LETTER

Fast Density-Based Clustering Using Graphics Processing Units*

Woong-Kee LOH[†], **Yang-Sae MOON**[††a)], ***Members***, ***and*** **Young-Ho PARK**[†††b)], ***Nonmember***

SUMMARY Due to the recent technical advances, GPUs are used for general applications as well as screen display. Many research results have been proposed to the performance of previous CPU-based algorithms by a few hundred times using the GPUs. In this paper, we propose a density-based clustering algorithm called *GSCAN*, which reduces the number of unnecessary distance computations using a grid structure. As a result of our experiments, GSCAN outperformed CUDA-DClust [2] and DBSCAN [3] by up to 13.9 and 32.6 times, respectively.

key words: *density-based clustering, graphics processing units, grid structure*

1. Introduction

A *Graphics Processing Unit* (*GPU*) is a processor that generates continuous image frames to output on a display device. Due to the recent advances in GPU technology, many approaches have been made to harness the high performance of GPU for general applications as well as screen display. A GPU consists of a number of *cores*; each core executes general instructions like an Arithmetic & Logic Unit (ALU) in a CPU, and thus a GPU is often regarded as a massively parallel processor. The results of many research efforts have improved the performance of previous CPU-based algorithms by a few hundred times using the GPUs [2], [6], [7].

Clustering is a task that assigns each object in a dataset to one of multiple groups or *clusters* so that the objects in the same cluster are more similar to each other than those in different clusters. Density-based clustering forms the clusters of densely gathering objects separated by sparse regions. It has the advantage that it can find the clusters of arbitrary shapes and filter out noise objects easily. Most widely referenced density-based clustering algorithms are DBSCAN [3], OPTICS [1], and DENCLUE [4]. CUDA-DClust proposed by Böhm et al. [2] improved the performance of DBSCAN by more than 15 times using a GPU.

In this paper, we propose a density-based clustering algorithm called *GSCAN* (*GPU-based DBSCAN*). GSCAN reduces the number of unnecessary distance computations using a grid structure. As a result of our experiments, GSCAN outperformed CUDA-DClust and DBSCAN by up to 13.9 and 32.6 times, respectively.

2. Graphics Processing Units

In this section, we briefly introduce Nvidia GPU [5]. Figure 1 shows a simplified architecture of Nvidia GPU. A GPU chip contains a few *multi-processors* (*MPs*) and each MP contains many *stream processors* (*SPs*). An SP is also called a *core*, which executes general instructions like an ALU in a CPU. Each MP performs a separate task different from the other MPs; in contrast, all the SPs in an MP must execute the same instruction simultaneously on possibly different data. Most recent GPUs contain hundreds or thousands of SPs. Each MP also contains *shared memory* (*SM*) which stores the common data shared by the SPs in the MP. The *device memory* (*DM*), which is also called *global memory*, outside the GPU chip is accessible from all the SPs in all the MPs.

Nvidia provides *Compute Unified Device Architecture* (*CUDA*) toolkit to support GPU application development [5]. A CUDA program is a simple variation of C program and consists of functions to be executed on both CPU and GPU. The functions executed in the threads on GPU are called *kernel functions*, and the CPU functions are called *host functions*. Each SP executes a thread for a same kernel function, and thus the kernel function is executed in parallel in a massive number of concurrent threads on a GPU.

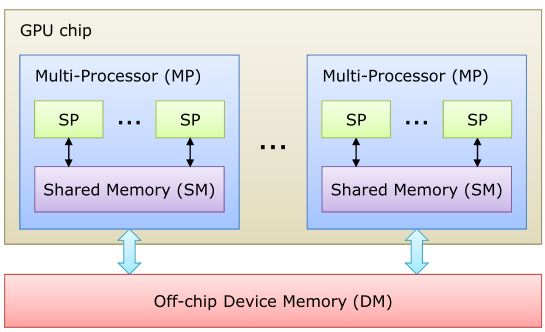


Fig. 1 Nvidia GPU architecture.

Manuscript received May 7, 2013.
Manuscript revised August 6, 2013.

[†]The author is with the Department of Multimedia, Sungkyul University, Korea.
[††]The author is with the Department of Computer Science, Kangwon National University, Korea.
[†††]The author is with the Department of Multimedia Science, Sookmyung Women's University, Korea.

*This research was partially supported by Basic Science Research Program through the National Research Foundation (NRF) funded by the Ministry of Education, Science and Technology (MEST) (No. 2010-0025001). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2011-0013235).

a) E-mail: ysmoon@kangwon.ac.kr (Corresponding author)
b) E-mail: yhpark@sm.ac.kr (Co-corresponding author)

DOI: 10.1587/transinf.E97.D.1349

The logical structure of kernel threads is as follows. The kernel threads are divided into multiple *blocks* so that the same number of blocks are contained in each block. A block has a unique zero-based *blockID*. For example, when there are 64 blocks, each block has a *blockID* 0 to 63. The threads in a block have unique zero-based *threadID*s in the same way. A block is assigned to an MP, and each thread is executed by an SP. An MP can be assigned multiple blocks, but a block cannot be executed by multiple MPs. In a CUDA program, the number of blocks and threads to execute a kernel function is specified, and the performance of the function is highly dependent on the number. If the number of blocks is larger than the number of MPs or the number of threads in a block is larger than the number of SPs in an MP, the corresponding kernel function is executed in the time-shared manner. If some blocks or threads are waiting for data accesses or synchronization, the GPU performs context switching for operational efficiency.

3. Related Work

In this section, we briefly explain about DBSCAN [3] and CUDA-DClust [2]. DBSCAN presented *density-connectivity* relationship for two objects and defined a cluster as a maximal set of density-connected objects. The density-connectivity relationship is defined using two input parameters ϵ and *MinPts*.

DBSCAN for a given dataset $\mathcal{D}$ is summarized in Algorithm 1. At first, the state of each object is set to be *undecided*. If an object is inserted into a cluster, its state turns into *decided*. Since, for each *undecided* object p, DBSCAN computes ϵ-neighbor $N_\epsilon(p)$ to determine whether p is a core object or a noise, DBSCAN's time complexity is $O(n^2)$, where n is the number of objects in $\mathcal{D}$.

Böhm et al. [2] proposed CUDA-DClust, which improved DBSCAN using a GPU. While DBSCAN forms a cluster for a single *undecided* object p ($\in \mathcal{D}$) at a time, CUDA-DClust assigns an *undecided* object p to each of thread blocks and forms multiple clusters for the assigned *undecided* objects simultaneously. To speed up computing ϵ-neighbor $N_\epsilon(p)$ in each block, CUDA-DClust performs multiple distance computations from p to a different object in each thread simultaneously. The sub-cluster formed in each block is called a *chain* in CUDA-DClust. If there is a collision between the chains being formed by different parallel blocks, i.e., an object is inserted into two or more different chains, it is recorded in a separate data structure called a collision matrix. In the final stage, the chains with collisions are merged to form a single cluster. Böhm et al. [2] showed through experiments that CUDA-DClust outperformed DBSCAN by up to 15 times. However, CUDA-DClust has a drawback that most of distance computations from an *undecided* object p to all the other objects in $\mathcal{D}$ to compute $N_\epsilon(p)$ is unnecessary.

Algorithm 1 DBSCAN.

```
1: Set the state of each object o in D to be undecided;
2: while there exist undecided objects in D do
3:    Choose any undecided object p ∈ D and compute N_ε(p);
4:    if |N_ε(p)| ≥ MinPts then
5:       Form a new cluster C and insert p in C;
6:       Seed ← N_ε(p) − p;
7:       while there are undecided or noise objects in Seed do
8:          for each undecided or noise object q ∈ Seed do
9:             Insert q in C and compute N_ε(q);
10:            if |N_ε(q)| ≥ MinPts then
11:               Seed ← Seed ∪ N_ε(q) − q;
12:            end if
13:         end for
14:      end while
15:   else
16:      Set the state of p as noise;
17:   end if
18: end while
```

4. GSCAN: GPU-Based DBSCAN

GSCAN is an extension of CUDA-DClust [2]. As shown in Algorithm 1, DBSCAN as well as CUDA-DClust performs distance computations from each *undecided* object p to all the other objects in $\mathcal{D}$ to compute ϵ-neighbor $N_\epsilon(p)$. However, as shown in Fig. 2, most of the other objects in $\mathcal{D}$ reside beyond ϵ-range from p and thus most of the distance computations are unnecessary. GSCAN improves the performance by reducing such unnecessary computations.

GSCAN reduces unnecessary distance computations as follows. GSCAN forms a grid covering the entire data space and divides the objects in $\mathcal{D}$ into grid cells as shown in Fig. 2. The grid dimension d' is less than or equal to data dimension d; in our experiments, we always set $d' = 4$ regardless of data dimension. Then, for each cell in the grid, a list of objects contained in the cell is generated. Unlike DBSCAN and CUDA-DClust, GSCAN performs distance computations from an *undecided* object p to only the objects contained in the cells overlapping with ϵ-range from

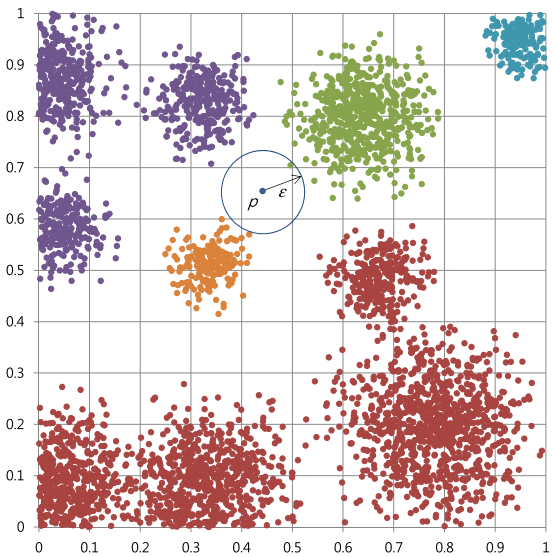


Fig. 2 Data objects divided into grid cells.

p. In Fig. 2, GSCAN considers only the objects contained in 9 cells around p and ignores all the other objects in different cells. Since GSCAN performs distance computations with much smaller number of objects than DBSCAN and CUDA-DClust, it achieves better performance.

A GPU kernel function is used to efficiently find the data objects contained in each grid cell in GSCAN. The entire dataset is evenly divided into *gridSize* data subsets and each subset is assigned to a kernel block. The data subset assigned to a block consists of data objects residing adjacently in device memory to make the most of device memory cache. Algorithm 2 shows the kernel function to divide the objects into grid cells. In lines (1) and (2), the range $[i_1, i_2)$ of data objects to be handled in each block is determined. Since each block is assigned a different *blockID* ($0 \leq blockID < gridSize$), there is no overlap between the ranges for different blocks. In line (3), *blockSize* indicates the number of threads in a block. In lines (3) to (7), since each thread in the block is assigned a different *threadID* ($0 \leq threadID < blockSize$), different data objects $\mathcal{D}[i]$ are handled in each thread; the cell containing $\mathcal{D}[i]$ is found, and $\mathcal{D}[i]$ is appended in the corresponding list $list[c]$. In line (5), $cellSize[c]$ can be accessed by multiple parallel threads at the same time. An atomic function atomicInc() prevents such a race condition by serializing concurrent accesses by different threads. atomicInc() increments $cellSize[c]$ by 1 and returns the value stored previously in $cellSize[c]$.

GSCAN finds the cells overlapping with ϵ-range from an object p efficiently as follows. Since there are a large number of grid cells, it is inefficient to compute the distance of each cell from p. GSCAN finds the set C_p of cells around the cell C_p containing p. Let c_i (≥ 0) be the coordinate of i-th dimension ($0 \leq i < d'$) of C_p. Then, the coordinates of i-th dimension of the cells in C_p are $c_i - 1$, c_i, or $c_i + 1$, and thus the number of cells in C_p is not more than $3^{d'}$, where d' is the grid dimension. For example, the x/y-coordinates of the cells in Fig. 2 are in the range $[0, 9]$. Since the x-coordinate of C_p containing p is 4, the cells in C_p have x-coordinates 3, 4, or 5. The condition necessary is that the size for every dimension of a cell should not be less than ϵ; otherwise, some objects in $N_\epsilon(p)$ may not be contained in the cells in C_p. Since the size for every dimension of the cells in Fig. 2 is 0.1, ϵ should not exceed 0.1.

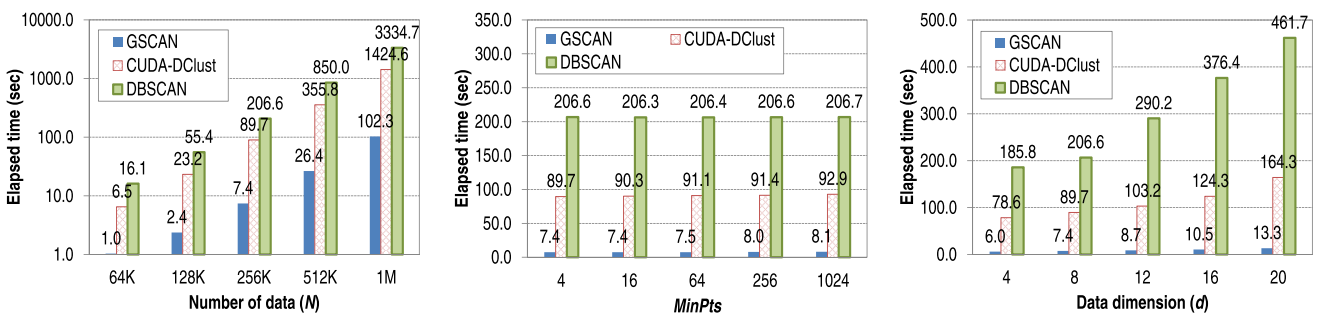
Algorithm 2 Kernel function to divide data objects into grid cells.

```
1: i1 ← N * blockID/gridSize;
2: i2 ← N * (blockID + 1)/gridSize;
3: for i ← i1 + threadID to i2 − 1 step blockSize do
4:     c ← ID of the cell containing D[i];
5:     j ← atomicInc(cellSize[c], N);
6:     list[c][j] = i;
7: end for
```

5. Performance Evaluation

In this section, to show the superiority of GSCAN, we compare its performance with CUDA-DClust [2] and DBSCAN [3]. We measure the elapsed time for their execution for various values of input parameters. For GSCAN and CUDA-DClust, the elapsed time starts when data objects are first copied to GPU device memory and ends when the final clustering result is copied to main memory. Experimental data is d-dimensional synthetic objects forming 10 arbitrary Gaussian clusters, and the coordinate for every dimension of the objects resides in the range [0.0, 1.0]. The input parameters are the number of data objects N, the minimum number of neighbors *MinPts*, and the data dimension d. We set the default parameter values as the same as in [2]: N = 256K, $MinPts$ = 4, d = 8, and ϵ = 0.05. The hardware platform is a PC equipped with the Intel Core i7-3960X 3.3GHz CPU, 16GB DDR3 main memory, 256GB solid-state drive (SSD), and Nvidia GTX 780 GPU with 3GB GDDR5 device memory. The software platform is Visual Studio 2010 and CUDA Toolkit 5.0 on Microsoft Windows 7 64bit Edition.

The first experiment was performed for various values of N (Fig. 3 (a)). Note that the vertical axis in Fig. 3 (a) is represented in the log scale. Table 1 shows the performance improvement ratio of GSCAN compared with CUDA-DClust (3rd row) and DBSCAN (4th row). GSCAN outperformed CUDA-DClust and DBSCAN by up to 13.9 and 32.6 times, respectively, when N = 1M. Table 2 shows the number of distance computations in the unit of 1G (2^{30}), and the values in parentheses are the reduction ratios in GSCAN. Compared with CUDA-DClust, while the reduc-



(a) Number of data (N). (b) $MinPts$. (c) Data dimension (d).

Fig. 3 Comparison of elapsed time.

Table 1 Performance improvement ratio of GSCAN.

	(a) Number of data (N)					(b) *MinPts*					(c) Data dimension (d)				
	64K	128K	256K	512K	1M	4	16	64	256	1024	4	8	12	16	20
CUDA-DClust	6.3	9.8	12.1	13.5	13.9	12.1	12.2	12.2	11.4	11.4	13.1	12.1	11.8	11.8	12.3
DBSCAN	15.6	23.5	28.0	32.2	32.6	28.0	27.8	27.6	25.9	25.4	30.9	28.0	33.3	35.9	34.7

Table 2 Number of distance computations.

N	GSCAN	CUDA-DClust	DBSCAN
64K	0.2	4.3 (18.7)	1.2 (5.2)
128K	0.9	16.6 (18.4)	4.8 (5.3)
256K	3.6	65.5 (18.4)	19.2 (5.4)
512K	13.7	258.5 (18.9)	76.8 (5.6)
1M	54.7	1,029.0 (18.8)	307.2 (5.6)

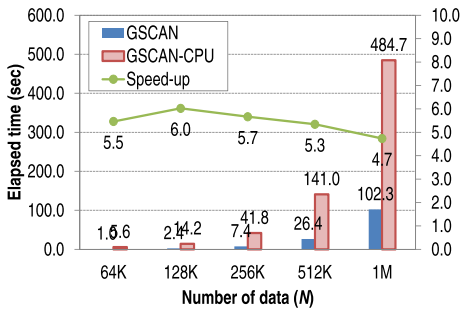


Fig. 4 Performance comparison with GSCAN-CPU.

tion ratio remains without significant changes, the performance improvement ratio by GSCAN keeps increasing as N increases. That is because the algorithms perform many operations such as data transfer other than distance computations, and the percentage of distance computations increases as N increases. The same discussion also applies to DBSCAN.

The second experiment was performed for various values of *MinPts* (Fig. 3 (b)). The reduction ratios of distance computations in GSCAN were 18.4 and 5.4 compared with CUDA-DClust and DBSCAN, respectively. As shown in the figure, the elapsed time of the algorithms is almost unchanged, since there is only little influence on the number of distance computations by *MinPts*.

The third experiment was performed for various values of d (Fig. 3 (c)). Although the elapsed time increases as d increases, there are only small changes in the performance improvement ratios as shown in Table 1. That is because, while the number of distance computations of the algorithms is almost unchanged, the more time is needed for distance computations for the larger d.

In the fourth experiment, we converted GSCAN for multi-core CPUs, which is called *GSCAN-CPU* in this paper, and compared its performance with GSCAN for GPUs (Fig. 4). GSCAN outperformed GSCAN-CPU by up to 6.0 times when N = 128K. Although the performance of each SP in the GPU is lower than the CPU core, the number of SPs in the GPU is much larger than the number of CPU cores. In our hardware, the CPU contains only six cores, while there are 2304 SPs in the GPU. Thus, the simple operations such as distance computations can be executed by the GPU more efficiently than the CPU. However, for the complicated operations with many conditional branches and frequent data communications, the CPU is the better choice, since such operations incur performance degradation by holding many GPU threads waiting.

References

[1] M. Ankerst, M.M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: Ordering points to identify the clustering structure," Proc. Int'l Conf. Management of Data, ACM SIGMOD, pp.49–60, Philadelphia, Pennsylvania, USA, June 1999.

[2] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," Proc. Conf. Information and knowledge management (CIKM), pp.661–670, Hong Kong, China, Nov. 2009.

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD), pp.226–231, Portland, Oregon, USA, 1996.

[4] A. Hinneburg and D.A. Keim, "An efficient approach to clustering in large multimedia databases with noise," Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD), pp.58–65, New York, New York, USA, Aug. 1998.

[5] Nvidia, CUDA C Programming Guide, Ver. 4.2, April 2012.

[6] S.A.A. Shalom, M. Dash, and M. Tue, "Efficient K-means clustering using accelerated graphics processors," Proc. Int'l Conf. Warehousing and Knowledge Discovery (DaWaK), pp.166–175, Turin, Italy, Sept. 2008.

[7] C. Trapnell and M.C. Schatz, "Optimizing data intensive GPGPU computations for DNA sequence alignment," Parallel Computing, vol.35, no.8, pp.429–440, Aug. 2009.