JMLR: Workshop and Conference Proceedings 36:104–119, 2014 BIGMINE 2014

Towards Optimal Execution of Density-based Clustering on Heterogeneous Hardware

Dirk Habich DIRK.HABICH@TU-DRESDEN.DE
Stefanie Gahrig STEFANIE.GAHRIG@TU-DRESDEN.DE
Wolfgang Lehner WOLFGANG.LEHNER@TU-DRESDEN.DE
Database Technology Group, Institute for System Architecture, TU Dresden, Germany

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

Data Clustering is an important and highly utilized data mining technique in various application domains. With ever increasing data volumes in the era of big data, the efficient execution of clustering algorithms is a fundamental prerequisite to gain understanding and acquire novel, previously unknown knowledge from data. To establish an efficient execution, the clustering algorithms have to be re-engineered to fully exploit the provided hardware capabilities. Shared-memory multiprocessor systems like graphics processing units (GPUs) provide extremely high parallelism combined with a high bandwidth transfer at low cost. The availability of such computing units increases with upcoming processors, where a common CPU and various computing units, like GPU, are tightly coupled using a unified shared memory hierarchy. In this paper, we consider density-based clustering for such heterogeneous systems. In particular, we optimize the configuration of *CUDA-DClust* – a density-based clustering algorithm for GPUs – and show that our configuration approach enables an efficient and deterministic execution. Our configuration approach is based on data as well as hardware properties, so that we are able to adjust the algorithm execution in both directions. In our evaluation, we show the applicability of our approach and present open challenges which have to be solved next.

Keywords: Clustering, GPU, Configuration

1. Introduction

In the era of big data, the role of data analysis techniques increases. Aside from development of specialized analysis techniques, the efficient execution is an additional major challenge. To establish an efficient execution, the analysis algorithms have to fully utilize the provided hardware capabilities. Looking at the hardware sector, architectures are more and more changing to heterogeneous systems, where a common CPU and various co-processors are coupled. Graphics Processing Units (GPUs) are a prominent example of such co-processors offering a high computing power. Generally, GPUs are shared-memory multi-processor systems providing extremely high parallelism combined with a high bandwidth transfer at low cost. To efficiently analyze massive data sets, these hardware heterogeneity (CPU and GPU) has to be exploited in an optimal way.

One of the most important data analysis or data mining techniques is data clustering, whereas clustering is described as the problem of (semi-)automatic partitioning of a set of objects into groups, so-called clusters, so that objects in the same cluster are similar, while

objects in different clusters are dissimilar. Typical application scenarios range from customer segmentation over pattern recognition to analysis of brain activities in neuro-imaging. A multitude of clustering algorithms has been developed over the years, e.g. partition-based or density-based algorithms. The advantage of density-based algorithms is that they are able to detect clusters of arbitrary shapes. In this case, clusters are considered as areas of high object density in the data space, which are separated by areas of lower density. This cluster notion has many advantages and those algorithms are rather robust against outliers and noisy data objects. DBSCAN Ester et al. (1996) is a famous representative algorithm in this domain, whereas it is a sequential algorithm with a linear space complexity $O(n)$ and a $O(n^2)$ time complexity with n as number of data objects.

To speedup DBSCAN as well as to be able to efficiently process huge numbers of multi-dimensional data objects, several approaches to parallelize DBSCAN have been introduced. On the one hand, most of the approaches have been focused on shared-nothing parallel architectures Arlia and Coppola (2001); Brecheisen et al. (2006); Dai and Lin (2012); Xu et al. (1999), where the main challenge is to coordinate clustering tasks between different processors using explicit message passings. On the other hand, Böhm et al. (2009) proposed *CUDA-DClust*, a density-based algorithm for shared-memory multi-processor systems like GPUs. Their approach is based on simultaneous expansion of multiple chains with different starting points, whereas each chain can be seen as a tentative cluster. During the parallel expansion of these chains, collisions of the chains have to be detected. Based on these collisions, the final clusters are computed. As the authors have shown, the algorithm outperforms the sequential DBSCAN algorithms by factors. However, the authors introduce new algorithm-specific configuration parameters, like e.g. number of chains, which complicates the applicability as well as it influences the performance of the algorithm.

Our Contribution

Unfortunately, the authors proposed only the core algorithm without precisely considering the specific configuration parameters. To overcome this configuration challenge, we propose an approach to compute the optimal parameter values for a given data set and hardware environment in this paper. Based on our configuration, we are able to guarantee an efficient execution with a deterministic runtime behavior. This is a major step regarding the algorithm applicability. In detail, our contributions are:

1. We review *CUDA-DClust* and present some slight extensions to optimize the algorithm from our point of view.

2. We empirically evaluate the configuration parameters with regard to their performance influence. As we are going to show, the chain configuration has a high influence on the performance and the optimal configuration depends on data characteristics.

3. Based on the result of the evaluation, we propose an approach to determine the optimal chain configuration parameters. To compute the optimal parameter values, we construct an equi-width multi-dimensional histogram. However, our approach is only efficiently useable on low-dimensional data. Nevertheless, we are going to show that with an optimal configured algorithm, a suitable speedup is achievable. Therefore, the role of this research in the context of modern heterogeneous hardware increases. In

general, our approach can be seen as first step to fully exploit the provided hardware capabilities in an optimal way.

4. In our evaluation, we are going to show that our configuration approach is applicable for a wide-range of low-dimensional data. Furthermore, we present open challenges in order to make our approach widely useable.

Outline

To motivate our work, we describe in the following section some current and upcoming hardware trends. In Section 3, we review *CUDA-DClust* Böhm et al. (2009) as a density-based clustering algorithm for a heterogeneous system consisting of CPU and GPU. This description follows an in-depth analysis of the specific configuration parameters and shows the benefits of an optimal setting. In contrast to the recommendation made in Böhm et al. (2009), a substantial speedup can be achieved with optimal parameters. In section 4, we present a first approach to determine the optimal configuration parameters for low-dimensional data. Then, we evaluate our approach in Section 5. Related work is presented in Section 6. We conclude the paper with future work and a short conclusion in Section 7.

2. Hardware Prerequisite

In this section, we review current and upcoming hardware architectures. As we are going to describe, heterogeneous hardware architectures combining a common CPU with various computing units evolve and this hardware architecture is highly interesting for large scale data analytics as required in the era of big data. Before we describe heterogeneous hardware more precisely, we review Graphical Processing Units as important co-processor.

2.1. Graphical Processing Units - GPUs

The original purpose of *Graphical Processing Units* is to perform mathematical calculations to determine the color of a specific pixel in a picture. With increasing requirements of the game and movie industries, GPUs have become a source of great computing power, which can be used for (i) high performance image processing and (ii) as a general purpose co-processor or accelerator for compute-intensive applications. The GPU consists of multi-processors that are optimized for Single Instruction-Multiple Data (SIMD) execution, whereas each multi-processor consists of multiple stream processors or cores. Within a group of threads, one instruction is executed simultaneously on multiple data objects. This reduces the need of individual control logic and chip space can be saved by sharing control structures between execution units (ALUs). This results in a higher density of ALUs leading to GPUs with over one thousand cores. In addition to the raw compute performance, GPUs differ from CPUs regarding their memory hierarchy. Usually, the GPU has no caches but rather fast registers per work thread, shared memory for a work-group and global shared-memory up to several gigabytes, accessible by all threads.

2.2. Heterogeneous Hardware Systems

Heterogeneous hardware systems combine a common CPU with one or multiple computing units, whereas each unit has its own specific characteristic. Well-known combinations are CPU and GPU or CPU and FPGA. Generally, these combinations can be classified into (i) loosely-coupled and (ii) tightly-coupled approaches.

In the first case, the different computing units do not share parts of their memory hierarchy or a common main memory, so that each data object has to be explicitly transferred between the units. The loosely-coupled combination of different computing units is well-known. In the second case, a shared main memory is available, which is directly accessible by all computing units, more or less. In the past, this was only done in a homogeneous way by placing multiple CPU cores on one chip to improve parallelism. Due to smaller form factors, lower power consumption guidelines, and the demand for system on chips (SoC), hardware vendors integrate different processors and accelerators into one chip. The most common examples for tightly-coupled heterogeneity are integrated graphic processing units (further denoted as iGPU). The iGPU is fully integrated and shares the main memory with the rest of the system. Examples of this kind of heterogeneous integration are Intels®core processors for desktop and mobile systems (starting from the 2nd generation) and the AMD®Fusion processors (also called Accelerated Procession Unit (APU)). There are different integration approaches by the manufacturers. For example in AMDs' Trinity architecture, the iGPU and the CPU cores can access every part of physical main memory. However, the memory is divided in an iGPU dedicated part and a part managed by the operating system. The partition sizes can be set on boot time. The iGPU and the CPU can access both parts of the memory; however, both compute units have faster transfer rates when accessing their own dedicated memory. It is possible for the CPU and the iGPU to work on the same data stored in one of the memory parts. On the host side of the main memory, there are two options to store data, i.e., as cached host memory and as uncached host memory. If the memory is cached, accesses from the GPU have to snoop the CPU caches, resulting in a loss of memory bandwidth. Uncached host memory can be accessed faster through an extra bus system, leading to higher bandwidths. At the moment, iGPUs are only integrated in desktop and mobile systems. There are also plans to equip micro servers with heterogeneous CPU/iGPU processors. In the future, processors with similar properties are likely to be used in large server environments, providing high performance for data- and compute-intensive applications.

2.3. Evaluation

To show the benefits and differences of loosely- and tightly-coupled heterogeneous hardware (CPU and GPU) for applications processing massive data such as data clustering, we evaluated the memory access bandwidth.

In the first part of this evaluation, we placed a predefined data block in the GPU dedicated memory. To measure the transfer bandwidth from GPU to CPU, we called a *map* and an *unmap* command on the CPU. The *map* command maps the specified data block into the main memory, where the CPU can modify the data. If the data is not residing in the CPU's main memory, mapping results in a copy operation to the main memory, while *unmap* releases the main memory and copies the data back. As expected and depicted in

	Loosely Env. Local Mem	Tightly Env. Local Mem	Tightly Env. CPU Mem
Avail. Mem.	1 GB	2 GB	30 GB
Map	1.47 GB/s	5.18 GB/s	3092 GB/s
Unmap	2.53 GB/s	5.09 GB/s	240 GB/s

Table 1: Memory Transfer between GPU and CPU.[1]

Table 1, the tightly-coupled approach has as better mapping bandwidth for its dedicated GPU memory, because data is not transferred via the PCIe bus as in the loosely-coupled approach. This is clearly beneficial for processing or analyzing large data sets, where lots of data has to be processed or can now be processed by different computing units. Additionally, when the memory already resides in the CPU main memory part as it is possible in tightly-coupled systems, there is no copy operation needed and the memory bandwidth increases massively (on our system, we achieved a *theoretical* bandwidth of 3092GB/s for the *map* and 240GB/s for the *unmap* operation[1]).

The second part of this evaluation focusses on bandwidth for read and write operations on the GPU, where the GPU reads and writes data to directly accessible memory blocks. In the loosely-coupled environment, the GPU can only access its local on-device memory, while in the tightly-coupled environment accessing the CPU's main memory is a second possibility. This time, the bandwidth of the loosely-coupled environment is much better than in the tightly-coupled environment as illustrated in Table 2. The reasons are specific optimizations for memory access and the usage of faster DRAM in our discrete GPU. However, the tightly-coupled environment shows two important properties:

1. There is no significant bandwidth difference in reading data from iGPU local memory or CPU memory.

2. There is a difference in write access times between dedicated and main memory. The relatively slow write access to CPU's main memory is caused by cache snooping to the CPU caches for coherent memory accesses.

	Loosely Env. Local Mem	Tightly Env. Local Mem	Tightly Env. CPU Mem
Avail. Mem.	1 GB	2 GB	30 GB
Kernel Read	35.94 GB/s	25.29 GB/s	24.6 GB/s
Kernel Write	57.99 GB/s	21.76 GB/s	6.57 GB/s

Table 2: GPU Memory Access Bandwidth.[1]

1. Measurements were taken with AMD OpenCL Sample BufferBandwidth using default parameters (0% cache-hit rate). Map/unmap function call times were used for calculation. If no data is copied, the bandwidth is theoretical.

Based on our bandwidth evaluation, we are able to conclude that dedicated memory accesses are slower for the tightly-coupled CPU/GPU than for the loosely-coupled variant. However, the data transfer times to the local memory are orders of magnitude faster if not obsolete at all. Looking at the still increasing CPU main memory sizes, it should be clear that only this memory is sufficient to support the analysis of large data sets. These results motivate our work to investigate data clustering algorithms in a heterogeneous hardware environment consisting of CPU and GPU. In this case, recent and upcoming hardware will efficiently support the processing of data residing in the CPU main memory with different computing units. However, the algorithms have to fully utilize the provided hardware capabilities, whereas several steps are necessary (i) decomposing algorithms, so that algorithm fragments can be executed on different computing units, (ii) design algorithm fragments being efficiently executable on a specific computing unit and (iii) executing the whole algorithm with an optimal placement as well as optimal configuration setting on heterogeneous hardware as demonstrated in Karnagel et al. (2013).

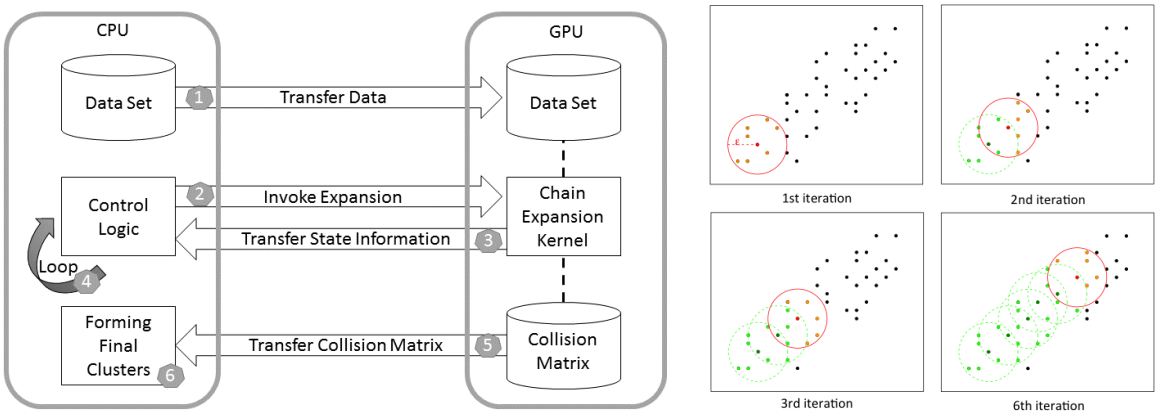
3. Density-based Clustering on GPU

The most well-known algorithm for density-based clustering is DBSCAN Ester et al. (1996). In this approach, clusters are defined as dense regions separated by regions of lower density. The algorithm uses two parameters, ε and $minPts$ to define a density threshold. With ε, a neighborhood is defined around each data point p. If this neighborhood contains at least $minPts$ additional data points, p is considered as a member of a dense area and is named *core-object*. Sets of core-objects with overlapping ε-neighborhoods are merged to create clusters. This is done recursively, that means, if p is a core-object each member of its ε-neighborhood is checked for the density condition. This way, DBSCAN iteratively expands a single cluster starting by a randomly chosen core-object. After that, the algorithm continues with an arbitrarily unprocessed data point until all data points have been considered.

Parallelization of DBSCAN is not straightforward as described in several papers Arlia and Coppola (2001); Brecheisen et al. (2006); Dai and Lin (2012); Xu et al. (1999), which is however required to efficiently execute the algorithm on recent multi-core or heterogeneous hardware. A GPU-specific version of DBSCAN has been proposed by Böhm et al. (2009). The so-called *CUDA-DClust* algorithm is briefly described next, whereas we present our interpretation of the algorithm due to an imprecise description in Böhm et al. (2009). Nevertheless, our interpretation works in a similar way. Furthermore, *CUDA-DClust* executes parts of the algorithm on GPU as well as CPU, which is highly interesting for heterogeneous hardware. Then, we present some issues and challenges with regard to the application and execution of this specific algorithm.

3.1. CUDA-DClust Algorithm

Generally, *CUDA-DClust* Böhm et al. (2009) relies on the basic process of DBSCAN, but refines it with some kind of speculation for parallelization. The basic concept used for parallel and speculative processing is the concept of so-called *chains*, whereas chains are *tentative clusters*. Instead of iteratively expanding a single cluster from a core-object as done in DBSCAN recursively, *CUDA-DClust* iteratively expands multiple chains from multiple arbitrary starting data points in parallel. While creating multiple chains in parallel,



(a) Algorithm Process (b) Chain Expansion on GPU

Figure 1: CUDA-DClust algorithm.

collisions between various chains emerge, because the same points are considered in different chains. These collisions are desired and required to determine the final clusters out of the chains at the last step. The collision detection is conducted during the chain expansion.

The whole *CUDA-DClust* algorithm follows a fixed roadmap, whereas parts of the algorithm are either executed on CPU or GPU. Figure 1(a) depicts the whole algorithm roadmap. In our interpretation, the algorithm consists of six steps. In step one, all data points have to be transferred from the CPU main memory to the GPU device memory. Afterwards, – second step – the GPU with its high number of parallel computing units is responsible to expand multiple chains simultaneously, whereas each single chain is iteratively expanded. In contrast to DBSCAN, in this expansion only one point within the ε-neighborhood of a core-object is considered as illustrated in Figure 1(b). Aside from expanding these tentative clusters, collisions between the chains are tracked using a collision matrix. The size of this collision matrix equals the square of the number of chains. After expanding the multiple chains on GPU, status information about the data point processing is transferred from the GPU to the CPU in step three. Based on the retrieved status information, a post-processing for the expanded chains is initiated. If there exist still unprocessed points in the set of data points, step two and three are executed again with new chains being expanded on the GPU. The loop – step four – is repeated until no more unprocessed data points exist. With every loop, the total number of chains increases by the number of chains. In the same way, the collision matrix also increases. To process all data points, an appropriate number of chains has to be expanded. If all points are processed, the final density-based clusters out of the chains are determined on the CPU (step six) by processing the collision matrix, which has to be transferred from GPU to CPU (step five). The final clusters are determined using breadth-first search.

To execute the algorithm on unknown data sets, several parameter values have to be specified. Aside from ε and $MinPts$, *CUDA-DClust* introduces three additional configuration parameters having a high influence on the performance of the algorithm. Those three parameters are directly connected to the chain approach and can be described as follows:

Number of chains: Using this parameter, the user has to specify how many parallel chains will be expanded on the GPU in one loop. As previously described, multiple loops are possible and in each loop the same number of chains is expanded. Then, the total number of chains is:

```
total_number_of_chains = number_of_chains * number_of_loops.
```

Aside from specifying the parallelization, this parameter directly influences the size of the collision matrix and therefore, the time needed to determine the final clusters out of the tentative clusters in step six. In Böhm et al. (2009), the authors recommended to set the number of chains to the number of available multi-processors of the GPU.

Starting points of the chains: Aside from the specification of the number of chains, each chain requires a starting data point. In Böhm et al. (2009), the authors suggested to choose the starting points randomly out the set of data points.

Size of the chains: The last influencing parameter is the size of the chains. Using this parameter, the user can specify how large each tentative cluster could be. In Böhm et al. (2009), the authors use a fixed size of 1024 points. That means, each chain could not be larger than 1024 points. In our interpretation, we are able to adjust the size of the chains by varying the number of iterations executed on the GPU for expansion.

Next, we present our results of an empirical evaluation of these *CUDA-DClust*-specific configuration parameters and show that the recommendation made in Böhm et al. (2009) is suboptimal. Based on these results, we propose a first approach to determine the optimal parameters in Section 4.

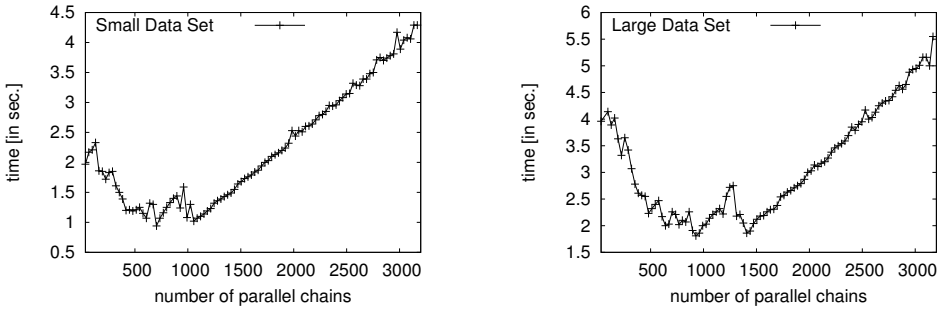
3.2. CUDA-DClust: Issues and Challenges

As shown in their evaluation Böhm et al. (2009), *CUDA-DClust* outperforms the classical DBSCAN implementation on CPU by factors. This speedup is possible due to the massive parallelization using chains and the appropriate hardware supporting the parallelization by GPU. Nevertheless, the application of the algorithm on an unknown set of data points is difficult. Aside from specifying values for ε and $minPts$, the *CUDA-DClust*-specific configuration parameters have to be set. To determine the influence of the specific parameters, we have empirically evaluated the parameters on different data sets in a loosely-coupled hardware environment.

Number of Chains

The number of chains being simultaneously expanded on GPU in each loop, is an important parameter of the algorithm, because it has a high influence on the execution time as shown next. In our empirical evaluation, we used a NVIDIA graphics card with 32 multi-processors with in summa 256 cores and $2GB$ device memory. The starting points are randomly chosen and the size of the chains is set to 1024 as suggested in Böhm et al. (2009).

Figure 2 shows the resulting runtimes – including all data transfer times – for varying numbers of chains, whereas our evaluation starts at 32 chains as recommended. The figures

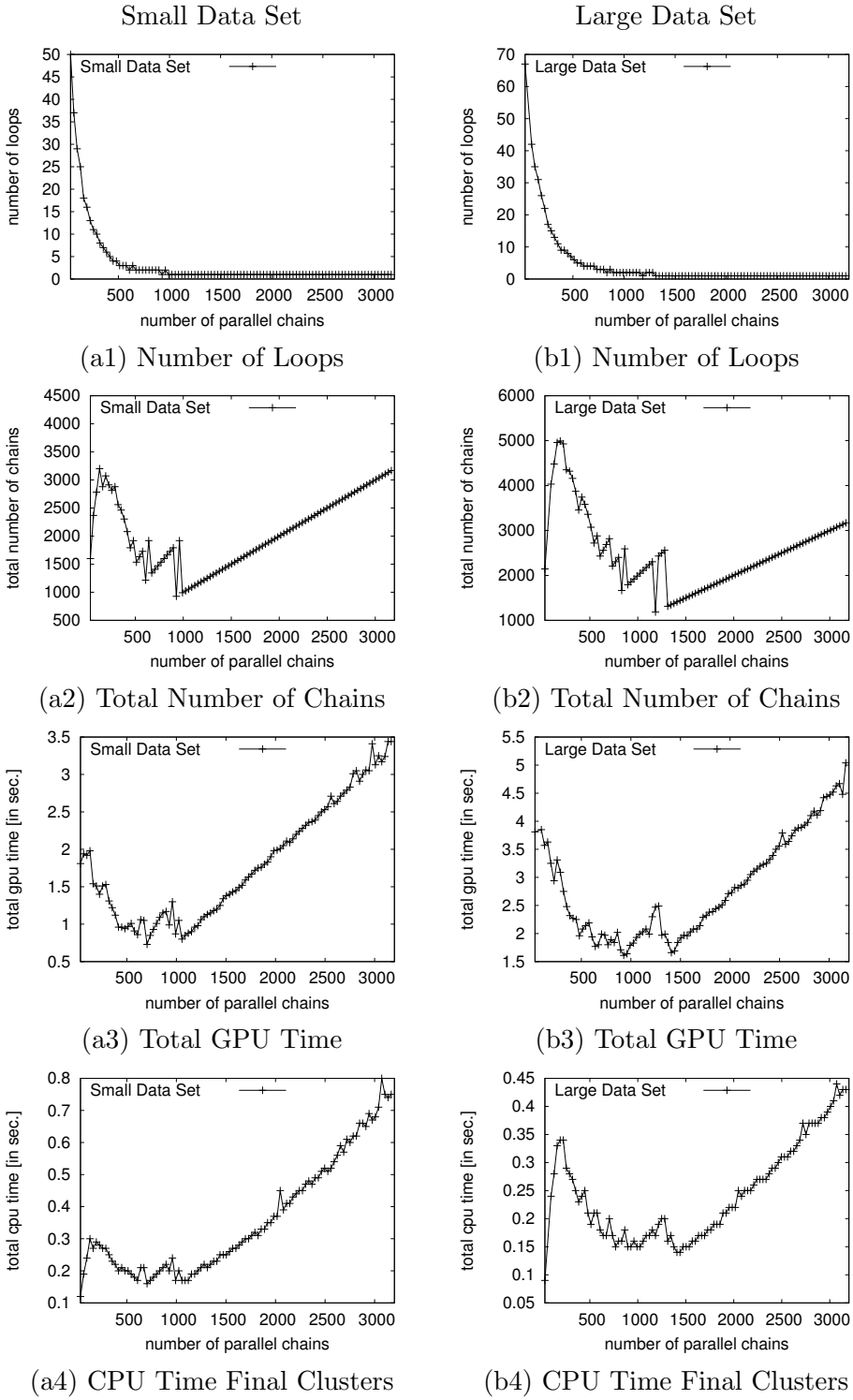


(a) Data Set with 25,000 Data Points (b) Data Set with 50,000 Data Points

Figure 2: *CUDA-DClust* runtime behavior for two different 2-dimensional data sets with varying number of chains between 32 and 3200.

depict the runtime behavior for two different 2-dimensional data sets, whereas the data sets differ in number of data points and number of clusters. Both are generated, so that we are able to cluster the data sets with the same ε and $MinPts$ values. As we can see, the runtime curves for both data sets are similar and the minimum runtime is not at the suggested configuration of 32 chains. The reason can be described as follows: Each algorithm execution starts with a fixed number of chains, e.g. 32. The 32 tentative clusters are simultaneously expanded on the GPU. However, the 32 chains do not cover all data points, therefore the loop – step four in Figure 1 – is executed multiple times resulting in expanding new chains, whereas again the same number of new chains is created. In each loop, we create as many chains as predefined by the parameter and the starting points are randomly chosen from the still unprocessed data points. Figures 3(a1) and (b1) illustrate the number of loops which are executed. With increasing numbers of chains, the number of loops decreases until only one loop is necessary to cover all data points. As to be expected, this behavior is due to the fact, that with increasing number of chains, a higher number of data points can be assigned to tentative clusters and less loops are necessary. However, in every case, a different total number of chains is expanded and therefore, the runtime sometimes increases or decreases. Figures 3 (a2) and (b2) depict the total number of chains which are evaluated. As we can see, the number of points with minimum runtime (compare with Figure 2) for both data sets corresponds to the minimum total number of chains. For the small data set with 25,000 points, the optimal parameter for the number of chains is 992 and for the large data set with 50,000 points 1312. In these cases, the algorithm simultaneously expands 992 or 1312 chains at once and the whole algorithm requires only one loop. In any other case, the algorithm expands more chains resulting in higher runtimes. After the optimal points, the runtime increases linear, because only one loop is necessary with increasing the number of chains. While Figures 3(a3) and (b3) show the total runtime on GPU, Figures 3(a4) and (b4) show the runtime for the determination of the final clusters on CPU. At the optimal number of chains, the runtimes on CPU and GPU are at the minimum, whereas most of the time is used for GPU computation.

Summarizing, the recommendation of Böhm et al. (2009) is suboptimal as previously presented and shown in Table 3. Using an optimal configuration, a speedup of approximately



(a1) Number of Loops (b1) Number of Loops

(a2) Total Number of Chains (b2) Total Number of Chains

(a3) Total GPU Time (b3) Total GPU Time

(a4) CPU Time Final Clusters (b4) CPU Time Final Clusters

Figure 3: Detailed Runtime Behavior Analysis.

Data Set	Runtime 32 Chains	Runtime Optimal Chains	Number of Optimal Chains	Speedup
Small	1.97 sec.	1.08 sec.	992	1.823
Large	3.96 sec.	2.18 sec.	1312	1.816

Table 3: Comparing runtime behavior between recommendation of 32 chains Böhm et al. (2009) and optimal number of chains.

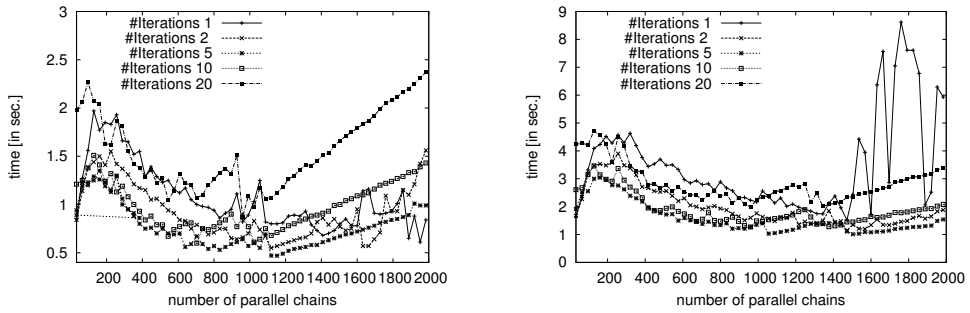
2 is achievable for the used data sets. Furthermore, an optimal value for this configuration parameter exist and the value is much larger than the number of multi-processors of the GPU. From a GPU perspective, a higher number of parallel threads than available physical execution units is beneficial, because this overload can then be used to schedule threads for execution, while others wait for a memory transfer. Thus, the high latency of memory access can be hidden and the whole execution is faster. Furthermore, the parameter does not only influence the GPU runtime but also the runtime on the CPU. Therefore, this parameter has a high influence on the overall algorithm performance. Moreover, the optimal number of chains is data dependent and has to be determined separately for each data set.

Starting Point of Chains

The second *CUDA-DClust*-specific parameter influences the choice of the chain starting points. Böhm et al. (2009) suggest to randomly choose the starting points. This randomness has a high influence on the runtime behavior of the algorithm, whereas the randomness introduces some runtime fluctuations. For less numbers of chains, the range between minimum and maximum runtimes is substantial. With higher numbers of chains, the fluctuation closes more and more. The reason can be described as follows. For less number of chains, the randomness has a high influence on the number of loops (step four of the algorithm). In the best case, the starting points are well-chosen, so that a minimal number of loops is necessary to cover all data points. In the worst case, the starting points are always placed badly, so that a high number of loops is executed resulting in a high total number of chains being expanded on GPU. The influence of this randomization decreases with increasing number of chains. To efficiently execute *CUDA-DClust*, a high number of chains is necessary and the starting points can be chosen randomly. Furthermore, the combination guarantees a deterministic runtime behavior with less time fluctuations.

Size of Chains

Using the third *CUDA-DClust*-specific configuration parameter, we are able to influence the size of the chains. In our algorithm interpretation and implementation, we expand chains using iterations as illustrated in Figure 1(b). In the first iteration, we consider only all reachable points within the ε-neighborhood of the starting point. In the second iteration, the ε-neighborhood of another point within the ε-neighborhood of the first iteration is evaluated. That means, with each iteration, we expand a chain by an ε-neighborhood. This iteration processing is done sequentially on GPU. Figure 4 shows the runtime curves for different number of chains and different number of iterations. As expected, the number of



(a) Small Data Set with 25,000 points (b) Large Data Set with 50,000 points

Figure 4: Evaluating the influence of chain size to the runtime behavior.

iterations has an influence on the runtime behavior. With less numbers of iterations, the sequential part of GPU processing is reduced, but the number of loops (step four) increases and therefore, the total number of chains increases. This behavior is correct: with small chains a higher number of chains is essential to cover all data points. With larger chain sizes – more iterations –, fewer loops and less chains are necessary in general. In this case, we need to find a tradeoff between chains' sizes and numbers of chains. In our evaluation, we achieved the best runtimes using 5 iterations as depicted in Table 4 for both data sets.

Number of Iterations	Small Data Number of Chains	Small Data Runtime	Large Data Number of Chains	Large Data Runtime
1	1888	0.65 sec.	2048	2.03 sec.
2	1504	0.77 sec.	1984	1.88 sec.
5	992	0.57 sec.	1312	1.37 sec.
10	992	0.77 sec.	1280	1.63 sec.
20	960	0.87 sec.	1216	2.71 sec.

Table 4: Influence analysis of the size of chains with regard to the optimal number of chains.

4. Determining Optimal Chain Configuration

To efficiently determine density-based clusters, the optimal configuration setting for *CUDA-DClust* has to be determined. From the previous section, we are able to draw the following implications for optimality:

- The GPU should expand as many chains as possible using one kernel invocation. In this case, step four of the *CUDA-DClust* algorithm – see Figure 1 – is prevented and this sequential portion of the *CUDA-DClust* algorithm is reduced to a minimum.
- The GPU should expand as many chains as needed for each data set using one kernel invocation. In this case, the size of the collision matrix is reduced to a minimum and the processing time of this collision matrix to identify the final clusters is minimized.

- Besides the number of chains, the starting points should be chosen in an appropriate way, so that in the expansion of the chains, all data points can be covered. With a large number of chains which are simultaneously expanded, the starting points can be chosen randomly.

To determine the optimal parameter setting, we introduce a pre-processing step for *CUDA-DClust*. In this additional step, we compute a multi-dimensional histogram to get specific information about the data set. From a conceptual point of view, the multi-dimensional histogram represents a coarse-grained clustering result of a predefined size. That means, non-empty histogram buckets can be seen as dense regions, while empty histogram buckets separate dense regions and do not contain any cluster. From a *CUDA-DClust* algorithm perspective, we can use the provided histogram information in that way, that we start chains only in non-empty histogram buckets, whereas the starting point of each chain is randomly chosen within the bucket. In this case, we are able to guarantee to distribute the chains over the whole data set, whereas the starting points are randomly selected in a smaller range.

The effectiveness of the multi-dimensional histogram for *CUDA-DClust* depends on the histogram size. In order to determine the right size, we have to take a deeper look at *CUDA-DClust* in combination with our empirical evaluation results. Generally, *CUDA-DClust* speculatively expands tentative clusters or chains from a selected starting point. On the one hand, the size of the chains should be small. On the other hand, the chains have to intersect, so that collisions arise. These collisions are used in the last step to determine the final clusters. Moreover, to expand a chain, we need an ε-neighborhood with $minPts$ data points in that region, whereas ε and $minPts$ have to be pre-determined for algorithm execution. In this way, we could determine a equi-width multi-dimensional histogram with a width of ε. Using this ε equi-width histogram, we could get the regions, where we have to start chains for expansion. Nevertheless, this histogram size is too fine-grained. To enable a slight growing of chains, we suggest to compute a $(2 * \varepsilon)$ equi-width multi-dimensional histogram. Based on this $(2 * \varepsilon)$ equi-width histogram, we determine the *CUDA-DClust*-specific parameter setting as follows:

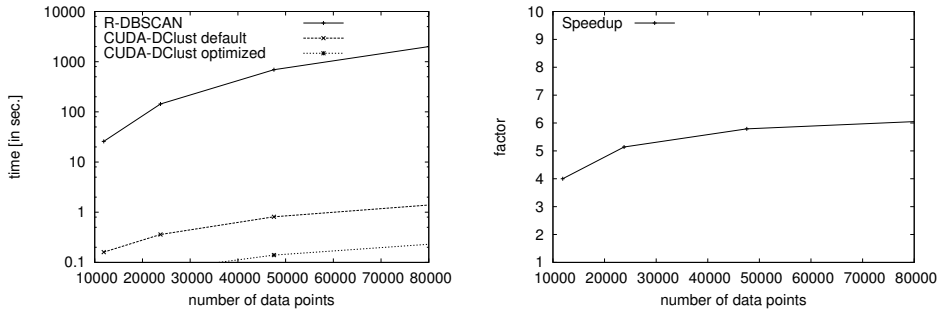
1. The number of chains corresponds to the number of non-empty histogram buckets. A further restriction to buckets with a minimum of $minPts$ entries is not possible. Even the core-object property is not satisfied in this bucket, a chain could start from here using the neighbor buckets, if the starting point locates on the border. Therefore, we consider all non-empty histogram buckets.

2. The starting points of the chains are randomly selected within the non-empty histogram buckets.

3. The size of the chains is restricted to 5 iterations. As evaluated in the previous section, we get the best runtime behavior with 5 iterations and the chains slightly growths for intersection.

5. Evaluation

In our evaluation, we use again synthetically generated data sets containing of 2-dimensional data points, whereas we varied the number of data points this time. In all experiments, we



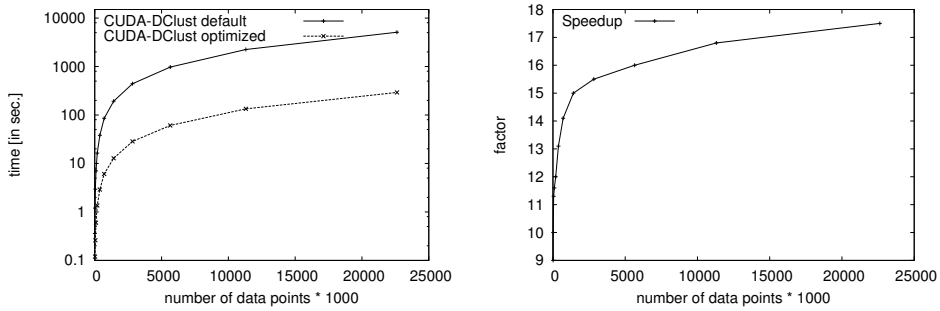
(a) Runtime (b) Speedup

Figure 5: Evaluation on loosely-coupled heterogenous system.

used the same values for ε and $minPts$. We compared a CPU-based DBSCAN of the *R* project (http://www.r-project.org), *CUDA-DClust* with the suggested configuration of Böhm et al. (2009) and our optimized *CUDA-DClust* approach. Our first experiment was done on a loosely-coupled NVIDIA graphics card *GeForce GTX 295* with 30 multiprocessors resulting in 240 cores and $1GB$ of device memory. The CPU was a Intel Core i7 with 2.7GHz and 6GB main memory. As we can see in Figure 5(a), our *CUDA-DClust* algorithm with an optimized configuration outperforms the CPU DBSCAN implementation as well as the original GPU-accelerated algorithm with standard configuration. In this experiment, the runtimes include all time for pre-processing and data transfer, whereas the data transfer is done via PCIe. The speedup as depicted in Figure 5(b) of our optimized configured *CUDA-DClust* compared to the standard *CUDA-DClust* increases with increasing data size.

We conducted the same experiment also on a tightly-coupled CPU/GPU system with a Intel Core i5 CPU with 1.3 GHz, a Intel HD Graphics 5000 (40 parallel compute units) and 4GB of main memory. *CUDA-DClust* is implemented using OpenCL. In this second experiment, all data resides in the CPU main memory, so that CPU and GPU access the same data objects. In this case, we do not need to explicitly transfer data between CPU and GPU. The results are depicted in Figure 6. As illustrated, we received the same behavior, whereas we varied the data size from $10,000$ to more than 20 million 2-dimensional data points. The speedup increases from 9 to 17 with increasing data size. In this case, the optimal configured *CUDA-DClust* reduces the processing activities (number of chains, chain size) compared to the standard approach, so that the processing time on GPU and CPU is massively reduced.

In our approach, the pre-processing is executed on CPU. For low-dimensional data, the pre-processing time is negligible. Nevertheless, the pre-processing step is our major bottleneck. Assuming that we are able to compute for each n-dimensional data set our equi-width histogram, then we can show that our determined parameter setting outperforms the original recommendation and we are able to achieve a substantial speedup. Unfortunately, the computation of our equi-width histogram for high-dimensional data is of complex nature and the effort increases exponentially with every dimension Hinneburg et al. (2003). Our next challenge is to develop an approach for high-dimensional data. As we have shown for low-dimensional data, we are able to substantially speedup density-based clustering.



(a) Runtime (b) Speedup

Figure 6: Evaluation on tightly-coupled heterogenous system.

6. Related Work

Various papers have been published with the focus to parallelize density-based clustering Arlia and Coppola (2001); Brecheisen et al. (2006); Dai and Lin (2012); Xu et al. (1999), whereas these papers use shared nothing architectures. The main advantage of shared nothing architectures is that they can be scaled up to a high number of computers. In recent years, algorithms for shared memory architectures in particular for GPUs have been also investigated. *CUDA-DClust* Böhm et al. (2009) is one example for density-based clustering on GPU. Andrade et al. (2013) have described a further approach for GPU accelerated density-based clustering. Their strategy is quite simple and is divided into two steps. The first step is to construct a graph that represents the data, where each object is represented as a node in the graph and an edge is created between two objects when the similarity measure between them is less than or equal to a threshold defined as an input parameter. After the graph construction, the second step is to identify the clusters, using a traditional breadth-first search (BFS) to traverse the graph created in the first step. Both steps are executed on GPU. Besides density-based clustering, GPU accelerated processing has been investigated in Böhm et al. (2009); Shalom et al. (2008). To best of our knowledge, no paper has done an exhaustive evaluation of the specific configuration and has presented an approach to determine the optimal configuration for execution.

7. Summary and Future Work

In this paper, we considered density-based clustering on heterogeneous hardware. In particular, we used the already published *CUDA-DClust* algorithm Böhm et al. (2009) and proposed a first approach to determine the optimal parameter setting for this algorithm. As we have demonstrated, we can achieve a substantial performance speedup using an optimal configuration on loosely- as well as tightly-coupled heterogeneous hardware. However, our approach with an equi-width histogram is only useable for low-dimensional data. Our next research step focusses on relaxing this limitation and on developing appropriate pre-processing techniques to determine an optimal parameter setting for high-dimensional data. From our point of view, this research direction offers a lot of potential to use the available and upcoming hardware resources in a more optimal way. Furthermore, we want

to investigate further algorithms in a more global context using our modular language for data clustering algorithms Hahmann et al. (2014).

Acknowledgments

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and by the European Union together with the Free State of Saxony through the ESF young researcher group "IMData" 100098198. Parts of the evaluation hardware were generously provided by Dresden CUDA Center of Excellence.

References

Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science*, 18:369–378, January 2013.

Domenica Arlia and Massimo Coppola. Experiments in parallel clustering with DBSCAN. In *Euro-Par 2001 Parallel Processing*, pages 326–331, 2001.

Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *CIKM '09*, page 661, 2009.

Stefan Brecheisen, HP Kriegel, and Martin Pfeifle. Parallel density-based clustering of complex objects. In *PAKDD*, pages 179–188, 2006.

Bi-Ru Dai and I-Chang Lin. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *CLOUD 2012*, pages 59–66, June 2012.

Martin Ester, HP Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

Martin Hahmann, Dirk Habich, and Wolfgang Lehner. Modular data clustering - algorithm design beyond mapreduce. In *EDBT/ICDT Workshops*, pages 50–59, 2014.

Alexander Hinneburg, Wolfgang Lehner, and Dirk Habich. Combi-operator: Database support for data mining applications. In *VLDB*, pages 429–439, 2003.

Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. The HELLS-join: a heterogeneous stream join for extremely large windows. In *DaMoN 2013*, page 2, 2013.

SAA Shalom, Manoranjan Dash, and Minh Tue. Efficient k-means clustering using accelerated graphics processors. In *DaWaK 2008*, pages 166–175, 2008.

Xiaowei Xu, J Jäger, and HP Kriegel. A fast parallel clustering algorithm for large spatial databases. *Data Min. Knowl. Discov.*, 3(3):263–290, 1999.