Project Report for GPU based DBSCAN algorithm

Madhav Poudel
School of Informatics, Computing & Cyber Systems
Northern Arizona University, Flagstaff, AZ, U.S.A.
mp2525@nau.edu

***Abstract*—Over the last decade, there has been an ever-increasing interest in parallel computation. The primary goal of adopting parallel computing is to improve the speed of intensive computation. Density-based clustering algorithms are widely used data mining algorithm used to find the clusters in data without any prior knowledge of its belongings [14]. It is one of the algorithms which require intensive computing and there have been many attempts to improve the performance of it. In this project, The DBSCAN algorithm is implemented in the Compute Unified Device Architecture (CUDA) programming model to achieve the parallelism and high-performance benefits offered by GPU. The result of the DBSCAN algorithm executed in parallel GPU is expected to be similar to the sequential CPU implementation. The demonstration of comparison in the performance between the sequential and parallel implementation of this algorithm is the main motive for the project.**

***Index Terms*—CUDA, GPU, CPU, parallel computing, clustering, dbscan**

I. Introduction

Nowadays, Parallel programming is becoming mainstream in the programming world being used in many different applications [2]. Parallelism is becoming the driving force of architecture design. GPU is one of the most essential hardware components of computer architecture that performs a rapid mathematical calculation in parallel. While the GPU was used primarily in processing and rendering images to display, GPU has been lately exploited in the most complex calculations in scientific research, big data research, data mining, machine learning, and artificial intelligence. A major benefit of GPU is that it provides extremely high parallelism combined with economically high bandwidth memory transfer. GPU is accelerating applications running on CPU by offloading compute-intensive and time-consuming portions of the program.

There has been progressive development in libraries, compiler and application programming interfaces supporting the trend of GPU and parallel programming. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve multiple complex computational problems more efficiently [5]. CUDA C is designed with an extension to a standard C programming language including a handful of APIs designed enabling developers to write a parallel algorithm that executes on GPU [13]. A typical CUDA program comprises the host program and kernel functions accompanying together. The host program is executed on the CPU while the kernel functions are executed in a massively parallel fashion on hundreds of processors on the GPU [1].

Clustering is one of the widely used unsupervised learning techniques in data processing. It helps to arrange the input data set into a collection of a finite range of semantically consistent groups with respect to its similarity [9]. Density-based clustering is among many of the different types of clustering algorithms with a purpose to group points that are closely packed together while noting the point outliers or noises that lies alone in low-density regions.

Density-based spatial clustering of applications with noise (DBSCAN) is a popular data clustering algorithm commonly used in data mining and machine learning applications. The key idea of this algorithm is that for each point of a cluster the neighborhood of a given radius has to include at least a minimum number of points [11]. The algorithm computes the distance from one point to another point, possibly multiple times to find the dense region which makes the algorithm complex and time consuming with the increase in many objects in dataset. There have been numerous attempts to enhance the performance of the algorithm. In light of this fact, The advantages of GPU over CPU can be exploited to build a DBSCAN algorithm which can drastically improve the performance of the algorithm. The project aims to implement a GPU based DBSCAN algorithm with CUDA to improve the performance compared to the CPU based sequential algorithm.

The rest of the report is organized as follows. Section 2 provides the background of DBSCAN and CUDA. Section 3 explains the methodology of DBSCAN based on CUDA. In Section 4, we present an experimental evaluation and comparision between CPU and GPU based DBSCAN algorithm. Finally, Section 5 discusses and concludes the project report and gives some future direction.

II. Background

A. An Overview of DBSCAN algorithm

Clustering is one of the most popular methods in data processing and analysis. It is being used in many areas such as machine learning, data mining, data compression, bioinformatics, and so on. Clustering, as a process, is a way of retrieving similar information in the data and its structure. In the real world where data is growing at a fast pace to an enormous size, noises are unavoidable problems that demand a robust clustering algorithm. There are many existing algorithms for finding clusters from the sparse data. However, these algorithms are not robust to noises and expect clusters number before finding them [7].

DBSCAN (density-based spatial clustering of applications with noise) is one of the most popular density-based clustering algorithms which do not expect cluster number beforehand and works well in a noisy dataset. It assumes that clusters are regions of high-density patterns, separated by regions of low density in the data space [6]. The algorithm is based on the intuitive concept of clusters and noise. The key idea is that for each point of a cluster, the neighborhood of a point has to contain at least a minimum number of points within a given range. DBSCAN algorithm have two main parameters and three types of points based on it as shown in 1.

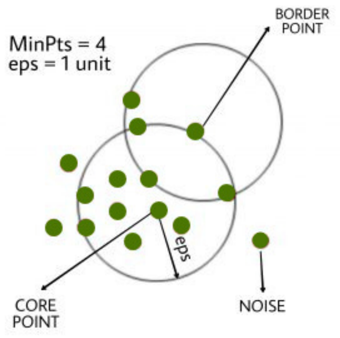


Fig. 1: Different parameters and types of points in DBSCAN

Two main parameters of DBSCAN algorithms are:

- ***eps***: It is a radius from a point in space used to find neighbors. if the distance between two points is lower or equal to this value, these points are considered neighbors.
- ***minPoints***: It is a minimum number of points within *eps* to form a dense region.

For example: If *eps* = 1.5 and *minPoints* = 4 then, It means that there are a minimum of 4 points required to form a dense cluster and a pair of points must be separated by a distance of less than or equal to *eps* to be considered as neighbors.

Furthermore, based on these two parameters, data points are classified into three categories listed below:

- ***Core Point***: A point that has more than minimum neighbor points within an *eps*.
- ***Border Point***: A point which has fewer than minimum neighbor points within *eps* but it is in the neighborhood of the core point.
- ***Noise Point***: A point which is neither a core point nor a border point.

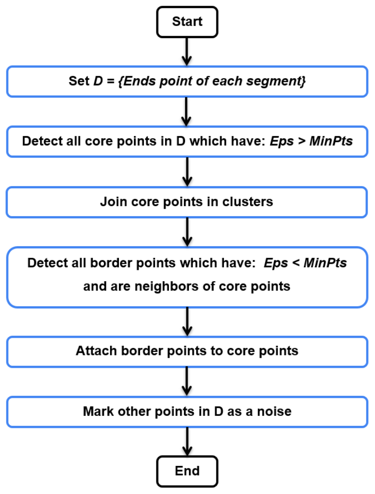

Start
Set *D = {Ends point of each segment}*
Detect all core points in D which have: *Eps > MinPts*
Join core points in clusters
Detect all border points which have: *Eps < MinPts* and are neighbors of core points
Attach border points to core points
Mark other points in D as a noise
End

Fig. 2: Flowchart of DBSCAN algorithm

The abstracted steps [12] involved in DBSCAN algorithm is given below and also shown in figure 2.

1) Find all the neighbor points within *eps* and identify the core points with more than *minPoints* neighbors.
2) For each core point if it is not already assigned to a cluster, create a new cluster.
3) Find recursively all its density connected points and assign them to the same cluster as the core point. A point a and b are said to be density connected if there exists a point c that has a sufficient number of points in its neighbors and both the points a and b are within the *eps*. So, if b is neighbor of c, c is neighbor of d, d is neighbor of e, which in turn is neighbor of a implies that b is neighbor of a.
4) Iterate through the remaining unvisited points in the dataset. The points that do not belong to any of cluster are considered as noise.

DBSCAN has a concept of noise and works well even with noisy datasets. Since the algorithm is robust with outliers, the algorithm is widely used in clustering applications [4]. However, the complexity of the algorithm increases with the increase in the number of datasets due to the tedious process of calculating neighbor points. To overcome the challenge and improve the performance of the DBSCAN algorithm is an active area of research.

B. CUDA Programming model

CUDA is a parallel computing platform and programming model with a small set of C extensions. With CUDA, we can build applications for a myriad of systems running parallel on GPUs, ranging from embedded devices, tablet devices, laptops, desktops, and workstations to HPC clustered systems [10].

Threads are the basic unit of CUDA programming. CUDA threads are extremely lightweight with less overhead of creation and fast switching. The parallelism is achieved by using thousands of threads inside GPU to execute a single instruction [3]. Each thread has an ID that it uses to compute memory addresses and make the control decisions. These threads cooperate by sharing different hierarchy of memory categorized by access time. Since the GPU has no direct access to the address spaces of CPU and the bandwidth of the system bus is very limited for transferring data from CPU to GPU, these memory models in GPU provide performance advantage to GPU. The management of memory in GPU is done by Host (CPU) code which includes memory allocation, memory freeing, copying data to and from a device, and so on.

A key component of the CUDA programming model is the kernel. The kernel is a code that is expressed as a sequential program and runs on the GPU device. A typical CUDA program is written in the host and it consists of serial code coexisting with parallel code [8]. When we execute a kernel function in a GPU, it launches a grid of thread blocks. The threads within a block cooperate through shared memory and synchronize their execution. CUDA applications use many such threads that are running at the same time, to achieve parallel computation.

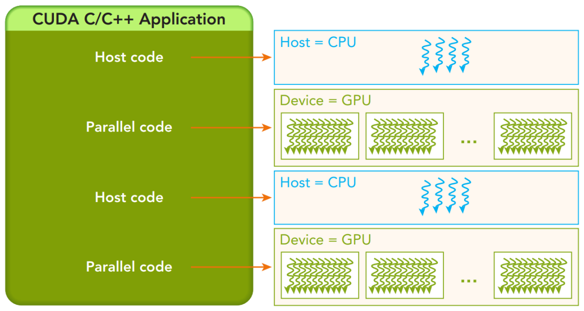


Fig. 3: Device and Host code execution in CUDA

A typical CUDA program consists of serial code coexisting together with parallel code. While the serial code is executed on the host/CPU, the parallel code is executed on the device/GPU. The execution model of host and device code in the CUDA program is shown in figure 3. A typical processing flow of a CUDA program follows the following steps.

- Copy data from Host memory to device memory.
- Host provides instruction to the device.
- Execute kernel function to operate on data stored on device memory.
- Copy data back from device memory to host memory.

III. Methodology

In the DBSCAN algorithm, there is a complex relationship between the data objects and an unknown number of clusters. The objects in a cluster are determined in multiple steps of finding, directly and indirectly, several unknown objects. It is not a straightforward process to do a parallelization. To overcome these challenges, we are using chains for parallelization. A chain is a collection of data objects belonging to a common density-based cluster. Each of these chains is processed by CUDA blocks. As these chains may belong to a single cluster, the concept of collision matrix is employed to determine common chains belonging to a single cluster. A data structure called seed list is used throughout the process to keep track of points that are being expanded or not. Each of these concepts is explained in detail following subsections.

A. Chains

Chains are a central concept in our algorithm to implement parallelism in DBSCAN using CUDA. Chains are basically blocks in CUDA architecture. Objects from the dataset are processed and expanded in the chain in a massively parallel way. The clusters are expanded by starting with a point and collecting cluster members belonging to the chain. As an example: If we use 32 blocks in our algorithm. Then at one instance, 32 points are expanded in each block forming 32 different chain clusters. Further, The threads assigned to each block are utilized for the tedious process of finding neighbor objects.

B. Seed List

Seed List is an important data structure of our algorithm that records the objects that need to be expanded to find the objects in the chain cluster. Its main objective is to provide a queue for the objects which are directly reachable from the core objects to expand the chain cluster where the core object belongs. The seed list is assigned to each chain or block in the algorithm before the execution of the DBSCAN kernel function. The seed list is updated while expanding the chain cluster and later the points in the seeds are processed to find more members of the chain cluster.

The size of the seed list is limited and there is a possibility of reaching its limit. In such a case, We have employed a refill seed list with a similar data structure to keep a record of objects overflown from the seed list. Once the seed list is empty, the seed list is filled with objects from the refill seed list. We have used two counters, seed list length and refill seed list length, to maintain the changing size of the seed list and refill seed list.

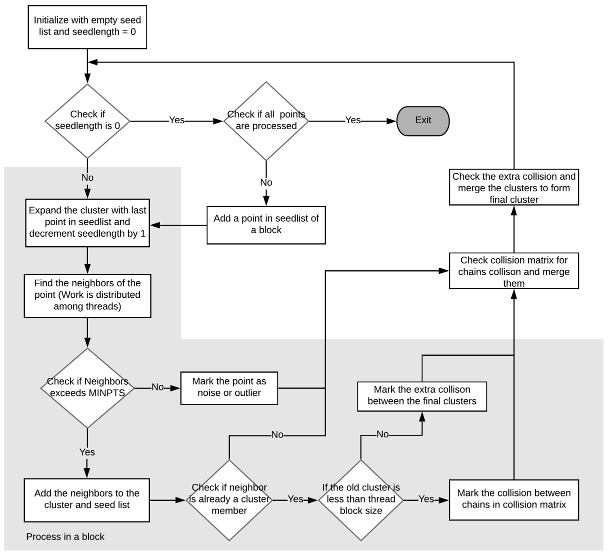
C. Cluster Collision

Since each of the chains expands different chain clusters starting from different objects. There is a high chance that these chain clusters expanded belong to a single cluster. We call this situation a collision and to deal with it, we have introduced another data structure called collision matrix which keeps a record of a collision between chains. The collision matrix record is used to merge the chain clusters belonging to the same group to form a final cluster. However, the collision matrix only keeps a record of chain clusters at one instance until the seed list is emptied and filled with new starting seed points. During this instance, the record of a collision between these final clusters formed after merging chain clusters in each of these instances is not recorded by the collision matrix. To this extent, we have introduced another data structure called extra collision which keeps track of collision in finalized clusters and merges to form the ultimate clusters. The extra collision record is kept along with the collision matrix for the collisions between the finalized cluster.

D. GPU based DBSCAN algorithm in CUDA

The GPU based DBSCAN algorithm starts by importing the datasets and allocating memory in GPU and CPU. There is rapid communication between CPU and GPU throughout the process to keep track of the expansion of objects and the merging of clusters. CPU manages the seed list objects from the dataset and merge the collision between clusters. GPU executes the main DBSCAN function by expanding the cluster for an object residing in the seed list by each block. There are two main functions in our algorithm, they are explained in detail in the following subsections along with the communication between them. Also, The flowchart of the DBSCAN algorithm with the communication between CPU and GPU is shown in figure 4.

1) Monitor Seed Points: Monitor Seed Points is the main function in CPU which keeps track of seed list, refill seed list, chains collision, and extra collision. In each step of monitoring seed points, it copies data in GPU memory to CPU. It first checks if the seed list is empty or not. If it is empty then for each chain, it assigns new points from unprocessed points in the dataset. If the seed list already contains objects, it passes its control to the DBSCAN kernel function to expand the cluster from the last object in the seed list. If the seed list is empty and there exist objects in the refill seed list then, the objects in the refill seed list are transferred into the seed list for further expansion. The seed list length and refill seed list length are maintained to keep track of objects correctly. If the seed list and refill seed list both are empty, then it checks the collision matrix for the record of chain collisions and merges the chain clusters. Further, It checks for the extra collision between finalized chain clusters and merges to form the final cluster. Once the processing of seed list and clusters merging are done then, the changed data in CPU are copied back to GPU memory for further dbscan expansion. Finally, It checks for unprocessed objects in each step to keep track of the completion of the algorithm. If all objects are processed then, It returns the flag that stops the whole dbscan process, and the results are shown from CPU. In case there are remaining objects to be processed, All the changes made in the CPU memory are copied in GPU memory for further expansion of the clusters.



Initialize with empty seed list and seedlength = 0
Check if seedlength is 0
Yes
Check if all points are processed
Yes
Exit
No
No
Expand the cluster with last point in seedlist and decrement seedlength by 1
Add a point in seedlist of a block
Check the extra collision and merge the clusters to form final cluster
Check collision matrix for chains collison and merge them
Find the neighbors of the point (Work is distributed among threads)
Check if Neighbors exceeds MINPTS
No
Mark the point as noise or outlier
Mark the extra collison between the final clusters
No
No
Yes
Add the neighbors to the cluster and seed list
Check if neighbor is already a cluster member
Yes
If the old cluster is less than thread block size
Yes
Mark the collision between chains in collision matrix
Process in a block

Fig. 4: Flowchart of DBSCAN algorithm in Cuda

2) DBSCAN kernel function: The DBSCAN kernel is the main method of our algorithm that executes in GPU parallelly. It determines the property of the object and expands it to find the members of the chain cluster. Once the seed list is assigned with new objects from the CPU then, each of the blocks expands the last object from the seed list decreasing the seed list length to prevent the multiple expansion of the object. The neighbors of the expanded object are searched among all the objects in the dataset. The workload in the process of finding neighbors is distributed among threads to achieve massive parallelism. The objects whose distance with the expanded objects is less than epsilon value are considered as neighbor points. The number of neighbors found in the dataset is counted by each thread in parallel using the atomic operation to prevent the potential race condition. If the number of neighbor objects is less than min points then, they are stored in the neighbor buffer. If the number of neighbors points is greater than and equal to min points then, these points are marked as the candidate of the cluster.

During the process of marking candidates, the potential collisions between the chains are checked by the knowledge of belonging of these neighbors object in multiple clusters. If these objects already belong to another chain cluster, then the collision is recorded between chains in the collision matrix. Similarly, If the object already belongs to the final clusters, then the clusters are recorded in an extra collision for final merging. If the object doesn't belong to any cluster and is unprocessed, then the object is kept in the chain cluster of the core object and added to the seed list for further expansion. By the same token, If the object is a noise before, it is also added to the chain cluster of core object. However, It is not added to the seed list as noise points cannot be expanded. Additionally, the remaining objects in the neighbor buffer are marked as a candidate and processed with similar steps.

3) Communication between CPU and GPU: There is a communication between CPU and GPU each time a new object needs to be expanded in the DBSCAN kernel function. The data structured used in the algorithm are defined in both CPU and GPU. In each communication step, data are transferred between them. The communication is initiated with a loop executing Monitor Seed Point function in CPU and DBSCAN kernel in GPU while processing the objects from the dataset and keeping track of remaining unprocessed objects. Once all the objects are processed, the loop is exited and the result of DBSCAN is printed in CPU.

IV. Experimental Evaluation

To evaluate the performance of the GPU based DBSCAN algorithm and to compare it with CPU algorithm, we have performed a set of experiments. First of all, we evaluated the runtime of the algorithm in CPU and GPU with a changing number of datasets. Then, from the runtime result, we evaluated the speedup gained in GPU based DBSCAN algorithm. Second, We evaluated the impact in runtime and speedup of CPU and GPU based algorithms with a changing number of elipson value. Third, we evaluated the impact in runtime and speedup of CPU and GPU based algorithms with a changing number of minimum points. Finally, We did evaluation on runtime profining of GPU based DBSCAN algorithm to find the contribution by components of algorithm in the total execution time. The implementation of the CPU algorithm is written in C++ and GPU algorithm is written in CUDA. We have used the following computational setup for the experiment.

- Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
- NVIDIA Quadro GP100 GPU
- CUDA version 9.0

Similarly, We have used the following specifications for the dataset, CUDA, and parameters in the DBSCAN algorithm unless specified in the particular experiment.

- Number of CUDA Blocks: 16
- Number of threads per block: 1024
- Size of Seed List: 2048
- Size of refill seed list: 4096
- Number of objects in the dataset: 1000000 (1M)
- Dimension: 2
- EPSILON: 1.5
- MINPTS: 4

A. DBSCAN vs GPU based DBSCAN

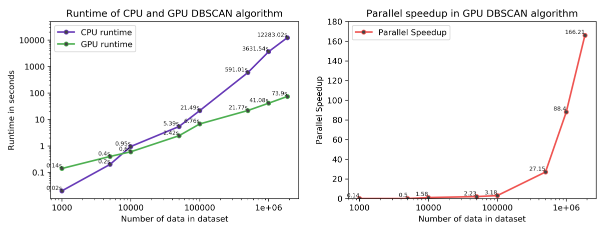


Fig. 5: Runtime and Speedup of CPU and CUDA based DBSCAN algorithms

Figure 5 shows the runtime of the CPU and GPU based DBSCAN algorithm along with the speedup achieved in the GPU based algorithm. For this experiment, we used the changing number of objects in the dataset ranging from 1000 to 1864620. From the plot, we can see that the runtime for the GPU algorithm is higher than the CPU algorithm until the objects are in the range of 10000. Because of the communication overhead between CPU and GPU, this uncertain behavior is seen in the plot. After the dataset exceeds 10000 objects, the runtime for the GPU algorithm is lower than the CPU algorithm leading to an increase in parallel speedup. For all objects in a dataset of size 1864620, the runtime for GPU algorithm is nearly equal to 1 minute and 15 seconds compared to the CPU algorithm with a runtime of CPU nearly equal to 3 hours and 25 minutes. The speedup gained by the GPU algorithm with a full dataset size is 166 compared to the CPU algorithm. By the same token, in GPU based DBACAN, for 1 million points, the runtime is one hour and the parallel speedup is 88. We can infer from this experiment that the speedup gained in the GPU algorithm is high with an increasing number of objects in the dataset.

B. *Impact of parameter* ***eps*** *(epsilon)*

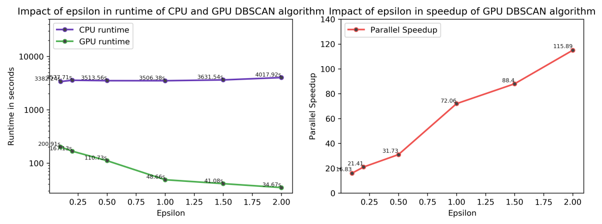


Fig. 6: Impact of EPS in runtime and Speedup of CPU and CUDA based DBSCAN algorithms

Figure 6 shows the impact of parameter EPS in the runtime of the CPU and GPU based DBSCAN algorithm along with the speedup gained in GPU approach. For this experiment, we used the changing value of EPS from 0.1 to 2. From the plot, we can see that the runtime for the GPU algorithm is higher than the CPU algorithm with changing the value in the EPS parameter. The run time of CPU is greatly affected by the changing value of EPS but, the run time of GPU is decreasing with increasing value of EPS. Thus, our algorithm has not any effect but, improvement with the change in the EPS parameter.

C. *Impact of parameter* ***minPts*** *(Minimum Points)*

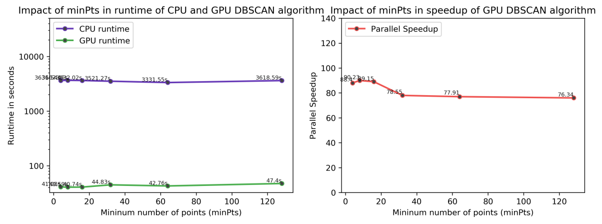


Fig. 7: Impact of minPts in runtime and Speedup of CPU and CUDA based DBSCAN algorithms

Figure 7 shows the impact of minPts parameter in the runtime of the CPU and GPU based DBSCAN algorithm along with the speedup gained in GPU approach. For this experiment, we used the changing value of minPts ranging from 4 to 128. The speedup of the GPU based algorithm tends to decrease with an increasing number of minPts parameter. The impact exists because many neighbors need to buffer before they are marked as a candidate. As threads use limited share memory while executing in parallel, higher the number of minPts, a large amount of share memory is used whose impact can be seen in overall algorithm performance.

D. *Runtime Profiling*

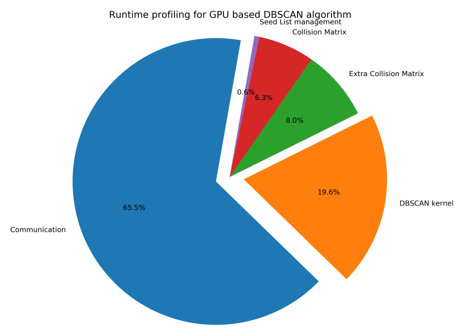


Fig. 8: Runtime profiling of GPU based DBSCAN algorithm

Figure 8 shows the runtime profiling for GPU based DBSCAN algorithm showing various components of the algorithm and it's execution time. From the diagram, we can infer that most of the execution time is used by communication between CPU and GPU. It is obvious with the fact that the algorithm executes in a loop with communication in each step. The use of several data structures has also contributed to the communication overhead. Similarly, the DBSCAN kernel has the second-largest execution time. The time execution for seed list management is negligible. However, Collision matrix and Extra collision matrix detection and cluster merging process seem to take the most portion of execution time in CPU.

V. Conclusion

We developed a GPU based DBSCAN algorithm using CUDA and utilizing the advantages of GPU. The algorithm produces nearly similar results compared to the CPU algorithm with a slight deviation. The deviation in the result is because of the parallel execution of the algorithm in random order and merging of a few extra clusters than necessary. From the experiment evaluation, we found that the performance of GPU based DBSCAN algorithm outperforms CPU based DBSCAN algorithm by a large factor. Also, we found that the algorithm has a slight impact when the number of minPts parameters is increased. In the runtime profiling experiment, we found that most of the execution time is consumed during communication between CPU and GPU. It infers that we haven't fully utilized the advantages of GPU leaving us room for optimization of the algorithm in the future. The optimization of the algorithm to fully utilize the advantages of GPU and reduce communication remains the future work of this project.

Acknowledgement

I would like to thank Dr. Michael Gowanlock for providing me an opportunity to work on this project. It has been a great learning experience for me. It has helped me to broaden my knowledge and skill in high-performance computing and technologies used in parallel and intensive data computing. Overall, I would like to thank you for encouraging me to choose a topic in CUDA.

References

[1] Christian Böhm et al. “Density-based clustering using graphics processors”. In: Jan. 2009, pp. 661–670. DOI: 10.1145/1645953.1646038.

[2] John Cheng. *Professional Cuda C programming*. Wrox programmer to programmer. Indianapolis, IN: John Wiley and Sons, Inc, 2014. ISBN: 9781118739327.

[3] Shane Cook. *CUDA programming: a developer’s guide to parallel computing with GPUs*. OCLC: ocn773025100. Amsterdam ; Boston: Elsevier, MK, 2013. ISBN: 9780124159334.

[4] Bryan Dixon. “Investigating clustering algorithm DBSCAN to self select locations for power based malicious code detection on smartphones”. In: *2017 Third International Conference on Mobile and Secure Services (MobiSecServ)*. Miami Beach, FL, USA: IEEE, Feb. 2017, pp. 1–7. ISBN: 9781509036325. DOI: 10.1109/MOBISECSERV.2017.7886567. URL: http://ieeexplore.ieee.org/document/7886567/ (visited on 05/11/2020).

[5] *Introduction to GPUs: CUDA*. URL: https://nyu-cds.github.io/python-gpu/02-cuda/ (visited on 04/17/2020).

[6] Li Ma et al. “G-DBSCAN: An Improved DBSCAN Clustering Method Based On Grid”. In: Dec. 2014, pp. 23–28. DOI: 10.14257/astl.2014.74.05.

[7] Adam Merk, Piotr Cal, and Michal Wozniak. “Distributed DBSCAN Algorithm – Concept and Experimental Evaluation”. In: May 2017, pp. 472–480. DOI: 10.1007/978-3-319-59162-9_49.

[8] John Nickolls. “GPU parallel computing architecture and CUDA programming model”. In: *2007 IEEE Hot Chips 19 Symposium (HCS)*. Stanford, CA, USA: IEEE, Aug. 2007, pp. 1–12. ISBN: 9781467388696. DOI: 10.1109/HOTCHIPS.2007.7482491. URL: http://ieeexplore.ieee.org/document/7482491/ (visited on 05/11/2020).

[9] Nidhi and Archana Patel. “An Efficient and Scalable Density-based Clustering Algorithm for Normalize Data”. In: *Procedia Computer Science* 92 (Dec. 2016), pp. 136–141. DOI: 10.1016/j.procs.2016.07.336.

[10] *Professional CUDA C Programming*. 1st. GBR: Wrox Press Ltd., 2014. ISBN: 1118739329.

[11] Anant Ram et al. “A Density Based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases”. In: *International Journal of Computer Applications* 3 (June 2010). DOI: 10.5120/739-1038.

[12] Anant Ram et al. “A Density Based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases”. In: *International Journal of Computer Applications* 3 (June 2010). DOI: 10.5120/739-1038.

[13] Ali Tourani. *CUDA Tutorial - How to Start with CUDA?* Dec. 2018. DOI: 10.13140/RG.2.2.22460.49289.

[14] Benjamin Welton, Evan Samanas, and Barton Miller. “Mr. Scan: extreme scale density-based clustering using a tree-based network of GPGPU nodes”. In: Nov. 2013. DOI: 10.1145/2503210.2503262.