Fast Outlier Detection Using a GPU

Fabrizio Angiulli*, Stefano Basta†, Stefano Lodi‡ and Claudio Sartori‡
* DIMES-UNICAL, Rende (CS), Italy
Email: f.angiulli@dimes.unical.it
† ICAR-CNR, Rende (CS), Italy
Email: basta@icar.cnr.it
‡ DISI-UNIBO, Bologna, Italy
Email: {stefano.lodi,claudio.sartori}@unibo.it

***Abstract*—The availability of cost-effective data collections and storage hardware has allowed organizations to accumulate very large data sets, which are a potential source of previously unknown valuable information. The process of discovering interesting patterns in such large data sets is referred to as data mining. Outlier detection is a data mining task consisting in the discovery of observations which deviate substantially from the rest of the data, and has many important practical applications. Outlier detection in very large data sets is however computationally very demanding and currently requires high-performance computing facilities. We propose a family of parallel algorithms for Graphic Processing Units (GPU), derived from two distance-based outlier detection algorithms: the *BruteForce* and the *SolvingSet*. We analyze their performance with an extensive set of experiments, comparing the GPU implementations with the base CPU versions and obtaining significant speedups.**

***Keywords*—Data mining exploiting GPUs, outlier detection.**

I. Introduction

In the last twenty years, the availability of cost-effective data collection and storage hardware has induced an unprecedented accumulation of very large data sets in organizations of all dimensions and kinds. types, nature, finality The process whose aim is to discover interesting patterns in such large data sets is referred to as data mining [1]. Many problems related to fundamental data mining tasks, like association rule discovery, data clustering and classifier learning, are difficult [2], [3], and also heuristic and approximate approaches are computationally demanding in practice. For this reason, a large amount of research in data mining has been directed to the design of parallel and distributed algorithms for high-performance computing architectures, in order to cope with the complexity of data mining problems [4]. The vast majority of non-sequential algorithms has been designed for the Distributed Memory Machine (DMM), and utilized on clusters of workstations or parallel supercomputers. Recently, Graphic Processing Units (GPU) with hundreds or thousands of cores have become widely available, and shared memory algorithms for fundamental data mining tasks exploiting the architecture of such many-core graphic processors have been proposed [5], [6]. Outlier detection, or anomaly detection, is a data mining task consisting in the discovery of observations which deviate substantially from the rest of the data, raising the hypothesis that they were generated by a different mechanism [1]. Outlier detection provides information which is readily usable to react in critical situations; therefore, it has many important practical applications in domains such as medical anomaly detection, sensor networks, industrial damage detection, cyber-intrusion detection, fraud detection, image processing, and textual anomaly detection [7]. Outlier detection is also one of the most computationally demanding data mining tasks, and its application to very large data sets currently requires high-performance computing facilities. For this reason, non-sequential outlier detection algorithms have already been proposed. Most algorithms have been designed for DMMs [8]–[13], and only a few for the shared memory machine, in particular for execution on a GPU [14], [15].

Our objective is to test the effectiveness of a GPU–based solution for distance-based outlier detection; the implementation is based on CUDA, a high–level programming environment for GPU. Our contributions are the following:

1) implementation for a GPU architecture of *BruteForce* and *SolvingSet* algorithms, to be used as baselines in comparisons;
2) implementation of several variants of the above algorithms, to experiment the effects of different usages of the memory hierarchy and of different implementation and optimization techniques;
3) execution of an extensive set of experiments, including five different datasets with dimensionality from two to ten and size up to more than one million, implemented and tested on a GPU hardware platform.

The structure of the paper is the following. Section II discusses the literature on parallel and distributed outlier detection. Section III recalls the centralized algorithms and introduces the GPU algorithms. Section IV describes the experimental setting and presents the results, mainly in terms of speedups of the GPU algorithms with respect to the CPU base algorithms and section V concludes the paper.

II. Related work

Hung and Cheung [8] presented *PENL*, a parallel version of *NL* [16], a block-oriented, I/O optimized nested loop algorithm to detect (p, δ) outliers, defined as objects lying within distance δ from at most a fraction p of the objects of the data set. *PENL*

978-1-4799-0838-7/13/$31.00 ©2013 IEEE

omits outlier ranking and an appropriate value of the parameter δ must be determined. A parallel version of Bay's algorithm has been proposed by Lozano and Acuña [9]. Bay's algorithm [17] iteratively loads consecutive blocks of objects in main memory. For each block, it scans the data set and for every retrieved object updates the neighborhood of the objects in the block. A cutoff outlier score is maintained; block objects having a score lower than the current cutoff are removed from the block. The outlier score is any function which is anti-monotonic in the nearest neighbor distances. Any such function can not increase under unions. Exploiting this fact, in Lozano and Acuña's algorithm, in every phase each processor scans its local data set in parallel and updates the neighborhood of the objects of the same data block maintaining a local cutoff. The neighborhoods are then merged at a master node, which distributes the global cutoff to all processors. The definition of distance-based outlier is compatible with ours. However, the scalability of the method is not consistent and the sensitivity of the centralized version to the order and distribution of the data are not discussed. A parallel version of the Local Outlier Factor (LOF) algorithm is proposed in the same paper [9]. In LOF, the outlier status of a point is determined by averaging the ratio of the density of the point and a neighbor, over all the point's k nearest neighbors. The density of a point is computed as the inverse of the averaged so-called reachability distances of the point with respect to its k nearest neighbors, where the reachability distance of a point p with respect to a second point q selects either the k nearest neighbor distance of q, if the p is one of q's k nearest neighbors, or the distance between the two points, otherwise. The main complexity source in LOF is the computation of the k nearest neighbors for all points. In Lozano and Acuña's parallel approach, each processor computes the k nearest neighbors of its data. The master site collects the results, computes the global neighbors and the LOF of all points. Alshawabkeh, Jang, and Kaeli [14] designed and tested an intrusion detection system based on LOF. The authors show that their GPU implementation outperforms a CPU implementation by up to two orders of magnitude, thereby providing a practical method for intrusion detection. Approaches based on LOF locally estimate density to define outliers, and therefore discover outliers different than our approach does. On the other hand, LOF essentially employs a local k nearest neighbor estimate, resulting in a overall higher complexity of the method. Finally, Matsumoto and Hung [15] propose a GPU-accelerated approach to the detection of outliers with uncertain data, following the outlier definition given in [18]. The method differs from ours as we not consider uncertainty in data values.

Algorithms designed for DMMs have also been proposed. Otey, Ghoting and Parthasarathy in [10] and Koufakou and Georgiopoulos in [11] proposed algorithms for distributed high-dimensional data in which outliers are defined on support, instead of distances. Such approaches are therefore completely different from ours. Dutta, Giannella, Borne and Kargupta [19] proposed algorithms for distributed principal components and top-k outlier detection, in which top-k outliers are deviant objects with respect to correlation: their sum of squared values in a fixed number of lowest-order, normalized principal components is at most the k-th largest such sum. This definition has no relation with the one employed in this work. Besides differing in the way outliers are defined, these approaches differ from ours in that the focus is not on achieving the maximum speedup possible with a given number of processors, but to avoid the transmission of the entire data set to all processors, because it is assumed the processors reside at different nodes of a communication network with limited bandwidth.

III. Algorithms

In this section we describe a family of parallel algorithms for computing the top n distance-based outliers on a GPU. The algorithms are derived either from the *BruteForce* algorithm or the *SolvingSet* algorithm [20].

A. Weights and outliers

In the following, we assume a data set D of objects, which is a finite subset of a given metric space.

Definition III.1 (Outlier weight) *Given an object $p \in D$, the* weight $w_k(p, D)$ *of p in D is the sum of the distances from p to its k nearest neighbors in D.*

Definition III.2 (Top n outliers) *Let T be a subset of D having size n. If there not exist objects $x \in T$ and y in $(D \setminus T)$ such that $w_k(y, D) > w_k(x, D)$, then T is said to be the* set of the top n outliers in D. *In such a case, $w^* = \min_{x \in T} w_k(x, D)$ is said to be the* weight of the top n-th outlier*, and the objects in T are said to be the* top n outliers in D.[1]

B. Sequential algorithms

1) The BruteForce algorithm: A sequential nested-loop algorithm to detect top-n outliers is described in Figure 1. The algorithm initially creates a min heap *Top* of n elements to store the top-n outliers and one max heap *nn* of k elements for each data point to store its k nearest neighbors. For each data point *P* having index i, the points having greater or equal index are accessed, and their distance from *P* are inserted both into their own heap and into the heap of *P* using the method *updateMin*. A distance value is inserted into the heap by *updateMin* if the heap is not full, or its maximum element exceeds the value. In the latter case, the maximum element is deleted. The method *updateMax* inserts the weight of *P* into the outlier heap *Top*, if the heap is not full, or the weight exceeds its minimum element. In the latter case, the

[1] In case of ties on the weight values, some objects y in $(D \setminus T)$ such that $w_k(y, D) = w^*$ could exist. In this case, the objects x in T such that $w_k(x, D) = w^*$ are nondeterministically selected among those scoring the same value of weight.

```
Input: Data set D, integer number n of outliers, integer number k of
nearest neighbors, a distance function dist(·,·).
Output: Set of the top-n outliers of D.

NestedLoopOutliers(D, n, k, dist) {
  Top = MinHeap.create(n);
  for i = 1 to D.count
      D.get(i).nn = MaxHeap.create(k);
  for i = 1 to D.count {
      P = D.get(i);
      for j = i to D.count
          if i == j
            then P.nn.updateMin(0);
            else {
                Q = D.get(j);
                d = dist(P, Q);
                P.nn.updateMin(d);
                Q.nn.updateMin(d);
            }
      Top.updateMax(P, P.nn.sum());
  }
  return(Top.getElements());
}
```

Figure 1. The *BruteForce* algorithm.

minimum element is deleted. The worst-case complexity of the *BruteForce* algorithm is $\Theta(|D|^2 \log k)$, and its best-case complexity is $\Omega(|D|^2)$, which makes it unsuitable for many data sets in real data mining applications. However, it can be readily noted that the outer loop of the algorithm fails to omit points which cannot be outliers. Let us call the sum of the distances of a point P from the points in its heap $P.nn$ the current weight of P. Obviously, if $P.nn$ is full, then the current weight of P cannot increase. Since the minimum weight w' of points in the outlier heap *Top* is a lower bound to the weight of a top-n outlier, any point having a full heap and a current weight smaller than w' should be excluded from further processing. This optimization, among others, is part of the *SolvingSet* algorithm.

2) The SolvingSet algorithm: In the sequel, we recall the notion of solving set and the *SolvingSet* algorithm.

Definition III.3 (Outlier Detection Solving Set) *An outlier detection solving set S is a subset S of D such that, for each $y \in D \setminus S$, it holds that $w_k(y, S) \leq w^*$, where w^* is the weight of the top n-th outlier in D.*

A solving set S always contains the set T of the top n outliers in D. Furthermore, a solving set can be used to predict novel outliers [20]. Our goal is to compute both S and T.

The *SolvingSet* algorithm is described in Figure 2. At each iteration (let us denote by j the generic iteration number), the *SolvingSet* algorithm compares all data set objects with a selected small subset of the overall data set, called C_j (for *candidate* objects), and stores their k nearest neighbors with respect to the set $C_1 \cup \ldots \cup C_j$. From these stored neighbors, an upper bound to the true weight of each data set object can thus be obtained. Moreover, since the candidate objects have been compared with all the data set objects, their true weights are known. The objects having weight upper bound lower than the n-th greatest weight associated with a candidate object, are called *non active* (since these objects cannot belong to the top-n outliers), while the others are called *active*. At the beginning, C_1 contains randomly selected objects from D, while, at each subsequent iteration j, C_j is built by selecting, among the active objects of the data set not already inserted in $C_1, \ldots, C_{j-1}$ during the previous iterations, the objects having the maximum current weight upper bounds. During the computation, if an object becomes non active, then it will not be considered anymore for insertion into the set of candidates, because it cannot be an outlier. As the algorithm processes new objects, more accurate weights are computed and the number of non active objects increases. The algorithm stops when no more objects have to be examined, i.e. when all the objects not yet selected as candidates are non active, and thus C_j becomes empty. The solving set is the union of the sets C_j computed during each iteration.

```
Input: Data set D, a distance function dist(·,·), integer number n of
outliers, integer number k of nearest neighbors, integer number m of
candidate points.
Output: Solving set of D, set of the top-n outliers of D.

SolvingSet(D, dist, n, k, m) {
  SolvSet = ∅;
  Top = MinHeap.create(n);
  for i = 1 to D.count
      D.get(i).nn = MaxHeap.create(k);
  Cand.set(D.RandomSelect(m));
  while Cand.count ≠ 0 {
    SolvSet = SolvSet ∪ Cand.getElements();
    D = D − Cand;
    for i = 1 to Cand.count {
        P = Cand.get(i);
        for j = 1 to Cand.count {
            Q = Cand.get(j);
            d = dist(P, Q);
            P.nn.updateMin(d);
            if i ≠ j then Q.nn.updateMin(d);
        }
    }
    NextCand = MinHeap.create(m);
    for i = 1 to D.count {
      P = D.get(i);
      for j = 1 to Cand.count {
          Q = Cand.get(j);
          if max(P.nn.sum(), Q.nn.sum()) ≥ Top.Min() {
            d = dist(P, Q);
            P.nn.updateMin(d);
            Q.nn.updateMin(d);
          }
      }
      NextCand.updateMax(P, P.nn.sum());
    }
    for i = 1 to Cand.count
        Top.updateMax(Q, Q.nn.sum());
    Cand.set(NextCand.getElements());
  }
  return(⟨SolvSet, Top.getElements()⟩);
}
```

Figure 2. The *SolvingSet* algorithm.

C. Parallelizing the BruteForce algorithm on the GPU

The *BruteForce* algorithm is amenable to parallelization by assigning threads to different portions of the distance matrix and updating nearest neighbor heaps efficiently. We present in the sequel two GPU algorithms following this approach.

1) The GPU-BruteForce algorithm: Kato and Hosino [21] presented a technique to compute on a GPU the result of a set of k nearest neighbors queries over a data set of points. A solution to such problem can be easily exploited for solving the top-n outlier problem, in two steps: after computing a k nearest neighbor list for each point, the points are ranked according to their weight. In the first step, the distance matrix is divided into groups of consecutive rows. Each group is assigned to a block and each row in the group is assigned to a thread of the block. Each thread loads a column of the matrix into the shared memory of the block, and computes the distance between the points corresponding to the row and column. In the second step, matrix cells in the same row, separated by a stride that equals the number of threads in the block, are assigned to a thread. The assigned cell values are inserted into a thread buffer when they are smaller than the k-th nearest neighbor distance of a max heap pertaining to the block. The buffer elements are then inserted into the heap. Finally, the points having the n largest weights are selected by using a multiblock reduction technique.

2) The GPU-BruteForce-SH algorithm: A variant of the previous algorithm employs a different, simpler technique to update the heaps. To each point a thread is assigned, which updates the heap of the point by inserting all distances into it.

D. The GPU-SolvingSet family of algorithms

We describe a family of parallel algorithms, based on the concept of solving set. The algorithms differ by the techniques employed to exploit the memory hierarchy and the architecture of a GPU. *GPU-SolvingSet* is our basic GPU algorithm.

1) The GPU-SolvingSet algorithm: The bounded nested loops in *SolvingSet*, which compute portions of the distance matrix pertaining to candidates, offer an opportunity to be executed by parallel threads in a GPU; however, incremental computation of nearest neighbors has to be dealt with care.

The main procedure of the algorithm is described in Figure 3; it runs on the CPU and iteratively calls GPU kernels, until no more candidates are available. The solving set, the outlier min heap and the candidates are stored on and initialized by the host. In the main while loop, *GPUSubtractCand* subtracts candidates from the current set of data points D by moving substitute points to their locations. Then, current candidates are added to the solving set. Next, the k nearest neighbors of candidate points with respect to all data points, and the k nearest neighbors of data points with respect to candidates are updated in three steps by the kernels *GPUCandNNCand*, *GPUDataNNCand*, and *GPUCandNNData*, which are are described in Figure 4 and 5.

GPUCandNNCand assigns a thread to each candidate and iterates over all candidates computing pairwise distances and inserting them into a max heap of k nearest neighbors. The procedure *updateMin* behaves similarly to the analogous procedure used in the sequential algorithms. The kernel *GPUDataNNCand* assigns a thread to each data point P and iterates over each candidate Q. Insertion into the max heap of P occurs when either the weight upper bound of P or the weight upper bound of Q is not smaller than the current minimum outlier weight, passed as parameter $minWeight$. The distance between P and Q is also stored in a distance matrix M by point index; row j of M stores the distances between the candidate with index j and all data points in D. If the previous condition is not met, an infinite distance value is stored in the matrix cell.

GPUCandNNData updates the k nearest neighbors of candidate points, visiting data points. Processor assignment and memory allocation are crucial for efficiently executing this step. A simple approach assigning one thread to each candidate for computing updates to the nearest neighbors max heaps would be very inefficient, as the number m of candidates is small compared to the number of processors. On the other hand, assigning one thread to each data point, which iterates through candidates to update their k nearest neighbors, would generate memory conflicts on the max heaps N of the candidates, as each thread tries to update the same heap synchronously. A substantial improvement over the solutions above consists in assigning a thread block to each row of the distance matrix, comprising p threads, with $p \leq D.count$. Thread t loops over distance matrix cells indexed $M[j,t]$, $M[j,t+p]$, $M[j,t+2\cdot p]$, ..., and possibly updates an own k nearest neighbors max heap *nbHeap* stored in fast shared block memory. The update is attempted when the weight upper bounds satisfy the current outlier weight lower bound $minWeight$. At the end of the iteration, in thread block j each thread heap *nbHeap*$[t]$ contains the k nearest neighbors of candidate j in the point set $\{D[t+k\cdot p] : t+k\cdot p \leq dataSize, k = 0,1,\dots\}$. The heaps are merged by parallel reduction in $\log p$ steps, and finally merged with $N[j]$, which contained the k nearest neighbors of candidate j in the set of current candidates, as computed by *GPUCandNNCand*. The kernel is executed by the threads of $candSize + 1$ blocks. The threads of the last block, indexed $candSize$, execute the second branch, which constructs the min heap *NextCand* of the candidates for the next iteration by a similar technique. Finally, *GPUSolvingSet* updates the heap of top-n outliers using current candidates and their weights, and sets the candidates for the next iteration.

2) Variants of GPU-SolvingSet: The distance matrix between data points and candidates arising in real problems can be very large, when compared to the size of high-bandwidth GPU memory. Our first variant, *GPU-SolvingSet-NDM*, does not store the entire matrix in device memory. *GPU-SolvingSet-NDM* initially employs *GPUCandNNCand* for computing inter-candidate distances. Then, for each candidate Q sequen-

Input: Data set D, a distance function $dist(\cdot,\cdot)$, integer number n of outliers, integer number k of nearest neighbors, integer number m of candidate points.
Output: Solving set of D, set of the top-n outliers of D.

```
GPUSolvingSet(D, dist, n, k, m) {
  SolvSet = ∅;
  Top = MinHeap.create(n);
  Cand.set(D.RandomSelect(m));
  while Cand.count ≠ 0 {
    GPUSubtractCand();
    SolvSet = SolvSet ∪ Cand.getElements();
    GPUCandNNCand(Cand.count);
    GPUDataNNCand(Cand.count, Top.min);
    GPUCandNNData(Cand.count, Top.min, D.count);
    for i = 1 to Cand.count
        Top.updateMax(D.get(i), D.get(i).nn.sum());
    Cand.set(GPUNextCand());
  }
  return(⟨SolvSet, Top.getElements()⟩);
}
```

Figure 3. The *GPU-SolvingSet* algorithm.

```
GPUCandNNCand(candSize) {
  i = Cand[blockDim * blockId + threadId];
  P = D[i];
  for j = 1 to candSize {
      d = dist(P, D[Cand[j]]);
      updateMin(N[i], d);
  }
}

GPUDataNNCand(candSize, minWeight) {
  i = blockDim * blockId + threadId;
  P = D[i];
  for j' = 1 to candSize {
      j = Cand[j'];
      Q = D[j];
      if max(N[i].weight, N[j].weight) ≥ minWeight
        then {
            M[j,i] = dist(P, Q);
            updateMin(N[i], M[j,i]);
        }
        else
            M[j,i] = ∞;
  }
}
```

Figure 4. Kernels for updating the nearest neighbors of data points in the set of candidates in the *GPU-SolvingSet* algorithm.

tially, in a grid of b blocks of p threads each, every thread: (i) inserts the distances between Q and $|D|/(b \cdot p)$ points P; each distance $dist(P, Q)$ is inserted both into max heap $N[P]$ and into max heap $nbHeap[t]$ in block shared memory, where t is the thread index in the block; (ii) contributes to hierarchically merge heaps that are local to the same block in parallel; if $t = 0$ saves the resulting heap in device memory, indexed by candidate and block. In the following grid, heaps indexed by the same candidate are merged in parallel by a single thread block in shared memory. Then, in a grid of b' blocks of p' threads each, every thread indexed t inserts the weight upper bound of P computed from $N[P]$ into a min heap $wtHeap[t]$ in block shared memory, for $|D|/(b' \cdot p')$ points P; it also contributes to merge heaps in the same block and save them

```
GPUCandNNData(candSize, minWeight, dataSize) {
  t = threadId;
  j = blockId;
  p = blockDim;
  if j < candSize
    then {
      nbHeap[t].size = 0;
      z = t;
      while z < dataSize {
        if max(N[z].weight, N[j].weight)
              ≥ minWeight
          updateMin(nbHeap[t], M[j, z]);
        z = z + p;
      }
      //Parallel reduction
      maxHeapMerge(N[j], nbHeap, t);
    } else {
      wtHeap[t].size = 0;
      z = t;
      P = D[z];
      while z < dataSize {
        if P.isActive
          then
            if (N[P].weight ≥ minWeight)
              then
                updateMax(wtHeap[t],
                          ⟨P, N[z].weight⟩);
            else P.isActive = false;
        z = z + p;
      }
      //Parallel reduction
      NextCand = minHeapMerge(wtHeap, t);
  }
}
```

Figure 5. Kernels for updating the nearest neighbors of candidates in the set of data points in the *GPU-SolvingSet* algorithm.

to device memory as in step (ii) above. Finally, new candidates are obtained by merging these heaps in parallel.

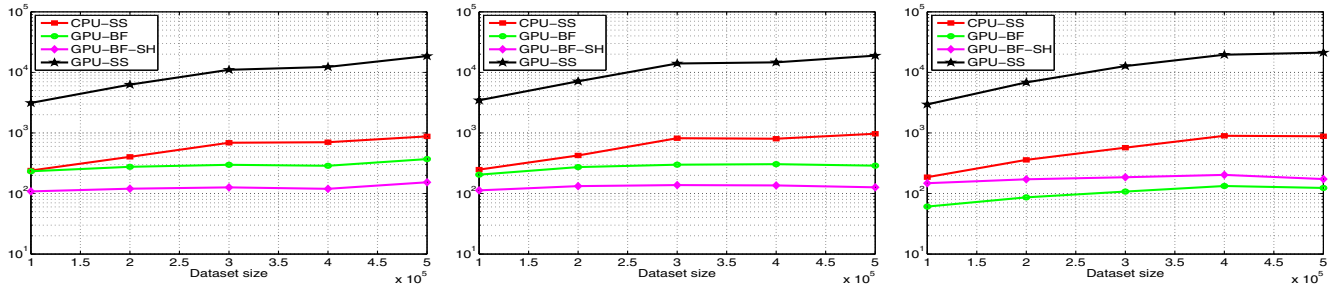
The second variant, *GPU-SolvingSet-NDM-TP*, after executing *GPUCandNNCand*, for each candidate Q sequentially executes step (i) above, in which however $nbHeap[t]$ in stored in device memory and t is the unique index of the thread in the grid. Step (ii) above is realized by a separate grid. New candidates are computed as in *GPU-SolvingSet-NDM* above.

IV. Experiments

In this section, experiments conducted by using the introduced methods are presented. The rest of the section is organized as follows: Section IV-A describes the experimental setting and the datasets employed; Section IV-B presents the comparison of *GPU-SolvingSet* with *SolvingSet* and *Brute-Force*-based strategies; finally, Section IV-C discusses the performances of *GPU-SolvingSet* variants here introduced.

A. Experimental setting and datasets

We ran the CPU/GPU codes by employing an Intel Xeon processor running at 2.40 GHz and hosting an Nvidia Tesla M2070 GPU with 448 cores at 1.15 GHz and 6 GB of global memory. The single precision floating point operations have been employed corresponding to a peak performance on the



(a) $k = 5$ (b) $k = 10$ (c) $k = 50$

Figure 6. Speedup over CPU−BF on *G2d* for various values of k.

GPU of 1.03 Teraflops. The CPU code is run on a single CPU core and makes use only of scalar single precision floating point operations. The algorithm codes are written in Java and the *NVIDIA CUDA Toolkit 4.1* is used for the GPU. As for the CUDA parameters, we point out that each experiment has been preceded by the execution of a tuning phase aimed to guarantee an optimal configuration for the run at hand. This step has carried by a code module providing the values to be assigned to the CUDA parameters used by the algorithms. In particular this module, which is the same for all the algorithms, works by taking into account the hardware specifications, the algorithm to be used and the settings of the running experiment, which are depending on the dataset to be mined and on the parameters for detection task. We omit the details of this module, but we can summarize that it computes the values for the CUDA parameters allowing an optimal matching between the GPU device at hand and the experiment, in order to optimize the performance of the algorithm on the specific data set. We note that the computational cost for the tuning module is negligible.

In the experiments, we considered the following datasets: *G3d* is synthetic and contains 500,000 3D real vectors, obtained by the union of the objects of three 3-d normal distributions having different mean vectors and the unit matrix as covariance matrix; *Covtype* includes the quantitative attributes of the real data set *Covertype*, available at the Machine Learning Repository of UCI [22], and consists of 581,012 instances of 10 attributes; *G2d* is a synthetic collection of 1,000,000 vectors generated from a 2-d normal distribution having the origin as mean vector and the unit matrix as covariance matrix; *Poker* is obtained from the real data set *PokerHands*, available at UCI repository, by removing the class label and consists of 1,000,000 instances of 10 attributes; *2Mass* contains data from the NASA/IPAC Infrared Science Archive[2] (IRSA) and is composed of 1,623,376 instances, consisting of three quantitative attributes associated with JHK filters, obtained from the database *2MASS Survey Atlas Image Info* of the 2MASS Survey Scan Working Databases catalog.

In the sequel, if not otherwise stated, the values for the detection task parameters, are $n = 10$, $k = 50$, and $m = 100$. We considered also other combinations of values for the above parameters, but the results are not completely reported in the paper, since the behavior of the algorithms does not change significantly. For the sake of brevity, in the rest of the section the experimented algorithms will be abbreviated by their acronyms, as detailed next: *BruteForce* as CPU−BF, *GPU-BruteForce* as GPU−BF, *GPU-BruteForce-SH* as GPU−BF−SH, *SolvingSet* as CPU−SS, *GPU-SolvingSet* as GPU−SS, *GPU-SolvingSet-NDM* as GPU−SS−NDM, and *GPU-SolvingSet-NDM-TP* as GPU−SS−NDM−TP.

[2]See http://irsa.ipac.caltech.edu/.

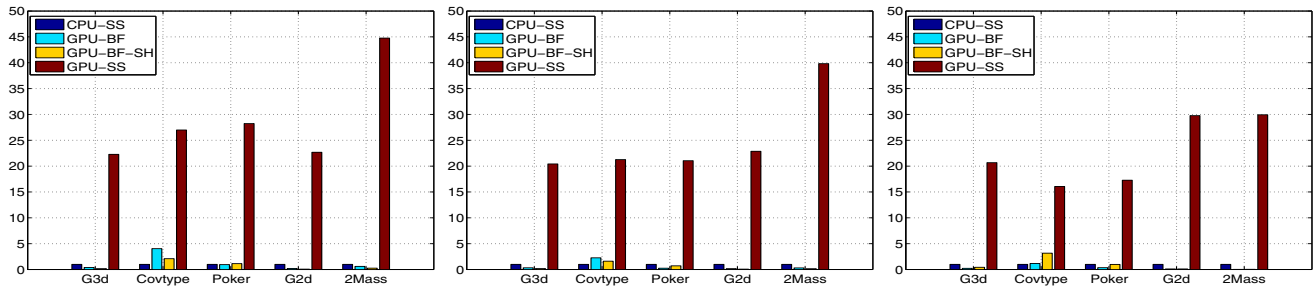
B. Comparison of GPU−SS *with* CPU−SS *and BruteForce-based strategies*

In this section, the CPU−BF, CPU−SS, GPU−BF, GPU−BF−SH, and GPU−SS algorithms are compared. In the first experiment the *G2d* dataset has been employed. In particular, the parameter k has been fixed to 5, the dataset size has been varied between 100K and 500K by sampling, and the execution time has been measured. Figure 6(a) reports the speedup of the methods with respect to the CPU−BF algorithm for $k = 5$. From the figure, it appears that the trend of the speedup is about monotonically non-decreasing with the dataset size for all the methods tested.

As for the various methods considered, the GPU−BF exhibits a great speedup, amounting to two orders of magnitude. For the larger dataset instance here considered, the speedup of GPU−BF approaches 400, a very satisfactory result. This behavior witnesses that the GPU−BF algorithm here proposed is able to fully exploit the GPU features. As far as the GPU−BF−SH method, the exhibited speedup is also good (two orders of magnitude), though lower than that of GPU−BF.

The CPU−SS method has a considerable speedup with respect to the CPU−BF one. Clearly, this is expected since the CPU−SS is like an optimized version of the CPU−BF. However, the curves show that CPU−SS outperforms even the GPU−BF and GPU−BF−SH algorithms. As a matter of fact, the CPU−SS technique, already introduced in the literature in [20], is able to vastly reduce the number of distance computations so that it outperforms even optimal parallelized versions of the CPU−BF algorithm.

As far as GPU−SS is concerned, the experiment confirms



(a) $k = 5$ (b) $k = 10$ (c) $k = 50$

Figure 7. Speedup over CPU−SS for various values of k.

that the strategy here employed to parallelize on the GPU is able to achieve time savings with respect to both the CPU−SS and the GPU/CPU-based versions of the brute force approach. We will elaborate on the relative speedups later in the section.

Figures 6(b) and 6(c) show the speedup of the methods for $k = 10$ and $k = 50$. In general, the behavior above described is maintained, the only difference concerns the GPU−BF and GPU−BF−SH methods for $k = 50$. In this case, the best speedup of GPU−BF−SH doubles, while the speedup GPU−BF gets smaller, though the order of magnitude of the speedup remains the same. However, it must be noticed that on the whole dataset GPU−BF comes back to outperform GPU−BF−SH (this datum is showed later in this section). In general, it can be observed that the speedup of the GPU versions of the CPU−BF worsens when k gets larger. For $k = 50$, that is the largest value of k here considered, the speedup of GPU−BF−SH gets better of that of GPU−BF. Thus, it can be concluded that GPU−BF is most sensitive to the value of the parameter k.

In order to validate the above analysis, the execution times of GPU−SS, CPU−SS, GPU−BF, GPU−BF−SH, and CPU−BF has been measured on the datasets *G3d*, *Covtype*, *Poker*, *G2d*, and *2Mass*. Figures 7(a)-7(c) report the speedup of the various methods w.r.t. the CPU−SS algorithm for $k \in \{5, 10, 50\}$. The results of the CPU−BF algorithm are not reported, since its execution time exceeded in any case the 24 hours (this was the maximum computation time allowed for a process allocated on the machine employed for the experimentation). The figures highlight that CPU−SS outperforms GPU−BF and GPU−BF−SH on all the datasets except for *Covtype*. In order to understand this behavior, the fraction

$$\frac{\#\text{distances computed by CPU−SS}}{\#\text{pairwise distances}}$$

has been measured. The results are reported in table below. Clearly, *Covtype* is the most demanding dataset for the CPU−SS method, as in this case the relative number of distances computed corresponds about to the 5% of the worst case number, that is the total number of pairwise distances among dataset objects and also the number of distances in charge of the brute force methods. Notice that in the GPU-based versions of the brute force method these distances are subdivided among all the GPU cores. Hence, for a GPU having 448 cores, the relative number of distances computed by each core amounts to 0.22%. By comparing this load with the values reported in the following table, it can be recognized a relationship between the distance computation savings obtained by CPU−SS and the speedup of GPU−BF and GPU−BF−SH w.r.t. CPU−SS.

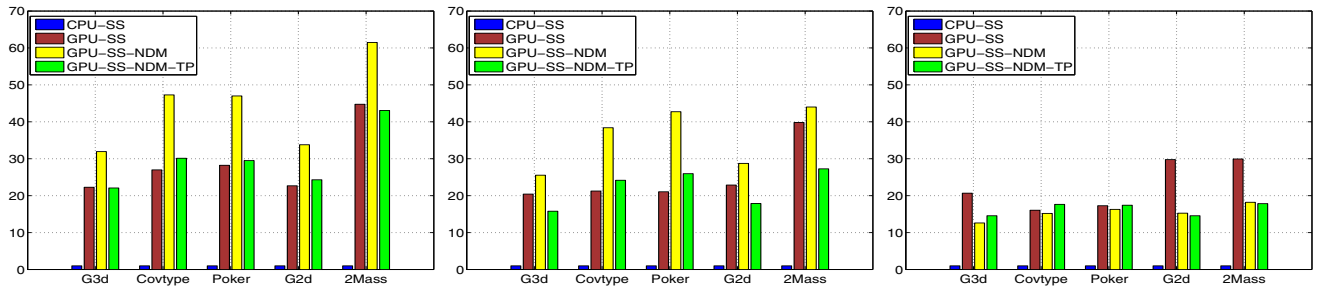
	$k = 5$	$k = 10$	$k = 50$
G2d	0.13%	0.11%	0.15%
G3d	0.34%	0.40%	0.64%
Covtype	5.05%	4.07%	6.83%
2Mass	0.67%	0.42%	0.04%
Poker	2.49%	1.59%	1.84%

Summarizing, in these experiments the speedup the GPU−SS may achieve over CPU−BF is enormous: up to four orders of magnitude. The CPU−SS method has a speedup over GPU−BF and GPU−BF−SH of one order of magnitude. Differently, the speedup of GPU−BF and GPU−BF−SH over CPU−BF is larger: two orders of magnitude, which is of the same order of the CUDA cores available on the GPU. However, as for the comparison of GPU−SS to CPU−SS the speedup is reduced (one order of magnitude: up to 45 times). This confirms that the brute force approach can exploit parallel architectures more efficiently than the solving set algorithm.

C. Comparison of the GPU−SS variants

In this section, the following algorithms are taken into account: CPU−SS, GPU−SS, GPU−SS−NDM, and GPU−SS−NDM−TP. Figure 8(a) reports the relative performance of the methods w.r.t. CPU−SS for $k \in \{5, 10, 50\}$ on all the datasets previously considered. As for GPU−SS−NDM, it can be seen that for small values of the parameter k the strategy guarantees improvements over GPU−SS, while for large k values it performs worse. As for GPU−SS−NDM−TP, from the experiment this strategy does not seem to offer particular advantages neither over GPU−SS−NDM nor over GPU−SS. Hence, it can be concluded that the reduced shared memory occupancy of this method is counterbalanced by the major cost to be paid on the additional operations involving the global memory.

The following table reports the performance increment of the GPU−SS variants w.r.t. the GPU−SS basic method.



(a) $k = 5$ (b) $k = 10$ (c) $k = 50$

Figure 8. Speedup over `CPU-SS` for various values of k.

	GPU−SS−NDM			GPU−SS−NDM−TP		
	$k = 5$	$k = 10$	$k = 50$	$k = 5$	$k = 10$	$k = 50$
G2d	+34.2	+15.5	−50.5	−5.1	−28.9	−53.4
G3d	+23.2	+11.2	−41.6	−16.6	−31.8	−34.2
Covtype	+59.4	+67.0	−8.9	+0.6	+3.9	+3.8
2Mass	+28.3	+4.6	−41.0	−12.5	−35.6	−43.3
Poker	+55.9	+89.5	−8.4	−2.9	+13.2	−3.9

As already pointed out, for $k < 10$, `GPU-SS-NDM` may achieve sensible time improvements (up to 67%). For $k = 50$, the role of `GPU-SS` and `GPU-SS-NDM` is exchanged, since the former method improves over the latter (up to 50%). Thus, this experiment makes clear what are the scenarios most suitable for the two kind of strategies and what are the improvements that can be obtained by each of them. Naturally, the `GPU-SS-NDM` has a spatial cost lower than that of `GPU-SS` and could become the only applicable *GPU-SolvingSet*-based strategy on very large dataset.

V. Conclusions

Due to the complexity of state-of-the-art algorithms for distance-based outlier detection, such algorithms may be impractical in on-line applications requiring very short response times, or applications handling very large data sets. The GPU algorithms presented in this work realize parallel, SIMT CUDA versions of the *SolvingSet* algorithm which outperform our implementation of an optimized CPU algorithm by a factor of at least 15, up to a factor of 60. The experiments we have conducted include both real and synthetic large data sets, and show a remarkable consistency of performance across data set sizes. We experimented several combinations of implementation choices, considering the proposals in the literature and introducing new variations witch differ in the usage of the GPU memory hierarchy and in the way the computation is distributed among the threads and the GPU cores. The experiments show that the approach is very promising, and the research will continue for a deep comprehension of the influence of the various implementation choices on computational properties and on performances.

Acknowledgment

The authors would like to thank M. Zanirati for his meticulous work on the implementation of the algorithms.

References

[1] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2011.
[2] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma, "Discovering all most specific sentences," *ACM Trans. Database Syst.*, vol. 28, no. 2, pp. 140–174, 2003.
[3] P. Hansen and B. Jaumard, "Cluster analysis and mathematical programming," *Mathematical Programming*, vol. 79, pp. 191–215, 1997.
[4] M. J. Zaki and Y. Pan, "Introduction: Recent developments in parallel and distributed data mining," *Distributed and Parallel Databases*, vol. 11, no. 2, pp. 123–127, March 2002.
[5] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *CIKM*, 2009, pp. 661–670.
[6] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *UCHPC-MAW*, 2009, pp. 1–6.
[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, 2009.
[8] E. Hung and D. W. Cheung, "Parallel mining of outliers in large database," *Distributed and Parallel Databases*, vol. 12, no. 1, pp. 5–26, 2002.
[9] E. Lozano and E. Acuña, "Parallel algorithms for distance-based and density-based outliers," in *ICDM*, 2005, pp. 729–732.
[10] M. E. Otey, A. Ghoting, and S. Parthasarathy, "Fast distributed outlier detection in mixed-attribute data sets," *Data Min. Knowl. Discov.*, vol. 12, no. 2-3, pp. 203–228, 2006.
[11] A. Koufakou and M. Georgiopoulos, "A fast outlier detection strategy for distributed high-dimensional data sets with mixed attributes," *Data Min. Knowl. Discov*, 2009 (Published online).
[12] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "A distributed approach to detect outliers in very large data sets," in *Euro-Par (1)*, 2010, pp. 329–340.
[13] ——, "Distributed strategies for mining outliers in large data sets," *IEEE TKDE*, vol. 99, no. PrePrints, 2012.
[14] M. Alshawabkeh, B. Jang, and D. Kaeli, "Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems," in *GPGPU*, 2010, pp. 104–110.
[15] T. Matsumoto and E. Hung, "Accelerating Outlier Detection with Uncertain Data Using Graphics Processors," in *LNCS*, 2012, vol. 7302, pp. 169–180.
[16] E. Knorr and R. Ng, "Algorithms for mining distance-based outliers in large datasets," in *VLDB*, 1998, pp. 392–403.
[17] S. D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule," in *KDD*, 2003.
[18] C. C. Aggarwal and P. S. Yu, "Outlier detection with uncertain data," in *SDM*, 2008, pp. 483–493.
[19] H. Dutta, C. Giannella, K. D. Borne, and H. Kargupta, "Distributed top-k outlier detection from astronomy catalogs using the DEMAC system," in *SDM*, 2007.
[20] F. Angiulli, S. Basta, and C. Pizzuti, "Distance-based detection and prediction of outliers," *TKDE*, vol. 18, no. 2, pp. 145–160, 2006.
[21] K. Kato and T. Hosino, "Solving k-nearest neighbor problem on multiple graphics processors," *CCGRID*, vol. 0, pp. 769–773, 2010.
[22] A. Asuncion and D. Newman, "UCI machine learning repository," 2007.