DBSCAN Parallelization on the GPU

1st Savannah Chappus
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
sjc497@nau.edu

2nd Kaitlyn Deacon
College of Environment, Forestry, and Natural Sciences
Northern Arizona University
Flagstaff, AZ, U.S.A.
krd327@nau.edu

3rd Nile Roth
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
ndr82@nau.edu

4th Adam Montano
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
ajm2327@nau.edu

5th Tomas Jauregui
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
taj262@nau.edu

***Abstract*—Algorithms that are capable of distinquishing between high and low density regions are especially benefitial in machine learning and unsupervised data mining. There are limitations to parallelizing the DBSCAN algorithm because of its inherent sequential operation. We acknowledge other work in the area, including that which parallelizes the algorithm for multi-core CPUs and many-core GPUs. In this paper, we propose a unique framework for GPU-accelerated density-based clustering that takes advantage of integration between the CPU and the GPU and navigates the sequential portions of the algorithm. In comparison with other work on the algorithm, our approach consists of four main aspects: (i) presorts the data to avoid expensive computations through large datasets, (ii) effectively divides work between the GPU and CPU to take advantage of their individual strengths, (iii) takes advantage of additional parameters to limit the searches for neighbors further than presorting the data does, and (iv) leverages additional data structures like the disjoint-set structure to effectively manage clustering on the CPU in a time-sensitive fashion. Additionally, we test our algorithm on two real-world datasets. Our optimized algorithm achieves a 8x speedup over baseline parallel GPU algorithm.**

***Index Terms*—DBSCAN, GPU, Parallel Clustering, Arbitrary Shape of Clusters, Efficiency on Large Spatial Datasets, Outlier Detection, Disjoint Set Data Structure**

I. Introduction

Clustering algorithms aim to organize data into clusters or groups based on similarities and inherent patterns within the data. They play important roles in processing and analyzing datasets in many areas such as pattern recognition, machine learning, and data mining by splitting a set of objects into disjoint classes [2].

Density-Based Spacial Clustering of Applications with Noise (DBSCAN) is a density-based clustering designed to take advantage of constructing clusters with arbitrary shapes [6], [7], [9], [10], [13]. This algorithm is efficient for large spacial databases with its robust behavior to noise through the use of **epsilon** and **minPts** parameters while omitting an input parameter for a required number of clusters [7].

The breadth-first search nature of DBSCAN along with its worst-case quadratic time complexity of $O(n^2)$ from distance computations between every two points, makes parallelization a challenge [7]. Due to the continuous increase in dataset sizes where n is the number of points, this technique may not scale well for large datasets [9]. Fortunately, there are several approaches that aim to address that drawback in part through spacial data structures to reduce the number of distance calculations.

Previous research focusing on utilizing multi-core CPU and GPU architectures to exploit the advantage of parallel computing, are readily available within a wide range of computer types. These existing works approach the scalability issues in a variety of ways ranging from spatial data structures like R-trees to the NVIDIA library of CUDA-DClust+ [2], [8], [12].

However, choosing the best GPU algorithm for a specific application is challenging and there is a lack of bench-marking and different datasets are used across previous studies [6]–[10], [13]. This paper reviews the existing DBSCAN algorithms specifically designed for GPUs and presents an experimental study of GPU-accelerated DBSCAN algorithm. For testing purposes, there were three datasets used: one generated dataset and two real-world datasets of over one billion data points pertaining to annual snowfall metrics and asteroid locations.

The paper is organized as follows: Section II establishes background information. Section III presents the algorithm and composed optimizations. Section IV displays performance results. Finally, Section V discusses indicated results and concludes the paper.

II. Background

A. G-DBSCAN: An Overview

The Density Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm is, as the name implies, clusters data using density specifications. Less-dense regions consider points as noise or outliers, while dense regions cluster points together according to parameter guidelines [9]. The **minPts** parameter indicates a point-density threshold, while the **Epsilon** (ϵ) parameter specifies a qualifying search distance to indicate a cluster region [9] [10]. However, since DBSCAN is inherently sequential, this severely limits parallelization opportunities [1], [10]–[12].

A concept proposed based on the original DBSCAN algorithm and clustering on GPUs is known as G-DBSCAN. This implementation strategy is divided into two steps: constructing density connections corresponding to the dataset in a graph-format and performing multiple breadth-first searches over the density graphs [6]. A drawback of the GPU is its limited global memory capacity, resulting in datasets that exceed global memory capacity and as such, must be divided into *n* partitions [1], [10].

B. Previous Studies of DBSCAN

There have been many studies addressing optimizations of DBSCAN to improve its efficiency including parallel multi-core CPU approaches [1], [6], [8], [12], though we will focus on ones directly related to GPU-accelerations.

Hybrid CPU/GPU Algorithm: One such optimization of DBSCAN introduces Hybrid-DBSCAN that utilizes both the GPU and CPUs to optimize throughput clustering [11]. The key innovation is leveraging the GPU's memory bandwidth for rapid index searching. In addition to memory leveraging, Hybrid-DBSCAN also manages data transfers between the GPU and the host to efficiently mitigate the negative impacts of memory buffers that incur a significant amount of allocation overhead from a large number of transfer batches [10].

Disjoint Set Data Structure: Proposed by Patwary et al. [1], a distributed-memory, multi-core DBSCAN algorithm based on the CPU had a major contribution by concurrently clustering a dataset in parallel with multiple processes clustering their own points [10]. The clusters were then merged into their final global clusters after all processes identified their local data points. This approach relied on using the disjoint set data structure where each point is considered to be within its own cluster initially [1], [7], [10], [12].

CUDA-DClust Implementation: A GPU-accelerated DBSCAN algorithm with several optimizations aims to improve performance by increasing efficiency of data mining interfaces that identify high and low-density regions [13]. Such enhancement is achieved by creating chains of points or sub-clusters that are density-reachable from each other (ie., they can be navigated by chaining together points that share neighbors with one another). This study computes the indexing structure on the GPU and utilizes kernel fusions for cluster expansion and index search combinations. This helps to reduce communication and overhead host synchronization. Our methods did not include CUDA-DCLUST+ however, by addressing the limitations of earlier designs it has significantly enhanced the efficiency and scalability of DBSCAN clustering for large datasets [7], [9], [11], [13].

III. Description of Methods used to Approaching the Problem

In this section we present our algorithm, GDBSCAN, that leverages the design of the DBSCAN clustering algorithm and the advantages of GPU-accelerated parallelization applications.

A. CPU Implementation

The CPU algorithm serves as a baseline for testing optimizations such as the GPU accelerations. It uses a sequential linear search to find the neighbors of each data point and construct a corresponding cluster array with a unique ID for each point by counting its respective neighbors. For ease of clustering, a disjoint-set data structure is used for refining the decision of merging points that belong to the same cluster.

TABLE I: Primary Notation of Variables used in this Study[1]

Variable	Definition
epsilon (ϵ)	qualifying distance radius between points for cluster regions
minPts	minimum number of neighbors required to define a *core point*
DIM	dimensionality of each point in the dataset
N	number of points in dataset
	$p_i = \{p_i^1, p_i^2, ..., p_i^{DIM}\}$ (1)
sortedDim	primary index of dimension on which the dataset is sorted
Core Point	point *p* that has at least *minPts* points within distance ϵ
Neighbor Point	point *p* that is within a specified distance ϵ from a *core point*
Border Point	point *p* which has fewer than *minPts* points within distance ϵ but is in the neighborhood of a core point
Noise Point	point *p* which is neither a border point or a core point

B. Baseline Kernel Implementation

In the baseline GPU implementation, the algorithm performs the neighbor search in parallel, with each thread representing a point in the dataset. This allows for concurrent calculation of neighbors compared to the entire dataset. After the GPU completes this kernel, it proceeds to create the *neighborsArray* for each point and assigns them to their respective clusters. Next,

[1]For ϵ and *minPts*, it is important to note that too large of an ϵ or too small of a value for *minPts* may result in a single cluster.

each point's array is processed by a thread on the GPU. Finally, after the GPU kernel completes its execution, the algorithm returns to a CPU kernel which handles combining clusters with common points using the disjoint-set data structure. Here, clusters are combined and reassigned based on the similarity of neighbors. Below is a detailed explanation of the kernels used in this implementation.

getNeighborFreqs: This kernel is step one of two for creating the neighbor array for the dataset. *getNeighborFreqs* is called with as many threads as there are points in the dataset. This allows for an easy division of labor (one thread per element), and each thread compares its designated point to each other point in the dataset. It does this by getting the difference in each dimension and running the distance formula between the two points. In each of these comparisons, the thread checks if the distance between the two points is less than the provided ϵ value. If it is, then these two points are considered neighbors, and it is stored in that threads local array of their designated point's neighbors. With this array we can easily count how many neighbors each thread (each point) has. Each thread stores their neighbors count into the *neighborFreqs* array **(see example 2)**. We must return from this kernel now so that we can ensure all threads in the grid complete their input in the *neighborFreqs* array before filling the neighbor position array.

getNeighborsArr: Entering this new kernel, we are ensured that the *neighborFreqs* array is completely populated and ready for access. Unfortunately the *localNeighbors* array associated with each thread must be repopulated due to the fact it was stored into registers and was lost when *getNeighborFreqs* returned. The same process from the last kernel is used to get each points' neighbors. The next step in the 'Get Neighbors' process is to create a neighbor position array. Each index is calculated by summing up the neighbor frequencies of all previous points before that index, resulting in its position in the neighbor array. Notice how the *neighborPos* array is created directly with the results of *neighborFreqs*. This is why we needed to ensure the entire *neighborFreqs* array is populated before beginning the *neighborPos* process. We can then use the *neighborPos* array to have each thread fill its point's neighbors into the correct foreseen position in the global neighbors array. Completing this entails completing the entire 'Get Neighbors' process. We now have the ability to compare neighbors and identify shared neighbors among points which is crucial for assigning clusters.

Disjoint Set Data Structure: In the DBSCAN implementation, a disjoint set data structure is employed to properly merge clusters. A disjoint set data structure has three operations: making a new set containing a single element *MAKE-SET*, finding the representative element of the set containing a given element *FIND-SET*, and merging two sets into one *UNION-SET*. The disjoint set data structure essentially maintains a forest of trees, where each tree is a set. The root of each tree is the representative element of the set. *UNION-SET* function works by making the root of one tree a child of the root of the other tree.

In the **expandClusters** kernel, which is run on the CPU, the disjoint set data structure merges clusters. First, a disjoint set is created with N elements, one for each data point. Then the function iterates through each point. If a point has at least *minPts* neighbors, then it forms a cluster. The function finds the set representatives for the point and its neighbors using *FindSet*, and merges them with *UnionSet*. The process ensures that all points in a cluster are in the same set. After processing all points, the final cluster labels are assigned based on the set representative of each point using *FindSet*.

C. Optimization 1: Presorted Dataset

The implementation of this method uses a sorted dataset as its input to enhance the time reduction in obtaining each point's neighbors. This input dataset must be sorted on one user-specified dimension as provided within argument 6 of the command line. For our dataset preparation, we utilized Python's Pandas library to presort our data with *read_csv* and *dfSortby* functions to read and sort any CSV file provided. The dimension with the largest variance is recommended for sorting to reduce the number of thread iterations.

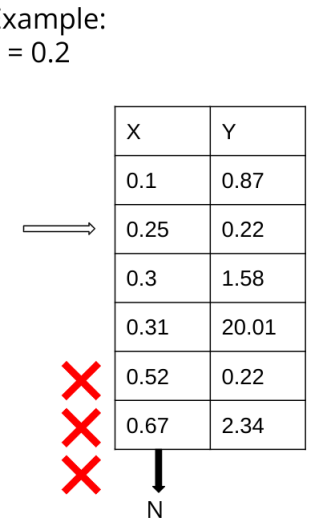

Fig. 1. Example of finding neighbors on sorted dataset.

By following the same process of both kernels, this optimization now obtains neighbors pertaining to each thread differently. In this case, each point neighbor is defined by having a distance less than ϵ from another specified point. Instead of comparing each thread's point to every other dataset point, a selective frame (or local array) of potential neighbors are obtained in its place.

The distance formula ensures that the difference between the sorted dimension of both specified points is greater than ϵ. As such, only points with a difference in the sorted dimension less than ϵ are potential neighbors, as can be seen in Fig. 1. This process must occur by iterating in both directions from the considered point for clustering purposes. This optimization

significantly reduces runtime as both kernels rely on obtaining a local neighbors array. With the proper ϵ value, each thread can perform up to 90% less work compared to the baseline kernel.

Algorithm 1 Algorithm for Sorted Data Kernel

procedure getNeighborFreqs(dataset, ϵ, N, DIM, minPts, neighborFreqs, neighborsArr, neighborPos)
if tid $<$ N **then**
  for all p_0 with sortedDimensionDifference $\leq \epsilon$ **do**
    distance $\leftarrow$ *getDistance()*
    if distance $\leq \epsilon$ **then**
      numNeighbors $\leftarrow$ numNeighbors + 1
      neighborFreqs[tid] $\leftarrow$ numNeighbors
    end if
  end for
end if
return
procedure getNeighborsArrs(dataset, ϵ, N, DIM, minPts, neighborFreqs, neighborsArr, neighborPos)
if tid $<$ N **then**
  for all p_0 with sortedDimensionDifference $\leq \epsilon$ **do**
    distance $\leftarrow$ *getDistance()*
    if distance $\leq \epsilon$ **then**
      storeNeighborLocally()
    end if
  end for
  for p_0 in *neighborFreqs* $<$ tid **do**
    startIndexInNeighborArr += neighborFreqs
  end for
  neighborPos $\leftarrow$ startIndexInNeighborArr
  for p_0 in *localNeighbors* **do**
    neighborsArr $\leftarrow$ localNeighbors
  end for
end if
return
procedure expandClusters(dataset, N, minPts, neighborFreqs, neighborsArr, neighborPos, clusterLabels)
for all p_0 in *dataset* **do**
  if numNeighbors $\geq$ minPts **then**
    expandClusterToNeighbors()
  end if
end for
for p_0 in *dataset* **do**
  assignClusterID()
end for
return

Note that Algorithm 1 fundamentally works the same in both the baseline and limited loop iterations implementations, with minor differences as explained in their respective sections.

D. Optimization 2: Limited Loop Iterations

This implementation is a very slight, yet crucial change to Optimization 1. In all previous modes, we are obtaining all neighbors of each point. With the logic of this implementation, we only obtain *minPts* neighbors for each point. *minPts* is generally a small, one or two digit number that is provided by the user. This variable defines what it means for a point to be a core point. A core point is a point with at least *minPts* neighbors. These core points share their *cluster ID* with all of its neighbors. If a point has less than *minPts* neighbors and is not neighbors with any core points, it is considered noise, and does not belong to a cluster. We noticed that finding just *minPts* neighbors would suffice for our clustering algorithm. Finding more than *minPts* neighbors, generally leads to more accurate clustering, but with the power of *unionSets()* from our disjoint set data structure, we will still receive similarly accurate results from Optimization 1.

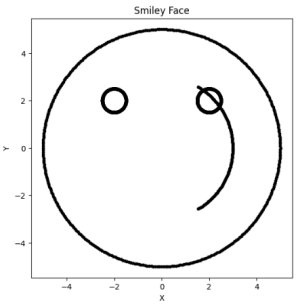

Fig. 2. Un-clustered DBSCAN Result of Engineered Dataset

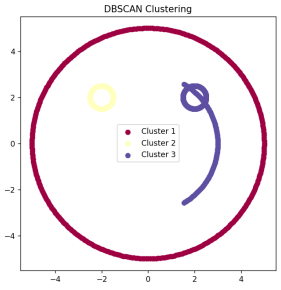

Fig. 3. Clustered DBSCAN Result of Engineered Dataset

E. NVIDIA Thrust Library Implementation Attempt

In this kernel, we aimed to utilize NVIDIA's Thrust library in order to more efficiently parallelize and expedite the calculation and fetching of each point's local neighbors for the global neighbor array. Due to the size of the datasets used in DBSCAN algorithms, computing and storing a Euclidean distance matrix is not feasible. Rather than computing a distance matrix, we opted for a "global neighbors array" which would hold each point in the dataset followed by all of its neighbors (based on ϵ).

Each point finds its local neighbors within ϵ, stores them in the global neighbor array, then sends that to the CPU for clustering operations. In our project, we hoped to use Thrust to expedite the process of finding the local neighbors of each point rather than (how it was done in mode 2).

We hoped to make use of Thrust's *lower_bound* and *upper_bound* functions which find the smallest and largest indices, respectively where the value passed in can be inserted into the data provided without disrupting the ordering of the array. This allows us to use a binary search that returns an index in the array even if the element does not actually exist in the array (see figure x). The pseudocode of the *getLocalNeighbors* function that utilizes the Thrust library is given in Algorithm 2.

The *getLocalNeighbors* function starts by taking in the data (*dataset*), number of points being considered (*N*), epsilon (ϵ) and the minimum points needed for a cluster (*minPts*). The

function then creates a thread for each point, and each thread computes its lower (*lowerBound*) and upper (*upperBound*) bounds based on epsilon (ϵ).

Then, in lines 5 and 6, we utilize Thrust's built-in functions for computing the lower and upper bounds as described earlier, and stores these in *lowerIndex* and *upperIndex* respectively. Once these have been computed, we copy the range between *lowerIndex* and *upperIndex* and store it in a local neighbors array for each point (*localNeighbors*).

Algorithm 2 Algorithm for obtaining local neighbors using Thrust

```
1: function GetLocalNeighbors(dataset, minPts, ϵ, N)
2: for all a_i in dataset do
3:     lowerBound ← a_i − ϵ
4:     upperBound ← a_i + ϵ
5:     lowerIndex ← lower_bound(dataset, lowerBound)
6:     upperIndex ← upper_bound(dataset, upperBound)
7:     CopyLocalNeighbors(lowerIndex, upperIndex, localNeighbors)
8: end for
9: return
```

The first issue we encountered was understanding how to incorporate Thrust into what we had already written. This meant including the necessary Thrust libraries and updating our Makefile. Once we were able to utilize the Thrust library, we needed to rearrange how our data was stored. On import, the CSV files are read in and stored in a linearized array row by row, so point 1 occupies the first (dimension) places in the array, and point 2 occupies the next (dimension) places and so on.

Since Thrust's *lower_bound* and *upper_bound* functions expect a linearly sorted section of an array, we needed to rearrange our dataset so that it was organized by dimension rather than by N.

TABLE II: Example Dataset of 5 Points and 3 Dimensions

Variable Imported Data
[a0, b0, c0, a1, b1, c1, a2, b2, c2, a3, b3, c3, a4, b4, c4]
Rearranged Data
[a0, a1, a2, a3, a4, b0, b1, b2, b3, b4, c0, c1, c2, c3, c4]

Once the data was in the proper format, we had to figure out how handle the data using Thrust. Thrust's library comes with both host and device vectors, which are to be used on the CPU and GPU respectively. In this open-source library, you cannot initialize device vectors directly on the GPU, and so we had to initialize a device vector and pass it in as a parameter to our GPU kernel.

Once in the kernel, we were able to write statements that made calls to *lower_bound* and *upper_bound* by specifying their occurrence on the device rather than the host. However, the results were consistently 0 for the indices for both the lower insertion location and the upper. Initially, we thought this meant the data had not been properly transferred to the device. As we continued researching the Thrust library and what functions and data types are available in the library, we discovered that the functionality for launching *upper_bound* and *lower_bound* calls on the device have been deprecated [14]. Please refer to section V for our comments about this.

IV. Results

A. Experimental Methodology

Our platform consists of 4x Intel Xeon Gold 6132 2.6 GHz CPUs with 28 total physical cores. The platform is also equipped with an NVIDIA Tesla V100 SXM2 16 GB GPU and runs CUDA 12.3 [4]. The host code is written in C/C++. All source code including the reference implementations are compiled with the O3 optimization flag, and experiments are averaged over three trials.

We would like the reader to also note the following about the timings reported:

1. All timings reported are on the order of seconds.
2. The speedups given in the tables are with respect to the baseline implementation (Mode 1) and are only given for the fastest block size.
3. In tables III and IV, Mode 1 refers to the **Baseline** implementation, Mode 2 refers to the **Sorted Data** optimization, and Mode 3 refers to the **Limited Loop Iterations** optimization. 4. Tables III and IV are tested using the asteroid dataset, which contains 5,000,000 data points.
5. We have not reported any CPU metrics because the CPU implementation was predicted to take days to complete the process.
6. In table V, 'Freqs' and 'Arr' refer to each mode's respective getNeighborsFreqsXXX kernel and getNeighborsArrXXX kernel.

TABLE III
Performance of Implementation and Optimization

Block Size	**Implementation**		
	Mode 1	Mode 2	Mode 3
32	99.552	49.354	11.714
64	96.588	49.122	11.136
128	96.686	49.455	11.142
256	97.758	50.050	11.080
512	98.259	49.912	11.241
1024	99.386	49.829	11.086
Speedup*	1.0	2.017	8.965

*Speedup is given for the fastest block size only

To better understand how our implementation performs, we used NVIDIA's Nsight Compute Profiler to analyze the performance of the different modes and their respective kernels. Table V provides insights into the achieved occupancy, memory throughput, and compute throughput for each kernel in each mode. Mode 1 shows a high achieved occupancy of 91.17%

TABLE IV
PERFORMANCE OF GPU KERNELS ONLY

Block Size	**Implementation**		
	Mode 1	Mode 2	Mode 3
32	0.001586	0.001207	0.000392
64	0.001253	0.001226	0.000382
128	0.001001	0.001316	0.000403
256	0.001332	0.001209	0.000403
512	0.001444	0.001204	0.000382
1024	0.001281	0.001298	0.000394
Speedup*	1.0	1.314	4.046

*Speedup is given for the fastest block size only

and 77.71% for the *getNeighborFreqs* and *getNeighborsArr* kernels, respectively. However, the memory throughput and compute throughput for Mode 1 are relatively lower compared to Mode 2. Mode 2 exhibits the highest achieved occupancy, with 94.36% for *getNeighborFreqs* and 97.9% for *getNeighborsArr*. It also demonstrates improved memory throughput and compute throughput compared to Mode 1, indicating better utilization of GPU resources. Mode 3 has significantly lower achieved occupancy and throughput metrics compared to the other modes. Comparing the profiler metrics with the response time tables, it can be observed that Mode 2 shows a significant speedup over Mode 1, both in terms of total time and GPU time. The higher achieved occupancy and improved throughput metrics of Mode 2 contribute to its better performance. Mode 3 further improves upon Mode 2, achieving the lowest response times overall. However, the profiler metrics for Mode 3 are relatively lower, suggesting that factors other than occupancy and throughput, like processing the minimum neighbors, play a crucial role in its performance gains. Overall, the Nsight Compute Profiler provides valuable insights into the performance characteristics of the different modes. Mode 2 demonstrates improved occupancy and throughput compared to Mode 1, resulting in faster execution times. Mode 3, despite having lower profiler metrics, achieves the best overall performance.

TABLE V
NVIDIA NSIGHT COMPUTE PERFORMANCE METRICS

	Implementation			
Kernel	**Metric**	Mode 1	Mode 2	Mode 3
Freqs	Achieved Occupancy	91.17	94.36	29.67
	Memory Throughput	55.67	75.25	39.89
	Compute Throughput	78.93	80.33	45.32
Arr	Achieved Occupancy	77.71	98.04	49.60
	Memory Throughput	63.08	57.17	51.35
	Compute Throughput	71.95	57.13	51.35

One of the concerns in our implementation of this algorithm was how to verify that our results on the GPU were correct - something that requires visualizing the data in some way. To verify the validity of our implementation, we also wrote a python script that could plot the dataset so that we'd be able to visualize the clustering. Because of the large data size for the test datasets, the python and CPU implementations could not realistically validate the clustering on asteroid and snow datasets. To validate that the CUDA and C implementations are correct, they were tested with the the sorted_smiley.csv dataset which has the following parameters: **N** = 1199 data points, ϵ = 1.0, and **minPts** = 5. This results in 3 clusters. The original dataset:

V. DISCUSSION & CONCLUSIONS

In conclusion, we implemented the DBSCAN algorithm with three distinct modes tailored to different optimization strategies, each designed to efficiently cluster large datasets.

Our first implementation (Mode 1) was the baseline GPU implementation, which parallelized the 'Get Neighbors' process, resulting in significantly improved performance compared to the CPU version. The second implementation (Mode 2) builds upon the baseline by utilizing a sorted dataset, allowing for a more efficient neighbor search process. This optimization drastically reduces the workload for each thread, leading to further speedups. The third implementation (Mode 3) introduces a slight modification to Mode 2, but is able to reduce this workload even further by setting a max number of neighbors provided by the user. Despite lower profiler metrics, Mode 3 achieves the best overall performance with a speedup of over 8x that of our baseline kernels (Mode 1).

We tested our implementation on many datasets including an asteroid data set containing the longitude and latitude of 5,000,000 asteroids in the Kuiper belt, and snow data with 12 dimensions! While validation against a Python implementation was achieved with smaller datasets, the large-scale datasets were left to be validated by data analysis. The consistency of cluster labels against the neighbor arrays in our output proved to be accurate.

In addition to our successful GPU kernel implementations, we also attempted to further improve our performance by utilizing NVIDIA's Thrust library. Upon discovering the deprecated nature of calling Thrust's lower and upper bound functions on the GPU, we note here that the NVIDIA C++ Standard Library (of which Thrust is a part of) is an open source project [3], and as such, a solution to the deprecated functionality may be provided in the future, in which case, we can pursue implementing Thrust's library in the ways described earlier in this paper. Overall, we are very pleased with the outcome of this DBSCAN implementation.

ACKNOWLEDGMENT

We thank the National Centers for Environmental Information for the Global Summary of the Year dataset, which we refined for our snow dataset. We also thank Dr. Gowanlock for his assistance in the implementation of this algorithm and for the asteroid data that we also used.

REFERENCES

[1] A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 2012, doi: https://doi.org/10.1109/sc.2012.9.

[2] A. Prokopenko, D. Lebrun-Grandie, and D. Arndt, "Fast tree-based algorithms for DBSCAN for low-dimensional data on GPUs," arXiv.org, Aug. 07, 2023. https://arxiv.org/abs/2103.05162. (accessed April 25, 2024)
[3] CCCL: CUDA C++ Core Libraries (2023), CCCL Development Team. Accessed Apr 25, 2024. [Online]. Available: https://github.com/NVIDIA/cccl
[4] "CUDA Toolkit Documentation 12.1," docs.nvidia.com. https://docs.nvidia.com/cuda/ (accesed May 06, 2024).
[5] "Dataset Overview — National Centers for Environmental Information (NCEI)," www.ncei.noaa.gov. https://www.ncei.noaa.gov/metadata/geoportal/rest/metadata/item/gov.noaa.ncdc:C00947/html# (accessed April 25, 2024).
[6] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering," Procedia Computer Science, vol. 18, pp. 369–378, 2013, doi: https://doi.org/10.1016/j.procs.2013.05.200.
[7] H. Mustafa, E. Leal, and L. Gruenwald, "An Experimental Comparison of GPU Techniques for DBSCAN Clustering," Dec. 2019, doi: https://doi.org/10.1109/bigdata47090.2019.9006169.
[8] M. Chen, X. Gao, and H. Li, "Parallel DBSCAN with Priority R-tree," Jan. 2010, doi: https://doi.org/10.1109/icime.2010.5477926.
[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." Available: https://cdn.aaai.org/KDD/1996/KDD96-037.pdf
[10] M. Gowanlock, "Hybrid CPU/GPU clustering in shared memory on the billion point scale," Jun. 2019, doi: https://doi.org/10.1145/3330345.3330349.
[11] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius, "A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 4, pp. 766–777, Apr. 2019, doi: https://doi.org/10.1109/tpds.2018.2869777.
[12] M. Gowanlock, D. M. Blair, and V. Pankratius, "Optimizing Parallel Clustering Throughput in Shared Memory," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 9, pp. 2595–2607, Sep. 2017, doi: https://doi.org/10.1109/tpds.2017.2675421.
[13] Madhav Poudel and M. Gowanlock, "CUDA-DClust+: Revisiting Early GPU-Accelerated DBSCAN Clustering Designs," Dec. 2021, doi: https://doi.org/10.1109/hipc53243.2021.00049.
[14] "[NVBug 3298282] Device-side CDP launch of 'thrust::upper_bound' returns incorrect result. · Issue 767 · NVIDIA/cccl," GitHub. https://github.com/NVIDIA/cccl/issues/767 (accessed May 06, 2024).

APPENDIX A

This section outlines how to run the code as well as an explicit outline of what parameters are expected. Our main function takes in many arguments that must all be populated in order for the algorithm to run.

The archive includes the following files:

dbscan_allModes.cu
dbscan_allModes.sh
dbscan.sh
Disjoint_set.h
sorted_smiley.csv
sorted_AST2.csv
sorted_AST1.csv
snowDataPreProcessed.csv

dbscan_allModes.sh is the job script to run all the above files, and contains different commented parameters for testing the code.

dbscan.sh is the job script to run each mode separately corresponding to the user selection.

These are run as directed below:

srun ./dbscan *N DIM epsilon minPts filename sortedDim*

N: the number of data points in the dataset
DIM: the number of dimensions. More specifically, the number of attributes associated with each point. For example, a database of library members has attributes ID, name, home address, and email. This dataset has a dimension of 4.
Epsilon (ϵ): a radius that defines a neighbor. A circle is drawn around each point with a radius of the supplied epsilon. If another point falls within this circle, it is considered a neighbor to that point. Identifying neighbors is crucial for the clustering process. *Note:* too large of an epsilon may result in a single cluster.
minPts: the number of neighbors that defines a core point. A core point expands its cluster ID to all of its neighbors. *Note:* too small of a minPts value may result in a single cluster.
fileName: name of the file stored in the current directory, generally a .csv or .txt file.
sortedDim: the index of the dimension on which the dataset is sorted.