Clustering Throughput Optimization on the GPU

Michael Gowanlock, Cody M. Rude, David M. Blair, Justin D. Li, Victor Pankratius
Massachusetts Institute of Technology, Haystack Observatory
[*gowanloc, cmrude, dblair, jdli, pankrat*]*@mit.edu*

***Abstract*—Large datasets in astronomy and geoscience often require clustering and visualizations of phenomena at different densities and scales in order to generate scientific insight. We examine the problem of maximizing clustering throughput for concurrent dataset clustering in spatial dimensions. We introduce a novel hybrid approach that uses GPUs in conjunction with multicore CPUs for algorithmic throughput optimizations. The key idea is to exploit the fast memory on the GPU for index searches and optimize I/O transfers in such a way that the low-bandwidth host-GPU bottleneck does not have a significant negative performance impact. To achieve this, we derive two distinct GPU kernels that exploit grid-based indexing schemes to improve clustering performance. To obviate limited GPU memory and enable large dataset clustering, our method is complemented by an efficient batching scheme for transfers between the host and GPU accelerator. This scheme is robust with respect to both sparse and dense data distributions and intelligently avoids buffer overflows that would otherwise degrade performance, all while minimizing the number of data transfers between the host and GPU. We evaluate our approaches on ionospheric total electron content datasets as well as intermediate-redshift galaxies from the Sloan Digital Sky Survey. Our hybrid approach yields a speedup of up to 50x over the sequential implementation on one of the experimental scenarios, which is respectable for I/O intensive clustering.**

***Keywords*-DBSCAN, Parallel Clustering, GPGPU, Query Optimization, Computer-Aided Discovery.**

I. Introduction

Applications in many domains utilize clustering algorithms such as Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [1]. Optimizations have been proposed, such as parallel implementations for multicore and manycore shared and distributed memory architectures that rely on a variety of indexing techniques, such as the R-tree [2], and grids [3].

DBSCAN takes as input two parameters: (i) the distance, ϵ, that is searched within the neighborhood of a point object; and (ii) the number of neighboring points, *minpts*, within ϵ that are required for each point to be a member of a cluster. To cluster a dataset, each point necessitates a search of its neighborhood, making index searches a significant fraction of the total computation. Table I shows the fraction of time spent searching an R-tree when executing a sequential implementation of DBSCAN [4] for various parameters and datasets from space weather (*SW-*) in geoscience and the Sloan Digital Sky Survey (*SDSS-*) in astronomy (see Section VII for more details), which ranges between 48.0% and 72.2% of the total response time. In this work, we examine how GPUs can reduce index search time and thereby improve the performance illustrated in Table I.

Table I: Fraction of total execution time spent searching the R-tree (*minpts* $= 4$).

Dataset	ϵ	Frac. Time	Dataset	ϵ	Frac. Time
SW1	0.20	0.522	*SDSS1*	1.40	0.480
SW1	1.40	0.483	*SDSS2*	0.15	0.679
SW4	0.15	0.525	*SDSS2*	0.45	0.512
SW4	0.45	0.510	*SDSS3*	0.07	0.722
SDSS1	0.20	0.703	*SDSS3*	0.12	0.629

To find spatial data defined by points, we focus on grid-based index optimizations for the GPU. As the search is constrained by a fixed ϵ, a grid can be constructed such that only neighboring grid cells need to be examined. This is naturally suited to the GPU, since the dimensions of the grid are constructed to account for a particular value of ϵ, the search is bounded and few branch instructions are needed, which reduces kernel thread divergence.

Other works that parallelize DBSCAN on the GPU [5], [6], [7] perform the clustering process in parallel, and then merge subclusters to make the final (larger) clusters. This process requires finding the neighbors of each point that are within ϵ. In contrast, we propose a hybrid approach that finds all of the direct neighbors of each point on the GPU and then sends this neighbor list to the host to perform the clustering. This approach relies on our efficient batching scheme that we propose to obviate memory limitations on the GPU. Furthermore, preprocessing the direct neighbors allows us to reuse the data to cluster with different parameters and thereby obtain clusters at different densities and scales.
This paper makes the following contributions:

- We propose and evaluate two GPU kernels that use a grid-based index to calculate the direct neighbors of the points to be used as input to DBSCAN.
- Major limitations in GPU computing are the host-GPU data transfers and limited memory. We develop an efficient batching scheme for DBSCAN to obviate the GPU memory constraints that overlaps computation and communication between the host and GPU.
- Combining the above, we introduce HYBRID-DBSCAN, and show its utility under three scenarios as follows. (a) we use it to cluster data with a single set of parameters; (b) a multi-clustering pipelined approach to maximize clustering throughput across a range of parameters; and

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.17

(c) a data reuse scheme that exploits the ϵ-neighborhood of each point being clustered.

- We evaluate our approaches on real-world space weather and galaxy datasets. Our approach outperforms the reference implementation of DBSCAN across a broad range of application scenarios.

The paper is organized as follows. Section II introduces DBSCAN and related work. Section III outlines the problem. Indexing methods and GPU kernels are outlined in Section IV. The hybrid approach and efficient batching scheme are detailed in Sections V and VI. The evaluation is given in Section VII and we conclude in Section VIII.

II. BACKGROUND

A. The DBSCAN Algorithm

We provide an outline of the DBSCAN algorithm and refer the reader to [1] for more details. Let D be a database of points to be clustered, where a point $p \in D$. The point p may be defined in arbitrary dimensions; here, we focus on spatial dimensions. The distance ϵ determines the ϵ-neighborhood of p, $N_\epsilon(p) = \{q \in D | dist(p,q) \leq \epsilon\}$. The distance function $dist(p,q)$ can be an arbitrary distance function (e.g., Euclidean). For a given point p, if $|N_\epsilon(p)| \geq$ *minpts*, p is referred to as a *core point*. There are a number of reachability criteria in the DBSCAN algorithm. First, given a core point p and a point $q \in D$, $q \in N_\epsilon(p)$ is *directly density reachable* from p, as it lies within the ϵ-neighborhood of p. However, points $(p,q) \in D$ may be *density reachable* if they are part of a chain of connected points belonging to a cluster, i.e., $p_1, \ldots, p_n$, where p_{i+1} is directly density reachable from p_i, for each of $1 < i < n$, $p = p_1$, and $q = p_n$. Points $(p,q) \in D$ are *density connected* if there exists a point $r \in D$, that is density reachable to both p and q. Finally, a point $p \in D$ is considered a *border point* if it is not a core point but is density reachable from another core point. Border points are assigned to clusters, and points unreachable by core points are outliers.

DBSCAN [1] is outlined in Algorithm 1. The algorithm takes as input: (i) the database of points to be clustered (D); (ii) the threshold distance (ϵ); (iii) the minimum number of points, *minpts*, within the ϵ-neighborhood to be considered a core point; and (iv) an index I (e.g., an R-tree [2]). We initialize four sets (lines 2-5): *visitedSet* stores the points that have already been visited by the algorithm, *clusterSet* is the set of points that have been assigned to a cluster, *noiseSet* is the set of outlier points, and $\mathcal{C}$ which is the set of independent clusters output by the algorithm. The algorithm examines all points in D that have not yet been visited (line 6). A new cluster set is initialized (line 7), and a given point is added to the visited set (line 8) denoting that the point has been visited. Next, the set of neighbors within ϵ are found by searching D (line 9) using an index. The algorithm determines if it is a core point having at least *minpts* within ϵ (line 10). If there are an insufficient number of neighbors, the point is marked as noise. If it is a core point, it is added to the set of points that form the current cluster (line 12), and to the set that accounts for those points that have been clustered (line 13). The algorithm loops over each of the directly density-reachable neighbors of the point defined by the set N (line 14). If a point $i \in N$ has not been visited (line 16), then it is marked as such (line 17). The neighbors of point i are found (line 18), and if there are a sufficient number, $|\hat{N}| \geq$*minpts*, the neighbors are examined to determine if they are part of the cluster (lines 19-20). The point is added to the cluster if it has not yet been assigned to one (lines 21-23). The algorithm outputs the set of individual clusters, $\mathcal{C}$, and the set of noise points.

Algorithm 1 The DBSCAN Algorithm.

```
1:  procedure DBSCAN($D$, $\epsilon$, minpts, Index $I$)
2:      visitedSet $\leftarrow \emptyset$
3:      clusterSet $\leftarrow \emptyset$
4:      noiseSet $\leftarrow \emptyset$
5:      $\mathcal{C} \leftarrow \emptyset$
6:      for all $p \in D | p \notin$ visitedSet do
7:          $C \leftarrow \emptyset$    $\triangleright$ $\mathcal{C}$ contains multiple clusters ($C$).
8:          visitedSet $\leftarrow$ visitedSet $\cup \{p\}$
9:          $N \leftarrow$ NeighborSearch($p, \epsilon, I$)
10:         if $|N| <$ minpts then noiseSet $\leftarrow$ noiseSet $\cup \{p\}$
11:         else
12:             $C \leftarrow C \cup \{p\}$
13:             clusterSet $\leftarrow$ clusterSet $\cup \{p\}$
14:             for all $i \in N$ do
15:                 $N \leftarrow N \setminus i$
16:                 if $i \notin$ visitedSet then
17:                     visitedSet $\leftarrow$ visitedSet $\cup \{i\}$
18:                     $\hat{N} \leftarrow$ NeighborSearch($i, \epsilon, I$)
19:                     if $|\hat{N}| \geq$ minpts then
20:                         $N \leftarrow N \cup \hat{N}$
21:                 if $i \notin$ clusterSet then
22:                     $C \leftarrow C \cup \{i\}$
23:                     clusterSet $\leftarrow$ clusterSet $\cup \{i\}$
24:         $\mathcal{C} \leftarrow \mathcal{C} \cup C$
25:     return
```

B. Related Work

Many studies have addressed improving the efficiency of DBSCAN [5], [8], [9], [6], [7], [10], [11], [4], but here we focus on those related to the GPU. CUDA-DClust [5] creates subclusters or chains of points that are density-reachable from each other. Multiple chains are constructed in parallel on the GPU and can make up a single cluster. When clustering in parallel, the algorithm keeps track of collisions that occur when two chains belong to the same cluster. After all points have been assigned to a chain, or labelled as noise, the algorithm resolves the collisions and assigns each point a cluster id. In [7], the authors reuse the main ideas of CUDA-DClust in [5], by making kernel optimizations that reduce host-GPU interaction. The GPU is part of a distributed memory implementation that scales to cluster billions of points. A GPU implementation is advanced in [6] that constructs graphs of points that are density-connected in parallel and then identifies the clusters by performing a breadth first search on the resulting graphs. In the above approaches, subclusters are formed and then are merged to form the final clusters.

Other prior work preprocesses points to find those within a given distance, or utilizes a distance matrix to find the distance between all pairs of points. A hybrid approach for DBSCAN [12] calculates the distance between all points in the dataset without an index, creating a symmetric distance matrix in $O(|D|^2/2)$. After the distance matrix has been computed on the GPU it is sent back to the host to be used for clustering. Research on k-nearest neighbor searches (kNN) [13] produces a distance matrix for the points on the GPU. To find the k nearest neighbors to a query point q, they use the merge path algorithm [14] for the GPU that uses an efficient truncated merge sort, and then selects the k nearest neighbors. An index is used to retrieve neighbors of a given point q in sublinear time. However, with high dimensionality, indexes become less efficient due to the curse of dimensionality [15], [16], [17], where a scan of all of the points in $O(|D|)$ time may be more efficient. The distance matrix in [13] is unlikely to be prohibitive because 64 dimensions are considered. The works above ([12] and [13]) are not applicable to our work because (i) we cannot accommodate a distance matrix because it would limit the size of the dataset that can be processed with a single kernel invocation, and (ii) since we focus on 2-D point objects, calculating the distance matrix would be inefficient in comparison to using an index.

With the proliferation of manycore architectures such as the GPU, new indexing approaches have been developed [18], [19], [17], [20], [21]. As noted in [18] and [20], indexes for the GPU may need to reach a trade-off between their complexity and selectivity. Many indexes developed for the CPU utilize trees that contain many branch conditions that can lead to thread divergence, thereby causing a loss of parallel efficiency [22] on the GPU. However, other work has addressed utilizing a GPU-friendly R-tree index [17] that can obtain high parallel efficiency by allowing for more regular memory accesses and without using recursion (which is limited by the stack on the GPU). The authors of [17] find that in comparison to other indexing techniques, their implementation is better when indexing fewer than 2^4 dimensions, and at 2^5 dimensions, the index is less efficient than an exhaustive scan. Other flatter indexing approaches for the GPU are advanced in [19], [20] and [21].

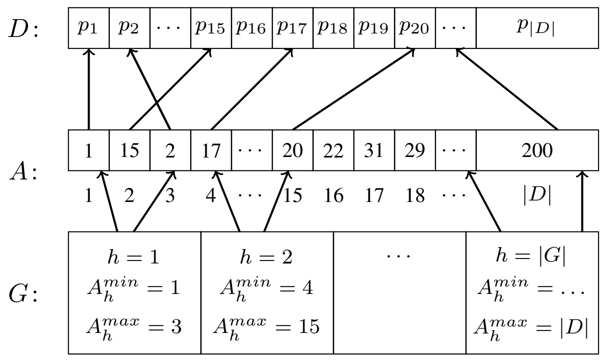
III. Motivation and Problem Statement

An overarching motivation is to maximize cluster throughput when clustering a dataset for a broad range of parameters, which is often required when examining datasets at different densities and scales. One such context is that of "Computer-Aided Discovery", which assists the researcher in examining a range of possible outcomes to assist in finding new phenomena [23].

Let D be a database of 2-D points to be clustered by DBSCAN. Each point p_i, $i = 1, \dots, |D|$, is defined by the coordinates $(x_i,\ y_i)$. We focus on grid-based index optimizations for GPU kernels. Instead of executing DBSCAN in parallel on the GPU, which is the focus of other works [5], [6], [7], we preprocess the 2-D data points in parallel, to find each of their neighbors within ϵ. We call this mapping of each $p_i \in D$ to each of its neighbors within ϵ the neighbor table, T. Each element in T is defined by $p_i \in D$ and the corresponding points within the ϵ neighborhood, $N_\epsilon(p_i)$. We then use the direct neighbors as input to the clustering algorithm instead of searching an index. We focus on this scenario as DBSCAN can be executed concurrently to maximize clustering throughput when clustering a dataset with varying input parameters. We define a variant as the input parameters to DBSCAN where we let V be a set of parameters defined as $v_i, i = 1, \dots, |V|$, where each v_i is defined by $(v_i^\epsilon, v_i^{minpts})$. We assume that there is sufficient memory to store all of the relevant data in memory on the host. However, as memory is limited on the GPU, we use a batched execution to construct T where appropriate.

IV. Indexing Methods

We outline our grid index for the GPU that is used to compute T (which is similar to an indexing scheme for trajectories [20]). We compute the index using D and ϵ. When searching the index, we improve locality when accessing D on the GPU by first binning $p_i \in D$ in x and y dimensions of unit width such that points in similar spatial locations will be stored nearby each other in memory. We then generate a grid, where each cell is of ϵ length in both the x and y dimensions and inserting points into these cells. We store the index, G, as an array of cells. Each cell is defined as C_h, $h = 1, \dots, |G|$, where h is a linear coordinate calculated from each cell's x and y coordinates. The point ids, i, of $p_i \in D$ found inside C_h are stored in a lookup array A as a range $[A_h^{min}, A_h^{max}]$, i.e., if a point p_i is located in C_h, then $i \in \{A[A_h^{min}], \dots, A[A_h^{max}]\}$. We use A because it minimizes memory usage; since a point can only be found inside a single cell, $|A| = |D|$. Alternatively, we could allocate a fixed amount of memory for each cell, C_h; however, this would require space for the cell with the greatest number of points, which would lead to a significant amount of wasted GPU memory. D, G, A, and ϵ are stored in global memory on the GPU. As an example, Figure 1 shows G, A and D, with cell C_1 containing a range into the lookup array A having the values [1, 3] that correspond to $p_i \in D$, where $i \in \{1, 15, 2\}$. Note that the points inside a cell are not guaranteed to be stored contiguously within D, whereas the indices from C_h into A are stored contiguously.



D:
p_1 p_2 $\cdots$ p_{15} p_{16} p_{17} p_{18} p_{19} p_{20} $\cdots$ $p_{|D|}$
A:
1 15 2 17 $\cdots$ 20 22 31 29 $\cdots$ 200
1 2 3 4 $\cdots$ 15 16 17 18 $\cdots$ $|D|$
G:
$h = 1$ $A_h^{min} = 1$ $A_h^{max} = 3$
$h = 2$ $A_h^{min} = 4$ $A_h^{max} = 15$
$\cdots$
$h = |G|$ $A_h^{min} = \ldots$ $A_h^{max} = |D|$

Figure 1: Grid indexing example, where G is the index array, A is the lookup array, and D is the database of points.

A. Global Memory Kernel

We outline a GPU kernel (Algorithm 2) that computes the ϵ-neighborhood of each point in D using an approach that does not use shared memory as implemented in CUDA. The global id for the thread is calculated from the thread id and block id (and block size) in a single CUDA memory dimension on line 2, and if the global thread id is larger than D the kernel returns (line 3). Two arrays (explained below) are initialized on line 4. The ϵ-neighborhood of each point in D is processed by a single thread, so we store the point in registers on line 5. There will be $|D|$ threads in total assuming the block size evenly divides $|D|$. For the point processed by the kernel, the linearized cell ids that should be checked are computed (line 6). Note that we elect to use cells of size $\epsilon \times \epsilon$, such that for a given point p_i, the points within ϵ are guaranteed to be within the point's cell or the adjacent cells (a maximum of 9 cells). We enter a loop on line 7 that iterates over each of the appropriate cells for a given point. For each cell, we compute the indices into the lookup array A of the points within the cell (lines 8-9). We iterate over these points in D (line 10) and calculate if they are within ϵ of the point assigned to the thread (line 5). If a point is within ϵ, it is added to the result set as a key/value pair, where the value is within ϵ of the key (line 13).

Algorithm 2 The GPUCalcGlobal Kernel.

```
1:  procedure GPUCalcGlobal($D$, $G$, $A$, $\epsilon$)
2:      gid ← getGlobalId()
3:      if gid $\geq |D|$ then return
4:      cellIDsArr ← $\emptyset$; resultSet ← $\emptyset$
5:      point ← $D$[gid]
6:      cellIDsArr ← getNeighborCells(gid)
7:      for cellID $\in$ cellIDsArr do
8:          lookupMin ← $A[G$[cellID].min]
9:          lookupMax ← $A[G$[cellID].max]
10:         for candidateID $\in$ {lookupMin,...,lookupMax} do
11:             result ← calcDistance(point, $D$[candidateID], $\epsilon$)
12:             if result $\neq \emptyset$ then
13:                 atomic: gpuResultSet ← gpuResultSet $\cup$ result
    return
```

B. Shared Memory Kernel

Algorithm 2 maps a single GPU thread to a point, $p_i \in D$, and finds those neighbors within ϵ only using global memory. In this section we outline a different approach that takes advantage of shared memory on the GPU (Algorithm 3). The main idea is that each thread block processes a single grid cell, where the points inside the block's cell and an adjacent cell (one that may contain points within ϵ of the points in the block's cell) are first paged into shared memory before the distance calculation occurs. This takes advantage of the shared memory bandwidth on the GPU. Because we need to page the data into shared memory from global memory, we tile this data transfer based on the granularity of the block size. Using this scheme, we are oblivious to the number of data points per cell, and we do not exceed the capacity of shared memory.

Algorithm 3 takes the same inputs as Algorithm 2 except that it also requires a schedule S that maps each block to a cell for processing, and a total number of threads, N. Since each block processes the points inside a single cell, the total number of threads, N, is the product of the number of non-empty grid cells and the block size, and not $|D|$ as in Algorithm 2.

For clarity, in Algorithm 3, we refer to CUDA keywords as follows: (a) we define arrays that are stored in shared memory using the *shared* keyword; (b) we refer to the block size as blockDim.x (we only use one memory dimension); (c) the thread id in a block is threadId.x; and (d) the block-level thread synchronization is denoted as *synchronize()*. Note that for illustrative purposes, we assume that the number of points in a given cell is at most the size of a block. If there are more points in a cell than the block size, then an additional loop is needed in Algorithm 3.

The algorithm begins similarly to Algorithm 2. The schedule S is checked on line 5 and the cell id assigned to the block is stored (only non-empty cells are processed). Lines 6-7 allocate arrays in shared memory that store the points processed by a single block and the points in adjacent cells, respectively. We call the points in the cell being processed by a block the origin cell (*pntsOriginCell*), and the points in an adjacent cell (including the origin cell) a comparison cell (*pntsCompCell*). Next, the first thread in the block retrieves the cell ids that are adjacent to the cell being processed, including the cell itself (lines 8-9), and the threads in the block are synchronized on line 10. Next, a loop is entered that iterates over all of the relevant cells (line 11). Each thread in the block copies a single point from the origin cell in global memory to shared memory using an offset into the lookup array A on lines 12-14. Had the number of points in the origin cell exceeded the block size, then another loop would be inserted that encompasses lines 12-22. Next, each thread loads a point from global into shared memory from a comparison cell on lines 15-17 and the threads in the block are synchronized (line 18). Then each thread compares a single point in the origin cell with all points in a given comparison cell (both of which are stored in shared memory) to find if they are within ϵ on lines 19-22. Comparing Algorithms 2 and 3, note that to exploit the shared memory on the GPU, GPUCALCSHARED requires more threads and block-level synchronization in comparison to GPUCALCGLOBAL.

Algorithm 3 The GPUCALCSHARED Kernel.

```
1:  procedure GPUCALCSHARED($D$, $G$, $A$, $\epsilon$, $S$, $N$)
2:      gid ← getGlobalId()
3:      if gid $\geq N$ then return
4:      shared cellIDsArr ← $\emptyset$; resultSet ← $\emptyset$
5:      cellToProc ← $S$[blockID]
6:      shared pntsOriginCell[blockDim.x]
7:      shared pntsCompCell[blockDim.x]
8:      if threadId.x = 0 then
9:          cellIDsArr ← getNeighborCells(cellToProc)
10:     synchronize()
11:     for cellID ∈ cellIDsArr do
12:         lookupOffset ← $G$[cellToProc].min + threadId.x
13:         dataID ← $A$[lookupOffset]
14:         pntsOriginCell ← pntsOriginCell ∪ {$D$[dataID]}
15:         compLookupOffset ← $G$[cellID].min + threadId.x
16:         compDataID ← $A$[compLookupOffset]
17:         pntsCompCell ← pntsCompCell ∪ {$D$[compDataID]}
18:         synchronize()
19:         for candidateID ∈ pntsCompCell do
20:             result ← calcDistance(pntsOriginCell[threadId.x],
                        pntsCompCell[candidateID], $\epsilon$)
21:             if result $\neq \emptyset$ then
22:                 atomic: gpuResultSet ← gpuResultSet ∪ result
    return
```

V. ALGORITHM OVERVIEW

Using the GPU kernels and indexing approach described above, an overview of the HYBRID-DBSCAN algorithm is as follows. The algorithm constructs the neighbor table, T, which is the ϵ-neighborhood of all $p_i \in D$. T is then used as input into a modified DBSCAN algorithm, which instead of taking as input ϵ and *minpts*, uses T and *minpts* as all of the direct neighbors are precomputed for a given ϵ. As discussed in Section III, for a fixed ϵ, T can be reused for multiple values of *minpts*.

We describe Algorithm 4 in greater detail. Let R be the result set returned from the GPU to the host which is stored as key value pairs. Each result item r_j, $j = 1, \ldots, |R|$, consists of a key and a value (k_j, v_j), where the key k_j is a point $p_i \in D$ and the value $v_j \in D$ is a point within ϵ of the key, i.e., $v_j \in N_\epsilon(k_j)$. R can be incrementally constructed if it and memory overheads exceed the size of global memory on the GPU (as will be described in Section VI). R (the result set buffer allocated on the host), *gpuResultSet* (the buffer for the result set on the GPU) and T (the neighbor table) are initialized on lines 2-4. Next, the index is constructed using D and ϵ as described in Section IV (line 5) and the GPU kernel is launched; either GPUCALCGLOBAL or GPUCALCSHARED can be used, although here we show GPUCALCGLOBAL (line 6). Note that if the GPUCALCSHARED is used, then we also compute and transfer S and N to the kernel as described in Section IV-B. In Algorithms 2 and 3, the kernel does not return *gpuResultSet* to the host. Since each key/value pair (k_j, v_j) may be stored in any order in global memory on the GPU, we leave the result set on the GPU after launching the kernel, and then use the CUDA Thrust [24] library to sort r_j by key (line 7), such that identical keys will be adjacent to each other in R. Finally, R is transferred back to the host into a pinned memory buffer that is used as a staging area. Next, we construct the neighbor table (line 8). There can be identical $k_j \in R$, as a point can have multiple neighbors within ϵ. We copy $v_j \in R$ from pinned memory to another buffer in memory on the host, denoted B.

We define T as having $\mathcal{T}_i^{min}$ and $\mathcal{T}_i^{max}$, where $i = 1, \ldots, |D|$, thus each point is an element in T. Then we find the minimum and maximum indices in k_j corresponding to a single key and store these values in T as $\mathcal{T}_i^{min}$ and $\mathcal{T}_i^{max}$, respectively. These are the points within the ϵ-neighborhood of p_i stored as a range of indices into array B denoted as $[\mathcal{T}_i^{min}, \mathcal{T}_i^{max}]$. Thus, if a point p_j is within ϵ of p_i, then $j \in \{B[\mathcal{T}_i^{min}], \ldots, B[\mathcal{T}_i^{max}]\}$.

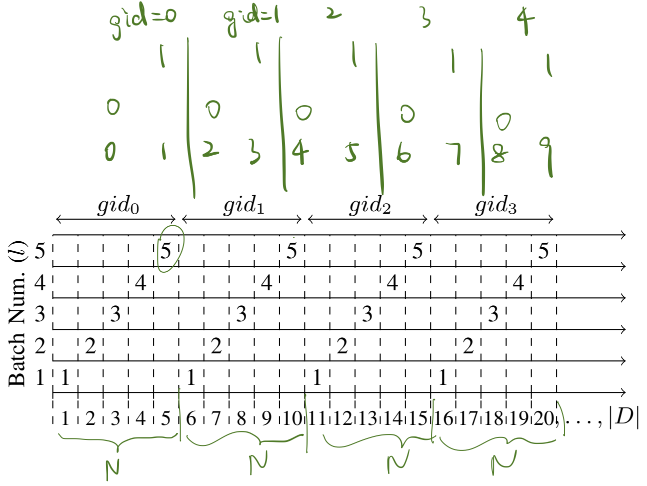
Using this scheme, we only copy the values to a buffer in memory from the pinned memory staging area without copying the keys. The reason we copy the values instead of leaving the data in the pinned memory location is because if multiple batches are required to generate T (which will be described in Section VI), then the pinned memory buffer needs to be available to write R for a future batch. After T has been constructed, DBSCAN is executed on line 9 which uses T and *minpts* (instead of ϵ and *minpts* as in Algorithm 1). The algorithm is modified such that the ϵ-neighborhood search (lines 9 and 18 in Algorithm 1) does not search the index I using ϵ, instead looking up the neighbors in T.

Algorithm 4 The HYBRID-DBSCAN Algorithm.

```
1:  procedure HYBRID-DBSCAN($D$, $\epsilon$, minpts)
2:      $R \leftarrow \emptyset$
3:      gpuResultSet ← $\emptyset$
4:      $T \leftarrow \emptyset$
5:      $(G, A)$ ← ConstructIndex($D$, $\epsilon$)
6:      GPUCALCGLOBAL($D$, $G$, $A$, $\epsilon$)
7:      $R$ ← gpuResultSet.SortByKey()
8:      ConstructNeighborTable($R$, $T$)
9:      DBSCAN($T$, minpts)
10:     return
```

VI. EFFICIENT BATCHING SCHEME

Due to the size of the dataset the data density and the value of ϵ, the result set size can be larger than the capacity of global memory on the GPU. To accommodate these result set sizes, we advance an efficient batching scheme to



0x5+5
1x5+5

Figure 2: Example assignment of points (x-axis) assigned to one of five batches using GPUCALCGLOBAL. The number above the point corresponds to the assigned batch number and the global thread id is shown for each respective kernel invocation.

incrementally compute the neighbor table T. Let n_b be the number of batches, a_b be an estimate of the total result set size across multiple batches, and b_b the buffer size allocated in global memory on the GPU for a single batch. If the result set from each batch is equal in size, and a_b is known exactly, then $n_b = a_b/b_b$. However, since the result set sizes will be different from each batch, and a_b is an estimate, then we need to overestimate a_b to ensure that the buffer on the GPU does not overflow. We introduce α as an overestimation factor. The equation to calculate n_b is as follows:

$$n_b = \left\lceil \frac{(1+\alpha)a_b}{b_b} \right\rceil. \tag{1}$$

Equation 1 relies on estimating a_b reasonably well and ensuring that the result set size between batches are fairly consistent in size. To minimize data transfer overhead, it is preferable to minimize n_b to reduce the number of transfers between the host and GPU. With a batched execution having multiple result set data transfers, we utilize pinned memory on the host to improve data transfer rates. However, this is at the expense of expensive pinned memory allocation [25]. Thus we want to minimize α for two reasons: (i) to not overallocate pinned memory; and (ii) to minimize the value of n_b. Given the above rationale, we advance a method to ensure a_b is estimated within a reasonable degree, and the result set sizes between batches are fairly consistent.

To calculate a_b we execute a kernel that counts the number of neighbors within ϵ of a uniformly distributed fraction f of $p_i \in D$ such that the kernel processes $f|D| < |D|$ points. The roughly uniform distribution of points occurs by simply sampling $f|D|$ points, since we initially sort D in both the x and y dimensions (Section IV). Since this kernel (similar to Algorithm 2 or 3) returns the number of neighbors within ϵ concerning a sample of the data points, which we denote e_b, and does not return a result set R (which requires significant overhead), the kernel executes once in negligible time. We let $f = 0.01$ (1% of $|D|$), thus $a_b = e_b \times 1/f$.

We describe an approach to ensure that the result set sizes are fairly consistent so that the buffer of size b_b does not overflow between batches. Let R_l be the result set from batch l, where $l = 1, \ldots, n_b$. We ensure $|R_l|$ is nearly the same size as b_b, i.e., $|R_l| \lesssim b_b$ (to maximize the usage of the buffer, and minimize n_b). As the result set size estimation value (e_b) takes a uniformly distributed sample of points, we execute the kernel (Algorithm 2 or 3) by computing the ϵ-neighborhoods of points that uniformly sample the dataset in each batch. In all that follows we refer to the GPUCALCGLOBAL kernel (Algorithm 2). Since the ϵ-neighborhood of $p_i \in D$ can be calculated in any order (i.e., there are no data dependencies between points), T can be constructed incrementally. Recall that in GPUCALCGLOBAL we assign a single thread to compute the ϵ-neighborhood of a single point. In Algorithm 2 we allow for the batching scheme by adding two additional inputs l (the batch number being executed) and n_b. We denote the global thread id described in Section IV-A as *globalID*. Thus on line 2 in Algorithm 2 (the point $p_i \in D$ being processed by the kernel) becomes *globalID*$\times n_b + l$, and line 3 returns if *gid* $\geq |D|/n_b$ (for illustrative purposes we assume that n_b evenly divides $|D|$). As an example, consider Figure 2 where $n_b = 5$. Each group of 5 adjacent points in D are processed by a different kernel invocation by accessing the data in a strided manner. Using this execution strategy, the values of $|R_l|$ are roughly consistent across batches. We set $\alpha = 0.05$, meaning we only overestimate b_b by $\gtrsim 5\%$. Since we take the ceiling in Equation 1, this is roughly the lower bound on the degree to which b_b will be overestimated.

Given that we batch the results from the GPU, we overlap the data transfers with kernel executions by assigning each batch to one of 3 CUDA streams (as we found that more streams achieved no performance gain). Therefore, in Algorithm 4, lines 6-8 are executed using 3 threads, where each thread launches the kernel, initiates sorting the result set as key/value pairs on the GPU using the Thrust API (which supports executions in streams), requests the result set data transfer from the GPU to the pinned memory staging area, and constructs a fraction of T on the host. The approach allows for significant overlap in both memory transfers between the GPU and host, within the memory on the host between the pinned memory staging area and pageable memory for the points in T, and in the construction of T. There is very little kernel execution overlap, as each invocation saturates GPU resources. Since 3 streams are used, we require 3 buffers in global memory on the GPU and in pinned memory on the host. When the result size estimation e_b is sufficiently large, we use a static buffer size, b_b, and if it is too small we use a variable buffer size as pinned memory allocation time can require a substantial fraction of the total response time for small datasets or ϵ values. When $e_b \geq 3 \times 10^8$ we set $b_b = 10^8$ (each stream has a buffer of this size), and if $e_b < 3 \times 10^8$ we set

$b_b = (e_b \times (1 + 2\alpha))/3$. We increase α by a factor of 2 for small result set sizes because the total result set size estimate e_b is more uncertain and there is more variability in $|R_l|$ between batches.

VII. Experimental Evaluation

A. Datasets

We utilize two real-world classes of datasets as follows.

- The *SW-* class consists of ionospheric total electron content datasets collected by GPS receivers (data preprocessing is described in [26]). *SW1* and *SW4* consist of 1,864,620 and 5,159,737 data points, respectively.
- The *SDSS-* class consists of a sample of galaxies from the Sloan Digital Sky Survey Data Release 12 [27] spanning a photometric redshift (z) of $0.30 \leq z \leq 0.35$. *SDSS1*, *SDSS2*, and *SDSS3* consist of 2×10^6, 5×10^6, and 15,228,633 data points, respectively.

We have chosen these datasets in the context of "Computer-Aided Discovery" where clustering can help researchers find new geophysical or astrophysical phenomena [23]. Regarding properties, *SW-* has many overdense regions as a function of the relative locations of GPS receivers; whereas *SDSS-* is more uniformly distributed. The *SW-* datasets are publicly available [28] and described in [4]. See [27] for *SDSS-* datasets.

B. Experimental Methodology

The multithreaded implementations of Hybrid-Dbscan on the host use OpenMP in C++ with the O3 compiler optimization flag. Executions use up to 16 dedicated cores on 2.40 GHz Intel Xeon E5-2630 v3 processors with 20 MB L3 cache. Response times are averaged over 3 trials. The GPU code is written in CUDA and executed on an NVIDIA Tesla K20c card with 5 GB of memory.

We select values of ϵ and *minpts* such that we do not get too few or two many clusters, which is a function of the dataset density and spatial distribution. We compare the performance to a reference implementation that executes a sequential version of Dbscan using an R-tree index on the CPU (details can be found in [4]). We do not report the time required to construct the index, as we did not optimize this in the reference implementation; however, we find that the grid indexes can be constructed faster than the R-tree.

C. Kernel Efficiency

We evaluate the performance of the two kernels, GPUCalcGlobal and GPUCalcShared. To assess efficiency, we only examine a single kernel invocation and do not analyze the other overheads, such as memory transfers. We obtain the metrics using the NVIDIA Visual Profiler. Table II shows the kernel response time and the total number of threads (n^{GPU}) launched by the kernels to process the datasets. n^{GPU} is the product of the total number of blocks launched by the kernel and the block size (set to 256). We use $\epsilon = 0.2$ on the datasets with $\sim 2 \times 10^6$ data points, and $\epsilon = 0.07$ with $\sim 5 \times 10^6$ data points. We decrease ϵ with increasing $|D|$ because when the number of data points increases, points are more likely to be chained together, generating fewer clusters.

Table II: Exp. Scenario 1 ($S1$) & Kernel Efficiency.

		GPUCalcGlobal		GPUCalcShared	
Dataset	ϵ	Time (ms)	n^{GPU}	Time (ms)	n^{GPU}
SW1	0.2	503.270	1,864,704	531.411	37,409,792
SW4	0.07	518.245	5,159,936	1258	255,272,704
SDSS1	0.2	72.677	2,000,128	544.745	110,757,120
SDSS2	0.07	80.038	5,000,192	1699	649,954,560

Comparing the response times in Table II, GPUCalcShared performs worse than GPUCalcGlobal on all datasets. The GPUCalcGlobal kernel assigns a single thread to find the ϵ-neighborhood of a point, thus the total number of threads required to process the kernel is roughly the same as the number of points in the dataset. With GPUCalcShared, a block is assigned to process each non-empty grid cell. Comparing the total number of threads across the datasets (n^{GPU}), Table II shows that GPUCalcShared uses far more threads than GPUCalcGlobal, and that a smaller ϵ value requires more threads because there are more non-empty cells. However, with smaller grid cells there are fewer points per cell, thus diminishing the opportunity to exploit shared memory per block. Assigning a grid cell to a block reduces GPU utilization due to increased overhead, particularly with a small ϵ.

The *SW4* dataset has many over-dense regions, whereas *SDSS2* is more uniformly distributed. We compare *SW4* to *SDSS2* (as they have roughly the same number of points). We see that GPUCalcGlobal is 143% faster than GPUCalcShared on *SW4*, and 2023% faster on *SDSS2*. Since *SDSS2* is more uniform, there are more grid cells containing points, requiring more thread blocks. The performance of GPUCalcShared is more sensitive to the data distribution, performing better on spatially skewed datasets. Furthermore, to reduce overhead, a block size should ideally be chosen to reflect the average data density, and we used a moderate value of 256. Overall however, GPUCalcShared performs poorly; therefore, in all that follows, we use GPUCalcGlobal. A potentially interesting future work direction would be to combine the two approaches such that GPUCalcShared processes the dense regions of a dataset and GPUCalcGlobal processes the remainder.

D. Performance of Hybrid-DBSCAN

We assess the performance of Hybrid-Dbscan (Algorithm 4) as compared to the reference implementation. As shown in Table III ($S2$), we individually execute Hybrid-Dbscan for 15 different parameters for *SW1* and *SDSS1*, and 9 for *SW4* and *SDSS2*, and 8 for *SDSS3*. We do not vary *minpts* because as will be shown, performance is determined by ϵ. Figure 3 plots the response time vs. ϵ. As ϵ increases, there are more distance calculations and larger result set sizes. The red curve shows the total time to execute HYBRID-DBSCAN, and the green and blue curves show the time required to construct T (the majority occurs on the GPU) and execute the modified version of DBSCAN, respectively. The time to construct T is roughly the same to execute DBSCAN across all datasets and ϵ values. Also, HYBRID-DBSCAN outperforms the reference implementation even with small values of ϵ and datasets (*SW1* and *SDSS1*). This is interesting because the GPU is usually ill-suited to small problem sizes. This suggests that despite slow host-GPU data transfers, HYBRID-DBSCAN is relevant across a broad range of application scenarios.

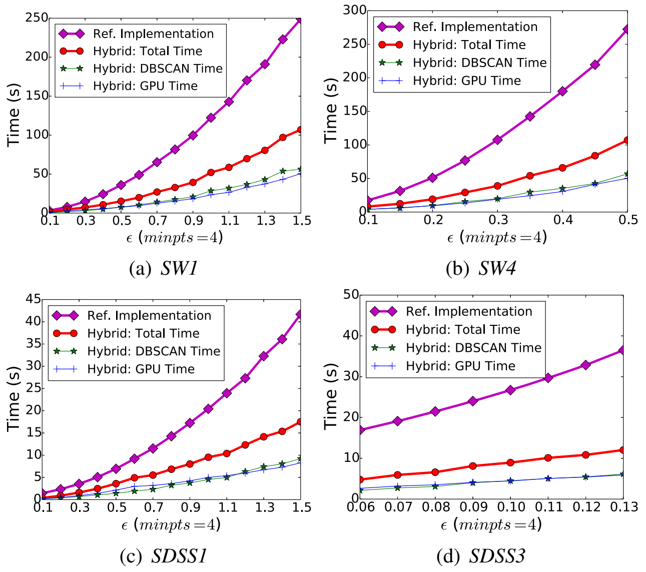


Figure 3: Response time vs. ϵ for HYBRID-DBSCAN and the reference implementation on scenario $S2$. GPU time refers to the time required to construct T, part of which occurs on the host. *SDSS2* has been omitted as its performance shows similar trends to *SDSS1* and *SDSS3*.

Table III: Scenario 2 ($S2$).

Dataset	v_i^{ϵ}	v_i^{minpts}
SW1	{0.1, 0.2, …, 1.5}	{4}
SW4	{0.1, 0.15, …, 0.5}	"
SDSS1	{0.1, 0.2, …, 1.5}	"
SDSS2	{0.1, 0.15, …, 0.5}	"
SDSS3	{0.06, 0.07, …, 0.13}	"

E. Multi-clustering Pipeline

Figure 3 illustrates how HYBRID-DBSCAN compares to the reference implementation when executing a single set of parameters. However, some applications may require executing DBSCAN for a broad range of input parameters. To maximize clustering throughput, we pipeline the construction of T and the DBSCAN phases of Algorithm 4 in a producer-consumer fashion; one thread executes lines 6-8 in Algorithm 4 for all of the variants in scenario $S2$ (see Table III) and another takes T as input into DBSCAN (line 9). The host that computes T (for a single ϵ) spawns 3 threads for batching (as described in Section VI), and up to 3 threads that can consume T for executing DBSCAN (although producing T and running DBSCAN require roughly the same response time as shown in Figure 3). Thus in the pipelined approach, when DBSCAN is being executed for v_i, T is being computed for v_{i+1}.

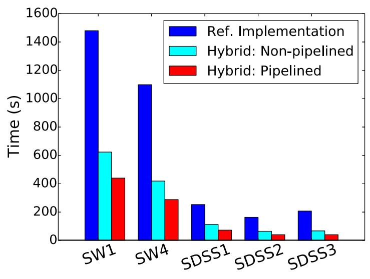


Figure 4: Comparison of the total response times of the three approaches on scenario $S2$.

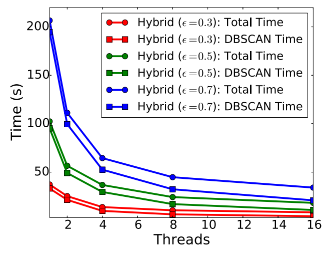
Figure 4 compares the response time of the reference implementation and non-pipelined HYBRID-DBSCAN (the sum of the times of the purple and red points in Figure 3) to the response time of the pipelined algorithm, and shows that pipelined HYBRID-DBSCAN is between 41.8% (*SW1*) and 66.5% (*SDSS3*) faster than the non-pipelined algorithm. Speedups over the reference implementation and non-pipelined version are shown in Table IV. Comparing the *SW*- and *SDSS*- datasets, both versions of HYBRID-DBSCAN perform better on *SDSS*- in comparison to the reference implementation, suggesting that HYBRID-DBSCAN is better for more uniform datasets. Furthermore, the largest dataset, *SDSS3* ($|D| \approx 1.52 \times 10^7$) yields the greatest performance gain over the reference implementation, suggesting that the performance of the R-tree begins to degrade with larger datasets, unlike HYBRID-DBSCAN which uses the grid-based index. Note that host-GPU data transfers are a significant bottleneck; therefore, future bandwidth increases will improve the relative performance of HYBRID-DBSCAN.

Table IV: Speedup using Pipelined HYBRID-DBSCAN on $S2$.

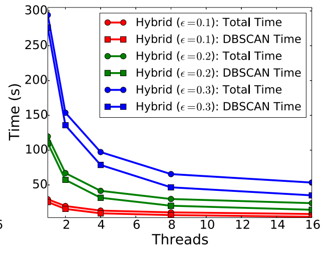
Dataset	Pipelined vs. Ref. Implementation	Pipelined vs. Non-Pipelined
SW1	3.36	1.42
SW4	3.81	1.45
SDSS1	3.48	1.56
SDSS2	4.04	1.60
SDSS3	5.13	1.66

F. Exploiting Data Reuse

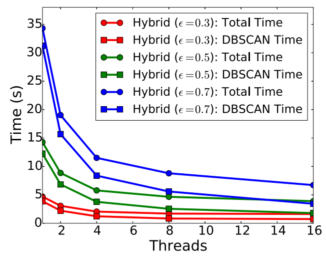
In the last section, we varied ϵ and kept *minpts* fixed (although *minpts* could have been varied, performance is driven by ϵ); therefore, each variant v_i required a different T. However, if ϵ is fixed and *minpts* is varied, then T can be used for all values of *minpts*. This is the opposite configuration of OPTICS [29], where *minpts* is fixed and ϵ is varied. Table V outlines scenario $S3$ where each dataset has a fixed ϵ and 16 values of *minpts*. In this scenario, T is computed once, and then up to 16 threads use it as input into DBSCAN for differing values of *minpts*.



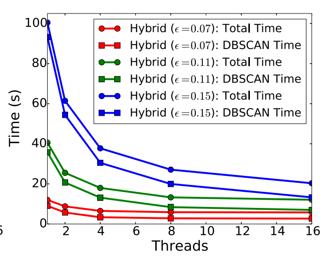
(a) *SW1*



(b) *SW4*



(c) *SDSS1*



(d) *SDSS3*

Figure 5: Response time vs. number of threads concurrently clustering scenario $S3$ when reusing a single table from a fixed ϵ value for multiple values of *minpts*. The difference between the same colored curves indicates the time to compute T. *SDSS2* has been omitted as its performance shows similar trends to *SDSS1* and *SDSS3*.

Figure 5 shows the response time vs. the number of threads used to execute the 16 variants on the datasets in $S3$. The difference between same color curves indicates the time to compute T. Executing HYBRID-DBSCAN between 1 and 16 threads on *SW1*, we find that the speedup ranges from $4.37\times$ ($\epsilon = 0.3$) to $6.07\times$ ($\epsilon = 0.7$), and on *SDSS1* it ranges from $2.89\times$ ($\epsilon = 0.3$) to $5.1\times$ ($\epsilon = 0.7$).

Figure 6 plots the speedup of HYBRID-DBSCAN using 16 threads with a single T (Figure 5 shows the response times) over individually clustering scenario $S3$ with the reference implementation. Reusing T to cluster multiple values of *minpts* leads to a relative speedup between $27\times$–$54\times$. This shows the utility of reusing T to maximize clustering throughput when using the GPU. Note that a CPU-only implementation could also compute and reuse T.

Table V: Scenario 3 ($S3$).

Dataset	v_i^{ϵ}	v_i^{minpts}
SW1	{0.3}	{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 400, 800, 1000, 2000, 3000}
SW1	{0.5}	"
SW1	{0.7}	"
SW4	{0.1}	"
SW4	{0.2}	"
SW4	{0.3}	"
SDSS1	{0.3}	{5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80}
SDSS1	{0.5}	{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150}
SDSS1	{0.7}	"
SDSS2	{0.2}	"
SDSS2	{0.3}	"
SDSS2	{0.4}	"
SDSS3	{0.07}	{5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80}
SDSS3	{0.11}	"
SDSS3	{0.15}	"

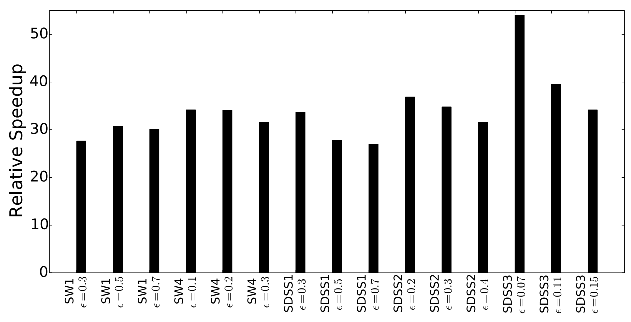


Figure 6: Speedup when reusing a single T with a fixed ϵ value in comparison to clustering with the reference implementation on scenario $S3$.

VIII. DISCUSSION AND CONCLUSION

This work advances a hybrid CPU-GPU approach to maximize DBSCAN clustering throughput. We demonstrate that despite host-GPU memory transfers with limited performance, combining GPU executions with an efficient index and batching scheme can overcome the memory bottlenecks associated with 2-D ϵ-neighborhood searches (Table I). This result has significant practical impact for DBSCAN and other applications, as host-GPU memory transfers are a typical bottleneck that make GPUs often unsuitable for memory-bound applications. The techniques described in this work are applicable to other similarity searches.

To maximize clustering throughput, our previous work [4] on DBSCAN exploited data reuse, where smaller clusters (having lower values of ϵ) could be used as seeds for clustering with larger values of ϵ. There, performance relied on data density and distribution. In contrast, HYBRID-DBSCAN is oblivious to the data characteristics to maximize clustering throughput, and when fixing ϵ and varying *minpts*, the ϵ-neighborhood of each point can be reused, which leads to significant performance gains over the reference implementation. The performance of HYBRID-DBSCAN is likely to improve over CPU algorithms as host-GPU bandwidth increases (e.g., with NVLink). Future work directions include revisiting shared memory kernels that do not suffer from the overheads outlined in Section VII-C, and modeling the performance of HYBRID-DBSCAN to predict how future increases in host-GPU bandwidth influence performance.

ACKNOWLEDGMENTS

We acknowledge support from NSF ACI-1442997.

REFERENCES

[1] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. of the 2nd KDD*, 1996, pp. 226–231.

[2] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1984, pp. 47–57.

[3] C. Xiaoyun, M. Yufang, Z. Yan, and W. Ping, "Gmdbscan: Multi-density dbscan cluster based on grid," in *Proc. of the IEEE Intl. Conf. on e-Business Engineering*, 2008, pp. 780–783.

[4] M. Gowanlock, D. M. Blair, and V. Pankratius, "Exploiting Variant-based Parallelism for Data Mining of Space Weather Phenomena," in *Proc. of the 30th IEEE Intl. Parallel & Distributed Processing Symposium*, 2016.

[5] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *Proc. of the 18th ACM Conf. on Information and Knowledge Management*. New York, NY, USA: ACM, 2009, pp. 661–670.

[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.

[7] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme Scale Density-based Clustering Using a Tree-based Network of GPGPU Nodes," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 84:1–84:11.

[8] M. Chen, X. Gao, and H. Li, "Parallel DBSCAN with Priority R-tree," in *The 2nd IEEE Intl. Conf. on Information Management and Engineering*, 2010, pp. 508–511.

[9] M. 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," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 62:1–62:11.

[10] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2014.

[11] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel approximate density-based clustering," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 560–571.

[12] P. Cal and M. Woźniak, "Data preprocessing with gpu for dbscan algorithm," in *Proc. of the 8th Intl. Conf. on Computer Recognition Systems*, 2013, pp. 793–801.

[13] S. Li and N. Amenta, "Brute-force k-nearest neighbors search on the gpu," in *Proc. of the 8th Intl. Conf. on Similarity Search and Applications*, 2015, pp. 259–270.

[14] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: A gpu merging algorithm," in *Proc. of the 26th ACM Intl. Conf. on Supercomputing*, 2012, pp. 331–340.

[15] R. J. Durrant and A. Kabán, "When is 'nearest neighbour' meaningful: A converse theorem and implications," *Journal of Complexity*, vol. 25, no. 4, pp. 385–397, 2009.

[16] I. Volnyansky and V. Pestov, "Curse of dimensionality in pivot based indexes," in *Proc. of the Second Intl. Workshop on Similarity Search and Applications*, 2009, pp. 39–46.

[17] J. Kim, W.-K. Jeong, and B. Nam, "Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, 2015.

[18] C. Böhm, R. Noll, C. Plant, and A. Zherdin, "Index-supported similarity join on graphics processors." in *BTW*, 2009, pp. 57–66.

[19] J. Zhang, S. You, and L. Gruenwald, "U^{2}STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs," in *Proc. of the ACM Workshop on City Data Management*, 2012, pp. 5–12.

[20] M. Gowanlock and H. Casanova, "Indexing of Spatiotemporal Trajectories for Efficient Distance Threshold Similarity Searches on the GPU," in *Proc. of the 29th IEEE Intl. Parallel & Distributed Processing Symposium*, 2015, pp. 387–396.

[21] ——, "Distance Threshold Similarity Searches: Efficient Trajectory Indexing on the GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2533–2545, 2016.

[22] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 3:1–3:8.

[23] V. Pankratius, J. Li, M. Gowanlock, D. M. Blair, C. Rude, T. Herring, F. Lind, P. J. Erickson, and C. Lonsdale, "Computer-Aided Discovery: Towards Scientific Insight Generation with Machine Support," *IEEE Intelligent Systems*, vol. 31, pp. 3–10, 2016.

[24] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," *GPU Computing Gems: Jade Ed.*, 2012.

[25] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *IEEE Intl. Parallel & Distributed Processing Symposium*, 2009, pp. 1–12.

[26] V. Pankratius, A. Coster, J. Vierinen, P. Erickson, and B. Rideout, *GPS Data Processing for Scientific Studies of the Earth's Atmosphere and Near-Space Environment*. Springer International Publishing, 2015, pp. 1–12.

[27] S. Alam *et al.*, "The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III," *The Astrophysical Journal Supplement Series*, vol. 219, p. 12.

[28] ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip, accessed 20-October-2016.

[29] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1999, pp. 49–60.