Data Mining Using Graphics Processing Units

Christian Böhm[1], Robert Noll[1], Claudia Plant[2], Bianca Wackersreuther[1], and Andrew Zherdin[2]

[1] University of Munich, Germany
{boehm,noll,wackersreuther}@dbs.ifi.lmu.de
[2] Technische Universität München, Germany
{plant,zherdin}@lrz.tum.de

Abstract. During the last few years, Graphics Processing Units (GPU) have evolved from simple devices for the display signal preparation into powerful coprocessors that do not only support typical computer graphics tasks such as rendering of 3D scenarios but can also be used for general numeric and symbolic computation tasks such as simulation and optimization. As major advantage, GPUs provide extremely high parallelism (with several hundred simple programmable processors) combined with a high bandwidth in memory transfer at low cost. In this paper, we propose several algorithms for computationally expensive data mining tasks like similarity search and clustering which are designed for the highly parallel environment of a GPU. We define a multidimensional index structure which is particularly suited to support similarity queries under the restricted programming model of a GPU, and define a similarity join method. Moreover, we define highly parallel algorithms for density-based and partitioning clustering. In an extensive experimental evaluation, we demonstrate the superiority of our algorithms running on GPU over their conventional counterparts in CPU.

1 Introduction

In recent years, *Graphics Processing Units* (GPUs) have evolved from simple devices for the display signal preparation into powerful coprocessors supporting the CPU in various ways. Graphics applications such as realistic 3D games are computationally demanding and require a large number of complex algebraic operations for each update of the display image. Therefore, today's graphics hardware contains a large number of programmable processors which are optimized to cope with this high workload of vector, matrix, and symbolic computations in a highly parallel way. In terms of peak performance, the graphics hardware has outperformed state-of-the-art multi-core CPUs by a large margin.

The amount of scientific data is approximately doubling every year [26]. To keep pace with the exponential data explosion, there is a great effort in many research communities such as life sciences [20,22], mechanical simulation [27], cryptographic computing [2], or machine learning [7] to use the computational capabilities of GPUs even for purposes which are not at all related to computer graphics. The corresponding research area is called General Processing-Graphics Processing Units (GP-GPU).

In this paper, we focus on exploiting the computational power of GPUs for *data mining*. Data Mining consists of 'applying data analysis algorithms, that, under acceptable efficiency limitations, produce a particular enumeration of patterns over the data' [9]. The exponential increase in data does not necessarily come along with a correspondingly large gain in knowledge. The evolving research area of data mining proposes techniques to support transforming the raw data into useful knowledge. Data mining has a wide range of scientific and commercial applications, for example in neuroscience, astronomy, biology, marketing, and fraud detection. The basic data mining tasks include classification, regression, clustering, outlier identification, as well as frequent itemset and association rule mining. Classification and regression are called supervised data mining tasks, because the aim is to learn a model for predicting a predefined variable. The other techniques are called unsupervised, because the user does not previously identify any of the variables to be learned. Instead, the algorithms have to automatically identify any interesting regularities and patterns in the data. Clustering probably is the most common unsupervised data mining task. The goal of clustering is to find a natural grouping of a data set such that data objects assigned to a common group called cluster are as similar as possible and objects assigned to different clusters differ as much as possible. Consider for example the set of objects visualized in Figure 1. A natural grouping would be assigning the objects to two different clusters. Two outliers not fitting well to any of the clusters should be left unassigned. Like most data mining algorithms, the definition of clustering requires specifying some notion of similarity among objects. In most cases, the similarity is expressed in a vector space, called the feature space. In Figure 1, we indicate the similarity among objects by representing each object by a vector in two dimensional space. Characterizing numerical properties (from a continuous space) are extracted from the objects, and taken together to a vector $x \in \mathbb{R}^d$ where d is the dimensionality of the space, and the number of properties which have been extracted, respectively. For instance, Figure 2 shows a feature transformation where the object is a certain kind of an orchid. The phenotype of orchids can be characterized using the lengths and widths of the two petal and the three sepal leaves, of the form (curvature) of the labellum, and of the colors of the different compartments. In this example, 5

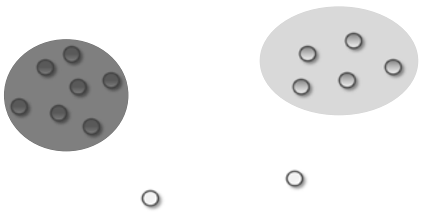


Fig. 1. Example for Clustering

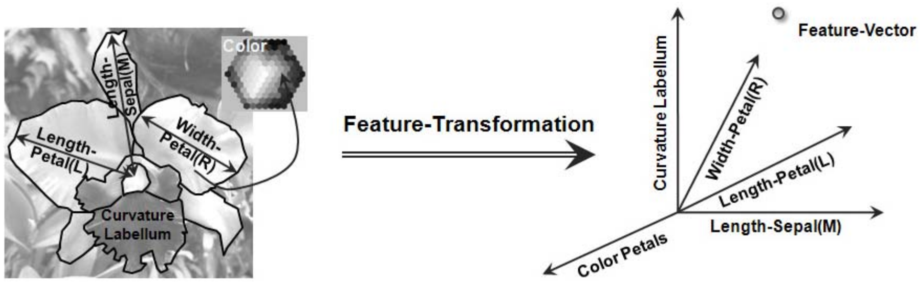


Fig. 2. The Feature Transformation

features are measured, and each object is thus transformed into a 5-dimensional vector space. To measure the similarity between two feature vectors, usually a distance function like the Euclidean metric is used. To search in a large database for objects which are similar to a given query objects (for instance to search for a number k of nearest neighbors, or for those objects having a distance that does not exceed a threshold ϵ), usually *multidimensional index structures* are applied. By a hierarchical organization of the data set, the search is made efficient. The well-known indexing methods (like e.g. the R-tree [13]) are designed and optimized for secondary storage (hard disks) or for main memory. For the use in the GPU specialized indexing methods are required because of the highly parallel but restricted programming environment. In this paper, we propose such an indexing method.

It has been shown that many data mining algorithms, including clustering can be supported by a powerful database primitive: The *similarity join* [3]. This operator yields as result all pairs of objects in the database having a distance of less than some predefined threshold ϵ. To show that also the more complex basic operations of similarity search and data mining can be supported by novel parallel algorithms specially designed for the GPU, we propose two algorithms for the similarity join, one being a nested block loop join, and one being an indexed loop join, utilizing the aforementioned indexing structure.

Finally, to demonstrate that highly complex data mining tasks can be efficiently implemented using novel parallel algorithms, we propose parallel versions of two widespread clustering algorithms. We demonstrate how the density-based clustering algorithm DBSCAN [8] can be effectively supported by the parallel similarity join. In addition, we introduce a parallel version of K-means clustering [21] which follows an algorithmic paradigm which is very different from density-based clustering. We demonstrate the superiority of our approaches over the corresponding sequential algorithms on CPU.

All algorithms for GPU have been implemented using NVIDIA's technology *Compute Unified Device Architecture* (CUDA) [1]. Vendors of graphics hardware have recently anticipated the trend towards general purpose computing on GPU and developed libraries, pre-compilers and application programming interfaces to support GP-GPU applications. CUDA offers a programming interface for the

C programming language in which both the *host program* as well as the *kernel functions* are assembled in a single program [1]. The host program is the main program, executed on the CPU. In contrast, the so-called *kernel functions* are executed in a massively parallel fashion on the (hundreds of) processors in the GPU. An analogous technique is also offered by ATI using the brand names Close-to-Metal, Stream SDK, and Brook-GP.

The remainder of this paper is organized as follows: Section 2 reviews the related work in GPU processing in general with particular focus on database management and data mining. Section 3 explains the graphics hardware and the CUDA programming model. Section 4 develops an multidimensional index structure for similarity queries on the GPU. Section 5 presents the non-indexed and indexed join on graphics hardware. Section 6 and Section 7 are dedicated to GPU-capable algorithms for density-based and partitioning clustering. Section 8 contains an extensive experimental evaluation of our techniques, and Section 9 summarizes the paper and provides directions for future research.

2 Related Work

In this section, we survey the related research in general purpose computations using GPUs with particular focus on database management and data mining.

General Processing-Graphics Processing Units. Theoretically, GPUs are capable of performing any computation that can be transformed to the model of parallelism and that allow for the specific architecture of the GPU. This model has been exploited for multiple research areas. Liu et al. [20] present a new approach to high performance molecular dynamics simulations on graphics processing units by the use of CUDA to design and implement a new parallel algorithm. Their results indicate a significant performance improvement on an NVIDIA GeForce 8800 GTX graphics card over sequential processing on CPU. Another paper on computations from the field of life sciences has been published by Manavski and Valle [22]. The authors propose an extremely fast solution of the Smith-Waterman algorithm, a procedure for searching for similarities in protein and DNA databases, running on GPU and implemented in the CUDA programming environment. Significant speedups are achieved on a workstation running two GeForce 8800 GTX.

Another widespread application area that uses the processing power of the GPU is mechanical simulation. One example is the work by Tascora et al. [27], that presents a novel method for solving large cone complementarity problems by means of a fixed-point iteration algorithm, in the context of simulating the frictional contact dynamics of large systems of rigid bodies. As the afore reviewed approaches in the field of life sciences, the algorithm is also implemented in CUDA for a GeForce 8800 GTX to simulate the dynamics of complex systems.

To demonstrate the nearly boundless possibilities of performing computations on the GPU, we introduce one more example, namely cryptographic computing [2]. In this paper, the authors present a record-breaking performance for the elliptic curve method (ECM) of integer factorization. The speedup takes advantage of two NVIDIA GTX 295 graphics cards, using a new ECM implementation relying on new parallel addition formulas and functions that are made available by CUDA.

Database Management Using GPUs. Some papers propose techniques to speed up relational database operations on GPU. In [14] some algorithms for the relational join on an NVIDIA G80 GPU using CUDA are presented.

Two recent papers [19,4] address the topic of similarity join in feature space which determines all pairs of objects from two different sets R and S fulfilling a certain join predicate.The most common join predicate is the ϵ-join which determines all pairs of objects having a distance of less than a predefined threshold ϵ. The authors of [19] propose an algorithm based on the concept of space filling curves, e.g. the z-order, for pruning of the search space, running on an NVIDIA GeForce 8800 GTX using the CUDA toolkit. The z-order of a set of objects can be determined very efficiently on GPU by highly parallelized sorting. Their algorithm operates on a set of z-lists of different granularity for efficient pruning. However, since all dimensions are treated equally, performance degrades in higher dimensions. In addition, due to uniform space partitioning in all areas of the data space, space filling curves are not suitable for clustered data. An approach that overcomes that kind of problem is presented in [4]. Here the authors parallelize the baseline technique underlying any join operation with an arbitrary join predicate, namely the nested loop join (NLJ), a powerful database primitive that can be used to support many applications including data mining. All experiments are performed on NVIDIA 8500GT graphics processors by the use of a CUDA-supported implementation.

Govindaraju et al. [10,11] demonstrate that important building blocks for query processing in databases, e.g. sorting, conjunctive selections, aggregations, and semi-linear queries can be significantly speed up by the use of GPUs.

Data Mining Using GPUs. Recent approaches concerning data mining using the GPU are two papers on clustering on GPU, that pass on the use of CUDA. In [6] a clustering approach on a NVIDIA GeForce 6800 GT graphics card is presented, that extends the basic idea of K-means by calculating the distances from a single input centroid to all objects at one time that can be done simultaneously on GPU. Thus the authors are able to exploit the high computational power and pipeline of GPUs, especially for core operations, like distance computations and comparisons. An additional efficient method that is designed to execute clustering on data streams confirms a wide practical field of clustering on GPU.

The paper [25] parallelizes the K-means algorithm for use of a GPU by using multi-pass rendering and multi shader program constants. The implementation on NVIDIA 5900 and NVIDIA 8500 graphics processors achieves significant increasing performances for both various data sizes and cluster sizes. However the algorithms of both papers are not portable to different GPU models, like CUDA-approaches are.

3 Architecture of the GPU

Graphics Processing Units (GPUs) of the newest generation are powerful coprocessors, not only designed for games and other graphics-intensive applications, but also for general-purpose computing (in this case, we call them GP-GPUs). From the hardware perspective, a GPU consists of a number of multiprocessors, each of which consists of a set of simple processors which operate in a SIMD fashion, i.e. all processors of one multiprocessor execute in a synchronized way the same arithmetic or logic operation at the same time, potentially operating on different data. For instance, the GPU of the newest generation GT200 (e.g. on the graphics card Geforce GTX280) has 30 multiprocessors, each consisting of 8 SIMD-processors, summarizing to a total amount of 240 processors inside one GPU. The computational power sums up to a peak performance of 933 GFLOP/s.

3.1 The Memory Model

Apart from some memory units with special purpose in the context of graphics processing (e.g. texture memory), we have three important types of memory, as visualized in Figure 3. The *shared memory* (SM) is a memory unit with fast access (at the speed of register access, i.e. no delay). SM is shared among all processors of a multiprocessor. It can be used for local variables but also to exchange information between threads on different processors of the same multiprocessor. It cannot be used for information which is shared among threads on different multiprocessors. SM is fast but very limited in capacity (16 KBytes per multiprocessor). The second kind of memory is the so-called *device memory* (DM), which is the actual video RAM of the graphics card (also used for frame buffers etc.). DM is physically located on the graphics card (but not inside the GPU), is significantly larger than SM (typically up to some hundreds of MBytes), but also significantly slower. In particular, memory accesses to DM cause a typical latency delay of 400-600 clock cycles (on G200-GPU, corresponding to 300-500ns). The bandwidth for transferring data between DM and GPU (141.7 GB/s on G200) is higher than that of CPU and main memory (about 10 GB/s on current CPUs). DM can be used to share information between threads

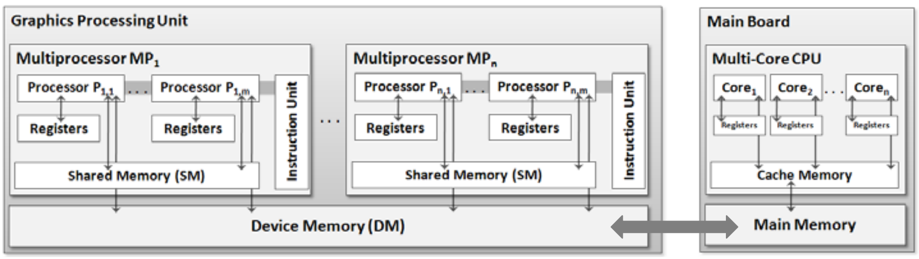


Fig. 3. Architecture of a GPU

on different multiprocessors. If some threads schedule memory accesses from contiguous addresses, these accesses can be coalesced, i.e. taken together to improve the access speed. A typical cooperation pattern for DM and SM is to copy the required information from DM to SM simultaneously from different threads (if possible, considering coalesced accesses), then to let each thread compute the result on SM, and finally, to copy the result back to DM. The third kind of memory considered here is the *main memory* which is not part of the graphics card. The GPU has no access to the address space of the CPU. The CPU can only write to or read from DM using specialized API functions. In this case, the data packets have to be transferred via the Front Side Bus and the PCI-Express Bus. The bandwidth of these bus systems is strictly limited, and therefore, these special transfer operations are considerably more expensive than direct accesses of the GPU to DM or direct accesses of the CPU to main memory.

3.2 The Programming Model

The basis of the programming model of GPUs are threads. Threads are lightweight processes which are easy to create and to synchronize. In contrast to CPU processes, the generation and termination of GPU threads as well as context switches between different threads do not cause any considerable overhead either. In typical applications, thousands or even millions of threads are created, for instance one thread per pixel in gaming applications. It is recommended to create a number of threads which is even much higher than the number of available SIMD-processors because context switches are also used to hide the latency delay of memory accesses: Particularly an access to the DM may cause a latency delay of 400-600 clock cycles, and during that time, a multiprocessor may continue its work with other threads. The CUDA programming library [1] contains API functions to create a large number of threads on the GPU, each of which executes a function called *kernel function*. The kernel functions (which are executed in parallel on the GPU) as well as the host program (which is executed sequentially on the CPU) are defined in an extended syntax of the C programming language. The kernel functions are restricted with respect to functionality (e.g. no recursion).

On GPUs the threads do not even have an individual instruction pointer. An instruction pointer is rather shared by several threads. For this purpose, threads are grouped into so-called *warps* (typically 32 threads per warp). One warp is processed simultaneously on the 8 processors of a single multiprocessor (SIMD) using 4-fold pipelining (totalling in 32 threads executed fully synchronously). If not all threads in a warp follow the same execution path, the different execution paths are executed in a serialized way. The number (8) of SIMD-processors per multiprocessor as well as the concept of 4-fold pipelining is constant on all current CUDA-capable GPUs.

Multiple warps are grouped into *thread groups* (TG). It is recommended [1] to use multiples of 64 threads per TG. The different warps in a TG (as well as different warps of different TGs) are executed independently. The threads in one thread group use the same shared memory and may thus communicate and share data via the SM. The threads in one thread group can be synchronized (let all threads wait until all warps of the same group have reached that point of execution). The latency delay of the DM can be hidden by scheduling other warps of the same or a different thread group whenever one warp waits for an access to DM. To allow switching between warps of different thread groups on a multiprocessor, it is recommended that each thread uses only a small fraction of the shared memory and registers of the multiprocessor [1].

3.3 Atomic Operations

In order to synchronize parallel processes and to ensure the correctness of parallel algorithms, CUDA offers *atomic operations* such as increment, decrement, or exchange (to name just those out of the large number of atomic operations, which will be needed by our algorithms). Most of the atomic operations work on integer data types in Device Memory. However, the newest version of CUDA (Compute Capability 1.3 of the GPU GT200) allows even atomic operations in SM. If, for instance, some parallel processes share a list as a common resource with concurrent reading and writing from/to the list, it may be necessary to (atomically) increment a counter for the number of list entries (which is in most cases also used as the pointer to the first free list element). Atomicity implies in this case the following two requirements: If two or more threads increment the list counter, then (1) the value counter after all concurrent increments must be equivalent to the value before plus the number of concurrent increment operations. And, (2), each of the concurrent threads must obtain a separate result of the increment operation which indicates the index of the empty list element to which the thread can write its information. Therefore, most atomic operations return a result after their execution. For instance the operation `atomicInc` has two parameters, the address of the counter to be incremented, and an optional threshold value which must not be exceeded by the operation. The operation works as follows: The counter value at the address is read, and incremented (provided that the threshold is not exceeded). Finally, the *old* value of the counter (before incrementing) is returned to the kernel method which invoked `atomicInc`. If two or more threads (of the same or different thread groups) call some atomic operations simultaneously, the result of these operations is that of an arbitrary sequentialization of the concurrent operations. The operation `atomicDec` works in an analogous way. The operation `atomicCAS` performs a Compare-and-Swap operation. It has three parameters, an address, a compare value and a swap value. If the value at the address equals the compare value, the value at the address is replaced by the swap value. In every case, the old value at the address (before swapping) is returned to the invoking kernel method.

4 An Index Structure for Similarity Queries on GPU

Many data mining algorithms for problems like classification, regression, clustering, and outlier detection use similarity queries as a building block. In many cases, these similarity queries even represent the largest part of the computational effort of the data mining tasks, and, therefore, efficiency is of high importance here. Similarity queries are defined as follows: Given is a database $\mathcal{D} = \{x_1, ...x_n\} \subseteq \mathbb{R}^d$ of a number n of vectors from a d-dimensional space, and a query object $q \in \mathbb{R}^d$. We distinguish between two different kinds of similarity queries, the range queries and the nearest neighbor-queries:

Definition 1 (Range Query)
Let $\epsilon \in \mathbb{R}_0^+$ be a threshold value. The result of the range query is the set of the following objects:

$$N_\epsilon(q) = \{x \in \mathcal{D} : \quad ||x - q|| \leq \epsilon\}.$$

where $||x - q||$ is an arbitrary distance function between two feature vectors x and q, e.g. the Euclidean distance.

Definition 2 (Nearest Neighbor Query)
The result of a nearest neighbor query is the set:

$$NN(q) = \{x \in \mathcal{D} : \quad \forall x' \in \mathcal{D} : \quad ||x - q|| \leq ||x' - q||\}.$$

Definition 2 can also be generalized for the case of the k-nearest neighbor query ($NN_k(q)$), where a number k of nearest neighbors of the query object q is retrieved.

The performance of similarity queries can be greatly improved if a multidimensional index structure supporting the similarity search is available. Our index structure needs to be traversed in parallel for many search objects using the kernel function. Since kernel functions do not allow any recursion, and as they need to have small storage overhead by local variables etc., the index structure must be kept very simple as well. To achieve a good compromise between simplicity and selectivity of the index, we propose a data partitioning method with a constant number of directory levels. The first level partitions the data set $\mathcal{D}$ according to the first dimension of the data space, the second level according to the second dimension, and so on. Therefore, before starting the actual data mining method, some transformation technique should be applied which guarantees a high selectivity in the first dimensions (e.g. Principal Component Analysis, Fast Fourier Transform, Discrete Wavelet Transform, etc.). Figure 4 shows a simple, 2-dimensional example of a 2-level directory (plus the root node which is considered as level-0), similar to [16,18]. The fanout of each node is 8. In our experiments in Section 8, we used a 3-level directory with fanout 16.

Before starting the actual data mining task, our simple index structure must be constructed in a bottom-up way by fractionated sorting of the data: First, the data set is sorted according to the first dimension, and partitioned into the specified number of quantile partitions. Then, each of the partitions is sorted individually according to the second dimension, and so on. The boundaries are stored using simple arrays which can be easily accessed in the subsequent kernel functions. In principle, this index construction can already be done on the GPU, because efficient sorting methods for GPU have been proposed [10]. Since bottom

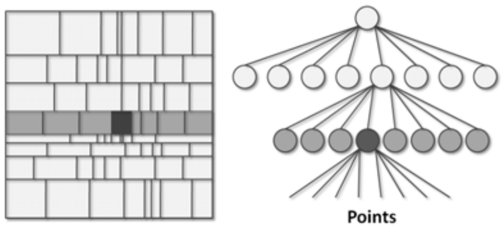


Fig. 4. Index Structure for GPU

up index construction is typically not very costly compared to the data mining algorithm, our method performs this preprocessing step on CPU.

When transferring the data set from the main memory into the device memory in the initialization step of the data mining method, our new method has additionally to transfer the directory (i.e. the arrays in which the coordinates of the page boundaries are stored). Compared to the complete data set, the directory is always small.

The most important change in the kernel functions in our data mining methods regards the determination of the ϵ-neighborhood of some given seed object q, which is done by exploiting SIMD-parallelism inside a multiprocessor. In the non-indexed version, this is done by a set of threads (inside a thread group) each of which iterates over a different part of the (complete) data set. In the indexed version, one of the threads iterates in a set of nested loops (one loop for each level of the directory) over those nodes of the index structure which represent regions of the data space which are intersected by the neighborhood-sphere of $N_\epsilon(q)$. In the innermost loop, we have one set of points (corresponding to a data page of the index structure) which is processed by exploiting the SIMD-parallelism, like in the non-indexed version.

5 The Similarity Join

The *similarity join* is a basic operation of a database system designed for similarity search and data mining on feature vectors. In such applications, we are given a database $\mathcal{D}$ of objects which are associated with a vector from a multidimensional space, the feature space. The similarity join determines pairs of objects which are similar to each other. The most widespread form is the ϵ-join which determines those pairs from $\mathcal{D} \times \mathcal{D}$ which have a Euclidean distance of no more than a user-defined radius ϵ:

Definition 3 (Similarity Join). *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be a set of feature vectors of a d-dimensional vector space and $\epsilon \in \mathbb{R}_0^+$ be a threshold. Then the similarity join is the following set of pairs:*

$$SimJoin(\mathcal{D}, \epsilon) = \{(x, x') \in (\mathcal{D} \times \mathcal{D}) : \quad ||x - x'|| \leq \epsilon\},$$

If x and x' are elements of the same set, the join is a similarity self-join. Most algorithms including the method proposed in this paper can also be generalized to the more general case of non-self-joins in a straightforward way. Algorithms for a similarity join with nearest neighbor predicates have also been proposed. The similarity join is a powerful building block for similarity search and data mining. It has been shown that important data mining methods such as clustering and classification can be based on the similarity join. Using a similarity join instead of single similarity queries can accelerate data mining algorithms by a high factor [3].

5.1 Similarity Join without Index Support

The baseline technique to process any join operation with an arbitrary join predicate is the nested loop join (NLJ) which performs two nested loops, each enumerating all points of the data set. For each pair of points, the distance is calculated and compared to ϵ. The pseudocode of the sequential version of NLJ is given in Figure 5.

```
algorithm sequentialNLJ(data set $\mathcal{D}$)
  for each $q \in \mathcal{D}$ do        // outer loop
    for each $x \in \mathcal{D}$ do      // inner loop: search all points $x$ which are similar to $q$
      if dist$(x, q) \leq \epsilon$ then
        report $(x, q)$ as a result pair or do some further processing on $(x, q)$
end
```

Fig. 5. Sequential Algorithm for the Nested Loop Join

It is easily possible to parallelize the NLJ, e.g. by creating an individual thread for each iteration of the outer loop. The kernel function then contains the inner loop, the distance calculation and the comparison. During the complete run of the kernel function, the current point of the outer loop is constant, and we call this point the *query point* q of the thread, because the thread operates like a similarity query, in which all database points with a distance of no more than ϵ from q are searched. The query point q is always held in a register of the processor.

Our GPU allows a truly parallel execution of a number m of incarnations of the outer loop, where m is the total number of ALUs of all multiprocessors (i.e. the warp size 32 times the number of multiprocessors). Moreover, all the different warps are processed in a quasi-parallel fashion, which allows to operate on one warp of threads (which is ready-to-run) while another warp is blocked due to the latency delay of a DM access of one of its threads.

The threads are grouped into thread groups, which share the SM. In our case, the SM is particularly used to physically store for each thread group the current point x of the inner loop. Therefore, a kernel function first copies the current point x from the DM into the SM, and then determines the distance of x to the query point q. The threads of the same warp are running perfectly simultaneously, i.e. if these threads are copying the same point from DM to SM, this needs to be done only once (but all threads of the warp have to wait until this relatively costly copy operation is performed). However, a thread group may (and should) consist of multiple warps. To ensure that the copy operation is only performed once per thread group, it is necessary to synchronize the threads of the thread group before and after the copy operation using the API function `synchronize()`. This API function blocks all threads in the same TG until all other threads (of other warps) have reached the same point of execution. The pseudocode for this algorithm is presented in Figure 6.

```
algorithm GPUsimpleNLJ(data set $\mathcal{D}$)          // host program executed on CPU
  deviceMem float $\mathcal{D}'$[][] := $\mathcal{D}$[][];          // allocate memory in DM for the data set $\mathcal{D}$
  #threads := $n$;                              // number of points in $\mathcal{D}$
  #threadsPerGroup := 64;
  startThreads (simpleNLJKernel, #threads, #threadsPerGroup);    // one thread per point
  waitForThreadsToFinish();
end.

kernel simpleNLJKernel (int threadID)
  register float $q$[] := $\mathcal{D}'$[threadID][];       // copy the point from DM into the register
                                                // and use it as query point $q$
                                                // index is determined by the threadID
  for $i := 0 \ldots n-1$ do                         // this used to be the inner loop in Figure 5
    synchronizeThreadGroup();
    shared float $x$[] := $\mathcal{D}'[i]$[];             // copy the current point $x$ from DM to SM
    synchronizeThreadGroup();                   // Now all threads of the thread group can work with $x$
    if dist$(x,q) \leq \epsilon$ then
      report $(x,q)$ as a result pair using synchronized writing
      or do some further processing on $(x,q)$ directly in kernel
end.
```

Fig. 6. Parallel Algorithm for the Nested Loop Join on the GPU

If the data set does not fit into DM, a simple partitioning strategy can be applied. It must be ensured that the potential join partners of an object are within the same partition as the object itself. Therefore, overlapping partitions of size $2 \cdot \epsilon$ can be created.

5.2 An Indexed Parallel Similarity Join Algorithm on GPU

The performance of the NLJ can be greatly improved if an index structure is available as proposed in Section 4. On sequential processing architectures, the indexed NLJ leaves the outer loop unchanged. The inner loop is replaced by an index-based search retrieving candidates that may be join partners of the current object of the outer loop. The effort of finding these candidates and refining them is often orders of magnitude smaller compared to the non-indexed NLJ.

When parallelizing the indexed NLJ for the GPU, we follow the same paradigm as in the last section, to create an individual thread for each point of the outer loop. It is beneficial to the performance, if points having a small distance to each other are collected in the same warp and thread group, because for those points, similar paths in the index structure are relevant.

After index construction, we have not only a directory in which the points are organized in a way that facilitates search. Moreover, the points are now clustered in the array, i.e. points which have neighboring addresses are also likely to be close together in the data space (at least when projecting on the first few dimensions). Both effects are exploited by our join algorithm displayed in Figure 7.

```
algorithm GPUindexedJoin(data set $\mathcal{D}$)
  deviceMem index idx := makeIndexAndSortData($\mathcal{D}$); // changes ordering of data points
  int #threads := $|\mathcal{D}|$, #threadsPerGroup := 64;
  for $i = 1$ ... (#threads/#threadsPerGroup) do
    deviceMem float blockbounds[$i$][] := calcBlockBounds($\mathcal{D}$, blockindex);
  deviceMem float $\mathcal{D}'$[][] := $\mathcal{D}$[][];
  startThreads (indexedJoinKernel, #threads, #threadsPerGroup); // one thread per data point
  waitForThreadsToFinish ();
end.

algorithm indexedJoinKernel (int threadID, int blockID)
  register float $q$[] := $\mathcal{D}'$[threadID][];       // copy the point from DM into the register
  shared float myblockbounds[] := blockbounds[blockID][];
  for $x_i$ := 0 ... indexsize.$x$ do
    if IndexPageIntersectsBoundsDim1(idx,myblockbounds,$x_i$) then
      for $y_i$ := 0 ... indexsize.$y$ do
        if IndexPageIntersectsBoundsDim2(idx,myblockbounds,$x_i$, $y_i$) then
          for $z_i$ := 0 ... indexsize.$z$ do
            if IndexPageIntersectsBoundsDim3(idx,myblockbounds,$x_i$, $y_i$, $z_i$) then
              for $w$ := 0 ... IndexPageSize do
                synchronizeThreadGroup();
                shared float $p$[] :=GetPointFromIndexPage(idx,$\mathcal{D}'$, $x_i$, $y_i$, $z_i$, $w$);
                synchronizeThreadGroup();
                if dist($p$, $q$) $\leq \epsilon$ then
                  report ($p$, $q$) as a result pair using synchronized writing
end.
```

Fig. 7. Algorithm for Similarity Join on GPU with Index Support

Instead of performing an outer loop like in a sequential indexed NLJ, our algorithm now generates a large number of threads: One thread for each iteration of the outer loop (i.e. for each query point q). Since the points in the array are clustered, the corresponding query points are close to each other, and the join partners of all query points in a thread group are likely to reside in the same branches of the index as well. Our kernel method now iterates over three loops, each loop for one index level, and determines for each partition if the point is inside the partition or, at least no more distant to its boundary than ϵ. The corresponding subnode is accessed if the corresponding partition is able to contain join partners of the current point of the thread. When considering the warps which operate in a fully synchronized way, a node is accessed, whenever at least one of the query points of the warps is close enough to (or inside) the corresponding partition.

For both methods, indexed and non-indexed nested loop join on GPU, we need to address the question how the resulting pairs are processed. Often, for example to support density-based clustering (cf. Section 6), it is sufficient to return a counter with the number of join partners. If the application requires to report the pairs themselves, this is easily possible by a buffer in DM which can be copied to the CPU after the termination of all kernel threads. The result pairs must be written into this buffer in a synchronized way to avoid that two threads write simultaneously to the same buffer area. The CUDA API provides atomic operations (such as atomic increment of a buffer pointer) to guarantee this kind of synchronized writing. Buffer overflows are also handled by our similarity join methods. If the buffer is full, all threads terminate and the work is resumed after the buffer is emptied by the CPU.

6 Similarity Join to Support Density-Based Clustering

As mentioned in Section 5, the similarity join is an important building block to support a wide range of data mining tasks, including classification [24], outlier detection [5] association rule mining [17] and clustering [8], [12]. In this section, we illustrate how to effectively support the density-based clustering algorithm DBSCAN [8] with the similarity join on GPU.

6.1 Basic Definitions and Sequential DBSCAN

The idea of density-based clustering is that clusters are areas of high point density, separated by areas of significantly lower point density. The point density can be formalized using two parameters, called $\epsilon \in \mathbb{R}^+$ and $MinPts \in \mathbb{N}^+$. The central notion is the *core object*. A data object x is called a core object of a cluster, if at least $MinPts$ objects (including x itself) are in its ϵ-neighborhood $N_\epsilon(x)$, which corresponds to a sphere of radius ϵ. Formally:

Definition 4. *(Core Object)*
Let $\mathcal{D}$ be a set of n objects from $\mathbb{R}^d$, $\epsilon \in \mathbb{R}^+$ and $MinPts \in \mathbb{N}^+$. An object $x \in \mathcal{D}$ is a core object, if and only if

$$|N_\epsilon(x)| \geq MinPts, \text{ where } N_\epsilon(x) = \{x' \in \mathcal{D} : ||x' - x|| \leq \epsilon\}.$$

Note that this definition is equivalent to Definition 1. Two objects may be assigned to a common cluster. In density-based clustering this is formalized by the notions *direct density reachability, and density connectedness.*

Definition 5. *(Direct Density Reachability)*
Let $x, x' \in \mathcal{D}$. x' is called directly density reachable from x (in symbols: $x \lhd x'$) if and only if

1. *x is a core object in $\mathcal{D}$, and*
2. *$x' \in N_\epsilon(x)$.*

If x and x' are both core objects, then $x \lhd x'$ is equivalent with $x \rhd x'$. The density connectedness is the transitive and symmetric closure of the direct density reachability:

Definition 6. *(Density Connectedness)*
Two objects x and x' are called density connected (in symbols: $x \bowtie x'$) if and only if there is a sequence of core objects $(x_1, ..., x_m)$ of arbitrary length m such that

$$x \rhd x_1 \rhd ... \lhd x_m \lhd x'.$$

In density-based clustering, a cluster is defined as a maximal set of density connected objects:

Definition 7. *(Density-based Cluster)*
A subset $C \subseteq \mathcal{D}$ is called a cluster if and only if the following two conditions hold:

1. *Density connectedness: $\forall x, x' \in C : x \bowtie x'$.*
2. *Maximality: $\forall x \in C, \forall x' \in \mathcal{D} \setminus C : \neg x \bowtie x'$.*

The algorithm DBSCAN [8] implements the cluster notion of Definition 7 using a data structure called *seed list* S containing a set of seed objects for cluster expansion. More precisely, the algorithm proceeds as follows:

1. Mark all objects as *unprocessed.*
2. Consider an arbitrary unprocessed object $x \in \mathcal{D}$.
3. If x is a core object, assign a new cluster ID C, and do step (4) for all elements $x' \in N_\epsilon(x)$ which do not yet have a cluster ID:
4. (a) mark the element x' with the cluster ID C and
   (b) insert the object x' into the seed list S.
5. While S is not empty repeat step 6 for all elements $s \in S$:
6. If s is a core object, do step (7) for all elements $x' \in N_\epsilon(s)$ which do not yet have any cluster ID:
7. (a) mark the element x' with the cluster ID C and
   (b) insert the object x' into the seed list S.
8. If there are still unprocessed objects in the database, continue with step (2).

To illustrate the algorithmic paradigm, Figure 8 displays a snapshot of DBSCAN during cluster expansion. The light grey cluster on the left side has been processed already. The algorithm currently expands the dark grey cluster on the right side. The seedlist S currently contains one object, the object x. x is a core object since there are more than $MinPts = 3$ objects in its ϵ-neighborhood ($|N_\epsilon(x)| = 6$, including x itself). Two of these objects, x' and x'' have not been processed so far and are therefore inserted into S. This way, the cluster is iteratively expanded until the seed list is empty. After that, the algorithm continues with an arbitrary unprocessed object until all objects have been processed.

Since every object of the database is considered only once in Step 2 *or* 6 (exclusively), we have a complexity which is n times the complexity of $N_\epsilon(x)$ (which is linear in n if there is no index structure, and sublinear or even $O(\log(n))$ in the presence of a multidimensional index structure. The result of DBSCAN is determinate.

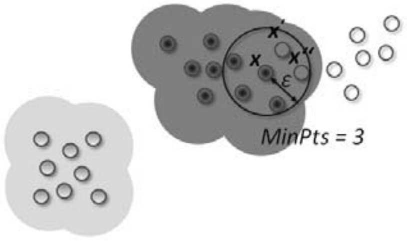


Fig. 8. Sequential Density-based Clustering

```
algorithm GPUdbscanNLJ(data set $\mathcal{D}$)          // host program executed on CPU
   deviceMem float $\mathcal{D}'[][] := \mathcal{D}[][]$;          // allocate memory in DM for the data set $\mathcal{D}$
   deviceMem int counter [$n$];                    // allocate memory in DM for counter
   #threads := $n$;                                // number of points in $\mathcal{D}$
   #threadsPerGroup := 64;
   startThreads (GPUdbscanKernel, #threads, #threadsPerGroup);      // one thread per point
   waitForThreadsToFinish();
   copy counter from DM to main memory ;
end.

kernel GPUdbscanKernel (int threadID)
   register float $q[] := \mathcal{D}'$[threadID][];          // copy the point from DM into the register
                                                   // and use it as query point $q$
                                                   // index is determined by the threadID
   for $i := 0$ ... threadID do                    // option 1 OR
   for $i := 0$ ... $n-1$ do                       // option 2
      synchronizeThreadGroup();
      shared float $x[] := \mathcal{D}'[i][]$;               // copy the current point $x$ from DM to SM
      synchronizeThreadGroup();                    // Now all threads of the thread group can work with $x$
      if dist$(x, q) \le \epsilon$ then
         atomicInc (counter[$i$]); atomicInc (counter[threadID]); // option 1 OR
         inc counter[threadID];                              // option 2
end.
```

Fig. 9. Parallel Algorithm for the Nested Loop Join to Support DBSCAN on GPU

6.2 GPU-Supported DBSCAN

To effectively support DBSCAN on GPU we first identify the two major stages of the algorithm requiring most of the processing time:

1. Determination of the core object property.
2. Cluster expansion by computing the transitive closure of the direct density reachability relation.

The first stage can be effectively supported by the similarity join. To check the core object property, we need to count the number of objects which are within the ϵ-neighborhood of each point. Basically, this can be implemented by a self join. However, the algorithm for self-join described in Section 5 needs to be modified to be suitable to support this task. The classical self-join only counts the total number of pairs of data objects with distance less or equal than ϵ. For the core object property, we need a self-join with a counter associated to each object. Each time when the algorithm detects a new pair fulfilling the join condition, the counter of both objects needs to be incremented.

We propose two different variants to implement the self-join to support DBSCAN on GPU which are displayed in pseudocode in Figure 9. Modifications over the basic algorithm for nested loop join (cf. Figure 6) are displayed in darker color. As in the simple algorithm for nested loop join, for each point q of the outer loop a separate thread with a unique *threadID* is created. Both variants of the self-join for DBSCAN operate on a array *counter* which stores the number of neighbors for each object. We have two options how to increment the counters of the objects when a pair of objects (x, q) fulfills the join condition. Option 1 is first to add the counter of x and then the counter of q using the atomic operation `atomicInc()` (cf. Section 3). The operation `atomicInc()` involves synchronization of all threads. The atomic operations are required to assure the correctness of the result, since it is possible that different threads try to increment the counters of objects simultaneously.

In clustering, we typically have many core objects which causes a large number of synchronized operations which limit parallelism. Therefore, we also implemented option 2 which guarantees correctness without synchronized operations. Whenever a pair of objects (x, q) fulfills the join condition, we only increment the counter of point q. Point q is that point of the outer loop for which the thread has been generated, which means q is exclusively associated with the *threadID*. Therefore, the cell *counter*[*threadID*] can be safely incremented with the ordinary, non-synchronized operation `inc()`. Since no other point is associated with the same *threadID* as q no collision can occur. However, note that in contrast to option 1, for each point of the outer loop, the inner loop needs to consider all other points. Otherwise results are missed. Recall that for the conventional sequential nested loop join (cf. Figure 5) it is sufficient to consider in the inner loop only those points which have not been processed so far. Already processed points can be excluded because if they are join partners of the current point, this has already been detected. The same holds for option 1. Because of parallelism, we can not state which objects have been already processed. However, it is still sufficient when each object searches in the inner loop for join partners among those objects which would appear later in the sequential processing order. This is because all other object are addressed by different threads. Option 2 requires checking all objects since only one counter is incremented. With sequential processing, option 2 would thus duplicate the workload. However, as our results in Section 8 demonstrate, option 2 can pay-off under certain conditions since parallelism is not limited by synchronization.

After determination of the core object property, clusters can be expanded starting from the core objects. Also this second stage of DBSCAN can be effectively supported on the GPU. For cluster expansion, it is required to compute the transitive closure of the direct density reachability relation. Recall that this is closely connected to the core object property as all objects within the ϵ range of a core object x are directly density reachable from x. To compute the transitive closure, standard algorithms are available. The most well-known among them is the algorithm of Floyd-Warshall. A highly parallel variant of the Floyd-Warshall algorithm on GPU has been recently proposed [15], but this is beyond the scope of this paper.

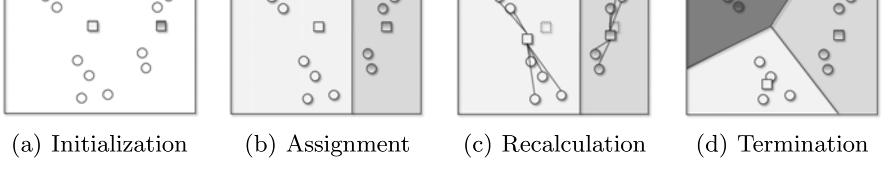
7 K-Means Clustering on GPU

7.1 The Algorithm K-Means

A well-established partitioning clustering method is the K-means clustering algorithm [21]. K-means requires a metric distance function in vector space. In addition, the user has to specify the number of desired clusters k as an input parameter. Usually K-means starts with an arbitrary partitioning of the objects into k clusters. After this initialization, the algorithm iteratively performs the following two steps until convergence: (1) Update centers: For each cluster, compute the mean vector of its assigned objects. (2). Re-assign objects: Assign each object to its closest center. The algorithm converges as soon as no object changes its cluster assignment during two subsequent iterations.

Figure 10 illustrates an example run of K-means for $k = 3$ clusters. Figure 10(a) shows the situation after random initialization. In the next step, every data point is associated with the closest cluster center (cf. Figure 10(b)). The resulting partitions represent the Voronoi cells generated by the centers. In the following step of the algorithm, the center of each of the k clusters is updated, as shown in Figure 10(c). Finally, assignment and update steps are repeated until convergence.

In most cases, fast convergence can be observed. The optimization function of K-means is well defined. The algorithm minimizes the sum of squared distances of the objects to their cluster centers. However, K-means is only guaranteed to converge towards a local minimum of the objective function. The quality of the result strongly depends on the initialization. Finding that clustering with k clusters minimizing the objective function actually is a NP-hard problem, for details see e.g. [23]. In practice, it is therefore recommended to run the algorithm several times with different random initializations and keep the best result. For large data sets, however, often only a very limited number of trials is feasible. Parallelizing K-means in GPU allows for a more comprehensive exploration of



(a) Initialization (b) Assignment (c) Recalculation (d) Termination

Fig. 10. Sequential Partitioning Clustering by the K-means Algorithm

the search space of all potential clusterings and thus provides the potential to obtain a good and reliable clustering even for very large data sets.

7.2 CUDA-K-Means

In K-means, most computing power is spent in step (2) of the algorithm, i.e. re-assignment which involves distance computation and comparison. The number of distance computations and comparisons in K-means is $O(k \cdot i \cdot n)$, where i denotes the number of iterations and n is the number of data points.

The CUDA-K-meansKernel. In K-means clustering, the cluster assignment of each data point is determined by comparing the distances between that point and each cluster center. This work is performed in parallel by the CUDA-K-meansKernel. The idea is, instead of (sequentially) performing cluster assignment of one single data point, we start many different cluster assignments at the same time for different data points. In detail, one single thread per data point is generated, all executing the CUDA-K-meansKernel. Every thread which is generated from the CUDA-K-meansKernel (cf. Figure 11) starts with the ID of a data point x which is going to be processed. Its main tasks are, to determine the distance to the next center and the ID of the corresponding cluster.

```
algorithm CUDA-K-means(data set $\mathcal{D}$, int $k$)            // host program executed on CPU
  deviceMem float $\mathcal{D}'[][] := \mathcal{D}[][]$;              // allocate memory in DM for the data set $\mathcal{D}$
  #threads := $|\mathcal{D}|$;                                // number of points in $\mathcal{D}$
  #threadsPerGroup := 64;
  deviceMem float $Centroids[][]$ := initCentroids();    // allocate memory in DM for the
                                                      // initial centroids
  double $actCosts := \infty$;                             // initial costs of the clustering

  repeat
    $prevCost := actCost$;
    startThreads (CUDA-K-meansKernel, #threads, #threadsPerGroup);     // one thread per point
    waitForThreadsToFinish();
    float $minDist := minDistances[threadID]$;            // copy the distance to the nearest
                                                        // centroid from DM into MM
    float $cluster := clusters[threadID]$;                // copy the assigned cluster from DM into MM
    double $actCosts$ := calculateCosts();                // update costs of the clustering
    deviceMem float $Centroids[][]$ := calculateCentroids(); // copy updated centroids to DM
  until $|actCost - prevCost| < threshold$                // convergence
end.


kernel CUDA-K-meansKernel (int $threadID$)
  register float $x[] := \mathcal{D}'[threadID][]$;     // copy the point from DM into the register
  float $minDist := \infty$;                     // distance of $x$ to the next centroid
  int $cluster$ := null;                        // ID of the next centroid (cluster)
  for $i := 1 \ldots k$ do                        // process each cluster
    register float $c[] := Centroids[i][]$     // copy the actual centroid from DM into the register
    double $dist$ := distance($x$,$c$);
    if $dist < minDist$ then
      $minDist := dist$;
      $cluster := i$;
  report($minDist$, $cluster$);                // report assigned cluster and distance using synchronized writing
end.
```

Fig. 11. Parallel Algorithm for K-means on the GPU

A thread starts by reading the coordinates of the data point x into the register. The distance of x to its closest center is initialized by ∞ and the assigned cluster is therefore set to `null`. Then a loop encounters all $c_1, c_2, \ldots, c_k$ centers and considers them as potential clusters for x. This is done by all threads in the thread group allowing a maximum degree of intra-group parallelism. Finally, the cluster whose center has the minimum distance to the data point x is reported together with the corresponding distance value using synchronized writing.

The Main Program for CPU. Apart from initialization and data transfer from main memory (MM) to DM, the main program consists of a loop starting the CUDA-K-meansKernel on the GPU until the clustering converges. After the parallel operations are completed by all threads of the group, the following steps are executed in each cycle of the loop:

1. Copy distance of processed point x to the nearest center from DM into MM.
2. Copy cluster, x is assigned to, from DM into MM.
3. Update centers.
4. Copy updated centers to DM.

A pseudocode of these procedures is illustrated in Figure 11.

8 Experimental Evaluation

To evaluate the performance of data mining on the GPU, we performed various experiments on synthetic data sets. The implementation for all variants is written in C and all experiments are performed on a workstation with Intel Core 2 Duo CPU E4500 2.2 GHz and 2 GB RAM which is supplied with a Gainward NVIDIA GeForce GTX280 GPU (240 SIMD-processors) with 1GB GDDR3 SDRAM.

8.1 Evaluation of Similarity Join on the GPU

The performance of similarity join on the GPU, is validated by the comparison of four different variants for executing similarity join:

1. Nested loop join (NLJ) on the CPU
2. NLJ on the CPU with index support (as described in Section 4)
3. NLJ on the GPU
4. NLJ on the GPU with index support (as described in Section 4)

For each version we determine the speedup factor by the ratio of CPU runtime and GPU runtime. For this purpose we generated three 8-dimensional synthetic data sets of various sizes (up to 10 million (m) points) with different data distributions, as summarized in Table 1. Data set DS_1 contains uniformly distributed data. DS_2 consists of five Gaussian clusters which are randomly distributed in feature space (see Figure 12(a)). Similar to DS_2, DS_3 is also composed of five Gaussian clusters, but the clusters are correlated. An illustration of data set

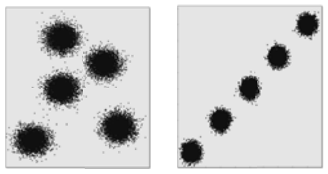

(a) Random Clusters (b) Linear Clusters

Fig. 12. Illustration of the data sets DS_2 and DS_3

Table 1. Data Sets for the Evaluation of the Similarity Join on the GPU

Name	**Size**	**Distribution**
DS_1	3m - 10m points	uniform distribution
DS_2	250k - 1m points	normal distribution, gaussian clusters
DS_3	250k - 1m points	normal distribution, gaussian clusters

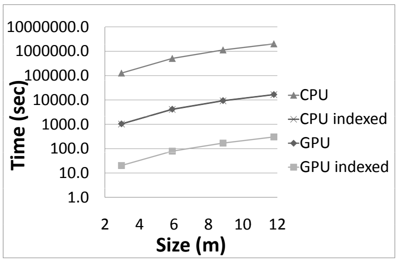
DS_3 is given in Figure 12(b). The threshold ϵ was selected to obtain a join result where each point was combined with one or two join partners on average.

Evaluation of the Size of the Data Sets. Figure 13 displays the runtime in seconds and the corresponding speedup factors of NLJ on the CPU with/without index support and NLJ on the GPU with/without index support in logarithmic scale for all three data sets DS_1, DS_2 and DS_3. The time needed for data transfer from CPU to the GPU and back as well as the (negligible) index construction time has been included. The tests on data set DS_1 were performed with a join selectivity of $\epsilon = 0.125$, and $\epsilon = 0.588$ on DS_2 and DS_3 respectively.

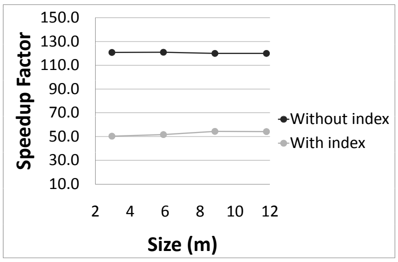
NLJ on the GPU with index support performs best in all experiments, independent of the data distribution or size of the data set. Note that, due to massive parallelization, NLJ on the GPU without index support outperforms CPU without index by a large factor (e.g. 120 on 1m points of normal distributed data with gaussian clusters). The GPU algorithm with index support outperforms the corresponding CPU algorithm (with index) by a factor of 25 on data set DS_2. Remark that for example the overall improvement of the indexed GPU algorithm on data set DS_2 over the non-indexed CPU version is more than 6,000. This results demonstrate the potential of boosting performance of database operations with designing specialized index structures and algorithms for the GPU.

Evaluation of the Join Selectivity. In these experiments we test the impact of the parameter ϵ on the performance of NLJ on GPU with index support and use the indexed implementation of NLJ on the CPU as benchmark. All experiments are performed on data set DS_2 with a fixed size of 500k data points. The parameter ϵ is evaluated in a range from 0.125 to 0.333.

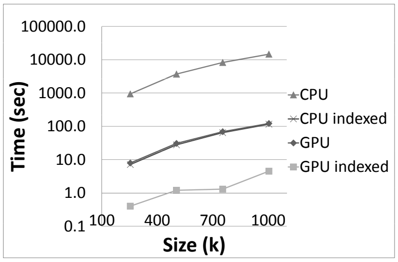
Figure 14(a) shows that the runtime of NLJ on GPU with index support increases for larger ϵ values. However, the GPU version outperforms the CPU implementation by a large factor (cf. Figure 14(b)), that is proportional to the value of ϵ. In this evaluation the speedup ranges from 20 for a join selectivity of 0.125 to almost 60 for $\epsilon = 0.333$.



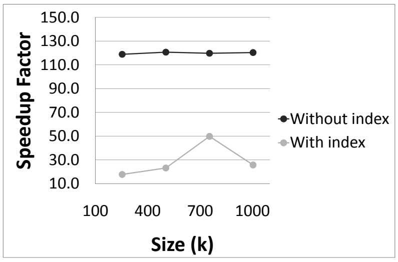
(a) Runtime on Data Set DS_1



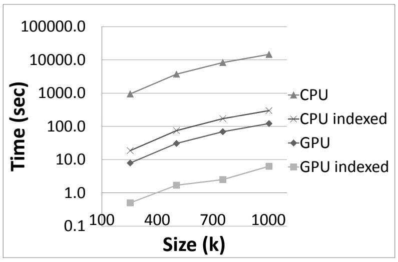
(b) Speedup on Data Set DS_1



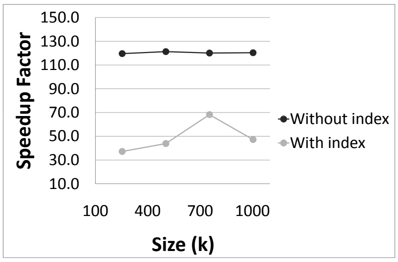
(c) Runtime on Data Set DS_2



(d) Speedup on Data Set DS_2



(e) Runtime on Data Set DS_3

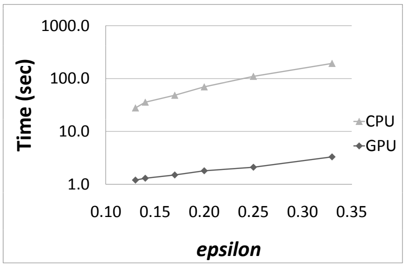


(f) Speedup on Data Set DS_3

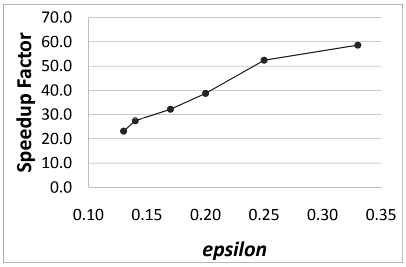
Fig. 13. Evaluation of the NLJ on CPU and GPU with and without Index Support w.r.t. the Size of Different Data Sets

Evaluation of the Dimensionality. These experiments provide an evaluation with respect to the dimensionality of the data. As in the experiments for the evaluation of the join selectivity, we use again the indexed implementations both on CPU and GPU and perform all tests on data set DS_2 with a fixed number of 500k data objects. The dimensionality is evaluated in a range from 8 to 32. We also performed these experiments with two different settings for the join selectivity, namely $\epsilon = 0.588$ and $\epsilon = 1.429$.

Figure 15 illustrates that NLJ on GPU outperforms the benchmark method on CPU by factors of about 20 for $\epsilon = 0.588$ to approximately 70 for $\epsilon = 1.429$. This order of magnitude is relatively independent of the data dimensionality. As in our implementation the dimensionality is already known at compile time, optimization techniques of the compiler have an impact on the performance of



(a) Runtime on Data Set DS_2



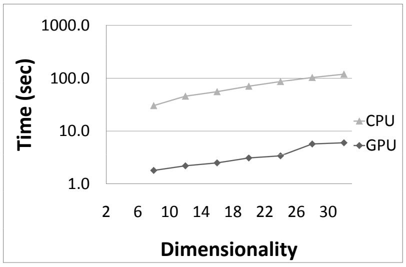
(b) Speedup on Data Set DS_2

Fig. 14. Impact of the Join Selectivity on the NLJ on GPU with Index Support

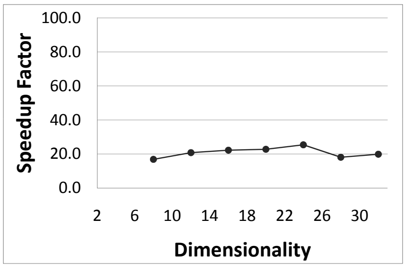
the CPU version as can be seen especially in Figure 15(c). However the dimensionality also affects the implementation on GPU, because higher dimensional data come along with a higher demand of shared memory. This overhead affects the number of threads that can be executed in parallel on the GPU.

8.2 Evaluation of GPU-Supported DBSCAN

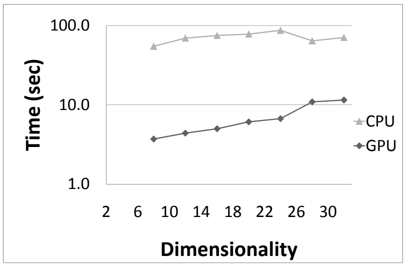
As described in Section 6.2, we suggest two different variants to implement the self-join to support DBSCAN on GPU, whose characteristic are briefly reviewed in the following:



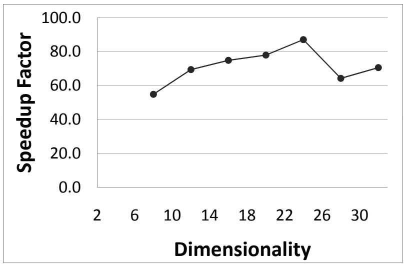
(a) Runtime on Data Set D_2 ($\epsilon = 0.588$)



(b) Speedup on Data Set D_2 ($\epsilon = 0.588$)



(c) Speedup on Data Set D_2 ($\epsilon = 1.429$)



(d) Speedup on Data Set D_2 ($\epsilon = 1.429$)

Fig. 15. Impact of the Dimensionality on the NLJ on GPU with Index Support

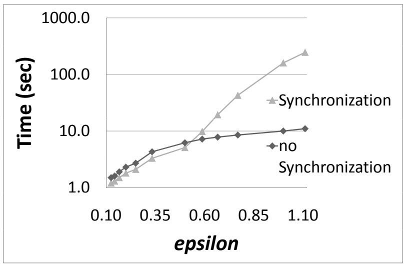


Fig. 16. Evaluation of two versions for the self-join on GPU w.r.t. the join selectivity

1. Increment of the counters regarding a pair of objects (x, q) that fulfills the join condition is done by the use of an atomic operation that involves *synchronization* of all threads.
2. Increment of the counters can be performed *without synchronization* but with duplicated workload instead.

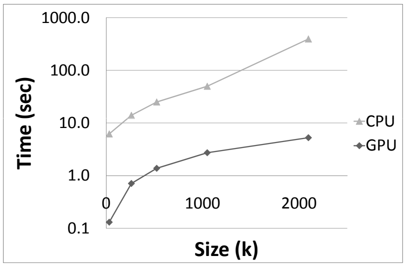
We evaluate both options on a synthetic data set with 500k points generated as specified as DS_1 in Table 1. Figure 16 displays the runtime of both options. For $\epsilon \leq 0.6$, the runtime is in the same order of magnitude, the synchronized variant 1 being slightly more efficient. From this point on, the non-synchronized variant 2 is clearly outperforming variant 1 since parallelism is not limited by synchronization.

8.3 Evaluation of CUDA-K-Means

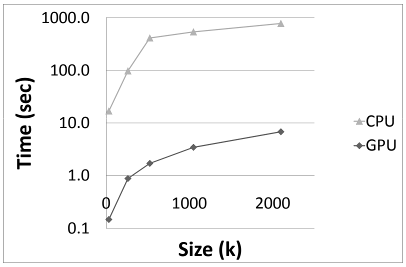
To analyze the efficiency of K-means clustering on the GPU, we present experiments with respect to different data set sizes, number of clusters and dimensionality of the data. As benchmark we apply a single-threaded implementation of K-means on the CPU to determine the speedup of the implementation of K-means on the GPU. As the number of iterations may vary in each run of the experiments, all results are normalized by a number of 50 iterations both on the GPU and the CPU implementation of K-means. All experiments are performed on synthetic data sets as described in detail in each of the following settings.

Evaluation of the Size of the Data Set. For these experiments we created 8-dimensional synthetic data sets of different size, ranging from 32k to 2m data points. The data sets consist of different numbers of random clusters, generated as as specified as DS_1 in Table 1.

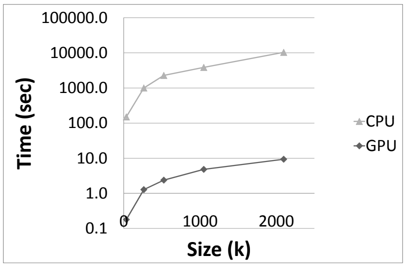
Figure 17 displays the runtime in seconds in logarithmic scale and the corresponding speedup factors of CUDA-K-means and the benchmark method on the CPU for different number of clusters. The time needed for data transfer from CPU to GPU and back has been included. The corresponding speedup factors are given in Figure 17(d). Once again, these experiments support the evidence that the performance of data mining approaches on GPU outperform classic



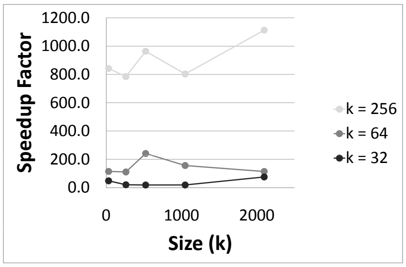
(a) Runtime for 32 clusters



(b) Runtime for 64 clusters



(c) Runtime for 256 clusters



(d) Speedup for 32, 64 and 256 clusters

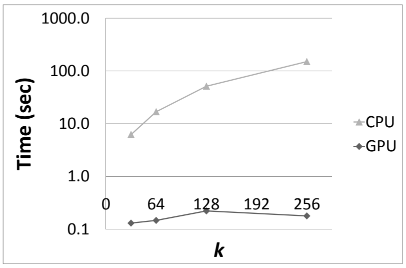
Fig. 17. Evaluation of CUDA-K-means w.r.t. the Size of the Data Set

CPU versions by significant factors. Whereas a speedup of approximately 10 to 100 can be achieved for relatively small number of clusters, we obtain a speedup of about 1000 for 256 clusters, that is even increasing with the number of data objects.

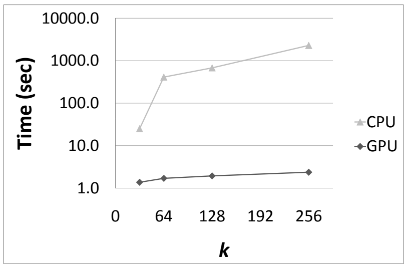
Evaluation of the Impact of the Number of Clusters. We performed several experiments to validate CUDA-K-means with respect to the number of clusters K. Figure 18 shows the runtime in seconds of CUDA-K-means compared with the implementation of K-means on the CPU on 8-dimensional synthetic data sets that contain different number of clusters, ranging from 32 to 256, again together with the corresponding speedup factors in Figure 18(d).

The experimental evaluation of K on a data set that consists of 32k points results in a maximum performance benefit of more than 800 compared to the benchmark implementation. For 2m points the speedup ranges from nearly 100 up to even more than 1,000 for a data set that comprises 256 clusters. In this case the calculation on the GPU takes approximately 5 seconds, compared to almost 3 hours on the CPU. Therefore, we determine that due to massive parallelization, CUDA-K-means outperforms CPU by large factors, that are even growing with K and the number of data objects n.

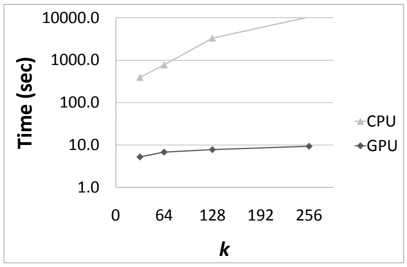
Evaluation of the Dimensionality. These experiments provide an evaluation with respect to the dimensionality of the data. We perform all tests on synthetic



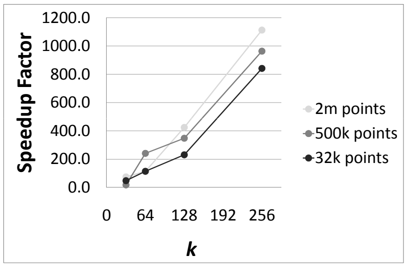
(a) Runtime for 32k points



(b) Runtime for 500k points



(c) Runtime for 2m points

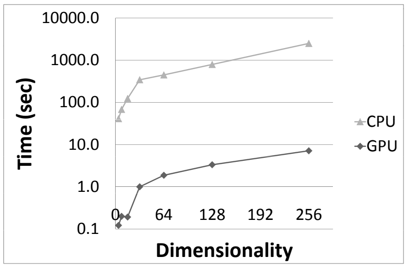


(d) Speedup for 32k, 500k and 2m points

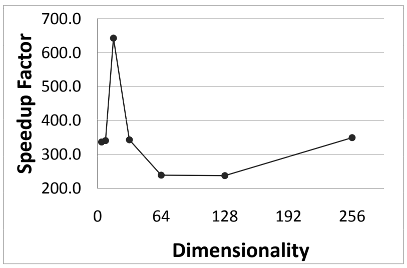
Fig. 18. Evaluation of CUDA-K-means w.r.t. the number of clusters K

data consisting of 16k data objects. The dimensionality of the test data sets vary in a range from 4 to 256. Figure 19(b) illustrates that CUDA-K-means outperforms the benchmark method K-means on the CPU by factors of 230 for 128-dimensional data to almost 500 for 8-dimensional data. On the GPU and the CPU, the dimensionality affects possible compiler optimization techniques, like loop unrolling as already shown in the experiments for the evaluation of the similarity join on the GPU.

In summary, the results of this section demonstrate the high potential of boosting performance of complex data mining techniques by designing specialized index structures and algorithms for the GPU.



(a) Runtime



(b) Speedup

Fig. 19. Impact of the Dimensionality of the Data Set on CUDA-K-means

9 Conclusions

In this paper, we demonstrated how Graphics processing Units (GPU) can effectively support highly complex data mining tasks. In particular, we focussed on clustering. With the aim of finding a natural grouping of an unknown data set, clustering certainly is among the most wide spread data mining tasks with countless applications in various domains. We selected two well-known clustering algorithms, the density-based algorithm DBSCAN and the iterative algorithm K-means and proposed algorithms illustrating how to effectively support clustering on GPU. Our proposed algorithms are accustomed to the special environment of the GPU which is most importantly characterized by extreme parallelism at low cost. A single GPU consists of a large number of processors. As buildings blocks for effective support of DBSCAN, we proposed a parallel version of the similarity join and an index structure for efficient similarity search. Going beyond the primary scope of this paper, these building blocks are applicable to support a wide range of data mining tasks, including outlier detection, association rule mining and classification. To illustrate that not only local density-based clustering can be efficiently performed on GPU, we additionally proposed a parallelized version of K-means clustering. Our extensive experimental evaluation emphasizes the potential of the GPU for high-performance data mining. In our ongoing work, we develop further algorithms to support more specialized data mining tasks on GPU, including for example subspace and correlation clustering and medical image processing.

References

1. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (2007)
2. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: Ecm on graphics cards. In: Soux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)
3. Böhm, C., Braunmüller, B., Breunig, M.M., Kriegel, H.-P.: High performance clustering based on the similarity join. In: CIKM, pp. 298–305 (2000)
4. Böhm, C., Noll, R., Plant, C., Zherdin, A.: Indexsupported similarity join on graphics processors. In: BTW, pp. 57–66 (2009)
5. Breunig, M.M., Kriegel, H.-P., Ng, R.T., Sander, J.: Lof: Identifying density-based local outliers. In: SIGMOD Conference, pp. 93–104 (2000)
6. Cao, F., Tung, A.K.H., Zhou, A.: Scalable clustering using graphics processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 372–384. Springer, Heidelberg (2006)
7. Catanzaro, B.C., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: ICML, pp. 104–111 (2008)
8. Ester, M., Kriegel, H.-P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD, pp. 226–231 (1996)
9. Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P.: Knowledge discovery and data mining: Towards a unifying framework. In: KDD, pp. 82–88 (1996)

10. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: Gputerasort: high performance graphics co-processor sorting for large database management. In: SIGMOD Conference, pp. 325–336 (2006)
11. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M.C., Manocha, D.: Fast computation of database operations using graphics processors. In: SIGMOD Conference, pp. 215–226 (2004)
12. Guha, S., Rastogi, R., Shim, K.: Cure: An efficient clustering algorithm for large databases. In: SIGMOD Conference, pp. 73–84 (1998)
13. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD Conference, pp. 47–57 (1984)
14. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational joins on graphics processors. In: SIGMOD, pp. 511–524 (2008)
15. Katz, G.J., Kider, J.T.: All-pairs shortest-paths for large graphs on the gpu. In: Graphics Hardware, pp. 47–55 (2008)
16. Kitsuregawa, M., Harada, L., Takagi, M.: Join strategies on kd-tree indexed relations. In: ICDE, pp. 85–93 (1989)
17. Koperski, K., Han, J.: Discovery of spatial association rules in geographic information databases. In: Egenhofer, M.J., Herring, J.R. (eds.) SSD 1995. LNCS, vol. 951, pp. 47–66. Springer, Heidelberg (1995)
18. Leutenegger, S.T., Edgington, J.M., Lopez, M.A.: Str: A simple and efficient algorithm for r-tree packing. In: ICDE, pp. 497–506 (1997)
19. Lieberman, M.D., Sankaranarayanan, J., Samet, H.: A fast similarity join algorithm using graphics processing units. In: ICDE, pp. 1111–1120 (2008)
20. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: Molecular dynamics simulations on commodity gpus with cuda. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 185–196. Springer, Heidelberg (2007)
21. Macqueen, J.B.: Some methods of classification and analysis of multivariate observations. In: Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297 (1967)
22. Manavski, S., Valle, G.: Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. BMC Bioinformatics 9 (2008)
23. Meila, M.: The uniqueness of a good optimum for k-means. In: ICML, pp. 625–632 (2006)
24. Plant, C., Böhm, C., Tilg, B., Baumgartner, C.: Enhancing instance-based classification with local density: a new algorithm for classifying unbalanced biomedical data. Bioinformatics 22(8), 981–988 (2006)
25. Shalom, S.A.A., Dash, M., Tue, M.: Efficient k-means clustering using accelerated graphics processors. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 166–175. Springer, Heidelberg (2008)
26. Szalay, A., Gray, J.: 2020 computing: Science in an exponential world. Nature 440, 413–414 (2006)
27. Tasora, A., Negrut, D., Anitescu, M.: Large-scale parallel multi-body dynamics with frictional contact on the graphical processing unit. Proc. of Inst. Mech. Eng. Journal of Multi-body Dynamics 222(4), 315–326