Noname manuscript No.
(will be inserted by the editor)

Scalable CAIM Discretization on Multiple GPUs Using Concurrent Kernels

Alberto Cano · Sebastián Ventura · Krzysztof J. Cios

Received: date / Accepted: date

Abstract CAIM (Class-Attribute Interdependence Maximization) is one of the state-of-the-art algorithms for discretizing data for which classes are known. However, it may take a long time when run on high-dimensional large-scale data, with large number of attributes and/or instances. This paper presents a solution to this problem by introducing a GPU-based implementation of the CAIM algorithm that significantly speeds up the discretization process on big complex data sets. The GPU-based implementation is scalable to multiple GPU devices and enables the use of concurrent kernels execution capabilities of modern GPUs. The CAIM GPU-based model is evaluated and compared with the original CAIM using single and multi-threaded parallel configurations on 40 data sets with different characteristics. The results show great speedup, up to 139 times faster using 4 GPUs, which makes discretization of big data efficient and manageable. For example, discretization time of one big data set is reduced from 2 hours to less than 2 minutes.

Keywords Supervised discretization · parallel implementation of CAIM algorithm · GPU · CUDA

A. Cano and S. Ventura are with the Department of Computer Science and Numerical Analysis University of Cordoba, Spain.
S. Ventura is also with the Information Systems Department. Faculty of Computing and Information Technology. King Abdulaziz University. 21589 Jeddah (Saudi Arabia).
E-mail: {acano,sventura}@uco.es

K. J. Cios is with the Department of Computer Science, Virginia Commonwealth University Richmond, VA, USA, and the IITiS Polish Academy of Sciences, Poland.
E-mail: kcios@vcu.edu

1 Introduction

Discretization is a data preprocessing technique which transforms continuous attributes into discrete values by dividing the continuous attribute into intervals, or bins [13,18,47]. CAIM (Class-Attribute Interdependence Maximization) [28] is a very popular top-down discretization algorithm that generates good discretization schemes for data for which the classes are known (supervised discretization). However, its performance becomes slow when the number of attributes and/or the number of distinct continuous data values increase, restricting its application to big data, meaning that they are either highly dimensional or have large number of instances. The problem of learning from such data is a challenging task that attracted attention of academia and industry [3,38]. Therefore, there is a need for new parallel solutions that perform within a reasonable time.

Multi-core parallel implementations have been used to speed up data mining algorithms over the last decade or so [7]. Increasing attention was focused on Graphic Processing Units (GPUs). GPUs are devices with multi-core architectures and massive parallel processor units, which provide fast parallel hardware solutions for a fraction of the cost of using traditional systems. Actually, since the introduction of the Computer Unified Device Architecture (CUDA) [34], researchers all over the world harnessed the power of the GPUs for general purpose GPU computing (GPGPU) [6, 12,35]. The use of GPGPU has been studied for speeding up algorithms within the framework of data mining [8,9,24,26], achieving high performance accompanied by good results. Owing to the great advantages provided by GPGPU, we would like to explore the performance of a GPU-based parallelization of the CAIM algorithm.

This paper presents a GPU-based implementation of the CAIM algorithm, which speeds up the discretization process and allows for handling big data. The GPU-based model is applicable to multiple GPU devices, enhancing its scalability to more complex and high-dimensional data sets. The GPU-based model takes advantage of the recent capabilities of Fermi and Kepler NVIDIA GPU architectures to improve efficiency of parallel computation by means of concurrent kernels execution. Experiments were carried out on 40 data sets to evaluate the run-time performance of the original CAIM algorithm (on a single-threaded CPU) vs. the multi-threaded CPU-based CAIM or vs. the GPU-based CAIM. The data sets had different number of instances, attributes, and classes, representing a varied collection of real-world problems. Experiments were performed having in mind these two objectives. Namely, measuring the speedup and analyzing the scalability to big data, regarding to the data set complexity. Experimental results show that current multi-core CPUs can speed up the discretization process up to 4 times, whereas great speedups up to 139 are obtained when using 4 GPUs. Results show good scalability of the GPU-based model to multiple GPU devices, as well as to big high-dimensional data sets, which is a significant advantage over the original CAIM. Specifically, the run-time on one of the most complex data sets evaluated was reduced from 2 hours to less than 2 minutes.

The remainder of this paper is organized as follows. The next section presents related work about GPU implementations of data mining techniques. Section 3 introduces the CAIM discretization scheme. Section 4 describes the GPU discretization model. Section 5 presents the results of the experimental study. Finally, Section 6 presents some concluding remarks.

2 Background

Data discretization is a challenging task that comprises a vast literature background, which has recently been reviewed by *García et al.* [18] using both theoretical and empirical perspectives. The foundations of the CAIM algorithm [28] have been considered in multiple research articles to propose new discretization methods. Specifically, *Sriwanna et al.* [43] proposed an enhanced CAIM discretization which addressed a new stopping criterion which eventually improved the accuracy. *Li et al.* [29] proposed an improvement of the CAIM algorithm based on the class-attribute coherence.

The increasing size of data sets increase the runtime of the data mining algorithms, and specifically, the data discretization methods. The use of parallel architectures has been a traditional approach to overcome the scalability of algorithms to large data sets. Mining very large databases with parallel processing addresses the problem of large-scale data mining [17,49].

Over the last few years, the use of GPUs to speed up computationally intensive algorithms has become a major niche for researchers [35]. Proof of this trend is the high number of research works about the use of GPU computing applied to general purpose computation [33]. GPUs demonstrated to achieve high performance on solving complex problems on physics [5], medicine [40,41,44], and computer science [6, 12,21]. Specifically, we focus on GPU implementations of data mining [24] and machine learning [45] algorithms. The use of GPU computing is justified due to the increasing size of databases, which requires parallel solutions capable of handling such data in reasonable time.

Classification and clustering are two common tasks in data mining which have been accelerated using GPU environments [9,46,50]. There are many works proposing GPU parallel implementation of algorithms on such tasks. However, there are few works addressing parallel implementations of discretization methods and none of them are on GPUs. Therefore, we think that it would be very interesting to analyze the performance of the GPU computing on speeding up the data discretization task.

Cerquides and Lopez [11] proposed a discretization method based on a distance metric between partitions that can be easily implemented in parallel. This similarity based method was very effective and efficient in very large data sets. *Yulong et al.* [48] proposed an efficient two-step parallel discretization algorithm by dynamic clustering. The algorithm first creates a decision table using a dynamic clustering algorithm, and then discretizes using cut importance breakpoints.

Parthasarathy and Ramakrishnan [36] proposed a parallel and incremental discretization algorithm for stream data, which dynamically maintains the required information even in the presence of data updates without re-executing the algorithm on the entire data set. *Zhao et al.* [51] presented a parallel discretization algorithm based on the Z-score idea of mathematical statistics. This algorithm reflected the dynamic semantic distance for different numerical attributes and significantly enhanced the precision rate and recall rate of data prediction algorithms.

Table 1: Quanta Matrix for Attribute F and Discretization Scheme D.

Class	Interval					Class Total
	$[f_1,f_2)$	...	$[f_{k-1},f_k)$	...	$[f_{n-1},f_n]$	
C_1	q_{11}	...	q_{1k}	...	q_{1n}	M_{1+}
:	:	...	:	...	:	:
C_i	q_{i1}	...	q_{ik}	...	q_{in}	M_{i+}
:	:	...	:	...	:	:
C_S	q_{S1}	...	q_{Sk}	...	q_{Sn}	M_{S+}
Interval Total	M_{+1}	...	M_{+k}	...	M_{+n}	M

3 CAIM discretization

The CAIM algorithm defines a discretization scheme D on every continuous attribute F of a data set consisting of M instances, where each instance belongs to only one of the S classes (supervised discretization). Scheme D discretizes a continuous attribute F into n discrete intervals bounded by the pairs of numbers arranged in ascending order:

$$D : \{[f_1, f_2), [f_2, f_3), ..., [f_{n-1}, f_n]\} \tag{1}$$

where f_1 is the minimal value and f_n is the maximal value of the continuous attribute F.

The quanta matrix shown in Table 1, represents a two-dimensional frequency matrix containing the number of instances that belong to each discretization interval and data class, where q_{ik} is the total number of continuous values belonging to the *i*-th class that are within interval $[f_{k-1}, f_k)$. M_{i+} is the total number of instances belonging to the *i*-th class and M_{+k} is the total number of continuous values of attribute F that are within the interval $[f_{k-1}, f_k)$, for $i = 1, 2, ..., S$ and $k = 1, 2, ..., n$.

The Class-Attribute Interdependency Maximization (CAIM) criterion measures the dependency between the class variable C and the discretization scheme D for attribute F as follows:

$$CAIM(C, D|F) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{{max_k}^2}{M_{+k}} \tag{2}$$

where n is the number of intervals, k iterates through all intervals, max_k is the maximum value among all q_{ik} values (maximum value within the *k*-th column of the quanta matrix), M_{+k} is the total number of continuous values of attribute F that are within the interval $[f_{k-1}, f_k)$. The CAIM criterion is a heuristic measure that is used to quantify the interdependence between classes and the discretized attribute. For details about the CAIM criterion and the CAIM algorithm, the reader is referred to the original CAIM article [28].

Algorithm 1 CAIM algorithm

Input: Data of M instances, S classes, and F attributes

```
 1: for every $F_i$ do
 2:     Sort all distinct values of $F_i$ in ascending order.
 3:     Find the minimum $f_{min}$, maximum $f_{max}$ values of $F_i$.
 4:     Initialize all possible interval boundaries $B$ with $f_{min}$, $f_{max}$, and all midpoints of adjacent pairs
        in the set.
 5:     Set discretization scheme $D$ = $\{[f_{min}, f_{max}]\}$.
 6:     Initialize iter = 1.
 7:     $\mathrm{CAIM_D} \leftarrow$ CAIM value of $D$.
 8:     Evaluate the CAIM value of the tentative schemes using $D$ and the points from $B$.
 9:     $\mathrm{CAIM_{max}} \leftarrow$ Select the highest valued midpoint.
10:     if ($\mathrm{CAIM_{max}} > \mathrm{CAIM_D}$ or iter $<$ S) then
11:         Update $D$ with the midpoint from $\mathrm{CAIM_{max}}$.
12:         iter = iter + 1 and go to step 7.
13:     else
14:         return Discretization scheme $D$ for attribute $F_i$.
15:     end if
16: end for
```

Output: Discretization scheme for all attributes

3.1 The CAIM algorithm

The CAIM algorithm uses a greedy approach searching for an approximate optimal value of the CAIM criterion by finding locally maximum values of the criterion. It consists of these steps:

- Initialization of the interval boundaries and creation of the discretization scheme D = $\{[f_1, f_n]\}$.
- Consecutive additions of new boundaries that result in the locally highest value of the CAIM criterion.

The pseudocode of the CAIM discretization algorithm is repeated here after [28], and shown as Algorithm 1. The algorithm begins with a single interval and splits it iteratively. From all possible division points, which are evaluated in line 8, the algorithm chooses the division boundary that provides the highest value of the CAIM criterion. The CAIM algorithm assumes that every discretized attribute needs at least a number of intervals equal to the number of classes.

The CAIM uses a trade-off between finding the optimal discretization scheme and having a reasonable computational cost. In spite of the greedy behavior of the algorithm, the discretization schemes that are generated have small number of discretization intervals and high class-attribute interdependency. For the data sets used in the experimental section, the CAIM algorithm generates discretization schemes with the (possibly) smallest number of intervals that assures low-computational cost. For details about the CAIM algorithm the reader is referred to the CAIM article [28].

4 CAIM discretization on GPUs

This section presents the GPU implementation of the CAIM algorithm based on the analysis of the steps of the algorithm to propose the most efficient way of addressing its computations. The GPU-based model comprises three different levels of coarse and fine-grained parallelism which are described in the following sections. First, we introduce the CUDA programming model and some definitions. Second, parallel sorting of the continuous attribute values is discussed. Third, parallel computation of the CAIM values for each tentative interval is explained and the new GPU kernel implementation is presented. Fourth, the extension of the GPU model to multiple GPU devices is described. Finally, execution of concurrent kernels on the GPU is presented.

4.1 CUDA programming model

CUDA is a parallel computing architecture developed by NVIDIA that allows programmers to take advantage of the parallel computing capacity of NVIDIA GPUs. The CUDA programming model executes kernels as batches of parallel threads [34]. These kernels run thousands to millions of lightweight GPU threads per each kernel call.

CUDA's threads are organized into thread blocks in the form of a grid. Thread blocks are executed on the GPU streaming multiprocessors. A stream multiprocessor can perform zero-overhead scheduling to interleave warps (a warp is a group of threads that execute together) and hide the overhead of long-latency arithmetic and memory operations. Current Fermi and Kepler NVIDIA GPU's architectures allow up to 16 kernels to be executed concurrently as long as there are enough multiprocessors available. Moreover, asynchronous data transfers can be performed concurrently with the kernel executions. These two features allow for speeding up execution of an algorithm as compared with sequential kernel pipeline and synchronous data transfers.

There exist some guidelines for improving performance of an algorithm on a GPU [19,34]. According to the NVIDIA CUDA programming guide and the best practices guide, the single most important performance consideration in programming for CUDA is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp are coalesced by the device into as few as one transaction when certain access requirements are met. Global memory resides in device memory and is accessed via 32, 64, or 128-byte segment memory transactions. It is better to perform fewer but larger memory transactions. When a warp executes an instruction which accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of memory transactions depending on the size of the word accessed by each thread and distribution of the memory addresses across the threads. In general, the more transactions are necessary the more unused words are transferred on the N-byte segment in addition to the words accessed by the threads, reducing the instruction throughput accordingly.

To achieve maximum performance and memory throughput memory accesses must be coalesced to global memory by following optimal memory access patterns, using data types that meet the size and alignment requirements, or padding data [34]. For these accesses to be fully coalesced, both the width of the thread block and the width of memory arrays must be a multiple of the warp size. We considered all these aspects when designing our GPU CAIM implementation, kernel implementations, data types, and memory alignments.

4.2 Parallel sorting of attribute values

The CAIM algorithm sorts the continuous attribute values in ascending order (Line 2 of Algorithm 1). Sorting is a building block of fundamental importance and is one of the most widely studied algorithmic problems [14,27]. The original CAIM implementation uses a tuned version of the quicksort algorithm [4,22]. This algorithm has O($m\ log\ m$) performance, where m is the number of distinct values of a discretized attribute, but in the worst case its performance may degrade to O(m^2). Therefore, when the number of distinct continuous values and the number of attributes increase, this step becomes very slow.

The importance of sorting in many areas dealing with big data has lead to the design of efficient sorting algorithms for a variety of parallel architectures [1]. Many studies focused on GPU parallel implementation of sorting algorithms [15,25,39,42]. Examples are bitonic merge [37] and quicksort [10]. The highest performance O(m) has a radix sort sorting algorithm on GPUs [30,32], which is publicly available in the Thrust library [23]. This highly-tuned radix sort algorithm [31] is considerably faster than alternative comparison-based sorting algorithms such as merge sort [39]. Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). The CAIM algorithm on GPUs also uses the radix sort to sort the continuous attribute values. It does not require any user-specified parameters or configuration settings, as the Thrust automatically selects them according to the sorting problem length. Thrust fast implementation of radix sort allows for sorting arrays with millions of values. It works by iterating over the d-bit digit-places of the values from the least-significant to the most-significant. For each digit-place, the method performs a stable distribution sort of the values based upon their digit at that digit-place. Given an m-element sequence of b-bit values, a radix sort of these values will require b/d iterations of a distribution sort over all m values [32].

4.3 Parallel computation of CAIM values

The CAIM algorithm evaluates the CAIM value for all tentative discretization schemes resulting from all possible division points of the attribute. This process becomes very slow when the number of attributes or the number of distinct continuous values increases. Fortunately, the computation of the CAIM value for a scheme is an independent operation from the computation of the rest of CAIM values. Therefore, the CAIM value for each tentative discretization scheme can be computed in parallel and concurrently.

The approach followed here to solve the parallel computation of the CAIM values on GPUs is one-dimensional. GPU threads are organized in an array of parallel

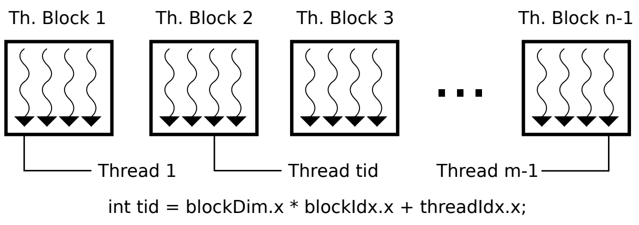

Fig. 1: CUDA grid of thread blocks.

threads, where each thread is responsible of evaluating the CAIM value of a single discretization scheme. Therefore, given m distinct continues values for a given attribute, the number of tentative discretization schemes to evaluate is $m - 1$, which represents the number of parallel threads to compute.

Threads are grouped into thread blocks whose dimension is recommended to be a multiple of the warp size (a warp is a group of threads that execute together and whose number depends on the compute capability of the GPUs), usually being 128, 192, 256, ..., up to 1024 threads per block. Choosing an appropriate number is important as it influences scalability of the model on future GPU devices that may have larger number of processors. NVIDIA recommends running at least twice as many thread blocks as the number of multiprocessors in the GPU, and provides an occupancy calculator which reports the GPU occupancy regarding the kernel requirements and the number of threads per block [34]. This spreadsheet reported that 256 threads per block is the best choice for our problem since it achieves full occupancy and provides more active thread blocks per multiprocessor. This way, it is possible to hide latency arising from register dependencies, and therefore, a wider range of possibilities are given to the scheduler to issue concurrent blocks to the multiprocessors. Moreover, while the GPU occupancy (usage of GPU resources) is maximal, the smaller number of threads per block the higher the number of blocks. This provides better scalability to future GPU devices which will have many more processors capable of handling more active blocks concurrently.

Threads and blocks within the kernel are identified by the built-in CUDA variables *threadIdx*, *blockIdx* and *blockDim*, which specify the grid and block dimensions and the block and thread indexes. Figure 1 shows the one-dimensional representation for the array of threads grouped into thread blocks, along with the unequivocal identification of a thread using the built-in variables. This one-dimensional representation limits the maximum number of distinct continuous values to 2^{24}, i.e., more than 16 million values per attribute, which enables handling all the benchmark problems evaluated so far. If higher number of values were required, the one-dimensional approach would be extended to the other two CUDA dimensions, forming a 3D grid of threads.

Code 1 shows the CAIM computation kernel on GPUs. It receives as input arguments the quanta matrix (described in Table 1), the current discretization scheme, and the number of current intervals. It computes the CAIM values for the tentative discretization schemes using Equation 2 and outputs their values as a float array. Finally, the discretization point with the highest CAIM value is selected, the discretization is performed, and the next iteration of the CAIM algorithm is run.

Code 1 CAIM computation kernel

```
// Kernel function
__global__ void computeCAIM(float* CAIMValues, int* quantaMatrix,
                            int* discretizationScheme, int numberIntervals)
{
  // Thread identifier
  int tid = blockDim.x * blockIdx.x + threadIdx.x;

  float CAIMValue = 0.0f;

  for(int i = 0; i < numberIntervals; i++)
  {
    int leftValueIdx = discretizationScheme[i];
    int rightValueIdx = discretizationScheme[i+1];

    if(leftValueIdx <= tid && tid < rightValueIdx)
    {
      // The current value splits the interval
      CAIMValue += CAIM(leftValueIdx, tid,   quantaMatrix);
      CAIMValue += CAIM(tid , rightValueIdx, quantaMatrix);
    }
    else
      CAIMValue += CAIM(leftValueIdx, rightValueIdx, quantaMatrix);
  }

  // Write the CAIM value in global memory
  CAIMValues[tid] = CAIMValue / numberIntervals;
}

// Device function
__device__ float CAIM(int leftValueIdx, int rightValueIdx, int* quantaMatrix)
{
  int columnSum [numberClasses] = {0}; // (M_{+k})

  // Calculate the sum of all rows for each column of the quanta matrix
  for(int i = leftValueIdx; i < rightValueIdx; i++)
    for(int j = 0; j < numberClasses; j++)
      columnSum[j] += quantaMatrix[i*numberClasses + j];

  int columnMax = 0; // (Max_k)
  int sumTotal  = 0; // (M)

  // Calculate the total sum of the quanta matrix and
  // the max value for each column
  for(int j = 0; j < numberClasses; j++)
  {
    sumTotal += columnSum[j];

    if(columnSum[j] > columnMax)
      columnMax = columnSum[j];
  }

  // Compute CAIM value
  float CAIM = columnMax / (float) sumTotal;
  CAIM = CAIM * columnMax;

  return CAIM;
}

//// Host call to kernel function

// Grid and block size
dim3 threadsCAIMValues(256);
dim3 gridCAIMValues((int) ceil(numbermidPoints / 256.0f));

// Copy tentative intervals to the GPU memory
cudaMemcpy(d_intervals, h_intervals, numberIntervals * sizeof(int), 1);

// Kernel computation
computeCAIM <<< gridCAIMValues, threadsCAIMValues >>>
(d_caimValues, d_quantaMatrix, d_intervals, numberIntervals);

// Copy CAIM results back to host memory
cudaMemcpy(h_caimValues, d_caimValues, numbermidPoints * sizeof(float), 2);
```

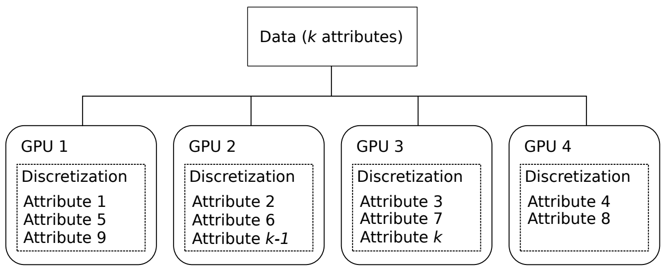

Fig. 2: Distribution of discretization on multiple GPUs.

The selection of the highest valued discretization point is again a parallelizable procedure since it consists on finding the largest value in an array. Thrust library provides the *max_element()* function which finds the largest value in an array and has $O(m)$ complexity.

4.4 Scalability to multiple GPU devices

As mentioned above, the computation of the CAIM values is independent for all the tentative discretization schemes of the attribute. This constitutes the first level of fine-grained parallelism. Moreover, the discretization process for each attribute is also an independent operation in the CAIM algorithm. Therefore, the second level of coarse-grained parallelism can be constituted for discretization of each attribute, which can also be done in parallel and concurrently. The way to take advantage of multiple GPUs and allow the scalability of the model to high-dimensional data sets is to distribute the discretization processes of the attributes into multiple GPU devices. This approach is scalable from a single GPU to as many GPUs as the number of attributes. Figure 2 shows the distribution of the discretization process of k attributes among 4 GPUs. The discretization processes within a GPU may overlap their executions. The same approach could be followed to distribute the discretization process among a cluster of GPUs, which would enable handling data sets with very many attributes.

4.5 Concurrent Kernels

Since introduction of Fermi and Kepler NVIDIA GPU architectures, applications have the ability to launch several kernels concurrently. This provides a mechanism by which applications can fill the device with several smaller kernel launches simultaneously as opposed to a single larger one.

Concurrent kernels execution constitutes the third level of the coarse-grained parallelism in our GPU model. Concurrent kernels are issued by means of CUDA streams, which allow asynchronous data transfers between host and device memory, overlapping data transfers and kernel executions [20]. In this way, discretization process for different attributes on the same GPU may overlap their executions, thus improving the efficiency of the implementation. Figure 3 shows the timeline for the

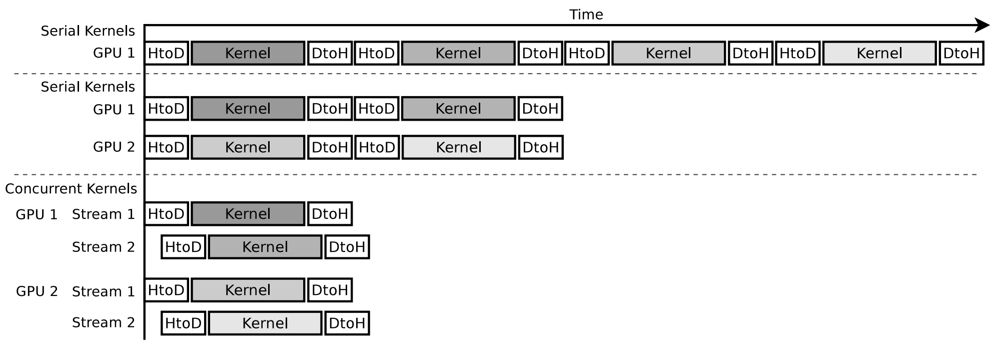

Fig. 3: Serial and concurrent kernels execution timeline.

serial execution of kernels on one and two GPUs, as well as the concurrent execution of kernels on two GPUs. The serial execution on a single GPU sequences memory transactions and kernel executions. Typically, it performs memory transfers from host to device (HtoD) to copy the data to the GPU memory. Then, the kernel is executed, and results are copied back to the host memory using device to host (DtoH) memory transfers. Having more than one GPU enables distributing kernels computation to multiple devices. Finally, the concurrent kernels execution capability of modern GPUs allows for distributing computation into multiple streams within the GPU. Therefore, the combination of concurrent kernels and distribution into multiple GPUs constitutes an efficient and scalable framework for parallel computation. However, it is also important to highlight that for small data sets, the size of the kernels are small, and therefore the GPU computation may be inefficient as compared to the CPU.

5 Experimental studies

This section presents the experiments and discusses the results. First, the hardware configuration is described. Second, the data problem is presented. Finally, the experiments and their results are discussed. Detailed information about the data sets, the algorithm's source code and experiments is provided as additional material at this website[1].

5.1 Hardware configuration

The experiments were run on a machine equipped with an Intel Core i7 quad-core processor running at 3.0 GHz and 12 GB of DDR3-1600 host memory. The video cards used were two dual-GPU NVIDIA GTX 690 equipped with 4 GB of GDDR5

[1]The data sets description, the algorithm's source code, the experimental settings and results are fully described and publicly available to facilitate the replicability of the experiments and future comparisons at the website: `http://www.uco.es/grupos/kdis/kdiswiki/CAIM-GPU`

video RAM per dual-GPU card. Each GTX 690 video card had two GPUs with 1,536 CUDA cores. In total there were 4 GPUs and 6,144 CUDA cores at default clock speeds. The host operating system was GNU/Linux Ubuntu 12.10 64 bit along with CUDA runtime 5.0, NVIDIA drivers 310.32, and GCC compiler 4.6.3 (O2 optimization level).

5.2 Data sets

The performance of the GPU-based CAIM implementation was evaluated on 40 data sets from the UCI machine learning [16] and the KEEL data sets repositories [2]. These data sets varied in terms of degrees of complexity. The number of instances ranged from 150 instances to about half million instances, while the number of distinct continuous values per attribute varied from 27 to 6,754. The number of attributes ranged from 4 to 649 and the number of classes varied from 2 to 28. The wide variety of data sets allowed us to evaluate the model performance on real-world problems of both low and high complexity.

5.3 Results

This section presents the results of the different experiments carried out. The speedup is measured and analyzed over 40 data sets. The speedup scalability is then analyzed in respect to the number of distinct values and the number of GPU devices used. Finally, the results of efficiency improvement using concurrent kernels are presented.

5.3.1 Speedup analysis

Table 2 shows information about the data sets (numbers of instances, attributes, classes, and the average number of distinct continuous values per attribute), the execution times and the speedups of the CPU and GPU parallel implementations versus the original single-threaded CAIM algorithm. The speedup using 4 CPU threads from the quad-core processor is up to 4 times faster than the original CAIM. However, overhead introduced due to threads creation reduces the final speedup to values around 3 times faster than original CAIM.

The GPU model obtained very good speedups up to 139 times faster than the original CAIM when using 4 GPUs (twonorm data set). Speedups are generally higher as the number of attributes and the number of distinct continuous values of the attributes increase. For instance, the *KDD99* data set has 41 attributes (only 26 are numeric), but the most important fact is that even though the number of instances is about half a million, the number of distinct continuous values is significantly lower (many real values are repeated). Therefore, the number of tentative discretization schemes on which to compute the CAIM value in parallel is relatively low. However, for the *twonorm* data set with significantly lower number of instances but all of them being different continuous values, the number of tentative schemes to be evaluated in parallel is significantly higher than for the *KDD99* data. On the other hand, there are some

Table 2: Data sets execution time and speedup as compared with CAIM.

Data set					Execution time (ms)					Speedup			
Name	#Inst	#Att	#C	#DV	1 CPU	4 CPUs	1 GPU	2 GPUs	4 GPUs	4 CPUs	1 GPU	2 GPUs	4 GPUs
Abalone	4174	8	28	756	17716	7391	1227	711	493	2.40	14.44	24.92	35.94
Arrhythmia	452	279	16	54	2246	657	2041	1013	561	3.42	1.10	2.22	4.00
Eye-mvmnts	10936	25	3	2434	5558	2122	215	146	110	2.62	25.85	38.07	50.53
Glass	214	9	7	101	37	11	37	22	12	3.36	1.00	1.68	3.08
Heart	270	13	2	27	8	5	9	8	7	1.60	0.89	1.00	1.14
Ionosphere	351	33	2	220	85	37	44	27	25	2.30	1.93	3.15	3.40
Iris	150	4	3	31	4	3	5	4	3	1.33	0.80	1.00	1.33
Isolet	7797	617	26	3232	7239654	2107783	421029	220416	110673	3.43	17.20	32.85	65.41
JM1	10885	21	2	1354	1166	419	53	36	31	2.78	22.00	32.39	37.61
KDD09-app	50000	204	2	2307	69206	22061	1404	1162	868	3.14	49.29	59.56	79.73
KDD09-chu	50000	204	2	2307	69459	21865	1420	1150	880	3.18	48.91	60.40	78.93
KDD09-ups	50000	204	2	2307	69597	22699	1411	1141	833	3.07	49.32	61.00	83.55
KDD99	494020	41	23	471	137417	79309	7267	5879	5637	1.73	18.91	23.37	24.38
Madelon	2600	500	2	140	1115	266	642	364	421	4.19	1.74	3.06	2.65
MC1	9466	38	2	193	244	74	63	38	41	3.30	3.87	6.42	5.95
Mfeat-factors	2000	216	10	215	3855	1071	1719	847	452	3.60	2.24	4.55	8.53
Mfeat-fourier	2000	76	10	1994	39851	10822	3583	1807	911	3.68	11.12	22.05	43.74
Mfeat-karhun	2000	64	10	1994	33138	8781	3020	1518	773	3.77	10.97	21.83	42.87
Mfeat-morph	2000	7	10	785	1231	499	136	91	55	2.47	9.05	13.53	22.38
Mfeat-zernike	2000	47	10	1994	24491	6469	2210	1133	578	3.79	11.08	21.62	42.37
Multiple	2000	649	10	657	96585	26695	11203	5661	2998	3.62	8.62	17.06	32.22
PC2	5589	36	2	165	120	50	51	33	36	2.40	2.35	3.64	3.33
Penbased	10992	16	10	100	172	73	99	56	35	2.36	1.74	3.07	4.91
Pendigits	10992	16	10	100	169	57	94	52	32	2.96	1.80	3.25	5.28
Pima	768	8	2	157	22	13	14	11	7	1.69	1.57	2.00	3.14
Ring	7400	20	2	3756	2154	593	79	49	38	3.63	27.27	43.96	56.68
Satimage	6435	36	7	78	164	51	126	75	47	3.22	1.30	2.19	3.49
Segment	2310	19	7	786	1315	501	219	128	86	2.62	6.00	10.27	15.29
Sensor-disc	2212	13	3	1104	258	140	51	35	24	1.84	5.06	7.37	10.75
Sonar	208	60	2	137	96	47	76	45	30	2.04	1.26	2.13	3.20
Spambase	4597	57	2	265	205	94	83	51	45	2.18	2.47	4.02	4.56
Sylva-agno	14395	41	2	424	651	242	75	49	28	2.69	8.68	13.29	23.25
Sylva-prior	14395	21	2	414	318	140	38	28	20	2.27	8.37	11.36	15.90
Texture	5500	40	11	989	7470	2060	1185	596	312	3.63	6.30	12.53	23.94
Thyroid	7200	21	3	67	43	22	18	15	15	1.95	2.39	2.87	2.87
Twonorm	7400	20	2	6754	5169	1397	102	55	37	3.70	50.68	93.98	139.70
Vowel	990	13	11	623	1215	378	270	138	82	3.21	4.50	8.80	14.82
Waveform	5000	40	3	625	538	181	127	74	63	2.97	4.24	7.27	8.54
Winequality$_r$	1599	11	11	132	108	64	89	51	32	1.69	1.21	2.12	3.38
Winequality$_w$	4898	11	11	210	269	180	119	74	52	1.49	2.26	3.64	5.17

data sets such as *heart* or *iris* in which the GPU computation is inefficient. This is due to the small size of these data sets, having very small number of distinct values. Under such circumstances, the GPU parallelization is not worthy since the data transfer times between host and device memory are more expensive than the compute time

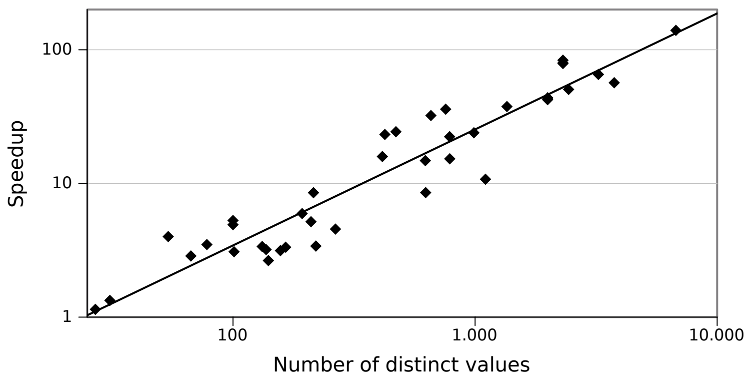

Fig. 4: Speedup vs. number of distinct values.

saved. Therefore, it is recommended to stay on the CPU when handling very small data sets with low small number of distinct values.

However, it is also important to highlight that the discretization process finishes only when the computation for all the attributes is finished. Therefore, when using multiple GPUs (each one discretizing a subset of the attributes set) the CPU main thread must wait until all the GPU devices have finished their work. It may happen that one GPU finishes sooner than the others, but all have to wait until the latest' work is done. The number of distinct values for each attribute is different (and not known a priori). Thus, one GPU may be assigned to discretize attributes having high number of distinct values, whereas other GPUs computing other attributes having smaller number of distinct values. Under such circumstances, synchronization between GPUs may introduce different delays depending on the attributes distribution and the number of GPUs available.

Therefore, the more concurrent threads are run for the *twonorm* data set the better speedups are obtained. In other words, in spite of the high number of instances of the *KDD99* data set the number of different continuous values is small so that it does not provide full occupancy to the GPU cores. Figure 4 shows the relation between the number of distinct continuous values and the speedup achieved when using 4 GPUs and the 40 data sets (each dot represent the speedup on a data set). It is easy to notice the increase of the speedup as the number of distinct values becomes higher, and it can be observed a linear regression line that highlights that.

It is also interesting to point out the limits of the GPU implementation regarding those data sets in which the number of values is small. For these data sets, such as *heart*, *iris*, or *glass*, the small number of values does not allow for a massive parallel computation of discretization schemes. Therefore, even though there is a lower execution time of the CAIM on GPU, the overhead introduced due to memory transfers between the host and device memory makes the total time similar or even slightly higher than using the original CPU algorithm. Thus, it is not recommended to run discretization on very small data sets on GPU since the data transfer times may be higher than the saved time due to GPU implementation. However, it is only a small drawback since the CAIM algorithm run on CPU on small data sets is very fast and therefore there is no need for yet faster execution. For big and high-dimensional data sets however, there is a strong need for GPU parallelization. For example, CAIM on GPUs reduced execution time from two hours to less than two minutes for the *isolet* data set.

5.3.2 Scalability analysis

Experimental results also indicate good scalability of the GPU implementation using one, two and four GPU devices. Note that in those data sets which stress the GPUs enough (run high enough number of threads to utilize the GPUs computing capabilities) by distributing computation of the discretization process into multiple devices significantly reduces the execution time. This behavior can be observed in data sets such as *isolet*, *twonorm*, or *ring*. Moreover, it is achieved not only in the best performance scenarios, but on the regular ones, in which doubling the number of GPU devices reduces the execution time almost by half. Figure 5 shows the speedup scalability (overall trendline among all data sets) using one, two, and four GPUs in respect to the increasing number of distinct values of the data sets. This way we outline the overall scalability of our model to multiple GPUs, which is promising for further extensions to larger GPU clusters with higher number of devices. The current model is limited to take advantage of as many GPU devices as the number of attributes, and also the maximum number of distinct values of an attribute is limited to the memory capacity of the GPU. Nevertheless, modern GPUs are plenty of memory and they can handle dozens of millions of distinct values. If the data set had higher number of values, the data would be split into multiple GPUs.

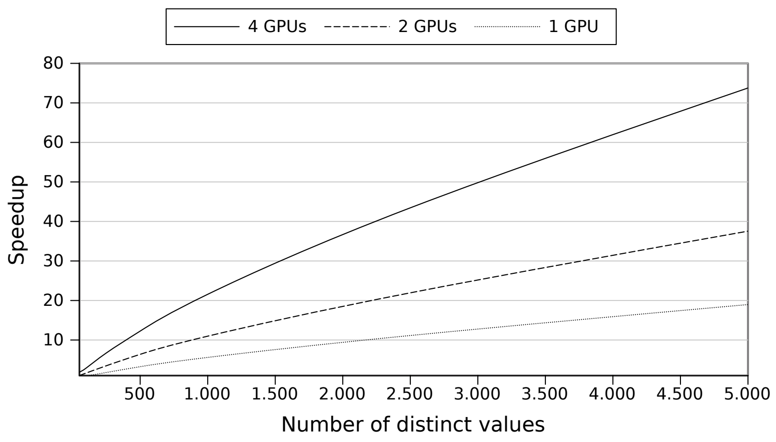

Fig. 5: GPUs scalability.

5.3.3 Concurrent kernels performance

This section presents the performance improvements when using concurrent kernels execution. Figures 6 and 7 show performance profiling of the serial and concurrent kernels execution on the GPUs, obtained from the NVIDIA Visual Profiler developer tool (each kernel represents the CAIM computation of a different attribute and it is drawn in different color). The data set evaluated were the *twonorm*, with 2 classes and 20 attributes to discretize using 4 GPUs. In this experiment each GPU is responsible for discretization of 5 of the 20 attributes. Since there are two classes, the kernels are executed two times according to the CAIM algorithm specification, as described in Section 3.1. Thus, the number of kernel executions per GPU is 10, and they are named from $K1$ to $K10$.

These figures show the timeline for each GPU, including the host to device (HtoD) memory transfers, device to host (DtoH) memory transfers, and kernel executions. In Figure 6, the execution of the kernels for the 5 respective attributes is serialized, i.e, one kernel runs only after the previous one finished. In Figure 7, the execution of the kernels is concurrent (overlapped in time). The maximum number of concurrent kernels for the Kepler GPU architecture used in the experiments is 16. Thus, distributing more than 16 discretization processes to a GPU would result in the execution of multiple batches of concurrent kernels. Nevertheless, this process is completely transparent to users and developers and NVIDIA reported that the maximum number of concurrent kernels will increase to 32 in the new GPU architectures [34], which guarantees good scalability on these future GPUs. The timelines from these figures show the expected behavior of the GPU execution as shown in the theoretical timeline in Figure 3.

It is interesting to look at the timeline prior to execution of the first kernel ($K1$). Figure 7 shows the execution of 5 small kernels within the same stream. These kernels belong to the sort procedure from the CUDA Thrust library, which sorts attribute values in the ascending order, as required by the CAIM algorithm. However, the Thrust library does not provide asynchronous execution for multiple streams. Therefore, the kernels which sort values are serialized in the main stream.

Figure 8 shows the execution time of the CAIM computation kernel in order to analyze the efficiency of the concurrent kernels execution versus the serial one. Notice the greater efficiency of the concurrent kernels allowing of speeding up the execution of the kernel about 4 times, which is expected as seen in the timeline from Figure 7. Figure 8 also demonstrates the O(m) performance of the GPU CAIM computation kernel, which increases the execution time linearly as the number of distinct values increases.

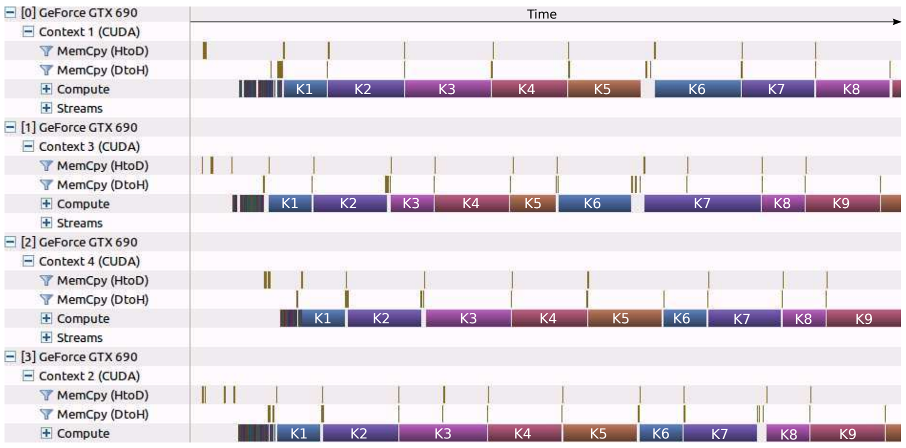

Fig. 6: Serial kernels execution timeline.

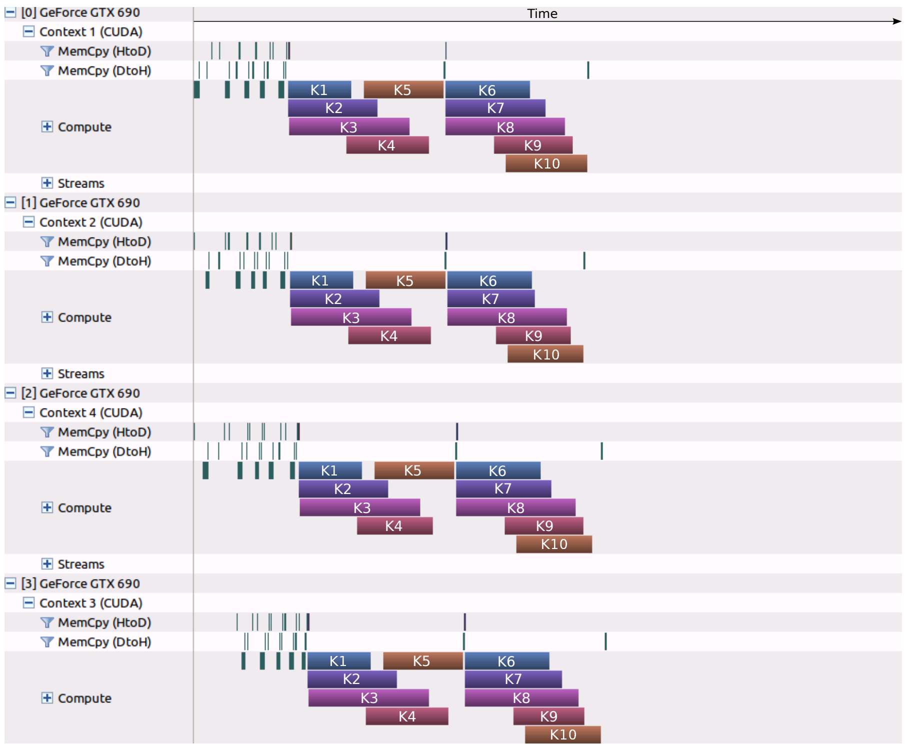

Fig. 7: Concurrent kernels execution timeline.

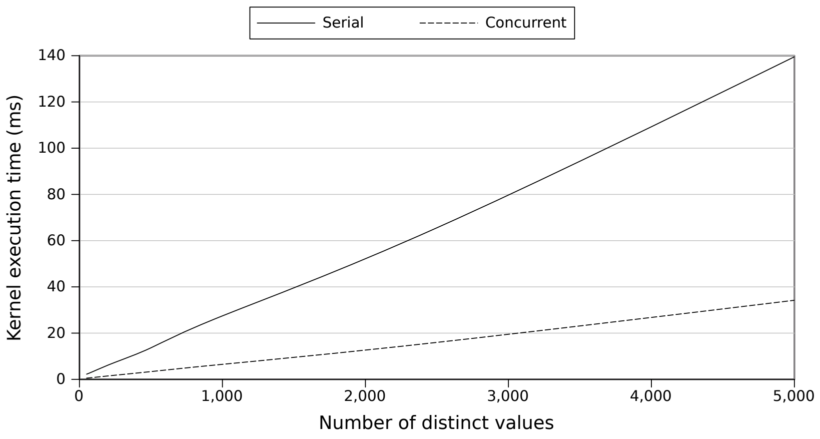

Fig. 8: Concurrent vs serial kernels execution time.

6 Concluding Remarks

In this paper we presented a GPU-based parallel implementation of the CAIM discretization algorithm. The implementation was based on the NVIDIA CUDA programming model and allowed for implementing thousands to millions threads to speed up sorting continuous attributes values, the CAIM criterion, and the discretization process. Two different levels of coarse-grained and one fine-grained parallelism were used to improve the efficiency of the GPU-based implementation. The proposed model is scalable and can be used on multiple GPU devices, which would allow for handling extremely large high-dimensional data sets within reasonable time. Concurrent kernels execution, featured in modern NVIDIA GPUs, was used to improve the efficiency of the kernels computation. The experiments were carried out to analyze performance and scalability of the GPU-based model over 40 data sets with different number of instances, attributes, and classes. Experimental results show very good performance of the GPU implementation in terms of significantly lower execution time. The highest speedups were observed on data sets having very high number of distinct values. Specifically, a speedup of up to 139$\times$ was achieved using 4 GPUs on the *twonorm* data set.

Acknowledgements This work has been supported by the grant from the National Institutes of Health 1R01HD056235-01A1 (KJC), the Regional Government of Andalusia and the Ministry of Science and Technology project TIN-2011-22408 (SV), and the Ministry of Education FPU AP2010-0042 (AC).

The authors also thank Duane Merrill and Sean Baxter from NVIDIA for their advise on improving efficiency of the sorting methods.

References

1. S. G. Akl. *Parallel sorting algorithms*. Notes and reports in computer science and applied mathematics. Academic Press, 1990.
2. J. Alcalá-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. García, L. Sánchez, and F. Herrera. KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework. *Analysis Framework. Journal of Multiple-Valued Logic and Soft Computing*, 17:255–287, 2011.
3. F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):203–215, 2005.
4. J. L. Bentley and M. D. McIlroy. Engineering a Sort Function. *Software-Practice and Experience*, 23(11):1249–1265, 1993.
5. M. Bernaschi, M. Bisson, M. Fatica, and E. Phillips. An introduction to multi-GPU programming for physicists. *The European Physical Journal Special Topics*, 210:17–31, 2012.
6. A. R. Brodtkorb, T. R. Hagen, and M. L. Stra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.
7. M. Cannataro, D. Talia, and P. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Computing*, 28(5):673–704, 2002.
8. A. Cano, J. M. Luna, and S. Ventura. High Performance Evaluation of Evolutionary-Mined Association Rules on GPUs. *Journal of Supercomputing*, 66(3):1438–1461, 2013.
9. A. Cano, A. Zafra, and S. Ventura. Speeding up the evaluation phase of GP classification algorithms on GPUs. *Soft Computing*, 16(2):187–202, 2012.
10. D. Cederman and P. Tsigas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14:4–24, 2010.
11. J. Cerquides and R. L. D. Mantaras. Proposal and empirical comparison of a parallelizable distance-based discretization method. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (1997)*, pages 139–142, 1997.
12. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.
13. K. J. Cios, W. Pedrycz, R. W. Swiniarski, and L. A. Kurgan. *Data Mining: A Knowledge Discovery Approach*. Springer, 2007.
14. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.
15. A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient Parallel Merge Sort for Fixed and Variable Length Keys. In *Proceedings of International Conference on Innovative Parallel Computing (2012)*, pages 1–9, 2012.
16. A. Frank and A. Asuncion. UCI machine learning repository, 2010.
17. A. A. Freitas and S. H. Lavington. *Mining Very Large Databases With Parallel Processing*. Kluwer International Series on Advances in Database Systems, 8. Kluwer, 1998.
18. S. García, J. Luengo, J. Saez, V. Lopez, and F. Herrera. A Survey of Discretization Techniques: Taxonomy and Empirical Analysis in Supervised Learning. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):734–750, 2013.
19. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.
20. J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing*, 72(9):1117–1126, 2012.
21. I. Green, RobertC., L. Wang, M. Alam, and R. A. Formato. Central force optimization on a GPU: a case study in high performance metaheuristics. *Journal of Supercomputing*, 62(1):378–398, 2012.
22. C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–16, 1962.
23. J. Hoberock and N. Bell. *Thrust: A productivity-oriented library for CUDA*, chapter 26, pages 359–372. Morgan Kaufmann, 2011.
24. L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi. Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). *Journal of Supercomputing*, pages 1–26, 2011.
25. F. G. Khan, O. U. Khan, B. Montrucchio, and P. Giaccone. Analysis of Fast Parallel Sorting Algorithms for GPU Architectures. In *Proceedings of the International Conference on Frontiers of Information Technology (2011)*, pages 173–178, 2011.

26. D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.
27. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, 2nd edition, 1998.
28. L. A. Kurgan and K. J. Cios. CAIM Discretization Algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):145–153, 2004.
29. M. Li, S. Deng, S. Feng, and J. Fan. An effective discretization based on class-attribute coherence maximization. *Pattern Recognition Letters*, 32(15):1962–1973, 2011.
30. D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (2010)*, pages 545–546, 2010.
31. D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.
32. D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.
33. C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.
34. NVIDIA Corporation. NVIDIA CUDA Programming and Best Practices Guide, http://www.nvidia.com/cuda, 2013.
35. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
36. S. Parthasarathy and A. Ramakrishnan. Parallel incremental 2d-discretization on dynamic datasets. In *Proceedings of the International Conference on Parallel and Distributed Processing Systems (2002)*, pages 247–254, 2002.
37. H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. *Concurrency Computation Practice and Experience*, 23(7):681–693, 2011.
38. A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.
39. N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the IEEE International Symposium on Parallel&Distributed Processing (2009)*, pages 1–10, 2009.
40. M. Schellmann, S. Gorlatch, D. Meilnder, T. Ksters, K. Schfers, F. Wbbeling, and M. Burger. Parallel medical image reconstruction: from graphics processing units (gpu) to grids. *Journal of Supercomputing*, 57(2):151–160, 2011.
41. R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley. A survey of medical image registration on multicore and the GPU. *IEEE Signal Processing Magazine*, 27(2):50–60, 2010.
42. E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.
43. K. Sriwanna, K. Puntumapon, and K. Waiyamai. An enhanced class-attribute interdependence maximization discretization algorithm. In *Proceedings of the 8th International Conference on Advanced Data Mining and Applications*, volume 7713 LNAI, pages 465–476, 2012.
44. N. Tatarchuk, J. Shopf, and C. DeCoro. Advanced interactive medical visualization on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1319–1328, 2008.
45. S. R. Upadhyaya. Parallel approaches to machine learning-a comprehensive survey. *Journal of Parallel and Distributed Computing*, 73(3):284–292, 2013.
46. P. Wittek and S. Darnyi. Accelerating text mining workloads in a mapreduce-based distributed GPU environment. *Journal of Parallel and Distributed Computing*, 73(2):198–206, 2013.
47. Y. Yang, G. I. Webb, and X. Wu. Discretization Methods. In *Data Mining and Knowledge Discovery Handbook*, pages 101–116. 2010.
48. X. Yulong, W. Xiaopeng, and X. Dawei. A two step parallel discretization algorithm based on dynamic clustering. In *Proceedings of the International Conference on Computer Science and Electronics Engineering (2012)*, volume 3, pages 192–196, 2012.
49. M. J. Zaki and C. T. Ho. *Large-Scale Parallel Data Mining*. Lecture Notes in Artificial Intelligence, State of the Art Survey. Springer, 2000.

50. Y. Zhang, F. Mueller, X. Cui, and T. Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *Journal of Parallel and Distributed Computing*, 71(2):211–224, 2011.
51. Y. Zhao, Z. Niu, X. Peng, and L. Dai. A discretization algorithm of numerical attributes for digital library evaluation based on data mining technology. In *Digital Libraries: For Cultural Heritage, Knowledge Dissemination, and Future Creation*, volume 7008 LNCS, pages 70–76, 2011.