Density-based Clustering using Graphics Processors

Christian Böhm,
University of Munich
Munich, Germany
boehm@dbs.ifi.lmu.de

Robert Noll
University of Munich
Munich, Germany
rnoll@fehler4.de

Claudia Plant
Technische Universität München
Munich, Germany
plant@lrz.tum.de

Bianca Wackersreuther
University of Munich
Munich, Germany
wackersb@dbs.ifi.lmu.de

ABSTRACT

During the last few years, GPUs have evolved from simple devices for the display signal preparation into powerful coprocessors that do not only support typical computer graphics tasks but can also be used for general numeric and symbolic computation tasks. As major advantage GPUs provide extremely high parallelism combined with a high bandwidth in memory transfer at low cost. We want to exploit these advantages in density-based clustering, an important paradigm in clustering since typical algorithms of this category are noise and outlier robust and search for clusters of an arbitrary shape in metric and vector spaces. Moreover, with a time complexity ranging from $O(n \log n)$ to $O(n^2)$ these algorithms are scalable to large data sets in a database system. In this paper, we propose CUDA-DClust, a massively parallel algorithm for density-based clustering for the use of a Graphics Processing Unit (GPU). While the result of this algorithm is guaranteed to be equivalent to that of DBSCAN, we demonstrate a high speed-up, particularly in combination with a novel index structure for use in GPUs.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications—*Data mining*; I.3.1 [**Computer Graphics**]: Hardware Architecture—*Graphics processors*

General Terms

Algorithms, Performance

1. INTRODUCTION

Graphic Processing Units (GPUs) have recently 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 image update. Therefore, today's graphics hardware contains a large number of programmable processors, which are optimized to cope with this high workload in highly a parallel way. In terms of peak performance, the GPU has outperformed state-of-the-art multi-core CPUs by a large margin. Hence, there is a great effort in many research communities to use the computational capabilities of GPUs even for purposes, which are not at all related to computer graphics, including life sciences, mechanical simulation or data mining. The corresponding research area is called *General Processing-Graphics Processing Unit*. A brief survey of recent approaches in the field of database management supported by GPUs is given in Section 2. Vendors of graphics hardware have anticipated that trend and developed libraries, precompilers and application programming interfaces. Most prominently, NVIDIA's technology *Compute Unified Device Architecture* (CUDA) offers free evelopment tools 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 or so-called *main program* is executed on the CPU. In contrast, the *kernel functions* are executed in a massively parallel fashion on hundreds of processors on the GPU. Analogous techniques are also offered by ATI using the brand names Close-to-Metal, Stream SDK, and Brook-GP.

Data mining tasks such as classification, clustering, or outlier detection are often very computationally demanding. Typical algorithms for the learning of complex statistical models from data are in quadratic or cubic complexity classes w.r.t. the number of objects and/or the dimensionality. To make these algorithms usable in a large and high dimensional database context is therefore a great challenge. Clustering is among the most important problems of data mining. Intuitively, it means to arrange the database objects such that similar objects are in the same group, called *cluster* and dissimilar objects are in different clusters. Typical applications include the segmentation of customers, the functional analysis of genes by detecting similar expression patterns, or the analysis of brain activities in neuro-imaging. The problem of clustering has attracted a huge volume of attention for several decades, with multiple books [11, 21], surveys [18] and papers presenting algorithms for partitioning clustering [19], subspace clustering [2] or density-based clustering [8, 3, 14] just to name a few directions. Whereas the widespread and well-known clustering algorithms of the k-means type focus on *Gaussian* or *spherically shaped* clus-

CIKM'09, November 2–6, 2009, Hong Kong, China.

ters it is generally agreed now that for many applications such assumptions are not beneficial. Hence, an important research focus has been to find clusters of arbitrary shape, by various methods of *spectral* clustering and *density-based* clustering. As spectral clustering requires a space complexity of $O(n^2)$ and a time complexity of $O(n^2)$ to $O(n^3)$ (where n is the number of objects), spectral clustering is not really scalable to database scenarios. In contrast, basic methods for density-based clustering have a linear space complexity and a $O(n^2)$ time complexity, that can even be reduced up to $O(n \log n)$ by the use of an appropriate indexing structure for similarity search. A brief review of density-based clustering approaches that enable one to find clusters of arbitrary shape, and that are very robust w.r.t noise and outliers is presented in Section 2.

To combine the high computational power and the multiple advantages provided by density-based clustering, we propose CUDA-DClust, a density-based clustering algorithm specially dedicated to the use of GPUs under NVIDIA's CUDA architecture and programming model. CUDA-DClust is highly parallel and exploits thus the high number of simple SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) processors of today's graphics hardware. The parallelization which is required for GPUs differs considerably from previous parallel algorithms which have focused on shared-nothing parallel architectures prevalent in server farms. In contrast, GPU-capable parallel algorithms do not only have shared main memory but groups of the processors even share very fast memory units at the speed of first level-cache or registers. Hence, the cooperation between different processes is not organized by message passing but by information sharing on different levels of fast and fastest memory. For even further acceleration, we also propose the algorithm CUDA-DClust*, a variant of CUDA-DClust that uses an index structure for similarity search which is particularly composed for the use in GPUs. We demonstrate the superiority of CUDA-DClust and CUDA-DClust* over the corresponding sequential density-based clustering methods on CPU. Both algorithms outperform their sequential counterpart by typically one order of magnitude, while the clustering results can be proven to be identical.

The rest of the paper is organized as follows: Section 2 surveys the related work in density-based clustering and GPGPU processing. Section 3 explains the graphics hardware and the corresponding programming model. A formal definition of density-based clustering is given in Section 4. Section 5 is dedicated to our highly parallel density-based clustering algorithm CUDA-DClust. Section 6 describes an index structure which can be applied in the simplified context of GPU processing. Section 7 demonstrates an impressive experimental evaluation of CUDA-DClust and its indexed version, and Section 8 concludes with a summary and some directions for future research.

2. RELATED WORK

Density-based Clustering. In density-based clustering, clusters are regarded as areas of high object density in the data space, which are separated by areas of lower density. This cluster notion has many benefits, as it allows to detect clusters of arbitrary shape, and most algorithms for density-based clustering are rather robust against outliers and noise points. The algorithms differ in the formal representation of the density-based cluster notion and in the search strategies to find the clusters. DBSCAN [8] determines a non-hierarchical, disjoint partitioning of the data into several clusters. Clusters are expanded starting at arbitrary seed points within dense areas. Objects in areas of low density are assigned to a separate noise partition. DBSCAN is robust against noise and the user does not need to specify the number of clusters in advance. A hierarchical extension of DBSCAN is presented via OPTICS [3]. The algorithm DENCLU [14] formalizes the cluster notion by non-parametric kernel density estimation based on modelling the local influence of each data object on the feature space by a simple kernel function, e.g. Gaussian. It defines a cluster as a local maximum of the probability density. A faster variant of DENCLU has recently been proposed [13]. [23] combines the idea of kernel density estimation with level set methods. Especially in difficult situations with overlapping clusters, this technique outperforms DENCLU and DBSCAN.

Parallel Variants of Density-based Clustering. Multiple papers cover parallel variants of the density-based clustering approach DBSCAN. Among these, PDBSCAN [22] is a parallel clustering algorithm for mining large distributed spatial databases. It uses the so-called *shared nothing* architecture, which has the main advantage that it can be scaled up to a high number of computers. In [4] a mapping of the overall structure of DBSCAN to separate sequential modules of the key functionalities is presented and among these a parallel cooperation scheme is devised. This version of DBSCAN is useful to improve performance on high-dimensional data. In [6], DBSCAN is parallelized by a conservative approximation of complex distance functions, based on the concept of merge points. The final clustering result is derived from a global cluster connectivity graph. However the parallel variants of density-based clustering can not straightforward be transfered on a GPU, because the parallelization, which is required for GPUs differs considerably. As the parallel algorithms described above focus on shared-nothing parallel architectures, GPU-capable parallel algorithms do not only have shared main memory but groups of the processors even share very fast memory units. Hence, the cooperation between different processes is not organized by message passing but by information sharing on different levels of extremely fast memory.

General Processing-Graphics Processing Unit. Some recent approaches, e.g. [9] and [10] demonstrate that important building blocks for query processing in databases, e.g. sorting, can be significantly speed up by the use of GPUs. In [12] some algorithms for the relational join on a GPU are presented. Two recent papers [17, 5] focus on GPU algorithms for the similarity join, which determines all pairs of objects from two different sets that are neighbors. [17] proposes an algorithm based on the concept of space filling curves, e.g. the z-order, to prune the search space. The z-order of a set of objects can be determined very efficiently on GPU by highly parallelized sorting. 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 problem is presented in [5]. Here, the baseline technique underlying any join operation with an arbitrary join predicate, the nested loop join (NLJ) is parallelized. Probably the most related approaches to ours are

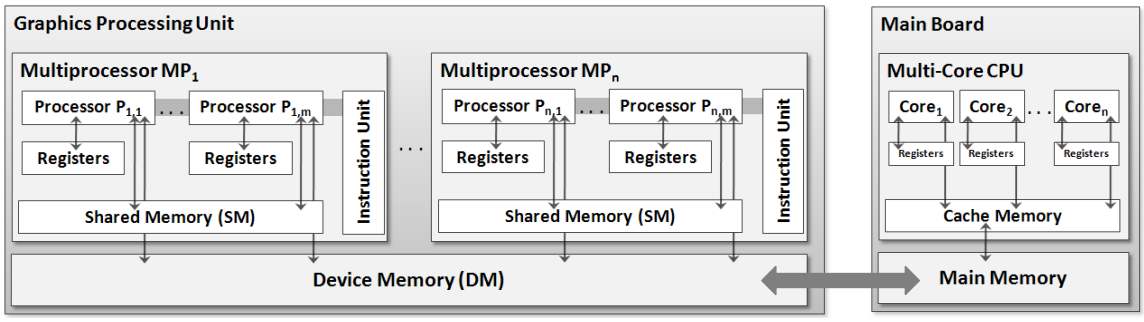


Figure 1: Architecture of the CUDA model including communications between CPU and GPU.

two papers on clustering on GPU. [7] extends the basic idea of k-means clustering by calculating the distances from a single input centroid to all objects at one time, which can be done simultaneously on GPU. [20] parallelizes k-means by using multi-pass rendering and multi shader program constants. Thus, the authors achieve significant increasing performances. However, the effectiveness of both approaches strongly depends on the shape of the clusters that are present in the data sets, and they are not portable to different GPU models, like CUDA-techniques are.

3. THE GPU: A MASSIVELY PARALLEL SUPERCOMPUTER

GPUs of the newest generation are powerful coprocessors, not only designed for games and other graphic-intensive applications, but also for general-purpose computing. From the hardware perspective, a GPU consists of a number of multiprocessors (MPs), each of which consists of a set of simple SIMD-processors. These SIMD-processors, which we also call just processors wherever non-ambiguous operate in a SIMD fashion. For instance, NVIDIA's GPU of the GT200 series, e.g. the Geforce GTX280 has 30 MPs, 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 four important types of memory (cf. Figure 1):

1. Set of *registers*: A fast memory area, which is completely private to each of the SIMD processors.
2. *Shared Memory* (SM): A memory unit with fast access but very limited in capacity, which is shared among all processors of a MP. It can be used to exchange information between threads on different processors of the same MP, but cannot be used for information, which is shared among threads on *different* MPs.
3. *Device Memory* (DM): The actual video RAM of the graphics processor, which is physically located on the graphics card but not inside the GPU. DM is significantly larger than SM but also significantly slower. It can be used to share information between threads on different MPs.
4. *Main Memory* (MM): A memory unit that 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 MM.

3.2 The Programming Model

The basis of the programming model of GPUs are threads. Threads are lightweight processes that 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. 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 MP 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 parallely executed kernel functions on the GPU as well as the sequentially executed host program on the CPU are defined in an extended syntax of the C programming language.

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 eight processors of a single MP 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 of SIMD-processors per MP, 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 TG use the same SM and may thus communicate and share data via the SM. When all warps of the same group have reached a

particular point of execution the threads in one TG can be synchronized. The latency delay of the DM can be hidden by scheduling other warps of the same or a different TG whenever one warp waits for an access to DM. To allow switching between warps of different TGs on a MP, it is recommended that each thread uses only a small fraction of the SM and registers of the MP [1].

3.3 Atomic Operations

In order to synchronize parallel processes and to ensure the correctness of parallel algorithms, CUDA offers *atomic operations*, e.g. `increment`, `decrement`, or `exchange`. Most of these operations work on integer data types in DM. However, the newest version of CUDA (Compute Capability 1.3 of the GPU GT200) even allows 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. In this case, atomicity implies the following two requirements: If two or more threads increment the list counter, then first the value counter after all concurrent increments must be equivalent to the old value plus the number of concurrent `increment` operations and, second, 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. Hence, 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. `atomicInc` works as follows: The addressed counter value is read, and incremented if the threshold is not exceeded. Finally, the *old* value of the counter is returned to the kernel method which invoked `atomicInc` before incrementing. If two or more threads of the same or different TGs 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. `atomicCAS` performs a Compare-and-Swap operation using 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 each case, the old value at the address is returned to the invoking kernel method before swapping.

4. FOUNDATIONS OF DENSITY-BASED CLUSTERING

The idea of density-based clustering is that clusters are areas of high point density, separated by areas of significantly lower point density that 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 P is called a core object of a cluster, if at least $MinPts$ objects (including P itself) are in its ϵ-neighborhood denoted by $N_\epsilon(P)$, which corresponds to a sphere of radius ϵ. Formally:

DEFINITION 1. *(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 $P \in \mathcal{D}$ is a core object, iff

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

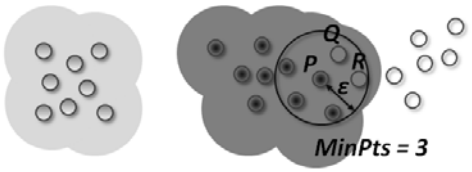


Figure 2: Sequential density-based clustering.

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 2. *(Direct Density Reachability)*
Let $P, Q \in \mathcal{D}$. Q is called directly density reachable from P (in symbols: $P \lhd Q$) iff

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

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

DEFINITION 3. *(Density Connectedness)*
Two objects P and Q are called density connected (in symbols: $P \bowtie Q$) iff there is a sequence of core objects $(P_1, ..., P_m)$ of arbitrary length m such that

$$P \rhd P_1 \rhd ... \lhd P_m \lhd Q.$$

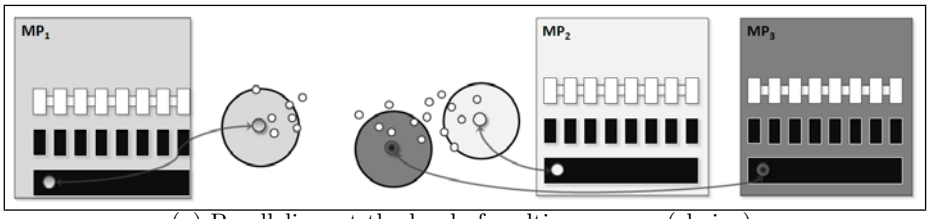
In density-based clustering, a cluster is defined as a maximal set of density connected objects:

DEFINITION 4. *(Density-based Cluster)*
A subset $C \subseteq \mathcal{D}$ is called a cluster iff the following two conditions hold:

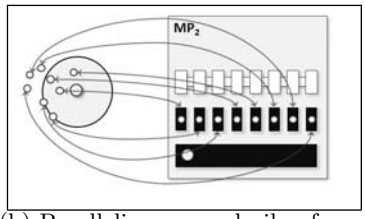
1. *Density connectedness: $\forall P, Q \in C : P \bowtie Q$.*
2. *Maximality: $\forall P \in C, \forall Q \in \mathcal{D} \setminus C : \neg P \bowtie Q$.*

The algorithm DBSCAN [8] implements the cluster notion of Definition 4 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 arbitrary unprocessed object $P \in \mathcal{D}$.
3. If P is core object, assign new cluster-ID C, and do step (4) for all elements $Q \in N_\epsilon(P)$, which do not yet have a cluster-ID:
4. (a) mark element Q with cluster-ID C and
   (b) insert object Q into seed list S.
5. While S not $\emptyset$ repeat step (6) for all elements $P' \in S$:
6. If P' is core object, do step (7) for all elements $Q \in N_\epsilon(P')$, which do not yet have any cluster-ID:
7. (a) mark element Q with cluster-ID C and
   (b) insert object Q into seed list S.
8. If $\mathcal{D}$ still contains unprocessed objects, do step (2).



(a) Parallelism at the level of multiprocessors (chains).



(b) Parallelism exemplarily of SIMD-processor MP_2.

Figure 3: Two different ideas of parallelization.

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 $N_\epsilon(P)$ (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. To illustrate the algorithmic paradigm, Figure 2 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. The seed list S only contains the object P. P is a core object since there are more than $MinPts = 3$ objects in its ϵ-neighborhood. Two of these objects, Q and R have not been processed so far and are therefore inserted into S. This way, the cluster is iteratively expanded until S is empty. After that, the algorithm continues with an arbitrary unprocessed object until all objects have been processed.

5. CUDA-DCLUST

5.1 Chains: The Main Idea of Parallelization

Since in density-based clustering there is a complex relationship between the data objects and an unknown number of clusters, where two objects share a common cluster whenever they are (directly or indirectly) connected by an unknown number of core objects, it is not possible to define a straightforward parallelization e.g. by assigning an individual thread to each data object, cluster, or dimension. Instead, our approach CUDA-DClust is based on a new concept especially introduced to allow for massive parallelism in density-based clustering, called *chain*. A chain is a set of data objects belonging to a common density-based cluster, formally:

Definition 5. *(Density-based Chain)*
A subset $C \subseteq \mathcal{D}$ is called a density-based chain iff the following condition holds:

$$\forall P, Q \in C : P \bowtie Q.$$

Note that a cluster may be composed by several chains but each chain belongs to one single cluster only. In contrast to Definition 4, we do not require maximality. Using the concept of chains, we basically perform the task of cluster expansion, i.e. the transitive closure computation of the relation of direct density reachability. Thus, a chain can be considered as a tentative cluster with a tentative cluster-ID (which we call *chain-ID*). The idea is, instead of sequentially performing *one* single cluster expansion like, for instance, in DBSCAN, we start many different cluster expansions at the same time via different chains from different starting points. Of course CUDA-DClust has to register every *collision* between two or more chains carefully. A collision means that CUDA-DClust has to notice that two chains actually participate in the same cluster. A detailed description of our collision handling is presented in Section 5.2.

We now introduce a concept for book-keeping of all information that is essential for collision handling, called *collision matrix*. A collision matrix C is an upper triangular boolean matrix of size $(p \times p)$, where p is the number of chains that are expanded simultaneously. If a collision of chains i and j is detected, we set $C_{i,j} :=$ **true** if $i < j$ and $C_{j,i} :=$ **true** otherwise. Since C is small, we later run a sequential algorithm to determine transitive collisions. Finally, a valid cluster-ID is assigned to each chain and is distributed to the points in a parallel fashion on GPU. Figure 3(a) illustrates parallelism based on the concept of chains. Here, three density-based chains are processed in different TGs and may, therefore, be computed in a parallel fashion on different MPs of the GPU. Each MP is responsible for the expansion of one chain. The seed point to be expanded is loaded into the SM in order to have it efficiently accessible for every thread in this TG.

Besides the approach of chains, we now introduce a second, but equally important idea of parallelization. Whenever a point S is considered for determining its core object property and for marking its neighbors as potential new seed points, all points in $N_\epsilon(S)$ must be determined. Therefor, a number of threads is generated in order to process many potential neighbors P_i of S simultaneously. Note that, as already mentioned in Section 3, the generation of a high number of threads in CUDA does not cause any considerable overhead. In contrast, it is of high importance to generate a high number of potentially parallel threads in order to keep the processors in a GPU busy, even in the frequent case of delays due to memory latency. Figure 3(b) shows the intra-MP parallism, exemplarily for MP_2 of the example illustrated in Figure 3(a). All possible mating partners Q_i of the seed point, which have to be tested whether or not they are in $N_\epsilon(P)$ are processed by different SIMD-processors in MP_2. Hence, the coordinates are loaded in the corresponding registers of the SIMD-processors, and each of them computes one distance function between P and Q_i simultaneously. Summarizing, we achieve a high degree of intra-MP and inter-MP parallelism by having one TG for each parallely executed chain and one thread for each potential partner point of the current seed point of a chain.

In CUDA, multiples of 32 threads must be grouped into TGs. As indicated in Section 3, threads of the same group

```
// Some global information available on Device Memory:

float pointSet [n][d];
int pointState [n];  // initialized with UNPROCESSED
                     // It will contain the ChainID/ClusterID of the object
                     // but also special values like NOISE.
float ϵ;
int minPts;
int threadGroupSize, numChains, maxSeedLength;
int seedList [numChains][maxSeedLength];
int seedLength [numChains];
boolean collisionMatrix[numChains][numChains];   // initialized with false

kernel ClusterExpansion (int threadGroupId, int threadID)
   chainID ≡ threadGroupID;   // ≡ means only shortcut for readability.
   mySeedList [] ≡ seedList [chainID][];
   mySeedLength ≡ seedLength [chainID];
   shared int neighborBuffer [minPts];
   shared int neighborCount = 0;

   mySeedLength := mySeedLength − 1;
   shared int seedPointID := mySeedList [mySeedLength];
   shared float P[] := pointSet [seedPointID][];   // copy from DM.
   syncThreads () ;
   for i := threadID to n − 1 step threadGroupSize do
      processObject (i);
   syncThreads () ;
   if neighborCount ≥ minPts then
      pointState[seedPointID] := chainID ;
      for i := 0 to minPts − 1 do              // Reconsider the neighbors
         markAsCandidate(neighborBuffer[i]); // in the quarantine buffer.
   else
      pointState[seedPointID] := NOISE ;
end kernel;

procedure processObject (int i)
   register float Q [] := pointSet [i][];
   register float δ := Distance (P, Q);
   if δ ≤ ϵ then
      register int h = atomicInc (neighborCount);
      if h ≥ minPts then
         markAsCandidate (i);
      else                                  // P is not yet confirmed as core object:
         neighborBuffer [h] := i;   // Therefore, put Q in quarantine buffer.
end procedure;

procedure markAsCandidate (int i)
   register int oldState=atomicCAS(pointState[i],UNPROCESSED,chainID);
   if oldState = UNPROCESSED then
      register int h := atomicInc (mySeedLength, maxSeedLength) ;
      if h < maxSeedLength then
         seedList [h] := i ;
   else
      if oldState ≠ NOISE ∧ oldState ≠ chainID then   // Collision !
         if oldState < i then
            collisionMatrix[oldState][i] := true;
         else
            collisionMatrix[i][oldState] := true;
end procedure;
```

Figure 4: The Cluster Expansion Kernel.

are potentially executed in a fully synchronous way in the same MP, may exchange information through the SM, and all threads of the same TG can be synchronized when all threads wait until all other threads of the same TG have reached the same point of execution. Therefore, it is obviously beneficial to collect all threads belonging to the same chain into one TG.

5.2 The Cluster Expansion Kernel

The Cluster Expansion Kernel is the main kernel method of CUDA-DClust. It performs the actual work, i.e. the determination of the core point property and the transitive expansion of the clusters. We generate a high number of threads, all executing the Cluster Expansion Kernel. Some other kernel methods for supportive and cleaning tasks will be shortly described later.

Consider the pseudocode presented in Figure 4. Each TG, generated by the Cluster Expansion Kernel starts with the ID of the seed point P that is going to be processed. Its main tasks are, determine the neighbor-points, assert the core object property of P, mark the neighbors as chain members, and record potential collisions in C. A TG starts by reading the coordinates of P into the SM, and determining the minimum bounding rectangle $\text{MBR}_\epsilon(P)$ of $N_\epsilon(P)$ allowing d-fold parallelism, where d is the dimension of the data space. Then the threads are synchronized and a loop encounters all points $Q \in \text{PointSet}$ from $\mathcal{D}$ and considers them as potential neighbors in $N_\epsilon(P)$. This is done by all threads in the TG allowing a maximum degree of intra-group parallelism. Therefore, this loop starts with the point with index *thread-ID*, and in each iteration, exactly the number of threads per TG is added to the index of the considered point Q. The further processing of Q by the procedure `processObject` depends on the core object property of P as clusters are expanded from core objects only. However, at an early stage it may be unclear if P is a core object. In any case, the distance δ between P and Q is determined. If δ exceeds ϵ nothing needs to be done. Otherwise, we have to distinguish: If P is already confirmed to be a core object, then we can immediately mark Q as a chain member, but have to perform further operations (described in the next paragraph). Otherwise, we increment (cf. Section 3.3) the counter *neighborCount*, which is individual to P atomically and take Q into a temporary list called *neighborBuffer* with at most $MinPts$ entries. We can say, that Q is taken under *quarantine* until the core status of P is really known. Whenever we mark an object as a chain member, we have to perform the following actions: Check if the object is already a member of the same or a different chain. In the first case, nothing needs to be done. Otherwise we have a *collision* of two chains. That means, we have to note that the two chain-IDs actually belong to the same cluster-ID. Figure 5 shows a typical situation of such a scenario. Note that three situations are possible: First the point was originally marked by the dark grey chain in the middle and later the white chain recognized the collision, or vice versa. The third opinion is, that the dark grey and the white chain tried to mark the collision object simultaneously. To cope with that case, we need atomic operations again. To label the neighbor points we use `atomicCAS` (compare-and-swap, cf. Section 3.3). This operation checks if a certain value is stored at the corresponding address. If the object had the status *UNPROCESSED* then `atomicCAS` replaces the old label by the current chain-ID and CUDA-DClust tries to include Q in the seed list of our chain. Otherwise, the label remains and the old value is returned to the kernel. Hence, we can decide unambiguously whether there was a collision or not.

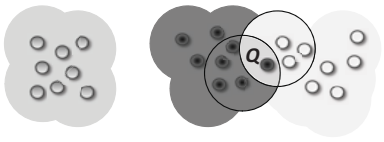


Figure 5: A collision of two chains.

5.3 Management of the Seed List

The seed list S is the central data structure of density-based clustering. Its purpose is to provide a waiting queue for those objects, which are directly density reachable from any object that has already been confirmed to be a core object of the current cluster, also named *candidates*. The core object property of the candidates in the queue will be checked later, and a confirmed core object P will be replaced by those objects, which are directly density reachable from P. S is a very simple data structure: An unpriorized queue for a set of objects with the only operations of storing and fetching an arbitrary object. In sequential density-based clustering algorithms such as DBSCAN or OPTICS, only one seed list must be maintained. In contrast, CUDA-DClust simultaneously performs cluster expansions of a moderately high number of chains, and, therefore, multiple seed lists must be managed, which potentially causes space problems. The index number of the actual seed list that is used by a TG can be determined from its TG-ID (which simulataneously serves as Chain-ID). To facilitate the readability of the code, at the beginning of the kernel ClusterExpansion, these simplifications are introduced by equivalence associations using the symbol $\equiv$. These are only meant as shortcuts, and no copying of data is needed.

For the insertion of objects into S the operation `atomicInc` (for increment) has to be used because some candidates may be simultaneously inserted by different threads of the same TG. The space in each seed list in the DM is limited by a parameter (*maxSeedLength*). In our experiments, we used 1,024 points per seed list. In case of a list overflow, new points are only marked by the current chain-ID but not inserted into the seed list. We use two different counters to keep track of the number of discarded candidates. Using a special kernel method, called *SeedRefill Kernel* we can search for candidates that have been discarded.

5.4 Load Balancing

Since a number of seeds is expanded simultaneously and a number of chains is processed in parallel, it naturally happens that one of the chains is completed before the others. In this case, we start a new chain from an unprocessed object. If no such object is available, we split a chain, i.e. one of the objects from S of a randomly selected chain is removed and we start a new chain with that candidate. Thereby, an almost constant number of threads working in the system is guaranteed, all with approximately the same workload.

5.5 The Main Program for CPU

Apart from initialization, the main program consist merely of a loop starting three different kernel methods on the GPU, until no more unprocessed data objects exist any more:

1. Transfer the data set from MM to DM;
2. Create *NumChains* new chains from randomly selected points of $\mathcal{D}$;
3. StartKernel (ClusterExpansion);
4. Transfer the states of the chains from DM to MM;
5. If necessary, StartKernel (NewSeeds);
6. If necessary, StartKernel (SeedRefill);
7. If unprocessed objects exist, continue with step (3);
8. Transfer the result (cluster-IDs) from DM to MM.

6. AN INDEX STRUCTURE TO SUPPORT CUDA-DCLUST ON GPU

The performance of CUDA-DClust can be significantly improved by a multidimensional index structure supporting the search of points in the ϵ-neighborhood of some object P $N_\epsilon(P)$. 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. 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 space $\mathcal{D}$ according to its first dimension, the second level according to its second dimension, and so on. Therefore, before starting the actual clustering method, some transformation technique should be applied that guarantees a high selectivity in the first dimensions, e.g. Principal Component Analysis. Figure 6 shows a simple, 2-dimensional example of a 2-level directory (plus the root node, which is considered as level-0), similar to [15, 16]. The fanout of each node is 8. In our experiments (cf. Section 7), we used a 3-level directory with fanout 16.

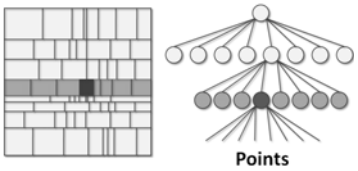


Figure 6: An index structure for the GPU.

To receive the indexed version of CUDA-DClust named CUDA-DClust*, our index structure must be constructed in a bottom-up way by fractionated sorting of the data $\mathcal{D}$ before starting the actual clustering method. First, $\mathcal{D}$ 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 arrays that 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 [9]. Since bottom up index construction is typically not very costly compared to the clustering algorithm, our method performs this preprocessing step on CPU. When transferring $\mathcal{D}$ from the MM into the DM in the initialization step of CUDA-DClust, CUDA-DClust* has to transfer the directory additionally. Compared to the size of $\mathcal{D}$, the directory is always small. The most important change in our cluster expansion kernel regards the determination of $N_\epsilon(P)$ of some given seed object P, which is done by exploiting SIMD-parallelism inside a MP. In CUDA-DClust, this was done by a set of threads inside a TG, each of which iterated over a different part of $\mathcal{D}$. In CUDA-DClust*, one of the threads iterates in a set of nested loops over those nodes of the index structure, which represent regions of $\mathcal{D}$ that intersect $N_\epsilon(P)$. 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 CUDA-DClust.

7. EXPERIMENTAL EVALUATION

To evaluate the performance of density-based clustering on GPU, we execute different experiments. First we evaluate CUDA-DClust vs. DBSCAN without index support and CUDA-DClust* vs. DBSCAN with index support w.r.t. the size of the data sets. Second we analyze the impact of the parameters $MinPts$ and ϵ. Finally we perform an evaluation w.r.t. the data dimensionality. All test data are synthetic feature vectors containing 20 randomly generated Gaussian clusters. Unless otherwise mentioned, the algorithms are parameterized with $\epsilon = 0.05$ and $MinPts = 4$. These parameters are used in an analogous way like the correspondent parameters of the sequential DBSCAN algorithm. For an extensive description of strategies for parameter selection, we refer the intrested reader to [8]. The time needed for index construction is not included in these experiments, but evaluated separately in Section 7.6. A detailed presentation of the complete runtime profile of CUDA-DClust is given in Section 7.7. 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. As a benchmark we apply a single-threaded implementation of DBSCAN on the CPU, that uses the same index structure as CUDA-DClust for all experiments with index support.

7.1 CUDA-DClust vs. DBSCAN

Figure 7(a) displays the runtime in seconds of CUDA-DClust compared with the DBSCAN implementation on the CPU without index support for various data size in logarithmic scale and the corresponding speedup factor. Due to massive parallelization, CUDA-DClust outperforms CPU without index by a large factor that is even growing with the number of points N. The speedup ranges from 10 for 30k points up to 15 for 1m points. Note that the calculation on GPU takes 40 minutes, compared to nine hours on CPU.

7.2 CUDA-DClust* vs. Indexed DBSCAN

The following experiments are performed with index support as introduced in Section 6. The runtime of CUDA-DClust* and the indexed version of DBSCAN on CPU and speedup are presented in Figure 7(b). CUDA-DClust* outperforms the benchmark again by a large factor, that is proportional to the size of the data set. In this evaluations the speedup ranges from 3.5 for 30k points to almost 15 for 2m points. A guaranteed speedup factor of at least 10 is obtained by data sets consisting of more than 250k points.

7.3 Impact of the Parameter MinPts

In these experiments we test the impact of the parameter $MinPts$ on the performance of CUDA-DClust* against the indexed version of DBSCAN on CPU. We use a synthetic data set with a fixed number of about 260k 8-dimensional points and choose $\epsilon = 0.05$. The parameter $MinPts$ is evaluated in a range from 4 to 2,048. Figure 7(c) shows that the runtime of CUDA-DClust* increases for larger $MinPts$ values. For $4 \leq MinPts \leq 512$ the speedup factor remains relatively stable between 10 and 5. Then the speedup decreases significantly. However CUDA-DClust* outperforms the implementation on CPU even for large $MinPts$-values by a factor of 2. The speedup decline can be explained by a closer look at the implementation details of CUDA-DClust* (cf. Section 5). It can be seen that the first $MinPts - 1$ found neighbors have to be buffered, because they can not be marked before it is guaranteed that the seed point is also a core point. As all threads that are executed in parallel have to share the limited shared memory, less threads can be executed in parallel if a high number of points have to be buffered in addition.

7.4 Impact of the Parameter ϵ

We also evaluated the impact of the second DBSCAN parameter ϵ on CUDA-DClust* using a synthetic data set consisting of 260k 8-dimensional points and fixed $MinPts = 4$. We test the impact of ϵ for values that range from 0.02 to 0.10 and present the results in Figure 7(d). Evidently, the impact of ϵ is negligible, as the range of the corresponding speedup factor is almost stable between 9 and 10.

7.5 Evaluation of the Dimensionality

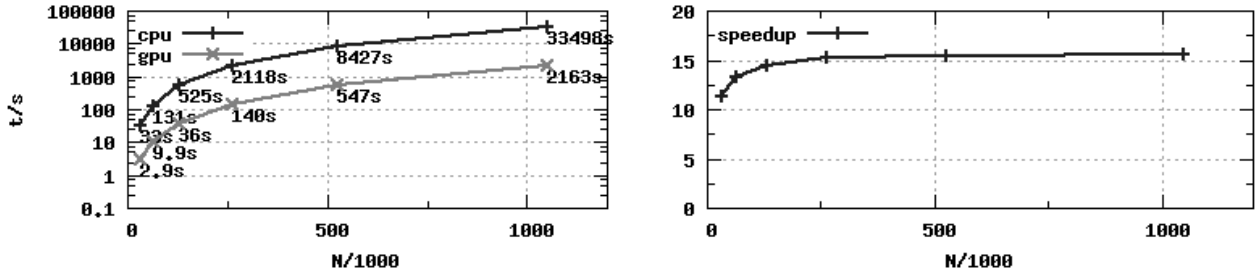
Here, we provide an evaluation w.r.t. the dimensionality of the data, ranging from 4 to 64. Again, we use our algorithm CUDA-DClust* and perform all tests on the synthetic data set consisting of 260k 8-dimensional points clustered with default parameter-settings for ϵ and $MinPts$. Figure 7 illustrates that CUDA-DClust* outperforms the benchmark by factors of about 7 to 12 depending on the dimensionality of the data. In our DBSCAN implementation the dimensionality is already known at compile time. Hence optimization techniques of the compiler have an impact on the performance of the CPU version. Obviously, also the size of the data structures for the seed points in the GPU implementation is affected by the dimensionality and hence influences the performance of CUDA-DClust*, as the data structures are stored in the shared memory. This overhead affects the number of threads that can be executed in parallel on the GPU.

7.6 Evaluation of the Index Construction

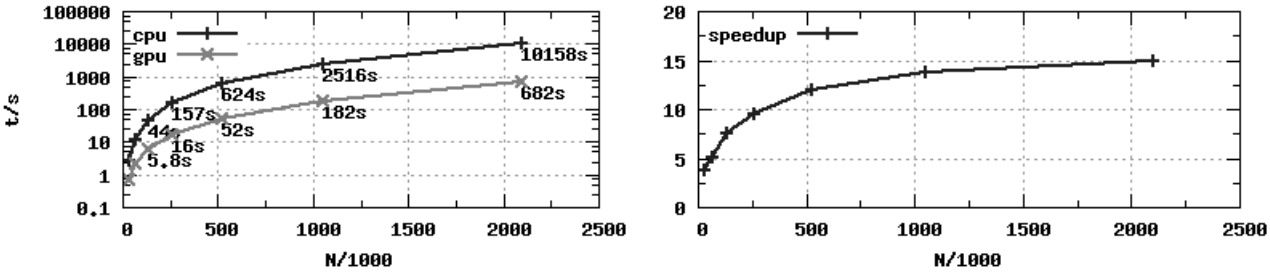
All experiments presented so far do not include the time that is needed for constructing the index structure (described in Section 6). Table 1 displays the time needed to construct the index, the runtime of CUDA-DClust and relates these two terms to each other for various sizes and dimensions of synthetic data sets. All experiments indicate that the time for building the index structure is negligible and independent of the parameter-setting by a fraction of lower than 5%. Further analysis demonstrate that larger data sets show an even smaller value of lower than 2%.

Table 1: Evaluation of the Index Construction.

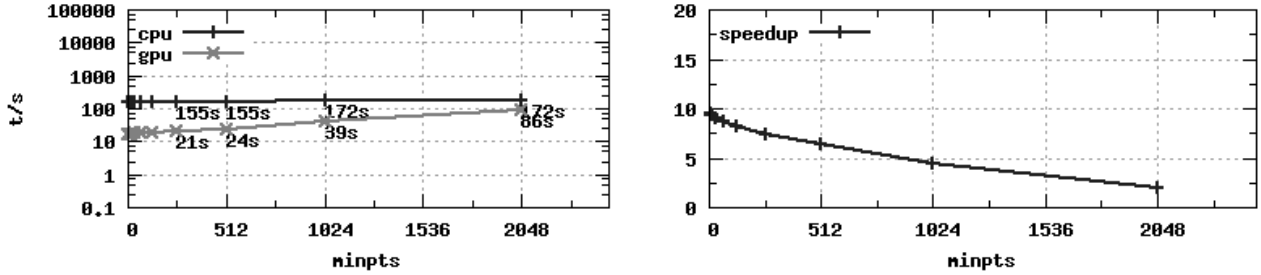
N	Dim	Index	GPU	Index / GPU
65k	8	0.1 s	2.2 s	4.5 %
130k	8	0.2 s	5.8 s	3.4 %
260k	8	0.4 s	16.5 s	2.4 %
260k	16	0.6 s	38.6 s	1.6 %
260k	32	1.0 s	45.9 s	2.2 %
260k	64	1.8 s	111.6 s	1.6 %
520k	8	0.8 s	52.1 s	1.5 %
1m	8	1.7 s	182.4 s	0.9 %
2m	8	3.6 s	681.8 s	0.5 %



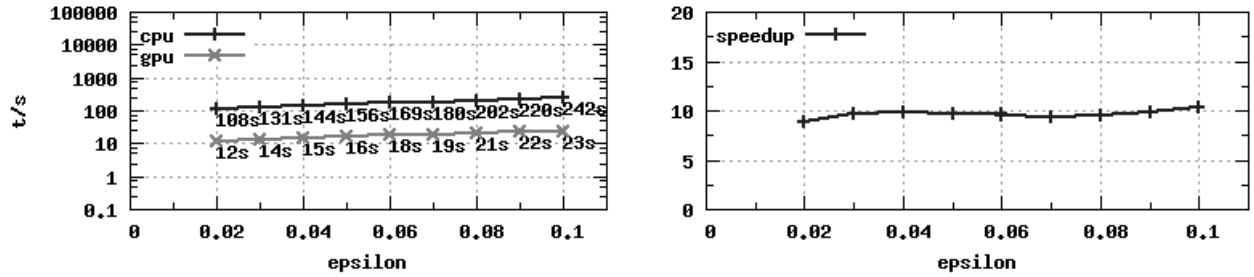
(a) Runtime and speedup of CUDA-DClust.



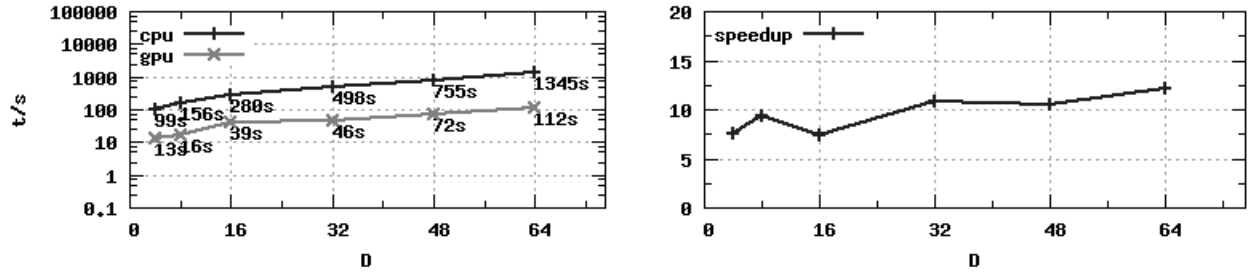
(b) Runtime and speedup of CUDA-DClust*.



(c) Impact of $MinPts$ on runtime and speedup of CUDA-DClust*.



(d) Impact of ϵ on runtime and speedup of CUDA-DClust*.



(e) Impact of the dimensionality on runtime and speedup of CUDA-DClust*.

Figure 7: Experimental evaluation of density-based clustering on CPU and GPU w.r.t. the size of the data set (Figures 7(a) and 7(b)), the parameters of DBSCAN (Figures 7(c) and 7(d)) and the dimensionality of the data (Figure 7(c)).

7.7 Runtime Profiling

In this section we present a detailed runtime profile of CUDA-DClust* concerning different parts of the algorithm that is evaluated on two different synthetic data sets. The corresponding charts are depicted in Figure 8. The left diagram illustrates the distribution of the execution time of the program parts cluster expansion kernel, new seeds kernel, collision matrix and seed refill kernel on a data set consisting of 520k 8-dimensional points and the right diagram for 4m points respectively. Thus the bulk of performance is executed by the cluster expansion kernel. The auxiliary program parts and computations on CPU only consume a small fraction of the complete runtime as desired. Remark that the CPU is not reserved during the execution of the cluster expansion kernel, that requires the main part of the computation. Hence the CPU can process multiple other jobs while the algorithm is executed by the cluster expansion kernel.

In all experiments the time needed to divide the clustering process and the time to integrate the result for both algorithms CUDA-DClust and CUDA-DClust* is negligible.

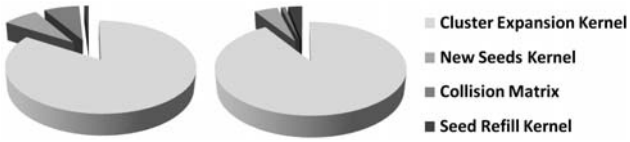


Figure 8: Runtime profile of CUDA-DClust* for two different data sets.

8. CONCLUSION

In this paper, we proposed CUDA-DClust, a novel algorithm for very efficient density-based clustering supported by the computing power of the GPU. This architecture allows for extreme parallelization at very low cost. CUDA-DClust combines several concepts to exploit the special characteristics of the GPU for clustering, as parallel cluster expansion supported by the concept of chains, parallel nearest neighbor search that can even be accelerated by the use of a hierarchical index structure, and effective load balancing among the microprocessors. Our experiments demonstrate that CUDA-DClust outperforms the algorithm DBSCAN on CPU by an order of magnitude and yields an equally accurate clustering. The impressive results can even be increased by the use of an appropriate index structure, which yields to our second proposed algorithm CUDA-DClust*. In our ongoing work, we analyze in-depth the effect of arbitrary shaped clusters and noise objects on the performances of CUDA-DClust and CUDA-DClust*. However, we do not expect significant losses, as first experiments let assume an independency of the runtime and the factors mentioned above.

9. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *SIGMOD Conference*, 1998.

[3] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In *SIGMOD Conference*, 1999.

[4] D. Arlia and M. Coppola. Experiments in Parallel Clustering with DBSCAN. In *Euro-Par*, 2001.

[5] C. Böhm, R. Noll, C. Plant, and A. Zherdin. Indexsupported Similarity Join on Graphics Processors. In *BTW*, 2009.

[6] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle. Parallel Density-Based Clustering of Complex Objects. In *PAKDD*, 2006.

[7] F. Cao, A. K. H. Tung, and A. Zhou. Scalable Clustering Using Graphics Processors. In *WAIM*, 2006.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, 1996.

[9] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD Conference*, 2006.

[10] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast Computation of Database Operations using Graphics Processors. In *SIGMOD Conference*, 2004.

[11] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[12] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Joins on Graphics Processors. In *SIGMOD*, 2008.

[13] A. Hinneburg and H.-H. Gabriel. DENCLUE 2.0: Fast Clustering Based on Kernel Density Estimation. In *IDA*, 2007.

[14] A. Hinneburg and D. A. Keim. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. In *KDD*, 1998.

[15] M. Kitsuregawa, L. Harada, and M. Takagi. Join Strategies on KD-Tree Indexed Relations. In *ICDE*, 1989.

[16] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, 1997.

[17] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*, 2008.

[18] F. Murtagh. A Survey of Recent Advances in Hierarchical Clustering Algorithms. *Comput. J.*, 26(4), 1983.

[19] R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *VLDB*, 1994.

[20] S. A. A. Shalom, M. Dash, and M. Tue. Efficient K-Means Clustering Using Accelerated Graphics Processors. In *DaWaK*, 2008.

[21] C. Van-Rijsbergen. *Information Retrieval*. Butterworths, London, England, 2nd edition, 1979.

[22] X. Xu, J. Jäger, and H.-P. Kriegel. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Min. Knowl. Discov.*, 3(3), 1999.

[23] A. M. Yip, C. H. Q. Ding, and T. F. Chan. Dynamic Cluster Formation Using Level Set Methods. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(6), 2006.