GPU Strategies for Distance-based Outlier Detection

Fabrizio Angiulli, *Senior Member, IEEE,* Stefano Basta, Stefano Lodi, and Claudio Sartori

Abstract—The process of discovering interesting patterns in large, possibly huge, data sets is referred to as data mining, and can be performed in several flavours, known as "data mining functions". Among these functions, Outlier Detection discovers 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 and distributed algorithms for Graphic Processing Units (GPU) derived from two distance-based outlier detection algorithms: the BruteForce and the SolvingSet. The algorithms differ in the way they exploit the architecture and memory hierarchy of the GPU and guarantee significant improvements with respect to the CPU versions, both in terms of scalability and exploitation of parallelism. We provide a detailed discussion of their computational properties and measure performances with an extensive experimentation, comparing the several implementations and showing significant speedups.

Index Terms—Distance-based outliers, outlier detection, parallel and distributed algorithms.

1 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. The process whose aim is to discover interesting patterns in such large data sets is referred to as data mining [11]. Many problems related to fundamental data mining tasks, like association rule discovery, data clustering and classifier learning, are difficult [10], [12], 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 [24]. 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 [8], [23]. GPU architectures are receiving increasing attention justified by the fact that these devices characterize computers that trade-off between performance and power consumption. As a matter of fact, according to the last update of the Green500 List, providing a ranking of the most energy-efficient supercomputers in the world, the first 23 supercomputers are equipped with GPU accelerators.

Outlier 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 [11]. 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 [9]. Outlier detection is also quite computationally demanding, 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. However, besides the many designed for DMMs [2], [3], [13], [16], [17], [20], there are only a few for shared memory architectures, in particular for execution on a GPU [1], [6], [18].

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:

- implementation for a GPU architecture of *Brute-*

A preliminary version of this article appears in the Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2014) [4].

- *F. Angiulli is with the DIMES Department, University of Calabria, Via P. Bucci, 41C, 87036 Rende (CS), Italy. E-mail: fabrizio.angiulli@unical.it.*
- *S. Basta is with the Institute of High Performance Computing and Networking, Italian National Research Council, Via P. Bucci 41C, 87036 Rende (CS), Italy. E-mail: basta@icar.cnr.it.*
- *S. Lodi and C. Sartori are with the Department of Computer Science and Engineering, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy. E-mail: {stefano.lodi,claudio.sartori}@unibo.it.*

Force and *SolvingSet* algorithms, to be used as baselines in comparisons;
- implementation of several variants, both centralized and distributed, of the above algorithms, to experiment the effects of different usages of the memory hierarchy and of different implementation and optimization techniques;
- execution of an extensive set of experiments, including five different datasets, both synthetic and real, with dimensionality from two to ten and size up to 100 million of objects, implemented and tested on a GPU hardware platform.

The structure of the paper is the following. Section 2 discusses the literature on parallel outlier detection. Section 3 recalls the centralized algorithms and introduces the GPU algorithms. Section 4 describes the experimental setting and presents the results, mainly in terms of speedup of the GPU algorithms with respect to the CPU base algorithms, and section 5 concludes the paper.

2 Related work

Hung and Cheung [13] presented *PENL*, a parallel version of *NL* [15], 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* 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 [17]. Bay's algorithm [7] 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 [17]. 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 [1] 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 more general outliers than our approach does. But LOF essentially employs a local k nearest neighbor estimate, resulting in a overall higher complexity of the method.

The work [6] presents the LoKDR algorithm, which seeks for the subset of features wherein normal data points fall in high-density regions and outliers fall in low-density regions of the feature space. Specifically, the criterion function takes the ratio of the local neighborhood density associated with inliers and outliers, calculated by exploiting kernel density estimation. In order to perform density estimation and to evaluate the criterion function, authors provide a GPU-based implementation of the method, which straightly takes advantage of the data parallel nature of k-nearest neighbor search and of the fact that the criterion function can be evaluated independently on different subsets of features. It is worth to notice that the LoKDR approach is loosely related to our tasks, which instead focuses on the discovery of the anomalies. Furthermore, this algorithm is designed to the centralized scenario and, although exploits parallelization through a GPU, the technical development concerning this aspect mainly consists in exploiting well-known efficient GPU-based k-NN searches and is a marginal contribution within the proposal.

Techniques for parallel GPU-based k-nearest neighbor search are also relevant in our context. Different approaches to this problem have been proposed in the literature. For a recent outlook we refer to [21].

3 Algorithms

3.1 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 3.1 (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 3.2 (Top n outliers) *Let Top 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 Top is said to be the* set of the top n outliers in D. *In such a case, $w^* = \min_{x \in Top} w_k(x, D)$ is said to be the* weight of the top n-th outlier, *and the objects in Top are said to be the* top n outliers in D.

3.2 Sequential algorithms

3.2.1 The BruteForce algorithm

The straightforward algorithm to detect top-n outliers is a sequential nested–loop. The algorithm initially creates a min heap *Top* of n elements to store the top-n outliers and one max heap N 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 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 $P = D[i]$ from the points in its heap $N[i]$ the current weight of P. Obviously, if $N[i]$ 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 is part of the *SolvingSet* algorithm.

3.2.2 The SolvingSet algorithm

Definition 3.3 (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 Top of the top n outliers in D. Furthermore, a solving set can be used to predict novel outliers [5]. Our goal is to compute both a solving set S and the set Top.

The logic of of the *SolvingSet* algorithm is described below, the meaning of the data set symbols is explained in Table 1. 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 dist(·,·), integer numbers n, k, m, r = |D|.
Output: Solving set of D, set of the top-n outliers of D.
(1)  SolvingSet(D, dist, n, k, m, r) {
(2)   Vector<int> SolvSet[ ], C[m], act[r];
(3)   Vector<Vector<float>> dm[m][r];
(4)   MinHeap<int, float> Top[n], NextCand[m];
(5)   MaxHeap<int, float> knn[r][k];
(6)   RandomSelect(C, m);
(7)   while C.length ≠ 0 {
(8)    SolvSet.append(C);
(9)    drop(D, C);
(10)   parallel for i = 0 to C.length − 1
(11)    for j = 0 to C.length − 1
(12)     if i ≠ j then updateMin(knn[C[i]], dist(D[C[i]], D[C[j]]));
(13)   parallel for i = 0 to D.length − 1
(14)    for j = 0 to C.length − 1 {
(15)     if max(weight(knn[i]), weight(knn[C[j]])) ≥ min(Top)
(16)     {d = dist(D[i], D[C[j]]);
(17)      updateMin(knn[i], <C[j], d>);
(18)     } else act[i] = 0;
(19)     dm[j][i] = d; }
(20)   parallel for j = 0 to C.length − 1
(21)    for i = 0 to D.length − 1 updateMin(knn[C[j]], <i, dm[j][i]>)
(22)   updateTopCand(C, knn, act); }
(23)  return(⟨SolvSet, Top⟩); }
```

Fig. 1. The *SolvingSet-MV* algorithm.

Performance improvements can be obtained by exploiting parallelism in the computation of neighbor lists. Figure 1 shows algorithm *SolvingSet-MV*, which realizes *SolvingSet* for a parallel multi-core environment. The sequential realization of *SolvingSet* [5], loops for each object pair in $D - \cup_{i=1}^{j} C_i$ and current candidates C_j, computing the distance and updating the neighbor lists of the objects if necessary. Whatever the loop ordering, a parallelization of the outer loop causes conflicts in the update of shared neighbor lists,

which accounts for a substantial part of the run time and therefore should not be enclosed in a critical region for efficiency. A better alternative consists in splitting the loop: first, the neighbors of $D - \cup_{i=1}^{j} C_i$ are updated and the distances are cached into a $m \times |D|$ candidate-point distance matrix *dm* (lines (13)–(19)); then, max heaps of candidates are updated from the cached distances (lines (20)–(21)). In *dm*, rows are ordered by candidate to enable local access in the second loop. Function at line (22) updates the top-n outliers and sets the active objects with the m largest weights as candidates for the next iteration.

Object	Allocation	Meaning
D	Host	Data set as a list of points
E	Device	Data set as an array of points
Cand	Host	Candidates as a list of pairs of points and indices
C	Device	Array of candidates
N	Device	Array of max heaps containing at most k nearest neighbors of data points and their distances.
LN	Device	Array of max heaps containing at most k nearest neighbors of candidate points and their distances.
H	Device	Array of intermediate min heaps in reductions
LC	Device	Array of min heaps. Each heap contains at most m candidates for the next iteration.
distM	Device	Matrix of floats
Top	Host	Min heap of the top-n outliers
SolvSet	Host	Solving set as a list of points

TABLE 1
Variables defined in *GPU-SolvingSet* algorithms.

3.3 Fermi GPU basics

In this paper we refer explicitly to the CUDA Fermi GPU architecture, produced by NVIDIA, on which we developed our prototypes. This device allows *thread parallelism*, threads are grouped in blocks, blocks are grouped in grids, each thread executes the same piece of code, the *kernel function*. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per–thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per–block shared memory space used for inter–thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel–wide global synchronization [19].

3.4 Parallelizing the *BruteForce* algorithm

The *BruteForce* algorithm is amenable to parallelization by assigning threads to different portions of the distance matrix and updating nearest neighbor heaps efficiently. For the comparisons with our proposals, we implemented two GPU algorithms following the above sketched approach, according to [14].

3.4.1 GPU-BruteForce

Kato and Hosino [14] presented a technique to compute on a GPU the result of a set of k nearest neighbors queries 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, using a multiblock reduction technique, the points having the n largest weights are selected.

3.4.2 GPU-BruteForce-SH

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. Distances are computed using the same technique utilized in *GPU-BruteForce*.

3.5 The *GPU-SolvingSet* family of algorithms

We describe a family of parallel (Sections 3.5.1-3.5.3) and distributed (Section 3.5.4) algorithms, based on the concept of solving set, and differing by the optimizations employed to exploit the memory hierarchy and the architecture of a GPU. *GPU-SolvingSet* is our basic GPU algorithm for computing top-n outliers.

The host part of our algorithms is described as object-oriented pseudo-code. In this part, identifiers beginning with *GPU* name wrapper procedures which call GPU kernels. In particular, *GPUExec* designates a generic wrapper for the execution of kernels. The device parts are described as structured kernel pseudo-code, which uses conventional predefined variables that correspond to the ones available in CUDA: $threadId$, $blockId$, $blockDim$ and $numBlocks$ denote the thread and thread block identifiers, the number of threads in a thread block, and the number of thread blocks in the current grid, respectively. Data structures will be allocated both in global device memory and fast shared memory. Shared structures are explicitly declared in kernels. Variables in device memory and host objects are summarized in Table 1.

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.

```
(1)  GPUSolvingSet(D, dist, n, k, m) {
(2)    PointSet SolvSet = new PointSet();
(3)    MinHeap Top = new MinHeap(n);
(4)    Point[ ] Cand = new Point[m];
(5)    float[ ] CandW = new float[m];
(6)    GPUAlloc(E, C, N, LN, H, LC, distM);
(7)    Cand = D.RandomSelect(m);
(8)    GPUCopyFromHost(C, Cand);
(9)    while Cand.length ≠ 0 {
(10)     GPUExec(Init(E, C, N, LN));
(11)     SolvSet.append(C);
(12)     GPUSync();
(13)     GPUExec(CToC(C, LN, k, dist));
(14)     GPUSync();
(15)     GPUExec(DToC1(E, C, N, LN, Top.min(), k, dist, distM));
(16)     GPUSync();
(17)     GPUExec(DToC2(E, C, LN, Top.min(), k, distM));
(18)     GPUSync();
(19)     for i = 1 to Cand.length {CandW[i] = getDistSum(LN[i])}
(20)     GPUSync();
(21)     GPUExec(Candidate(C, LC));
(22)     for i = 1 to Cand.length {
(23)       Top.updateMax(Cand[i], CandW[i]);  }
(24)     GPUSync();
(25)     GPUCopyToHost(Cand, C);  }
(26)   return(⟨SolvSet, Top.getElements()⟩);
```

Fig. 2. The *GPU-SolvingSet* algorithm.

```
CToC(C, LN, k, dist) {
  i = blockDim · blockId + threadId;
  for j = 0 to |C| − 1 {
    Q = C[j];  d = dist(C[i], Q);  updateMin(LN[i], ⟨Q, d⟩);  } }
```

Fig. 3. Neighbors of candidates w.r.t. candidates.

3.5.1 GPU-SolvingSet

The bounded nested loops in *SolvingSet*, computing parts of the distance matrix can be executed by parallel threads in a GPU; however, the incremental computation of neighbors has to be dealt with care, and can be implemented by different techniques.

The main procedure of the algorithm is described in Figure 2; it runs on the CPU and consists of an initialization sequence and a while loop which calls GPU kernels, until no more candidates are available. The solving set *SolvingSet*, the top-n outlier min heap *Top*, the candidates *Cand*, and their weights *CandW* are stored on and initialized by the host. In particular, *Cand* is initialized to a randomly selected m-subset of D. In the while loop, kernel *Init* subtracts candidates from the current on-device array of points E and compacts the array by moving substitute points to their locations. Then, current candidates are added to the solving set. Next, the k nearest neighbors of candidates in the data set and the k nearest neighbors of data points with respect to candidates are updated in three steps by kernels *CToC*, *DToC1*, and *DToC2*, described in Figures 3, 6, and 8, respectively. Phase *DToC2* also collects the candidates for the next iteration. Figure 4 shows an example dataset, including the solving set and the outliers. Figure 5 sketches the computations performed by kernel *CToC*. This kernel

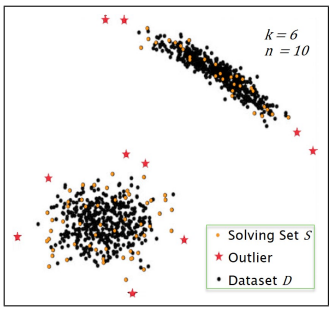


Fig. 4. Example data set

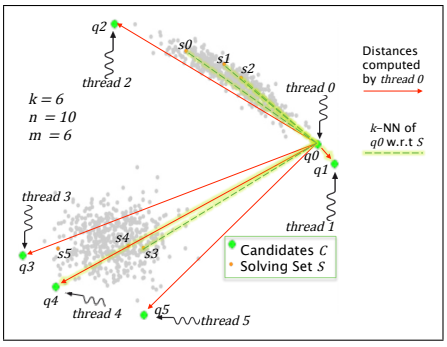


Fig. 5. Sketch of the *CToC* kernel computations.

computes the k nearest neighbors of each candidate in the set of current candidates C and stores them into the max heap of the candidate, which is an element of array LN. To this end, each thread of *CToC* computes the distances between all candidates and a fixed candidate $C[i]$ which is uniquely assigned to the thread. The neighbors of $C[i]$ and their distances are inserted into a max heap $LN[i]$ of size k by *updateMin*, which is analogous to *updateMin* in sequential algorithms. *GPU-SolvingSet* then improves the neighbor list of points in E in kernel *DToC1*, extends the k nearest neighbors of candidates in C to all E in *DToC2*, and selects the candidates for the next iteration.

Kernel *DToC1* in Figure 6 computes the distances between E and C, and updates the array N of max heaps of nearest neighbors of points in E. The computations of this thread are sketched in Figure 7. Each thread in *DToC1* is uniquely associated to an index i in E, and iterates over C. A candidate and its distance to $E[i]$ are inserted into max heap $N[i]$ when the weight upper bound of $E[i]$, or of the candidate, is not smaller than the current minimum outlier weight, in *Top* and passed as $minWeight$. The distance between $E[i]$ and the candidate is also stored in the distance matrix *distM*; row j of *distM* stores the distances between the candidate with index j and all data points in E. If the condition is not met, an infinite distance value is stored in the matrix cell.

In *DToC2*, processor assignment and memory allocation are crucial for efficiency. The assignment of a single thread to each candidate for computing

```
DToC1(E, C, N, LN, minWeight, k, dist, distM) {
  i = blockDim · blockId + threadId;
  for j = 0 to |C| − 1 {
    Q = C[j];
    if max(DistSum(N[i]), DistSum(LN[j])) ≥ minWeight
      then { d = dist(E[i], Q);
             if DistSum(N[i]) ≥ minWeight
               then updateMin(N[i], ⟨Q, d⟩); }
      else d = ∞;
    distM[j][i] = d; } }
```

Fig. 6. Neighbors of data points w.r.t. candidates.

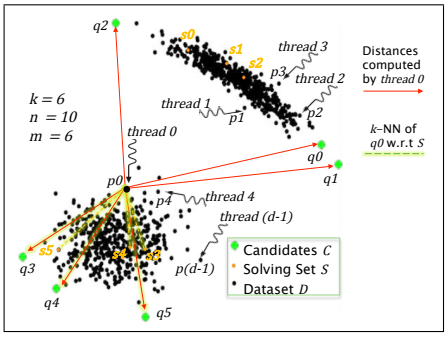


Fig. 7. Sketch of the *DToC1* kernel computations.

updates to its nearest neighbors max heap does not fully utilize computing capacity, as the candidates are fewer than the processors. On the other hand, the assignment of one thread to each data point, which iterates through candidates to update their neighbors, generates conflicts on their max heaps LN, because all threads access the same heap synchronously. In our implementation, a thread block is assigned to each candidate, that is, to each row of the current distance matrix portion, and each thread visits cells with stride $blockDim$. The outcome is max heap $LN[blockId]$ containing the k nearest neighbors of the candidate.

Kernel *DToC2* is described in Figure 8. Shared array *nbHeap* of $blockDim$ per-thread max heaps in function *CandNN* is used in the selection of the k nearest neighbors of a candidate. A shared max heap *cdHeap* is also used in the selection of the points having largest weights as candidates for the next iteration. *DToC2* is executed in a grid of $|C| + 1$ thread blocks, in which each block among the first $|C|$ selects the k nearest neighbors of a distinct candidate, and the last block selects m points having the largest weights. The initial conditional statement separates the paths. In function *CandNN*, called by the "then" part, each thread in a block processes a subset of row $blockId$ of matrix *distM*. Recall that row $distM[blockId]$ stores the distances from candidate indexed $blockId$ in C to all data points. To this end, the thread visits its points with stride $blockDim$, starting from point $threadId$, and inserts the distance between the point and candidate $blockId$ into heap $nbHeap[threadId]$. Each thread executes *MaxHeapReduction*, described in Figure 9, a reduction of the heaps in a thread block in the heap of thread 0, which finally merges

```
DToC2(E, C, LN, minWeight, k, distM) {
  shared MinHeap cdHeap[blockDim];
  if blockId < |C|
    then CandNN(E, C, LN, minWeight, k, distM);
    else {
      i = threadId;
      while i < |E| {
        if E[i].active
          then if DistSum(N[i]) ≥ minWeight
                 then updateMax(cdHeap[threadId],
                                ⟨i, E[i], DistSum(N[i])⟩);
                 else E[i].active = false;
        i = i + blockDim; }
      synchronize();
      MinHeapReduction(threadId, cdHeap, blockDim/2);
      if threadId == 0
        then minHeapMerge(LC[threadId], cdHeap[0]); } }

CandNN(E, C, LN, minWeight, k, distM) {
  shared MaxHeap nbHeap[blockDim];
  if DistSum(LN[blockId]) < minWeight then return;
  i = threadId;
  while i < |E| {
    updateMin(nbHeap[threadId], ⟨E[i], distM[blockId][i]⟩);
    i = i + blockDim; }
  synchronize();
  MaxHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then maxHeapMerge(LN[blockId], nbHeap[0]); }
```

Fig. 8. Neighbors of candidates w.r.t. data points.

```
maxHeapMerge(H1, H2) {
  for i = 1 to H2.length do updateMin(H1, H2.get(i)); }

MaxHeapReduction(t, maxHeap, h) {
  while h ≥ 1 {
    if t < h then maxHeapMerge(maxHeap[t], maxHeap[t + h].get(i));
    if h > 32 then synchronize();
    h = h/2; } }
```

Fig. 9. Merging and reducing max heaps.

its heap with the heap of the candidate, $LN[blockId]$. The function exits immediately for all threads of block $blockId$ if the corresponding candidate cannot be a top-n outlier, because its current weight is smaller than parameter $minWeight$, the n-th outlier weight lower bound. In the "else" part, the threads of a single block process interleaved subsets of the data set. A point is processed only if its *active* field is set. If its weight is not lower than $minWeight$, then it is inserted into the min heap of thread *cdHeap*. Otherwise, it cannot be a top-n outlier and is marked as nonactive. After synchronization, shared heaps are hierarchically merged into the heap of thread zero, which finally copies it to the output heap variable *LC*. In the last steps of the main loop (lines 19–25), the exact weights of current candidates are computed from heap array *LN* and written into array *CandW*; candidates are then copied from max heap *LC* to the candidate set *C* in device memory. Max heap *Top* of current best top-n outliers is updated; finally the candidates for the next iteration are copied into program object *Cand*.

3.5.2 Variants of GPU-SolvingSet with reduced space requirements

The matrix of distances between the data set and the candidates arising in real outlier detection problems

```
(1)  for i = 1 to Cand.length {
(2)    j = Cand.getIndex(i); Q = Cand.get(i);
(3)    GPUExec(DToC1NDM(E, C, N, LN, Q, j,
(4)                     Top.min(), k, dist, H)); }
(5)  GPUExec(DToC2NDM(LN, k, H));
(6)  GPUSync();
(7)  GPUExec(WeightUpd1(E, N, H, m, Top.min()));
(8)  for i = 1 to Cand.length {
(9)    CandW[i] = getDistSum(LN[i]); }
(10) GPUSync();
(11) GPUExec(WeightUpd2(LC, H, m));
```

Fig. 10. Modified code in *GPU-SolvingSet-NDM*.

```
DToC1NDM(E, C, N, LN, Q, j, minWeight, k, dist, H) {
  shared MaxHeap nbHeap[blockDim];
  i = blockId · blockDim + threadId;
  while i < |E| {
    if max(DistSum(N[i]), DistSum(LN[j])) ≥ minWeight
      then { d = dist(E[i], Q);
             updateMin(N[i], ⟨Q, d⟩);
             updateMin(nbHeap[threadId], ⟨E[i], d⟩); }
    i = i + blockDim · numBlocks; }
  synchronize();
  MaxHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then H[j][blockId] = nbHeap[0]; }
```

Fig. 11. Neighbors in *GPU-SolvingSet-NDM*.

```
DToC2NDM(k, LN, H) {
  shared MaxHeap nbHeap[blockDim];
  t = threadId;
  while t < numBlocks {
        maxHeapMerge(nbHeap[threadId], H[blockId][t]);
        t = t + blockDim; }
  synchronize();
  MaxHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then
    maxHeapMerge(LN[blockId], nbHeap[0]); }
```

Fig. 12. Reduction of heaps in *GPU-SolvingSet-NDM*.

```
DToCTP1(E, C, N, LN, Q, j, minWeight, k, dist, H1) {
  i = blockId · blockDim + threadId;
  while i < |E| {
    if max(DistSum(N[i]), DistSum(LN[j])) ≥ minWeight
      then {
        d = dist(E[i], Q);
        updateMin(N[i], ⟨Q, d⟩);
        if |LN[j]| < k ∨ d < DistMax(LN[j])
          then updateMin(H1[threadId], ⟨E[i], d⟩); }
    i = i + blockDim · numBlocks; } }

DToCTP2(k, j, H, H1) {
  shared MaxHeap nbHeap[blockDim];
  t = threadId;
  while t < blockDim1 · numBlocks1 {
        maxHeapMerge(nbHeap[threadId], H1[t]);
        t = t + blockDim · numBlocks; }
  synchronize();
  MaxHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then H[j][blockId] = nbHeap[0]; }
```

Fig. 13. Neighbors of candidates and data in *GPU-SolvingSet-NDM-TP*.

can be very large, compared to high-bandwidth GPU memory. We describe *GPU-SolvingSet-NDM*, a variant to *GPU-SolvingSet*, which does not store the entire distance matrix in device memory, and a second variant, *GPU-SolvingSet-NDM-TP*, which also allocates temporary heaps in shared memory during reductions.

3.5.2.1 *GPU-SolvingSet-NDM*: The algorithm differs from *GPU-SolvingSet* at lines 15–18, which are replaced by the code in Figure 10. In this variant, both the computation of nearest neighbors of data points and candidates, and the selection of points having largest weights are split into two phases. Kernel *DToC1NDM*, described in Figure 11, is called once for each candidate, to update the nearest neighbor max heaps N and LN of data points and candidates, respectively. The kernel is executed in a grid of $numBlocks$ thread blocks of $blockDim$ threads each, and employs a max heap array *nbHeap* of size $blockDim$ in each block. Each thread in a block manages a unique heap in the array to store neighbors of the candidate parameter Q. Points are visited starting from a unique point among the first $blockDim \cdot numBlocks$, and continuing with stride $blockDim \cdot numBlocks$. Each thread thus computes the distance between Q and each one of $|D|/(numBlocks \cdot blockDim)$ points and inserts it into the max heap $N[i]$ of the point, as well as into max heap $nbHeap[threadId]$, by executing the respective *updateMin* functions; the distance is inserted only into a non-full heap or when its max element is larger than the distance. Threads are then synchronized and each thread in parallel executes function *MaxHeapReduction*. At each iteration, each heap having index t smaller than h absorbs the entire heap at index distance h, and h is halved. Note that in CUDA all threads in the same warp are automatically synchronized; since only threads having index in the set $\{0, 1, \ldots, h-1\}$ execute heap updates, and a warp consists of 32 threads having contiguous indices, only iterations for $h > 32$ need an explicit synchronization at their end. At function exit, the $blockDim$ max heaps of each thread block have been merged in shared memory yielding one heap for each thread block in $nbHeap[0]$. Since shared memory is local to a block, the resulting heap is stored in device memory by thread 0 into an array of heaps $H[j, blockId]$, which is indexed by candidate and block identifier. In the following step of Figure 10, heaps in device memory are merged by candidate by kernel *DToC2NDM* described in Figure 12. The kernel takes three parameters k, LN, and H. Note that H is the same heap matrix which is assigned to in the last line of *DToC1NDM* and that by $numBlocks$ we denote the number of blocks in the execution grid of *DToC1NDM*, which equals the number row elements, or of heaps per candidate, stored in device memory in H. The kernel is executed by one thread block per candidate. Each thread in $blockId$ absorbs the heaps for candidate having index $blockId$ in the candidate set, with stride $blockDim$, into an initially empty max heap *nbHeap* in shared memory. Per-thread heaps are then merged hierarchically into the heap of the first thread of each block, which finally merges it into heap LN. The subsequent steps

```
for i = 1 to Cand.length {
  j = Cand.getIndex(i); Q = Cand.get(i);
  DToCTP1(E, C, N, LN, Q, j, Top.min(), k, dist, H1);
  GPUSync();
  DToCTP2(k, j, H, H1);
  GPUSync(); }
```

Fig. 14. Modified loop in *GPU-SolvingSet-NDM-TP*.

```
CandNN(E, C, LN, Top.min(), k, distM);
GPUSync();
GPUExec(WeightUpd1(E, N, H, m, Top.min()));
for i = 1 to Cand.length {CandW[i] = getDistSum(LN[i]); }
GPUSync();
GPUExec(WeightUpd2(LC, H, m));
```

Fig. 15. Modified code in *GPU-SolvingSet-S*.

```
WeightUpd1(E, N, H, m, minWeight) {
  shared MinHeap nbHeap[blockDim];
  i = blockId · blockDim + threadId;
  while i < |E| {
    if E[i].active ∧ DistSum(N[i]) ≥ minWeight
      then updateMax(nbHeap[threadId], ⟨i, E[i], DistSum(N[i])⟩);
      else E[i].active = false;
    i = i + blockDim · numBlocks; }
  synchronize();
  MinHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then H[blockId] = nbHeap[0]; }

WeightUpd2(LC, H, m, numBlocks) {
  shared MinHeap nbHeap[blockDim];
  i = threadId;
  while i < numBlocks {
    minHeapMerge(nbHeap[threadId], H[i]);
    i = i + blockDim; }
  synchronize();
  MinHeapReduction(threadId, nbHeap, blockDim/2);
  if threadId == 0 then LC = nbHeap[0]; }
```

Fig. 16. Best candidates update in *GPU-SolvingSet*.

select the most promising candidates for the next iteration and update the active status of data set points. Kernel *WeightUpd1* is executed in a grid of $blockDim \cdot numBlocks$ thread blocks, in which each thread is associated to a data point. The kernel uses a shared array *nbHeap* of $blockDim$ per-thread min heaps in each thread block to select the points having largest weight, visiting points with a stride of $blockDim \cdot numBlocks$ to cover the whole data set. A hierarchical merge of the min heaps stores the m points with largest weights visited by each block into the per-block min heap $H[blockId]$. In parallel with kernel execution, the weights of the candidates of the current iteration are copied from device memory into program memory. After the synchronization barrier, each thread of of a single thread block in kernel *WeightUpd2* visits the min heap array H with a stride of $blockDim$, merging the visited heaps into its own heap *nbHeap*$[threadId]$. Finally, the per-thread heaps are hierarchically merged into *nbHeap*$[0]$, which is then copied into *LC*.

3.5.2.2 *GPU-SolvingSet-NDM-TP*: As data points are associated to threads in *DToC1NDM*, a many threads are employed by the kernel, thus requiring a large amount of shared memory in per-thread heaps and possibly limiting processor occupancy. This problem is approached in *GPU-SolvingSet-NDM-TP* by further splitting kernel *DToC1NDM* into two parts, both reported in Figure 13. Furthermore Figure 14 shows a modified loop which substitutes the first one in Figure 10. The first part, *DToCTP1*, iterates over data points as *DToC1NDM*, but inserts neighbors to candidates into a max heap array *H1* in global memory, and does not perform any hierarchical reduction. Each thread of the second part, *DToCTP2*, merges into its per-thread max heap *nbHeap* the heaps in array *H1*, iterating through its elements with stride equal to the total number of threads $blockDim \cdot numBlocks$. The iteration bound refers to the number of threads $blockDim1$ and the number of blocks $numBlocks1$ of kernel *DToCTP1*. After a synchronization barrier, a hierarchical reduction merges per-thread heaps of each block into *nbHeap*$[0]$, which is then copied to the global heap element pertaining to candidate j and the current block. The final reduction of the global heap matrix rows $H[j]$ is performed by kernel *DToC2NDM*.

3.5.3 Variants of GPU-SolvingSet

3.5.3.1 *GPU-SolvingSet-S*: Lines 17–20 are replaced by the code in Figure 15, combining *CandNN* of Figure 8 to compute the nearest neighbors of candidates, and functions *WeightUpd1* and *WeightUpd2*, shown in Figure 16, to select candidates for the next iteration, thus operating as lines 6–11 in Figure 10.

3.5.3.2 *GPU-SolvingSet-S-MS*: Kernel *CandNN* is substituted by kernel *CandNNMS*, which splits the execution in steps to stop iterations early. Per-thread max heaps *nbHeap* are declared in kernel *CandNNMS*, shown in Figure 17, as in *CandNN*. However, the single while loop is substituted by nested while loops to allow for a finer flow control. The outer loop divides the computation of *LN*$[blockId]$ into $nSteps$ steps and verifies at the beginning of each step that the candidate assigned to the block cannot be ruled out as a top-n outlier, by checking that its current weight upper bound, computed from *LN*$[blockId]$, is not smaller than $minWeight$. In each step, after resetting the per-thread heap *nbHeap*$[threadId]$, the inner loop iterates through $stepSize$ elements of the matrix portion row *distM*$[blockId]$, with stride $blockDim$ as in the previous implementation, inserting pairs of neighbors and distances into the per-thread heap. After the synchronization barrier, per-thread max heaps are hierarchically merged into the heap of the first thread, which is finally merged into *LN*$[blockId]$. As in *CandNN*, an early return is executed by all threads of block $blockId$ if candidate $blockId$ cannot be a top-n outlier.

3.5.4 Distributed approach

To take full advantage of state-of-the-art supercomputers, we designed the GPU-*DistributedSolvingSet* al-

```
CandNNMS(E, C, LN, minWeight, k, distM) {
  shared MaxHeap nbHeap[blockDim];
  if DistSum(LN[blockId]) < minWeight then return;
  nSteps = |E|/stepSize; s = 0; i = threadId;
  while s < nSteps ∧ DistSum(LN[blockId]) ≥ minWeight {
    nbHeap[threadId].size = 0;
    while i < (s + 1) · stepSize ∧ i < |E| {
      d = distM[blockId][i];
      updateMin(nbHeap[threadId], ⟨D[i], d⟩);
      i = i + blockDim; }
    synchronize();
    MaxHeapReduction(threadId, nbHeap, blockDim/2);
    if threadId == 0 then maxHeapMerge(LN[blockId], nbHeap[0]);
    s = s + 1; } }
```

Fig. 17. Heaps update in *GPU-SolvingSet-S-MS*.

gorithm, representing a combined distributed and many-core approach to the computation of the top-n outliers, which merges the features of the *LazyDistributedSolvingSet* [3] with the *GPU-SolvingSet*.

It is assumed that the global data set D is horizontally partitioned into ℓ *local datasets* D_i located at *local nodes* N_i and that a *supervisor node* N_0 is connected to all N_is by a high-performance network. The GPU-*DistributedSolvingSet* algorithm runs at N_0 the *LazyDistributedSolvingSet* algorithm (cf. Figure 3 in [3]), and at nodes N_i the *NodeComp*$_i$ procedures. Each procedure *NodeComp*$_i$ performs the computation within the main cycle of the *GPU-SolvingSet* algorithm (cf. Figure 2, lines (10)-(25)) on its own D_i.

The supervisor works as follows. The current set of candidates (initially randomly selected) is added to the solving set and broadcast to local nodes where procedures *NodeComp*$_i$ are executed. Local nodes return the local nearest neighbors of each candidate and also a set of candidates for the next iteration. The supervisor exploits the local nearest neighbors in order to determine the true weight of current candidates and update the top-n outliers. The computation ends when there are no more candidates to consider.

As for the local nodes, when the distance matrix to be materialized does not fit into the device memory, the local dataset D_i is partitioned in a suitable number of *chunks* and computations are performed on a per chunk basis by running the kernels multiple times.

3.6 Cost analysis

Let N_p denote the number of threads that can be simultaneously executed on the GPU. In the following, we assume the hypothesis of optimal thread allocation. W.l.o.g., it is assumed that the cost of computing the distance between two objects is $O(d)$, where d is the number of attributes of each object. Moreover, by F it is denoted the number of bytes employed to store a floating point number or an integer number.

3.6.1 GPU-SolvingSet

Consider the *DToC1* kernel. Each thread computes the distance between a fixed dataset object and all the m candidates and updates the heap of its k nearest neighbors. Thus, the cost in charge of each thread is $O(m(d+\log k))$ and the temporal cost of executing N threads is

$$O\left(m\frac{N}{N_p}(d+\log k)\right).$$

Consider now the *DToC2* kernel. Then number of threads associated with this kernel is $(m+1)N_t$, where $m+1$ is the number of blocks executed, while N_t is the number of threads per block. During the executing of the portion of code preceding the parallel reduction, each thread populates an heap of k (m, resp.) elements by inspecting disjoint subsets of the distance matrix (of the object weights, resp.) each having size $\frac{N}{N_t}$. This costs $O(\frac{N}{N_t}\log(\max\{k,m\}))$ operations per thread.

As for the parallel reduction, it requires $\log N_t$ steps during which each involved thread merges two heaps of k (m, resp.) elements, with temporal cost $\max\{k,m\}\log(\max\{k,m\})$, thus with total temporal cost $O(\max\{k,m\}\log(N_t)\log(\max\{k,m\}))$. Overall, the cost is

$$\frac{(m+1)N_t}{N_p}\cdot\frac{N}{N_t}\cdot\Big(\log(\max\{k,m\})+ \\ +\log(N_t)\max\{k,m\}\log(\max\{k,m\})\Big).$$

For simplicity, assume that $k \geq m$ (for otherwise, in what follows it suffice to replace k with m). Hence, after simplifying, the total temporal cost of this kernel is proportional to

$$O\left(m\frac{N}{N_p}\log N_t\cdot(k\log k)\right).$$

As for the update of the heap *Top* of the top n outliers, its cost is negligible, amounting to $O(m\log n)$.

Putting things together, the temporal cost of the *GPU-SolvingSet* algorithm is

$$O\left(tm\frac{N}{N_p}\Big(d+\log N_t\cdot(k\log k)\Big)\right),$$

where t denotes the number of main iterations performed by the method until reaching convergence.

As for the space cost, consider the device memory occupancy: the dataset occupies NdF bytes, the heaps storing the k nearest neighbor distances occupy $N(k+1)F$ bytes, and the distance matrix requires NmF bytes. The total amount of global memory needed is hence $O(N(d+k+m)F)$.

As for the shared memory occupancy, this kind of memory is employed mainly for storing the intermediate heaps employed by the kernel *DToC2*. Maintaining all the heaps associated with threads within the same block requires $N_t(k+1)F$ bytes ($N_t(m+1)F$ bytes, resp.).

3.6.2 GPU-SolvingSet-NDM and GPU-SolvingSet-NDM-TP

Here, N_b denotes the number of blocks (also denoted as $numBlocks$ in the pseudo-code) and N_t the number of threads per block (also denoted $blockDim$ in the pseudo-code). Hence, the total number of threads which are executed per kernel is N_bN_t.

The temporal cost of *DToC1NDM* is

$$O\left(\left(\frac{N_bN_t}{N_p}\right)\cdot\left(\frac{N}{N_bN_t}\right)\cdot(d+\log k)\right),$$

where N_bN_t is the total number of threads executed, $\frac{N}{N_bN_t}$ is the number of distances per thread, d is the cost of computing distances, and $\log k$ is the cost of updating heaps.

The parallel reduction in shared memory performed at the end of the kernel *DToC1NDM* costs

$$O\left(\left(\frac{N_bN_t}{N_p}\log N_t\right)\cdot(k\log k)\right),$$

since its depth is $\log N_t$ (due to the necessity to merge the N_t heaps of each block) and $k\log k$ is the cost of merging heaps. At the end of this reduction we have N_b heaps, one for each block, containing the k smallest distances among the current candidate and the objects processed by the threads belonging to the same block. These heaps are stored in device memory in a bi-dimensional data structure of m rows (one for each candidate) and N_b columns (one for each block of the kernel *DToC1NDM*). This data structure occupies $N_b(k+1)mF$ bytes and substitutes the data matrix employed by *GPU-SolvingSet* algorithm, which instead occupies NmF bytes. Thus, in order to save space it must be guaranteed that $N_b(k+1)<N$.

From now we use the notation N_{b1} and N_{t1} (N_{b2} and N_{t2}, resp.) to denote the values for the parameters N_b and N_t employed in kernel *DToC1NDM* (*DToC2NDM*, resp.). Summarizing, the cost of the loop involving lines 1-6 in Figure 10 is

$$O\left(m\left(\frac{N}{N_p}(d+\log k)+\left(\frac{N_{b1}N_{t1}}{N_p}\log N_{t1}\right)\cdot(k\log k)\right)\right).$$

Kernel *DToC2NDM* takes care of merging the N_{b1} heaps associated with the k nearest neighbors in the N_{b1} blocks of each current candidate. The number of blocks N_{b2} of this kernel equals the number m of candidates (hence $N_{b2}=m$), while the number N_{t2} of threads per block is sensibly smaller than the number of blocks of the preceding kernel (hence $N_{t2}\ll N_{b1}$), since the threads in the same block are in charge of merging the N_{b1} heaps pertaining to the same candidate object. The cost of the while loop of kernel *DToC2NDM* is

$$O\left(\left(\frac{N_{b2}N_{t2}}{N_p}\right)\cdot\left(\frac{N_{b1}}{N_{t2}}\right)\cdot(k\log k)\right)\equiv O\left(m\frac{N_{b1}}{N_p}k\log k\right),$$

while the parallel reduction at the end of kernel costs

$$O\left(\frac{N_{b2}N_{t2}}{N_p}\log N_{t2}k\log k\right)\equiv O\left(m\frac{N_{t2}}{N_p}\log N_{t2}k\log k\right).$$

Assuming that $N_{t2}\log N_{t2}<N_{b1}$ (recall that $N_{t2}\ll N_{b1}$ holds) the latter cost is subsumed by the former one, which in its turn is subsumed by the cost of the parallel reduction of kernel *DToC1NDM*.

As for kernels *WeightUpd1* and *WeightUpd2* that are in charge of determining the candidates for the next iteration, their cost determination is similar to that of the two previous kernels, leading to

$$O\left(\left(\frac{N}{N_p}\right)\cdot\log m+\left(\frac{N_bN_t}{N_p}\log N_t\right)\cdot(m\log m)\right).$$

For the sake of simplicity, assume that $k\geq m$. Since these kernels are executed t times, the temporal cost of *GPU-SolvingSet-NDM* algorithm is eventually

$$O\left(\frac{tm}{N_p}\Big(N(d+\log k)+N_bN_t\log N_t\cdot(k\log k)\Big)\right).$$

Since $N_bN_t\ll N$, the cost of algorithm *GPU-SolvingSet-NDM* is dominated by the cost of algorithm *GPU-SolvingSet*.

During the execution of the kernel *DToC1NDM* each thread allocates its own max heap of k elements in shared memory. In order to alleviate this cost, the *GPU-SolvingSet-NDM-TP* algorithm avoids this allocation by employing the kernels *DToCTP1* and *DToCTP2* in place of *DToC1NDM*.

3.6.3 GPU-SolvingSet-S and GPU-SolvingSet-S-MS

Kernel *CandNN* pretty corresponds to the first branch of the kernel *DToC2*, having cost

$$O\left(\frac{m}{N_p}\Big(N\log k+N_t\log N_t\cdot(k\log k)\Big)\right),$$

which also corresponds to the worst case temporal cost of kernel *CandNNMS*. Since kernel *CandNN* (*CandNNMS*, resp.) is the heaviest operation performed by algorithm *GPU-SolvingSet-S* (*GPU-SolvingSet-S-MS*, resp.), the final cost of this algorithm is

$$O\left(\frac{tm}{N_p}\Big(N\log k+N_t\log N_t\cdot(k\log k)\Big)\right),$$

which is in its turn dominated by the cost of the algorithm *GPU-SolvingSet-NDM*.

4 Experiments

4.1 Experimental setting and datasets

We ran the distributed experiments and the *SolvingSet-MV* algorithm on Intel Haswell processors running at 2.40 GHz and hosting an NVIDIA Tesla K80 GPU with 4992 cores and 24GB of global memory, while for the other centralized CPU/GPU codes we used an Intel Xeon processor running at 2.40 GHz and hosting an NVIDIA Tesla M2070 GPU with 448 cores and 6GB of global memory. The CPU code is run on a single CPU core and makes use only of scalar single precision floating point operations. The 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 been 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.

For a given kernel function f, the aim of the parameter tuning module is to determine the optimal values (N_b^{opt}, N_t^{opt}) for the CUDA parameters N_b, number of thread blocks, and N_t, number of threads per block, respectively. The module reaches its goal by determining the values for N_b and N_t that allow to reach a pre-defined level of *occupancy*. Occupancy is a metric related to the number of active warps on a multiprocessor which is important in determining how effectively the hardware is kept busy. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. The *hardware specifications* are characteristic of the GPU model on which the algorithm is run and are already known to the module. Among these characteristics there are the *Threads per Multiprocessors*, the *Thread Blocks per Multiprocessor*, the *Shared Memory per Multiprocessor*, and others (for a complete list, see the NVIDIA *CUDA GPU Occupancy Calculator*). Given the value of the parameter k, which represents an input of the module, and a certain value for the CUDA parameter N_t (the *Threads per Block*), the *Registers per Thread* and the *Shared Memory per Block* can be obtained from the kernel function f structure. These three parameters are also referred to as *resource usage* parameters. The module must also take into account some constraints associated with the specific kernel. E.g., in some cases N_t is required to be a power of two since a reduction has to be performed, while when the kernel makes use of a data structure whose size is related to the block size, the shared memory per block depends on N_t. The combination of the resource usage parameters and of the hardware specifications allows to directly compute the occupancy. Higher occupancy does not always equate to higher performance, since there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation. Based on the literature, we selected as a rule of thumb to not exceed $occ_{\max} = 0.5$ occupancy [22]. Hence, the module enumerates all the admissible values for the parameter N_t and selects the pairs (N_b', N_t') associated with the largest occupancy not greater than $occ_{\max}$. We notice that the cost in charge of each kernel is directly proportional to the number of thread blocks. Moreover, during the tuning phase we verified that for a fixed number of threads per block, the best performance is obtained by taking the smallest number of thread blocks guaranteeing the pre-defined level of occupancy. Thus, among the pairs (N_b', N_t'), the pair (N_b^*, N_t^*) is selected by minimizing the value N_b', by taking either the minimum or the second minimum. The optimal number of thread blocks can be eventually obtained as $N_b^{opt} = N_b^* \cdot P$, where P is the number of multiprocessors. If N_b is set by the user, it is preferable that $N_b \geq N_b^{opt}$ and a multiple of N_b^{opt}. As for the optimal number of threads per block N_t^{opt}, it is set to $N_t^{opt} = N_t^*$.

In the experiments, we considered the following five datasets: *G3d*, synthetic, contains $500{,}000$ 3-d vectors; *Covtype*, real, includes $581{,}012$ instances of 10 attributes; *G2d*, synthetic, collection of $1{,}000{,}000$ 2-d vectors; *Poker*, real, consists of $1{,}000{,}000$ objects of 10 attributes; *2Mass*, real, contains $1{,}623{,}376$ 3-d instances. Further details on previous datasets are included in [3].

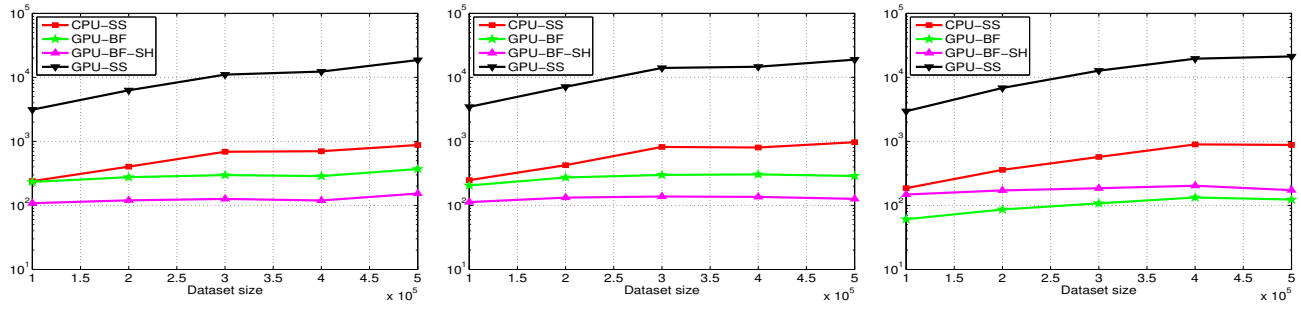
In the sequel, if not otherwise stated, the values for the detection task parameters, are $n = 10$, $k = 50$, and $m = 100$. We considered also other combinations of values for the above parameters, but the results are not completely reported in the paper, since the behavior of the algorithms does not change significantly.

For the sake of brevity, in the sequel we shorten algorithm names by using uppercase characters.

4.2 Comparison of the basic strategies

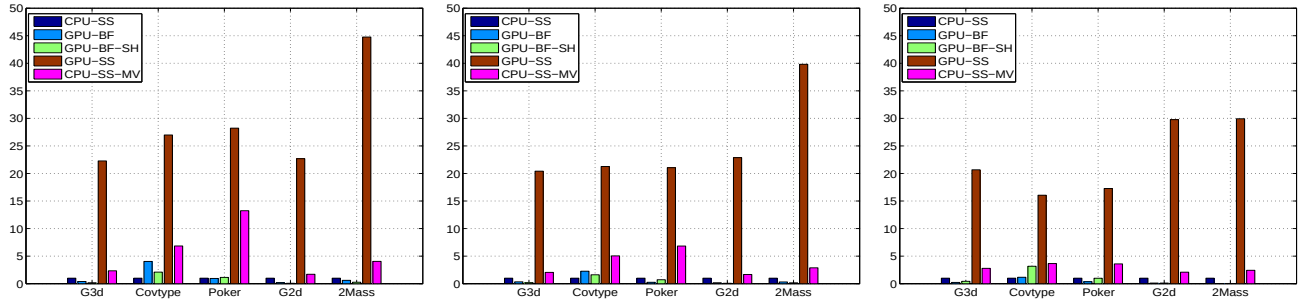
In this section, CPU-BF, CPU-SS, GPU-BF, GPU-BF-SH, GPU-SS, and CPU-SS-MV algorithms are compared.

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 18(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 approximatively 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 speed up 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 [5], 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 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.



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

Fig. 18. Speedup over CPU-BF on *G2d* for various values of k.



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

Fig. 19. Speedup over CPU-SS for various values of k.

Figures 18(b) and 18(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 shown later in this section). In general, it can be observed that the speed up 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 sensitive to the value of the parameter k.

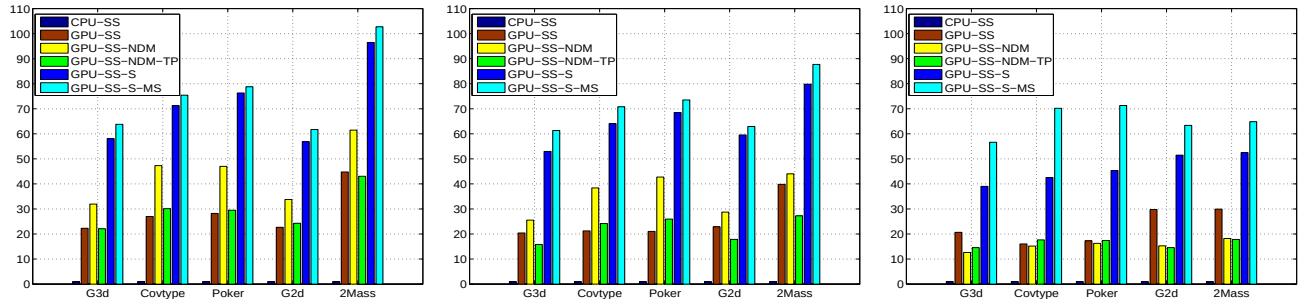
Figures 19(a)-19(c) report the speedup of the various methods w.r.t. CPU-SS for $k \in \{5, 10, 50\}$ on the datasets *G3d*, *Covtype*, *Poker*, *G2d*, and *2Mass*. The figures highlight that CPU-SS outperforms GPU-BF and GPU-BF-SH on all the datasets except for *Covtype*. In order to understand this behavior, the ratio between the number of distances computed by CPU-SS and the number of pairwise distances has been measured. The results are reported in the table below.

	$k = 5$	$k = 10$	$k = 50$
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%

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 previous 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.

We also measured the speedup of CPU-SS-MV. It can be observed that its performances are directly related both to the number of distances to be computed and the size of the dataset and are inversely related to the number k of nearest neighbors, and that the GPU version GPU-SS is able to achieve larger speedup.

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 speed up 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).



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

Fig. 20. Speedup over CPU-SS for various values of k.

This confirms that the brute force approach can exploit parallel architectures more efficiently than the solving set algorithm However, there is still room for improvements as accounted for in the next section.

4.3 Comparison of the GPU-SS variants

In this section, the CPU/GPU *SolvingSet*-based algorithms analysed in the earlier sections are compared. Specifically, the following algorithms are taken into account: GPU-SS (the basic method computing the full distance matrix), GPU-SS-NDM (the variant of the GPU-SS which does not materialize the full distance matrix and which stores the heaps in shared memory), GPU-SS-NDM-TP (the variant of the previous GPU-SS-NDM which stores the heaps in global memory during distance calculations and in shared memory during reduction), GPU-SS-S (the variant of the GPU-SS which uses two kernels to separately compute the *updateMin* and the *updateMax* operations), GPU-SS-S-MS (the variant of the previous GPU-SS-S method which optimizes the executions of *updateMin* operations by splitting the entire calculation in multiple steps), and CPU-SS.

Figures 20(a)-20(c) report 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 GPU-SS-NDM-TP variant is counterbalanced by the major cost to be paid on the additional operations involving the global memory. As for GPU-SS-S, the figures show a noteworthy improvement either over GPU-SS or over the two earlier variants. Finally, as for GPU-SS-S-MS, this variant presents a performance quite similar to its parent version (i.e. GPU-SS-S) for small values of k, whereas it offers a significant advantage for $k = 50$.

Thus, these experimental results make clear what are the scenarios most suitable for the presented strategies and what are the improvements that can be obtained by each of them. Naturally, GPU-SS-S-MS offers the best performance, whereas GPU-SS-NDM allows to reduce execution time with respect to the CPU version (and for $k < 50$ also with respect to GPU base version) maintaining a space cost lower than that of GPU-SS and of all the variants. Hence, GPU-SS-NDM could be the only applicable solving set GPU-based strategy on very large dataset (demanding very large amount of memory resources) or in the case the available hardware is limited in terms of storage.

4.4 Speedup of GPU-DSS

Figure 21 on the left shows the speedup $S_\ell = T_1/T_\ell$ of GPU-DSS exploiting GPU-SS-S-MS at local nodes, where T_ℓ denotes the execution time when ℓ local nodes are employed, for $k = 5$. Here, we considered three additional synthetic data sets, called 5G2d, 10G2d, and 100G2d, which have been generated in the same manner as G2d, but consist of 5, 10, and 100 million of instances, respectively. The figure highlights that the speedup increases with the size of the data to be processed. This behavior depends on two factors. The first one is the ratio between the supervisor node time and the total execution time. It can be seen from Figure 21 on the right that this ratio is sensible for the smallest dataset (up to 18%), and becomes less influential as the dataset size grows (less than 0.5% for the largest dataset). The second factor is the ratio between the dataset size and the device memory. We notice that the local dataset was not partitioned into chunks except for the first two executions of the largest dataset. This explains why the speedup tends to the ideal one for the three first datasets and why the curve of the last dataset exhibits initially a superlinear growth (up to 5 nodes) and then slows down approaching the unitary growth rate.

5 Conclusions

Due to the complexity of state-of-the-art algorithms for distance-based outlier detection, they may be impractical in on-line applications requiring short response times or handling very large data sets. Starting from the baseline sequential *BruteForce* and the

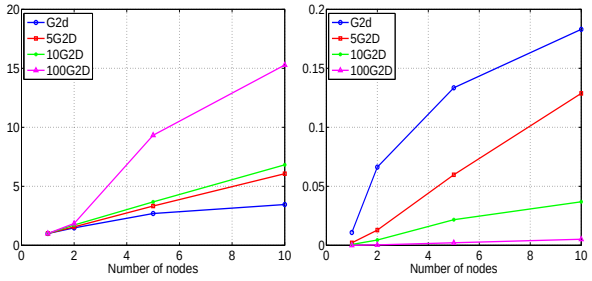


Fig. 21. Speedup (left) and ratio between the supervisor time and the total execution time (right).

SolvingSet algorithms, we implemented several GPU algorithms, both centralized and distributed. Some of them are aimed to find the best way to parallelize operations in the given architecture and memory hierarchy, other to reduce memory occupancy, and therefore more suitable for very large data sets. We have conducted an extensive set of experiments, including both real and synthetic large data sets, and we found a remarkable consistency of performance across data set sizes. To assess performance improvements, we have considered also a multicore version of the solving set algorithm, though the design of fully optimized multicore and hybrid CPU/GPU strategies deserves further investigation that we leave as a subject for future work. The experiments show that the approach is quite effective, with the speedup of the centralized strategies reaching two orders of magnitude in the best cases. The best results are given by the variants which explicit address the best usage of the parallel threads and memory hierarchy rather than the variants with a reduction of memory occupancy. As for the distributed approach, experiments highlight that ts speedup over the centralized GPU algorithm tends to the ideal one for datasets of increasing sizes.

REFERENCES

[1] M. Alshawabkeh, B. Jang, and D. Kaeli. Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems. In *GPGPU*, pages 104–110, 2010.
[2] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. A distributed approach to detect outliers in very large data sets. In *Euro-Par (1)*, pages 329–340, 2010.
[3] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. Distributed strategies for mining outliers in large data sets. *TKDE*, 25(7):1520–1532, 2013.
[4] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. Accelerating outlier detection with intra- and inter-node parallelism. In *HPCS*, pages 476–483. IEEE, 2014.
[5] F. Angiulli, S. Basta, and C. Pizzuti. Distance-based detection and prediction of outliers. *TKDE*, 18(2):145–160, 2006.
[6] F. Azmandian, A. Yilmazer, J. G. Dy, J. A. Aslam, and D. R. Kaeli. Gpu-accelerated feature selection for outlier detection using the local kernel density ratio. In *ICDM*, p. 51–60, 2012.
[7] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD*, 2003.
[8] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. Density-based clustering using graphics processors. In *CIKM*, pages 661–670, 2009.
[9] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 2009.
[10] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *Trans. Database Syst.*, 28(2):140–174, 2003.
[11] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. 3rd edition, 2011.
[12] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79:191–215, 1997.
[13] E. Hung and D. W. Cheung. Parallel mining of outliers in large database. *Distributed and Parallel Databases*, 12(1):5–26, 2002.
[14] K. Kato and T. Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *CCGRID*, p. 769-773, 2010.
[15] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403, 1998.
[16] 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).
[17] E. Lozano and E. Acuña. Parallel algorithms for distance-based and density-based outliers. In *ICDM*, p. 729-732, 2005.
[18] T. Matsumoto and E. Hung. Accelerating outlier detection with uncertain data using graphics processors. In *PAKDD II*, pages 169–180, 2012.
[19] NVIDIA Corporation. NVIDIA's next generation CUDA computing architecture: Fermi. Technical report, NVIDIA, 2009.
[20] M. E. Otey, A. Ghoting, and S. Parthasarathy. Fast distributed outlier detection in mixed-attribute data sets. *Data Min. Knowl. Discov.*, 12(2-3):203–228, 2006.
[21] N. Sismanis, N. Pitsianis, and X. Sun. Parallel search of k-nearest neighbors with synchronous operations. In *HPEC*, pages 1–6, 2012.
[22] V. Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC*, 10, 2010.
[23] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using GPUs. In *UCHPC-MAW*, pages 1–6, 2009.
[24] M. J. Zaki and Y. Pan. Introduction: Recent developments in parallel and distributed data mining. *Distributed and Parallel Databases*, 11(2):123–127, 2002.



Fabrizio Angiulli received the Laurea degree in Computer Engineering in 1999 from the University of Calabria (UNICAL). In 2001 he joined the ICAR Institute of the Italian CNR. Since 2006, he is with the DIMES Dept. of the UNICAL, where he is currently an Associate Professor of Computer Science. In 2013, he obtained the Full Professor qualification. His research interests include data mining and artificial intelligence.



Stefano Basta received the PhD degree in Systems Engineering and Informatics in 2000 from University of Calabria. Since 2000, he is Researcher for the Institute of High Performance Computing and Networking (ICAR) of Italian CNR. His research interests include logic programming, deductive databases, knowledge representation, information integration, and data mining.



Stefano Lodi received the PhD degree in computer science and electronic engineering in 1993 from the University of Bologna. He is an associate professor of Systems for Information Processing at the University of Bologna. His research interests include data clustering and classification by support vector machines in distributed and streaming environments, and semantic peer-to-peer systems.



Claudio Sartori is Full Professor in Systems for Information Processing, in the Department of Computer Science and Engineering of the University of Bologna. He has been doing research since 1984 in databases, information systems, artificial intelligence, distributed and peer-to-peer systems, data mining. He is the director of the graduate program "Computer Engineering", in the University of Bologna.