of samples).

The work in [8] introduces outlier detection on uncertain data as records in a database, with each record having a number of attributes (dimensions). Each dimension has a *pdf*, and the objective is to find data points in an area with data density η (expressed as the η-probability of a data point) less than a threshold value δ, i.e. a (δ, η)-outlier. As it is assumed that outliers have low density in some subspace of the data, each subspace is explored and in each subspace and outliers are removed.

It is noted [8] that it is impractical to determine η-probabilities directly, so a sampling approach of the *pdf*s is used. As the sampling process is a very time consuming operation, [8] also proposes a 'microclustering' technique to reduce the number data objects into clusters. However, in this paper we do not explore microclustering of the data to focus on the performance of density sampling on a GPU.

3 Algorithm for Outlier Detection with Uncertain Data

In this paper, we propose modification of the density sampling algorithm proposed in [8] to optimize it for GPU acceleration. We will first recap the outlier detection approach and terminology.

Let dataset $\mathcal{D}$ contain n uncertain data objects, with each object d_i having m dimensions. For each object, each dimension has a *pdf* that is assumed to be independent. Each object is represented by its mean value $\bar{X}_i$. The *pdf* for $\bar{X}_i$ along dimension j is denoted by $h_i^j(\cdot)$ and the standard deviation of $h_i^j(\cdot)$ is denoted $\psi_j(\bar{X}_i)$. In the case of data stored in certain form (i.e. without standard deviation), uncertainty can be estimated from the calculated standard deviation of each dimension using the Silverman approximation suggested in [8].

It is defined that the η-probability of object $\bar{X}_i$ is the probability that $\bar{X}_i$ lies in a subspace with overall data density of at least η. A subspace is defined as the objects in a subset of the m dimensions, while overall data density is defined by *pdf* s of each object. The probability p_i of $\bar{X}_i$ in a subspace of dimensionality r with overall data density of at least η can be found by solving the following integral:

$$p_i = \int_{h(x_1,\ldots,x_r)} \prod_{j=1}^{r} h_i^j(x_j)\, dx_j \tag{1}$$

Note that $h(x_1, \ldots, x_r)$ represents the overall probability density function on all coordinates in the given subspace. However, as p_i is difficult to calculate precisely with Eq. 1, it can be estimated using a density sampling algorithm, using s samples:

```
*EstimateProbability*($d_i$, $\eta$, $r$, $s$)
Let $F_i^j(\cdot)$ be the inverse cumulative distribution function of *pdf* $h_i^j(\cdot)$
$success = 0$, $runs = 0$
**for** $s$ times **do**
  **for** $j = 1$ to $r$ **do**
    $y =$ a uniform random value $[0, 1]$
    **for** all $d_k$ in $\mathcal{D}$ **do**
      $density_i = density_i + h_k^j(F_i^j(y))$
    **end for**
    $density_i = density_i / |\mathcal{D}|$
  **end for**
  **if** $density_i > \eta$ **then**
    $success = success + 1$
  **end if**
  $runs = runs + 1$
**end for**
**return** $(success/runs)$
```

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. It is also evident that the requirement to calculate overall data density at each sampled point presents a large computational workload.

Finally, object $\bar{X}_i$ is defined as a (δ, η)-outlier if the η-probability of $\bar{X}_i$ in any subspace is less than a user parameter δ, that is, the object has a low probability of lying in a region of high data density.

The overall algorithm is presented in pseudo code form as follows:

```
*DetectOutlier*($\mathcal{D}$, $\eta$, $\delta$, $r$, $s$)
$\mathcal{O} = null$
$i = 1$
$\mathcal{C}_i = \{$ First dimension of data points in $\mathcal{D}$ $\}$
**while** $\mathcal{C}_i$ is not empty and $i \leq r$ **do**
  For each object $d \in \mathcal{C}_i - \mathcal{O}$ calculate *EstimateProbability*($d$, $i$, $s$, $\eta$)
  Add any objects with $\eta$-probabilities $< \delta$ to $\mathcal{O}$
```

```
        $\mathcal{C}_{i+1}$ = for each point in $\mathcal{C}_i - \mathcal{O}$ append corresponding dimension $i+1$ from $\mathcal{D}$
        $i = i + 1$
    **end while**
```

Note that the outlier detection algorithm uses a roll-up approach [8], starting with a one dimensional subspace and adding more dimensions for each iteration. Each subspace is tested for outliers and any outliers are discarded from further consideration in other subspaces.

4 Serial and Parallel Implementations

In order to compare performance, we implemented the algorithm described in the previous section in two ways: a traditional serial implementation in C++ (for the CPU) and a parallel implementation optimized for the OpenCL framework (for both CPU and GPU). In this section, we detail the key points to our serial and parallel implementations.

Note that in this paper, we assume all dimensions and their *pdf*s are independent of each other. Density at a single point in a given data object is estimated by taking the *pdf* of all dimensions of all data objects. This process is repeated for each sample of each data object. In this case, the *pdf* is given by the Gaussian function $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. The calculation of the inverse cumulative distribution function (inverse *cdf*) is more complicated due to the absence of a closed form representation. In this implementation, it is calculated numerically using the technique described in [9].

4.1 Serial Methods

Within the serial implementation are two methods referred to as 'iterative' and 'single pass'. As noted previously, the density sampling algorithm uses a roll-up approach to outlier detection starting with a single dimension and adding more dimensions for each subsequent iteration. Since each dimension is considered independent, every combination of subspaces does not need to be considered without loss of generality. Any outliers found in a given subspace are excluded from later subspaces. The 'iterative' method uses this roll-up approach as described in Sect. 3. In contrast, the simpler 'single pass' method only tests the entire problem space, that is, rather than looping through subspaces of dimensionality 1 to r in *DetectOutlier*, one subspace of dimensionality r is tested. This effectively averaging out all the densities in each dimension. As shown in Sect. 5, this has a marked impact on performance (running time) as well as quality.

4.2 Parallel Methods

The parallel OpenCL implementation follows a similar path, with two methods referred to as 'early reject' and 'no early reject'. As described in Sect. 3, 'early reject' generates comparable results to the serial iterative method, and 'no early reject' generates comparable results to the serial single pass approach. However, the OpenCL implementation differs in some key ways to the relatively straightforward serial implementation.

When a kernel executes on a computing device such as a GPU, it should take into account a different architecture to a regular CPU. To better leverage the GPU using

OpenCL, this implementation uses several optimizations such as the use of single precision floating point values and special hardware accelerated mathematical functions (*native* functions). Single precision floating point values are used extensively in graphics, and thus GPUs are optimized for many single precision functions. By avoiding double precision there are significant performance improvements at the cost of a small amount of quality. However this is dependent on the hardware platform, and on our test platform the CPU offered worse performance running OpenCL code with these optimizations. As such, for fairness the CPU OpenCL implementation uses a simpler version that is functionally identical but using double precision and without additional math functions.

The main kernel that is called from *DetectOutlier* on the host contains an implementation of *EstimateProbability*, along with a number of other functions such as the uniform random number generator, as well as calculation of *pdf* and *cdf*. The current OpenCL framework and the GPU imposes certain additional restrictions, such as a lack of recursion (a function calling itself) and lack of dynamic memory allocation on the GPU. This is a problem as refinement methods used in math libraries often use recursion. These were re-written to remove recursion and to take advantage of additional OpenCL functionality (e.g. the complementary error function *erfc*).

In addition, branching logic can cause a reduction in performance and should be avoided where possible, as GPUs must execute the same code path for each worker thread executing in parallel. As such, counter-intuitive methods such as an arithmetic approach to η-density is used and not removing objects already detected as outliers from further calculation until later are used to avoid branching as far as possible. Memory management is also important on a GPU, with the fastest private memory available for each worker thread used as a scratch area and a slower global memory space that can be accessed by all workers used to store the dataset $\mathcal{D}$. As copying data to and from the GPU is an expensive operation, data transfers are minimized and as much preprocessing and processing done on the GPU as possible.

For optimum performance using a GPU, clearly there must be a high level of data parallelism, with many worker threads to divide the problem. In this implementation, each data object is assigned one worker thread, and each worker is responsible for density sampling of that object's space. To further improve parallelism, vectors are used in each worker's private memory to hold the *pdf* variables. This allows multiple dimensions of each object to be operated on simultaneously on hardware that supports vector operations (4 dimensions in this implementation). The preprocessor will zero additional empty dimensions to ensure the vectors are filled. The simplified overview of the modified *EstimateProbability* (no early reject) that executes on each worker thread is as follows:

```
EstimateProbabilityWorker($r$, $s$, $\eta$, $\delta$)
Let $i$ be the data object of the current worker
Let $F_i^j(\cdot)$ be the inverse cumulative distribution function of pdf $h_i^j(\cdot)$
Let vector length $x = r/4$, for vectors of width 4
$prob = 0$
for $x$ times do
    Let zero vectors $successes$, $densities$, $y$ be of length $x$
    $runs = 0$, $subtotal = 0$
    **for** $s$ times **do**
        $y =$ uniformly random variables in $[0, 1]$
        **for** all $d_k$ in $\mathcal{D}$ **do**
            $densities = densities + h_k^j(F_i^j(y_{1,\ldots,x}))$
        **end for**
        $densities = densities/|\mathcal{D}|$
        $successes = successes +$ clamp(ceil($densities - \eta$), 0, 1)
        $runs = runs + 1$
    **end for**
    **if** using 'early reject' **then**
        **for** each vector dimension $l$ **do**
            **if** $(subtotal + successes_l)/l > \delta$ **then**
                $subtotal = (subtotal + successes_l)/l$
            **else**
                $successes_l = 0$
            **end if**
        **end for**
    **else**
        $prob = sum(successes)/(r \times runs)$
    **end if**
**end for**
Copy $prob$ into global memory for host program
```

Note that while the early reject method adds some overhead to check each dimension against δ, the performance impact is negligible unless the vectors are large relative to the number of objects in the dataset. In our testing, there was no detectable no performance difference between the early reject and no early reject methods.

In addition, the *DetectOutlier* loop that calls *EstimateProbability* is replaced with the following:

```
Copy $\mathcal{C}_r$ from the host to the GPU
Call *EstimateProbabilityWorker*($d$, $r$, $s$, $\eta$, $\delta$) for every object $d \in \mathcal{C}_r$
Copy $\eta$-probabilities from the GPU back to the host
Add any objects with $\eta$-probabilities $< \delta$ to $\mathcal{O}$
```

The host loop is thus replaced by multiple workers executing in parallel on the GPU.

The following section shows the results of our testing using a synthetic and a real dataset, as well as the parameters used for optimal results. Note that rather than directly manipulating η and δ, a parameter $uncertainty$ is used both in the generation of synthetic data and to adjust the value of η and δ, compensating for the differences in the underlying standard deviation. When operating on data in which the standard deviation is not known, it can be estimated on a sample of the data by the preprocessor.

5 Experimental Results

The following tests were conducted on a PC running Microsoft Windows Vista SP2 with an Intel Core 2 Duo E8200 dual core CPU and an NVIDIA GeForce GT440 (96 stream processors) GPU. The serial and host code was compiled using Microsoft Visual Studio 2010. The OpenCL code was run using NVIDIA CUDA Toolkit 4.0 and driver 280.26 (OpenCL 1.1) for the GPU, and AMD Stream SDK 2.5 (OpenCL 1.1) for the CPU.

5.1 Performance

To test performance, we generate simple synthetic datasets with a fixed percentage of outliers (10%). In all cases, data objects that are not outliers have a mean value of 0 and a standard deviation of 1, while outliers are offset by 3 (simulating an outlier with high confidence).

In these performance tests, we compare the running time of the serial iterative (CPU-Iterative) and single pass (CPU-Single Pass) methods, as well as the OpenCL implementation on the CPU (CPU-OpenCL) and the GPU. As noted in Sect. 3, the early reject and no early reject methods demonstrated identical performance with the datasets used, so are not shown individually for clarity. All objects in this test have 12 dimensions and use 800 samples per object.

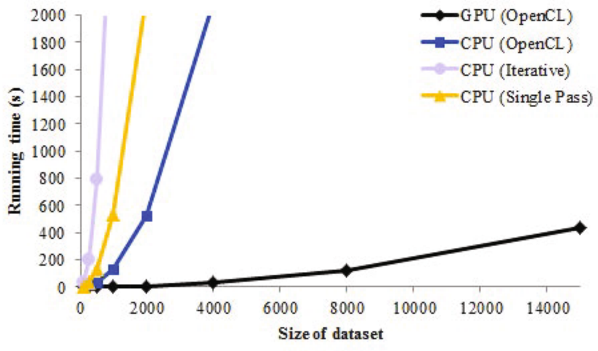

Fig. 1. A comparison of running time with increasing numbers of data objects

It is evident from Fig. 1 that none of the CPU based methods (including the parallel CPU-OpenCL method) offer acceptable performance, with running times increasing rapidly with larger numbers of objects. The GPU offers significant performance improvement over the tested CPU implementations, from 8× at very small sizes against the parallel CPU-OpenCL method up to over 1500× compared against the slowest CPU-Iterative method at larger sizes. Fig. 2 shows the relative increase in running time as dataset size doubles.

The scaling of performance is quadratic with respect to the number of objects in the dataset, due to the algorithm's design. The CPU-based implementations demonstrate this clearly (with some deviation at very small dataset sizes due to overhead).

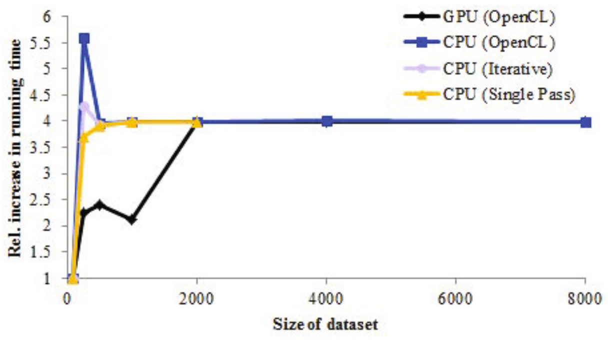

Fig. 2. A comparison of relative increase in running times with each doubling of data objects

At smaller sizes, the GPU demonstrates linear scaling, exceeding the expected quadratic scaling. However the GPU's actual scaling behavior is still quadratic, as the algorithm is unchanged. The processing overhead on the GPU skews performance at smaller dataset sizes, but at sizes exceeding 1000 objects, the worker threads' contention for processing resources and global memory access becomes the performance limiter. It is possible that with the microcluster compression technique described in [8], this behavior can be used advantageously. Overall, the GPU methods maintain a 67× performance improvement over CPU-OpenCL (parallel) and a 273× improvement over CPU-Single Pass (serial).

Figs. 3 and 4 look at two other scaling considerations, the number of dimensions per object and the number of samples per object.

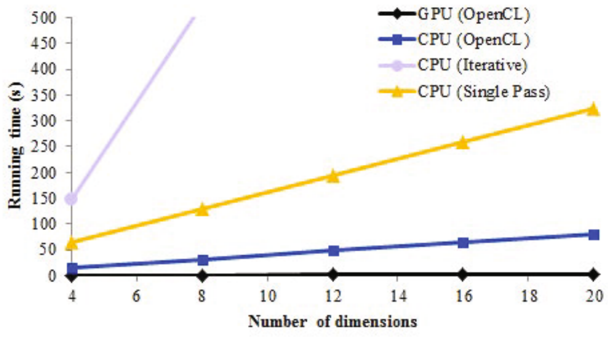

Fig. 3. A comparison of running time with increasing dimensionality

It is clear from Figs. 3 and 4 that the number of dimensions and number of samples offers a strictly linear increase in running time, consistent with the increase in processing load without significant additional memory access load. For the GPU in particular, the increase in running time is negligible.

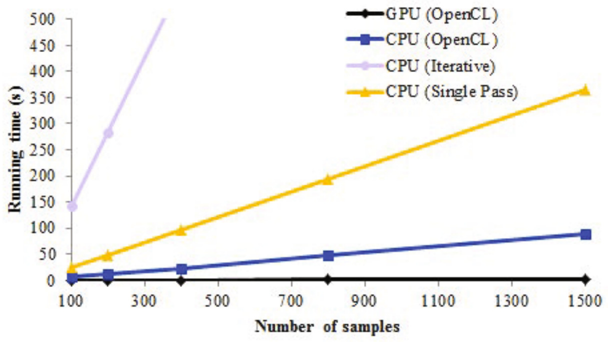

Fig. 4. A comparison of running time with increasing numbers of samples per object

5.2 Quality

Although the focus of this paper has been on performance, the quality of the results must also be acceptable. In the following tests of outlier detection quality, we make the following assumptions: the source data is recorded in a certain form, with data points each having some number of dimensions. To represent the inherent uncertainty, the values of each data point's dimensions are mapped to the mean values of an equivalent uncertain data object's dimensions. The uncertainty of each dimension was estimated from the standard deviation.

To adjust uncertainty in the synthetic dataset, the standard deviation is adjusted in a range from 1 to 3 (i.e. at uncertainty level 3, standard deviation is three times the actual standard deviation). Algorithm parameters η and δ are automatically scaled from 0.3 to 0.6 as uncertainty increases. This is done in an attempt to control the large decline in quality originally seen [8] as uncertainty increases. Although the scaling factors are hard-coded in this implementation, the preprocessor could be extended to better dynamic control of the algorithm parameters and reduce the number of parameters to tune by hand.

The following tests use a real dataset: the Breast Cancer Wisconsin (Diagnostic) Data Set (labeled 'wdbc') from the UCI Machine Learning Repository. This dataset contains 569 records with 30 attributes. This dataset is divided into records marked 'benign' and 'malignant', for the purposes of this test the 'malignant' records are deemed outliers, resulting in a relatively high outlier rate of 37%.

Fig. 5 shows the related parallel and serial methods yielding similar quality, with the CPU methods leading slightly in quality due to the GPU's use of single precision floating point values. Dynamically adjusting the algorithm parameters allows for recall to remain fairly static as uncertainty increases. Note is that the simplest single pass method of averaging density over the entire problem space works well in this relatively small dataset. However as shown in Fig. 6, quality rapidly drops off as more dimensions are added, limiting its usefulness.

For a clearer overview, Fig. 6 represents quality using F1 score, the harmonic mean of precision and recall. It is clear that while adding dimensions results in a slight gain in quality for the iterative and early reject methods, the single pass and no early reject

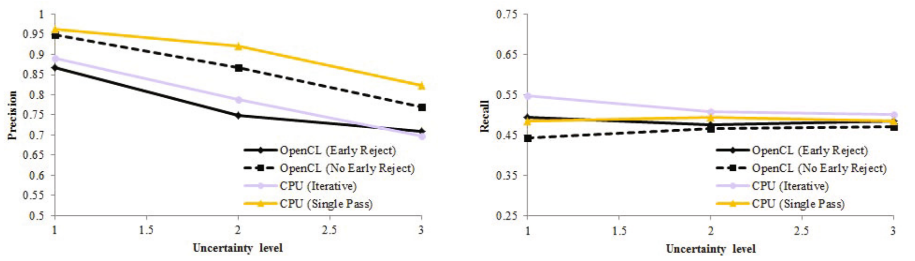

Fig. 5. Precision (left) and Recall (right) with different levels of uncertainty for the ‘wdbc’ dataset

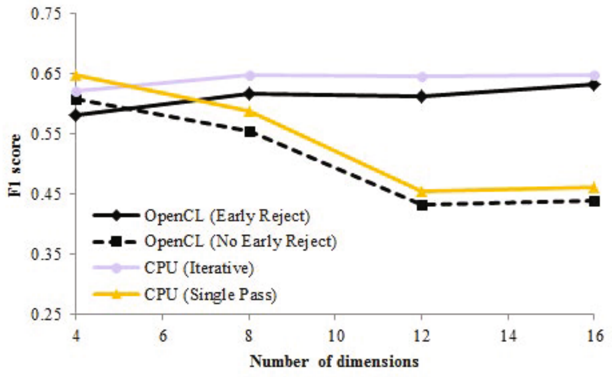

Fig. 6. F1 score with different numbers of dimensions

methods show a significant loss in quality. As outliers are not necessarily of low density in all dimensions, averaging out an object’s density over all dimensions leads to areas of low density being lost, thus recall declines significantly.

6 Conclusion

Through this paper, we have demonstrated the use of a density sampling algorithm for outlier detection on uncertain data can be greatly accelerated by leveraging both a GPU and the OpenCL framework. With our implementation, experimental results demonstrate significant reductions to running time from a worst case very small dataset yielding a $8\times$ performance improvement over the parallel CPU-OpenCL implementation and the best case yielding over $1500\times$ improvement compared to the serial CPU-Iterative method. This could enable large numbers of uncertain objects to be scanned in time critical situations, such as in fault detection on sensor networks.

In the future, we would like to explore other techniques to detecting outliers with uncertain data both in comparison to this implementation and in consideration for more methods that can be parallelized for GPU acceleration. Density and distance based calculations are often used in clustering and outlier detection applications, and there are demonstrable gains from using GPU acceleration in calculation intensive tasks, such as when operating on uncertain data. There are also still opportunities to improve quality and performance (e.g. the microclustering compression technique proposed in [8]), and further testing with more datasets is planned.

Acknowledgments. The work described in this paper was partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (PolyU 5191/09E, PolyU A-SA14).

References

1. Aggarwal, C.C. (ed.): Managing and Mining Uncertain Data. Springer (2009)
2. 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)
3. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discoverying clusters in large spatial databases with noise. In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (1996)
4. Hung, E., Cheung, D.W.: Parallel mining of outliers in large database. Distributed and Parallel Databases 12(1), 5–26 (2002)
5. 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)
6. 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)
7. Huhle, B., Schairer, T., Jenke, P., Strasser, W.: Robust non-local denoising of colored depth data. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Workshop on Time of Flight Camera based Computer Vision (2008)
8. Aggarwal, C.C., Yu, P.S.: Outlier detection with uncertain data. In: Proceedings of the SIAM International Conference on Data Mining 2008 (2008)
9. Acklam, P.J.: An algorithm for computing the inverse normal cumulative distribution function. Technical report (2003)