A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU

Michael Gowanlock, *Member, IEEE*, Cody M. Rude, David M. Blair, Justin D. Li, and Victor Pankratius

Abstract—We introduce Hybrid-Dbscan, that uses the GPU and CPUs for optimizing clustering throughput. The main idea is to exploit the memory bandwidth on the GPU for fast index searches, and optimize data transfers between host and GPU, to alleviate the potential negative performance impact of the PCIe interconnect. We propose and compare two GPU kernels that exploit grid-based indexing schemes to improve neighborhood search performance. We employ a batching scheme for host-GPU data transfers to obviate limited GPU memory, and exploit concurrent operations on the host and GPU. This scheme is robust with respect to both sparse and dense data distributions and avoids buffer overflows that would otherwise degrade performance. We evaluate our approaches on ionospheric total electron content datasets as well as intermediate-redshift galaxies from the Sloan Digital Sky Survey. Hybrid-Dbscan outperforms the reference implementation across a range of application scenarios, including small workloads, which typically are the domain of CPU-only algorithms. We advance an empirical response time performance model of Hybrid-Dbscan by utilizing the underlying properties of the datasets. With only a single execution of Hybrid-Dbscan on a dataset, we are able to accurately predict the response time for a range of ϵ search distances.

Index Terms—DBSCAN, GPGPU, in-memory database, parallel clustering, query optimization, spatial databases

1 Introduction

Unsupervised machine learning methods, such as clustering, are used in many application domains. In this paper, we present a hybrid CPU/GPU approach for the Density-Based Spatial Clustering of Applications with Noise (Dbscan) [1] clustering algorithm. The parameters to the algorithm are: (i) the distance ϵ, that is searched within the neighborhood of each point; and (ii) the number of points within ϵ that are required of each point to be a member of a cluster, *minpts*. Dbscan requires expensive neighborhood searches around each point in the dataset. As shown in our previous work [2], sequential R-tree searches on a galaxy dataset require up to 72.2 percent of the total response time.

To find neighboring points on the GPU, we propose grid-based index optimizations. As the search is constrained by ϵ, a grid can be constructed such that only neighboring grid cells need to be examined. In comparison to using index-trees[3], grids are well-suited to the GPU, as the search is bounded and fewer branch instructions are needed.

- *M. Gowanlock is with the School of Informatics, Computing & Cyber Systems, Northern Arizona University, Flagstaff, AZ 86011. E-mail: michael.gowanlock@nau.edu.*
- *C.M. Rude and V. Pankratius are with the Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: {cmrude, pankrat}@mit.edu.*
- *D.M. Blair is with the Department of Chemistry and the Department of Earth, Environmental and Planetary Sciences, Brown University, Providence, RI 02912. E-mail: david_blair@brown.edu.*
- *J.D. Li is with the AAAS Science & Technology Policy Fellow, Washington, DC 20005. E-mail: jdli@alumni.stanford.edu.*

Manuscript received 30 Apr. 2018; revised 2 Aug. 2018; accepted 1 Sept. 2018. Date of publication 13 Sept. 2018; date of current version 13 Mar. 2019. (Corresponding author: Michael Gowanlock.) Recommended for acceptance by J. Wang. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2869777

Other GPU approaches to DBSCAN [4], [5], [6] perform the clustering process in parallel, and then merge subclusters to make the final clusters. In contrast to these works, 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, which also allows us to reuse the data when clustering with different parameters. Our hybrid approach relies on obviating GPU memory limitations with our batching scheme. The paper organization and major contributions are as follows:

- We compare two GPU kernels that utilize a grid-based index for finding the ϵ-neighborhood of each point in the dataset. On the CPU, the neighbors of each point are then used for Dbscan point assignment (Sections 4 and 5).
- We develop an efficient batching scheme for Dbscan that allows for processing result sets that exceed global memory capacity, and overlaps GPU computation and GPU-host communication. Our approach does not require expensive kernel restart strategies (Section 6).
- We evaluate our Hybrid-Dbscan algorithm under the following scenarios: (i) clustering data with a single set of parameters; (ii) a pipelined multi-clustering approach to maximize clustering throughput; and (iii) a data reuse scheme to reuse the ϵ-neighborhood of each point being clustered. In general, our approach outperforms the reference implementation of Dbscan (Section 7).
- To understand the relative performance of Hybrid-Dbscan, we advance a performance model. With only a single execution of Hybrid-Dbscan, a set of data-dependent model parameters are obtained, and can be used to accurately predict the response time of Hybrid-Dbscan for other relevant values of the ϵ parameter.

1045-9219 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

The remainder of the paper is organized as follows: Section 2 introduces DBSCAN and related work. Section 3 outlines the problem. Finally, we discuss and conclude the work in Sections 8 and 9, respectively.

2 Background

2.1 The DBSCAN Algorithm

We provide a brief outline of DBSCAN and refer the reader to [1] and our previous work [2] for more details. Let D be a database of points to be clustered. A point $p \in D$ 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. For a given point p, if $|N_\epsilon(p)| \geq minpts$, p is referred to as a *core point*. When beginning a new cluster, a core point adds neighboring points to the cluster, ϵ-neighborhood searches occur around these points, and points are added to the cluster if they are found in regions with sufficient density. The cluster stops growing when there are no more points reachable by points in the cluster that meet the density criteria. The process continues until all points in the dataset have been assigned to a cluster or are classified as outliers.

2.2 Related Work

Many studies have addressed improving the efficiency of DBSCAN [4], [5], [6], [7], [8], [9], [10], [11], but here we focus on those related to the GPU. CUDA-DClust [4] 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 labeled as noise, the algorithm resolves the collisions and assigns each point a cluster id. In [6], the authors reuse the main ideas of CUDA-DClust in [4], 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 [5] 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-dimensional (2-D) point objects, calculating the distance matrix would be inefficient in comparison to using an index.

Our previous works [11], [18] advanced a parallel CPU-only approach for optimizing clustering throughput. That approach relied on using a previous clustering result as a seed to produce a new set of clusters with a larger ϵ value. This approach is largely sensitive to the data distribution. In contrast, the hybrid approach that we advance in this paper is oblivious to the data distribution [2].

With the proliferation of manycore architectures such as the GPU, new indexing approaches have been developed [17], [19], [20], [21], [22]. As noted in [19] and [21], 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 [23] 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 [20], [21] and [22].

3 Motivation and Problem Statement

A typical scenario for domain scientists is that they need to examine a range of possible outcomes to assist in finding new phenomena. Thus, the motivation of this work is to maximize cluster throughput when clustering a dataset for a broad range of parameters.

Let D be a database of 2-D points to be clustered by DBSCAN. Each point p_i, $i = 1, \ldots, |D|$, 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 [4], [5], [6], 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 $v_i, i = 1, \ldots, |V|$, 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.

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

We give an overview of HYBRID-DBSCAN in Algorithm 1 and describe where each component is executed (CPU or GPU), and later refer to the individual components of the algorithm. The algorithm begins by constructing an index on the host (line 3), and then a GPU kernel is launched that finds the points within ϵ of each point (line 4). The result set is stored in a buffer in a nondeterministic order by the GPU threads, so it is sorted by each query point on the GPU (line 5), then the data is copied to the host, and duplicate query points are removed (line 6). A modified DBSCAN algorithm is executed (line 7) that only performs point assignment, as T has been computed on the GPU.

Algorithm 1. The HYBRID-DBSCAN Algorithm

```
1: procedure HYBRID-DBSCAN(D, ε, minpts)
2:   R ← ∅ ; T ← ∅
3:   (G, A) ← ConstructIndex(D, ε)
4:   GPUCALCGLOBAL(D, G, A, ε)
5:   R ← gpuResultSet.SortByKey()
6:   ConstructNeighborTable(R, T)
7:   DBSCAN(T, minpts)
8:   return
```

4 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 [21]). 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 insert points into these cells. We store the index, G, as an array of cells. Each cell is defined as C_h, $h = 1, \ldots, |G|$, 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 $i \in \{A[A_h^{min}], \ldots, A[A_h^{max}]\}$. 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. See Fig. 1 in our previous work [2] for an illustration of the index.

4.1 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 11). 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, ε)
2:   gid ← getGlobalId()
3:   if gid ≥ |D| then return
4:   cellIDsArr ← ∅; gpuResultSet ← ∅
5:   point ← D[gid]
6:   cellIDsArr ← getNeighborCells(gid)
7:   for cellID ∈ cellIDsArr do
8:     lookupMin ← A[G[cellID].min]
9:     lookupMax ← A[G[cellID].max]
10:    for candidateID ∈ {lookupMin,...,lookupMax} do
11:      result ← calcDistance(point, D[candidateID], ε)
12:      if result ≠ ∅ then
13:        atomic: gpuResultSet ← gpuResultSet ∪ result
   return
```

4.2 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. Thus, we are oblivious to the number of data points per cell, and do not exceed shared memory capacity.

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

In Algorithm 3, we refer to CUDA keywords as follows: (a) arrays that are stored in shared memory use the *shared* keyword; (b) the block size is blockDim.x (we only use one memory dimension); (c) the thread id in a block is threadId.x; and (d) block-level thread synchronization is denoted as *synchronize()*. 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

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

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. 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 11-13. Had the number of points in the origin cell exceeded the block size, then another loop would be inserted that encompasses lines 11-23. Next, a loop is entered that iterates over all of the relevant cells (line 15). Next, each thread loads a point from global into shared memory from a comparison cell on lines 16-18 and the threads in the block are synchronized (line 19). 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 20-23. 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, ε, S, N)
 2:   gid ← getGlobalId()
 3:   if gid ≥ N then return
 4:   shared cellIDsArr ← ∅; gpuResultSet ← ∅
 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:   lookupOffset ← G[cellToProc].min + threadId.x
12:   dataID ← A[lookupOffset]
13:   pntsOriginCell ← pntsOriginCell ∪ {D[dataID]}
14:   synchronize()
15:   for cellID ∈ cellIDsArr do
16:     compLookupOffset ← G[cellID].min + threadId.x
17:     compDataID ← A[compLookupOffset]
18:     pntsCompCell ← pntsCompCell ∪ {D[compDataID]}
19:     synchronize()
20:     for candidateID ∈ pntsCompCell do
21:       result ← calcDistance(pntsOriginCell[threadId.x],
            pntsCompCell[candidateID], ε)
22:       if result ≠ ∅ then
23:         atomic: gpuResultSet ← gpuResultSet ∪ result
    return
```

5 Algorithm Overview

We describe Hybrid-Dbscan (Algorithm 1) in greater detail as follows. 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 the Hybrid-Dbscan components exceed the size of global memory on the GPU (Section 6). R, and T are initialized on line 2. Next, the index is constructed using D and ϵ as described in Section 4 (line 3) and the GPU kernel is launched; either GPUCalcGlobal or GPUCalcShared can be used, although here we show GPUCalcGlobal (line 4). Note that if the GPUCalcShared is used, then we also compute and transfer S and N to the kernel as described in Section 4.2. 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 5), 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 6). 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, which allows the pinned memory buffer to be available to write R for a future batch (Section 6). After T has been constructed, Dbscan is executed on line 7 which uses T and *minpts* (instead of ϵ and *minpts* in the original Dbscan algorithm). The algorithm is modified such that the ϵ-neighborhood search does not search the index I using ϵ, instead looking up the neighbors in T.

6 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. Furthermore, the result set size is *nondeterministic*; therefore, we do not know a priori how large the result set will be. We are able to accommodate large result set sizes with an efficient batching scheme that incrementally computes the neighbor table, T, and addresses nondeterministic sizes. There are two primary methods in the literature for addressing large/non-deterministic result set sizes as follows: (i) allocate a buffer for the result, and use the strategy of kernel result buffer overflow prevention and subsequent kernel restart [21], [25], [26]; and, (ii) execute the entire algorithm twice, once to compute the exact size of the result set, and the second to store the result set [27]. The drawback of both of these approaches is that they are not work-efficient. In (i), a buffer overflow may occur while only part of the result set for a number of query points have been computed (the neighbors would need to be recomputed when reattempting the kernel for these points); and (ii) requires double the total work.

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

There are two motivations for our batching scheme design that avoid (i) and (ii) above. First, we do not use the strategy of executing the algorithm twice, or kernel result buffer overflow prevention and subsequent kernel restart. Second, we do not require additional memory for buffer overflow purposes. We achieve these objectives by first *estimating* the total result set size, and then execute a number of batches such that the buffers never overflow.

We describe the batching scheme as follows. 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 γ as an overestimation factor. We calculate n_b as follows:

$$n_b = \lceil [(1+\gamma) \cdot a_b]/(b_b) \rceil. \quad (1)$$

Equation (1) relies on estimating a_b accurately 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 and overlap computation and communication. However, this is at the expense of expensive pinned memory allocation [28]. Thus we want to minimize γ 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 4). Since this kernel (similar to Algorithms 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 percent 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 (Algorithms 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 4.1 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 $|D|$). See Fig. 2 in our previous work [2] for an example of 5 adjacent points in D processed by different kernel invocations. Using this execution strategy, the values of $|R_l|$ are roughly consistent across batches. We set $\gamma = 0.05$, 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 (we found that additional streams do not improve performance). Therefore, in Algorithm 1, lines 4-6 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, 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 $a_b \geq 3 \times 10^8$ we set $b_b = 10^8$ (each stream has a buffer of this size), and if $a_b < 3 \times 10^8$ we set $b_b = (a_b \times (1+2\gamma))/3$. We increase γ 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.

7 Experimental Evaluation

7.1 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 [29]). *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 [30] 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.

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 [31] and described in [11]. See [30] for *SDSS-* datasets.

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

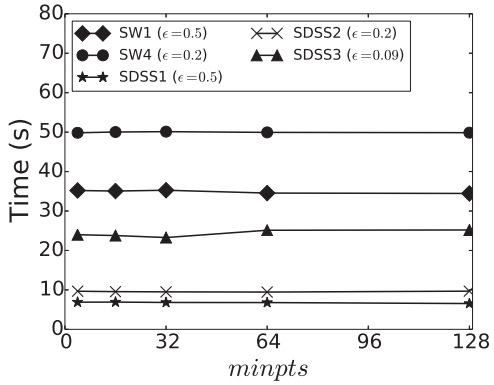


Fig. 1. Reference implementation response time versus *minpts* for all datasets. The following epsilon values for clustering the datasets are used: *SW1* and *SDSS1* ($\epsilon = 0.5$), *SW4* and *SDSS2* ($\epsilon = 0.2$), and *SDSS3* ($\epsilon = 0.09$).

7.2 Experimental Methodology

The multithreaded implementations of HYBRID-DBSCAN on the host use OpenMP in C++ and are compiled with the O3 compiler optimization flag. The GPU code is written in CUDA. Response times are averaged over 3 trials, unless otherwise noted. We use two similar platforms as follows: *Platform 1*. Consists of 2.6 GHz Intel Xeon E5-2660 v3 processors (20 total cores), and an Nvidia Tesla K40m card with 12 GB of memory. *Platform 2:* Consists of 2.4 GHz Intel Xeon E5-2630 v3 processors (16 total cores), and an Nvidia Tesla K20c card with 5 GB of memory.

All experiments in Section 7.3 are performed on Platform 1; and the remainder are performed on Platform 2. We found no significant performance difference between the two GPUs (they are the same Tesla generation). 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 [11]). 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.

For all experiments excepting the single kernel invocation experiment in Section 7.3, we use at least $n_b = 3$ and a maximum buffer size, $b_b = 10^8$ (Section 6). In the scenario where the total result set size exceeds global memory capacity, we still limit b_b to its maximum size, and may not fully utilize the entire memory capacity. We do not maximize global memory usage as: (i) pinned memory is expensive to allocate, so we select b_b such that it does not significantly degrade performance; and (ii) executing multiple batches allows for overlapping data transfers and computation. Thus, even if the entire result set can fit within global memory, we still use a batched execution. In all experiments that require the maximum buffer size, we are able to store the 3 buffers of size $b_b = 10^8$ (~ 2.24 GiB), the entire dataset, and the index, within the 5 GB of global memory on Platform 2.

To demonstrate that performance is independent of the *minpts* parameter, we show the reference implementation response time versus *minpts* in Fig. 1. It is clear that the selection of the *minpts* parameter does not significantly impact performance. As HYBRID-DBSCAN performs the point assignment on the CPU, the same performance behavior occurs in HYBRID-DBSCAN.

7.3 Kernel Efficiency and Overheads

We evaluate the performance of the two kernels, GPUCALCGLOBAL and GPUCALCSHARED. To assess kernel efficiency, we only examine a single kernel invocation (we do not use batching) and do not analyze the other overheads. Table 1 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.

Comparing the response times in Table 1, GPUCALCSHARED performs worse than GPUCALCGLOBAL on all datasets but *SW1*. 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 1 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. Furthermore, the occupancy of GPUCALCSHARED is lower than the GPUCALCGLOBAL kernel, thus reducing throughput.

The threads in a block are assigned a cell in GPUCALCSHARED; therefore, they will have more coalesced memory accesses than GPUCALCGLOBAL, as they page the points from an adjacent cell into shared memory. Also, each block of threads in GPUCALCSHARED will have similar instruction flows. Thus, the warp execution and global load efficiency is higher for GPUCALCSHARED than GPUCALCGLOBAL. Despite these performance advantages, GPUCALCSHARED is still generally slower due to lower occupancy and the larger total number of threads needed.

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 90.2 percent faster than GPUCALCSHARED on *SW4*, and 1,683 percent faster on *SDSS2*. Since *SDSS2* is more uniform, there are more grid cells containing points, thus 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.

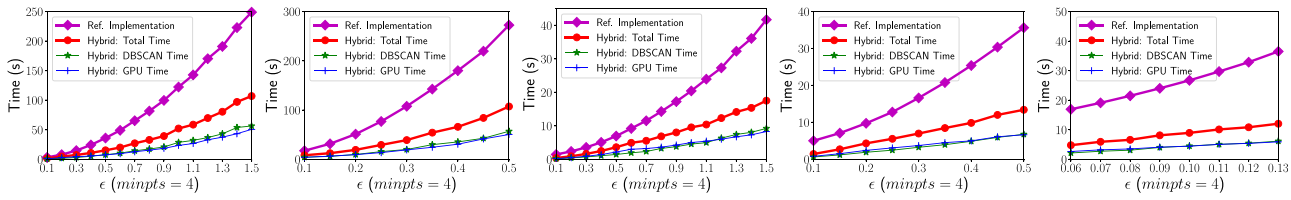
As discussed in Section 6, we use a batching scheme to overlap computation and data transfers between the host and GPU. With the exception of the results presented Table 1, we use a minimum of three streams (and batches) which requires asynchronous data transfers and pinned memory. Table 2 compares data transfer overheads to the total execution time of the kernels ($n_b = 3$ for all datasets).

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Scenario 1 ($S1$) & Kernel Efficiency

Dataset	ϵ	GPUCalcGlobal					GPUCalcShared				
		Kernel Time (ms)	n^{GPU}	Occupancy (%) Theoretical (Achieved)	Warp Exec. Eff. (%)	Global Load (%)	Kernel Time (ms)	n^{GPU}	Occupancy (%) Theoretical (Achieved)	Warp Exec. Eff. (%)	Global Load (%)
SW1	0.2	515.476	1,864,704	75 (66.5)	58.4	20.6	495.146	37,409,792	62.5 (58.7)	77.0	40.4
SW4	0.07	520.806	**5,159,936**	75 (**66.1**)	56.0	20.5	990.816	**255,272,704**	62.5 (**58.4**)	72.6	44.9
SDSS1	0.2	73.408	**2,000,128**	75 (**67.5**)	54.9	22.0	429.064	**110,757,120**	62.5 (**59.1**)	70.8	45.6
SDSS2	0.07	75.202	**5,000,192**	75 (**68.1**)	45.0	19.9	1340.64	**649,954,560**	62.5 (**59.6**)	79.8	48.4

The theoretical and achieved occupancy, warp execution efficiency, and global load values are given in percentages. Comparing the values in bold illuminates the poor performance of GPUCalcShared in comparison to GPUCalcGlobal.



(a) *SW1* (b) *SW4* (c) *SDSS1* (d) *SDSS2* (e) *SDSS3*

Fig. 2. Response time versus 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.

While the kernels are executed in serial (i.e., their execution on the GPU cannot be overlapped), the data transfers occur concurrently with GPU computation. Table 2 shows that the total time for data transfers is less than the time to execute the GPUCalcGlobal kernels for a given dataset; however, these data transfers are at least partially overlapped with computation on the GPU. Therefore, the overhead is less than that elucidated in Table 2. We report the mean bandwidth utilization, and find that the pinned memory data transfers are able to saturate a majority of the bandwidth over PCIe v.3. Thus, result set data transfers are not a significant bottleneck, and future bandwidth increases will improve the relative performance of Hybrid-Dbscan. Furthermore, as n_b increases, there is more opportunity to overlap data transfers with computation. Since these data transfers occur with the smallest value of n_b possible, the ratio of computation to data transfer overhead is likely to be higher when $n_b > 3$.

The kernel times in Table 1 are slightly lower than Table 2, as we launch 3 kernels for the results in Table 2 to overlap data transfers and computation. The greater total kernel response times are due to the overhead of launching the additional kernels, and the loss of continuity in the computation. However, the performance gain from overlapping data transfers and computation outweighs the relatively small increase in total kernel execution time.

TABLE 2
Scenario 1 ($S1$) & Data Transfer Overhead

Dataset	ϵ	Kernel Time (ms)	Data Transfer Time (ms)	Mean Rate (GB/s)
SW1	0.2	561.317	215.995	9.866
SW4	0.07	572.648	195.690	10.007
SDSS1	0.2	76.811	29.535	9.383
SDSS2	0.07	77.270	23.836	10.328

GPUCalcGlobal is executed with three batches for each dataset. The total time to execute the GPUCalcGlobal kernels, the time needed to transfer the result set over PCIe, and bandwidth utilization, are reported for a single trial.

7.4 Performance of Hybrid-DBSCAN

As shown in Fig. 1, performance is independent of *minpts*. Thus, in this section, we will focus on only varying ϵ.

We assess Hybrid-Dbscan (Algorithm 1) as compared to the reference implementation. As shown in Table 3 ($S2$), we individually execute Hybrid-Dbscan for 15 different parameters for *SW1* and *SDSS1*, and 9 for *SW4* and *SDSS2*, and 8 for *SDSS3*. Fig. 2 plots the response time versus 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 the smallest datasets (*SW1* and *SDSS1*). This is an interesting result 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.

7.5 Multi-Clustering Pipeline

Some applications may require executing Dbscan for a range of input parameters. To maximize clustering throughput,

TABLE 3
Scenario 2 ($S2$)

Dataset	v_i^{ϵ}	Dataset	v_i^{ϵ}
SW1	{0.1, 0.2, ..., 1.5}	*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}	$v_i^{minpts} = \{4\}$ for all datasets.	

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

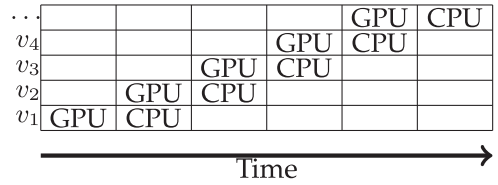


Fig. 3. Illustration of the pipelined execution. The neighbor table, T, is computed on the GPU for v_{i+1}, while the CPU performs the point assignment (modified DBSCAN) for v_i. This figure depicts the optimal case where the CPU and GPU components have perfect load balancing.

we pipeline the construction of T and the DBSCAN phases of Algorithm 1 in a producer-consumer fashion; one thread executes lines 4-6 in Algorithm 1 for all of the variants in scenario $S2$ (see Table 3) and another takes T as input into DBSCAN (line 7). The host computes T (for a single ϵ) spawns 3 threads for batching (as described in Section 6), 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 Fig. 2). In the pipelined approach, when the modified DBSCAN algorithm is being executed for v_i on the CPU, T is being computed for v_{i+1}, as shown in Fig. 3.

We find that pipelined HYBRID-DBSCAN is between 41.8 percent (*SW1*) and 66.5 percent (*SDSS3*) faster than the non-pipelined algorithm. To achieve an optimal performance gain from the pipelined implementation, there would need to be perfect load balancing between CPU and GPU components, as depicted in Fig. 3. Since the GPU needs to be executed before the CPU, it is impossible to achieve a pipelined execution that is 100 percent faster than the non-pipelined approach. The optimal pipelined response time (T^{opt}) is calculated from the non-pipelined approach below:

$$T^{opt} = T^{np} \cdot [(|V| + 1)/(2|V|)], \quad (2)$$

where T^{np} is the non-pipelined response time, and $|V|$ is the number of variants executed (see Table 3).

Speedups over the reference implementation and non-pipelined version are shown in Table 4. Comparing the *SW-* and *SDSS-* datasets, both versions of HYBRID-DBSCAN perform comparatively better on *SDSS-*. Furthermore, in comparison to the optimal pipeline response time calculated by Equation (2), *SW1* was 32.26 percent from the optimal pipelined response time, whereas *SDSS3* was only 6.79 percent from the optimal response time.

In comparison to the reference implementation, the largest dataset, *SDSS3*, yields the greatest performance gain, suggesting that the performance of the R-tree begins to degrade with larger datasets, unlike HYBRID-DBSCAN which uses the grid-based index.

7.6 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 [32], where *minpts* is fixed and ϵ is varied. Table 5 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*.

Fig. 5 in our previous work [2] shows the response time versus 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, we find that the speedup ranges from $4.37\times$ ($\epsilon = 0.3$) to $6.07\times$ ($\epsilon = 0.7$) on *SW1*, and it ranges from $2.89\times$ ($\epsilon = 0.3$) to $5.1\times$ ($\epsilon = 0.7$) on *SDSS1*. Larger values of ϵ achieve a bigger performance improvement across the datasets.

Fig. 4 plots the speedup of HYBRID-DBSCAN using 16 threads with a single T 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.

7.7 Performance Model

As was shown in Section 7.4, the performance of HYBRID-DBSCAN (and DBSCAN) degrades with increasing ϵ. This is because as ϵ increases, the spatial search radius increases requiring more computation to find the neighbors within ϵ of a given point. In HYBRID-DBSCAN, this yields a larger neighbor table that needs to be sent back to the host for subsequent point assignment.

The size of the neighbor table, T, may be a good overall indicator of the quantity of work that will be executed by the GPU. Because the size of the neighbor table is a function of both ϵ and properties of the dataset (number of data points and spatial data distribution), it is not possible to estimate the size of the neighbor table without metrics regarding a given dataset.

Fig. 5 shows the average number of neighbors as a function of ϵ and the area of the ϵ-neighborhood for (a) *SW1* and (b) *SDSS1* (other datasets omitted as they are similar). The figure shows that the average number of neighbors increases linearly with the area of the ϵ-neighborhood. Therefore, with a linear increase in area defined by ϵ, there will be a linear increase in the size of the neighbor table, T.

TABLE 4
Speedup Using Pipelined HYBRID-DBSCAN on $S2$

Dataset	Speedup Pipelined versus Ref. Implementation	Speedup Pipelined versus Non-Pipelined	Time (s) Non-Pipelined	Time (s) Optimal Pipelined (T^{opt})	Time (s) Pipelined	% from Optimal
SW1	3.36	1.42	623.40	332.48	439.77	32.26
SW4	3.81	1.45	418.86	232.70	288.27	23.88
SDSS1	3.48	1.56	113.50	60.53	72.69	20.09
SDSS2	4.04	1.60	64.67	35.93	40.40	12.45
SDSS3	5.13	1.66	67.27	37.84	40.40	6.79

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
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}	″

7.7.1 Model Components

Given that the amount of work on the GPU may increase linearly as a function of area (Fig. 5), we advance a performance model that scales with area. Let T^{Total} be the total modeled response time, T^{GPU} be the modeled time for the GPU component of HYBRID-DBSCAN, and T^{CPU} be the modeled time required for the CPU component of the algorithm (the point assignment after the neighbor table has been computed). Therefore, the total modeled response time is of the following form: $T^{Total} = T^{GPU} + T^{CPU}$.

From Fig. 2, the CPU and GPU components of HYBRID-DBSCAN are nearly equal. Thus, we set $T^{CPU} = T^{GPU}$. For other platforms, there will be a similar ratio between the average time required for each of the CPU to GPU components of HYBRID-DBSCAN and this is entirely platform dependent. As a result, we obtain $T^{Total} = 2T^{GPU}$.

7.7.2 GPU Component

As mentioned above, the total amount of work should be proportional to the size of the neighbor table, T. Therefore, we aim to predict the response time of HYBRID-DBSCAN for any relevant value of ϵ by using only measured values from a *single* HYBRID-DBSCAN execution, corresponding to a single ϵ value. As was shown in Fig. 1, performance is not dependent on the *minpts* parameter.

We define several measured quantities that will be used in the performance model that are a function of the dataset and the value of ϵ, and thus need to be determined empirically. We denote the following measured values for a given ϵ value and dataset as follows:

- m^{ϵ} – the value of ϵ selected for executing HYBRID-DBSCAN to collect the empirical values for the model (parameters shown below).
- $v(m^{\epsilon})$ – the average number of neighbors in a dataset for m^{ϵ} (see Fig. 5).
- $a(m^{\epsilon})$ – the area defined by the value of m^{ϵ}, which is the radius of the ϵ-neighborhood, $a(m^{\epsilon}) = \pi(m^{\epsilon})^2$.

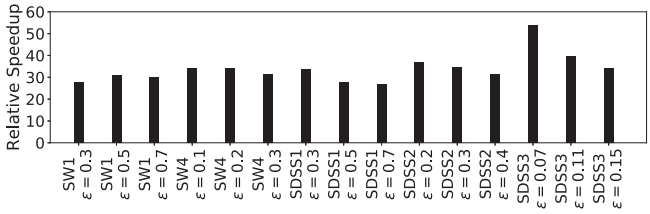
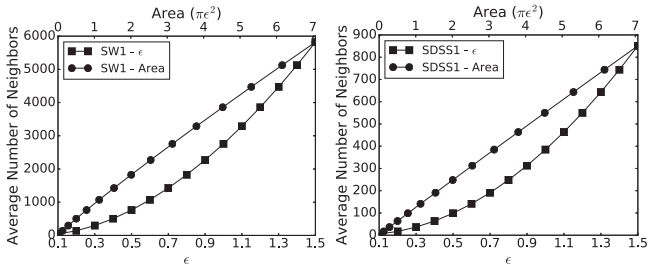


Fig. 4. Speedup over the reference implementation when reusing a single T with a fixed ϵ on scenario $S3$.



(a) *SW1* (b) *SDSS1*

Fig. 5. The mean number of neighbors of each point in datasets (a) *SW1* and (b) *SDSS1* versus ϵ (bottom horizontal axis), and area (top horizontal axis). The mean number of neighbors in a dataset grows linearly with the ϵ-area.

- $T(m^{\epsilon})$ – is the response time to execute the GPU component of HYBRID-DBSCAN with m^{ϵ}.
- $\theta(m^{\epsilon})$ – the time required for pinned memory allocation for the result set buffers when using m^{ϵ}.

Let the the performance model for the GPU be denoted as: $T^{GPU}(\epsilon, m^{\epsilon}) = \theta(m^{\epsilon}) + \pi\epsilon^2(\alpha \cdot \beta)$, where the time is estimated for a given ϵ value. We define $\alpha = v(m^{\epsilon})/a(m^{\epsilon})$, where for a given m^{ϵ}, this is the ratio of the average number of neighbors to the area defined by m^{ϵ}. Next, we define $\beta = (T(m^{\epsilon}) - \theta(m^{\epsilon}))/v(m^{\epsilon})$, which is the ratio of the response time to execute the GPU component of HYBRID-DBSCAN to the average number of neighbors in the dataset for a given m^{ϵ}, without the overhead of pinned memory allocation. The pinned memory is allocated once and then used for transferring the results for multiple batches to incrementally build the neighbor table; therefore, it needs to be removed from $T(m^{\epsilon})$, such that this overhead component does not scale with area. The expanded performance model is as follows:

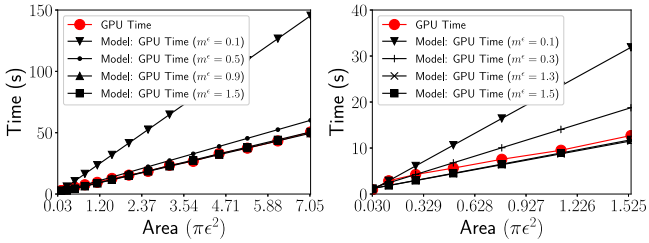
$$T^{GPU}(\epsilon, m^{\epsilon}) = \theta(m^{\epsilon}) + \pi\epsilon^2\left(\frac{T(m^{\epsilon}) - \theta(m^{\epsilon})}{a(m^{\epsilon})}\right). \quad (3)$$

Thus, with all of the measured quantities for a given m^{ϵ} and dataset: $\theta(m^{\epsilon})$, $v(m^{\epsilon})$, $a(m^{\epsilon})$, and $T(m^{\epsilon})$, the GPU response time is modeled as a function of ϵ, where the model, T^{GPU}, scales with area defined by ϵ. Note that in Equation (3), when $m^{\epsilon} = \epsilon$, then $T^{GPU}(\epsilon, m^{\epsilon}) = T(m^{\epsilon})$, i.e., the modeled and measured times are equal. All of the measured quantities can be recorded after executing HYBRID-DBSCAN once for a given m^{ϵ} value.

7.7.3 Model Evaluation

Fig. 6a shows the measured GPU response time and modeled response time (T^{GPU}) versus area on the *SW1* dataset (the response times are also shown in Fig. 2a versus ϵ).

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.



(a) *SW1* (b) *SW1*

Fig. 6. Comparison of the modeled (T^{GPU}) versus actual GPU response time as a function of area. (a) Family of models for *SW1*, where different selections of m^ϵ are shown. (b) Is the same as (a) but focusing on smaller areas and low and high values of m^ϵ. Low values of m^ϵ overestimates response time, whereas a high value of m^ϵ will underestimate the response time for small areas defined by ϵ.

Different models are shown for varying m^ϵ. Recall that for a given m^ϵ, the set of model parameters are obtained and used as input into Equation (3). From Fig. 6a, if we execute HYBRID-DBSCAN and use the parameters for a small value of ϵ, such as $m^\epsilon = 0.1$, the performance model significantly overestimates the response time for larger values of ϵ. This is because the ratio of work performed on the GPU to the overheads associated with using the GPU is lower for low values of ϵ in comparison to higher values of ϵ. However, when $m^\epsilon \geq 0.9$, the model is in good agreement with the measured response time. Fig. 6b highlights low values of ϵ in Fig. 6a. When selecting a higher value of ϵ to use as input to the model, such as $m^\epsilon = 1.5$, for smaller values of ϵ, the model slightly underestimates the response time. This is due to the above mentioned reason, that the ratio of computation on the GPU to the associated overhead is lower for larger ϵ values. Thus, the linear scaling as a function of area does not fully capture this effect.

Fig. 7 compares the performance model to the measured response times as a function of area (instead of ϵ as shown in Fig. 2) for each dataset. In the figure, we only select a single performance model, computed by executing HYBRID-DBSCAN for the respective values of ϵ. The parameters are summarized in Table 6. The GPU components of the models for *SW1* and *SW4* (Figs. 7a and 7b) exhibit a high degree of accuracy, as the median difference between the measured and modeled response times is at most 5.05 percent. The largest differences between the model and measured values occurs at low ϵ values. For instance, on *SW1* the largest difference is 35.34 percent and occurs at $\epsilon = 0.2$. This is because the model was generated using $m^\epsilon = 1.5$, and it has higher pinned memory overhead ($\theta(m^\epsilon)$ parameter) than the overhead which would have occurred for allocating pinned memory buffers for $\epsilon = 0.2$. As described in Section 6, the buffers for batching the results are allocated based on the estimate of the total size of the neighbor table. For a low value of ϵ, there will be smaller memory allocations for the buffers.

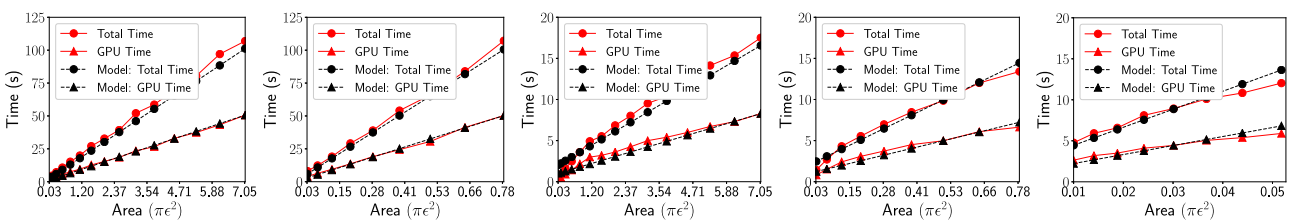
The median percentage difference between the GPU model (T^{GPU}) and measured values on the *SDSS-* datasets are 15.31 percent (*SDSS1*), 11.81 percent (*SDSS2*), and 10.49 percent (*SDSS3*); therefore, the simple linear performance model is consistent with the observed measurements. The dataset with the greatest number of points (*SDSS3*) is the most accurate of the *SDSS-* datasets, which was also observed for the *SW-* datasets. The worst difference between the measured and modeled response times was 365 percent on the *SDSS1* dataset, corresponding to $\epsilon = 0.1$. Like the model of *SW1* described above, this is due to the $\theta(m^\epsilon)$ parameter used.

We have shown that a performance model that scales with area can broadly capture the performance behavior of the GPU component of HYBRID-DBSCAN (T^{GPU}). However, achieving good accuracy across all values of ϵ is challenging, due to the following reasons. (i) For low values of ϵ, smaller buffers are allocated. Otherwise, the total response time would be dominated by pinned memory allocation. Thus, when predicting the model response time of a low value of ϵ when using the model parameters determined by a higher

TABLE 6
GPU Model Parameter Values and Accuracy Overview

Dataset	m^ϵ	$v(m^\epsilon)$	$a(m^\epsilon)$ (units^2)	$T(m^\epsilon)(s)$	$\theta(m^\epsilon)(s)$	T^{GPU} Model Accuracy (% Difference)		
						Best	Worst	Median
SW1	1.5	5818.84	7.069	50.61	0.98	0.819	35.34	5.05
SW4	0.5	2027.41	0.785	50.28	0.98	0.985	25.99	4.56
SDSS1	1.5	851.25	7.069	8.28	0.98	0.574	365.0	15.31
SDSS2	0.4	158.07	0.503	4.98	0.98	0.846	53.24	11.81
SDSS3	0.1	32.48	0.031	4.44	0.98	2.165	16.26	10.49

The model accuracy statistics exclude the model prediction for $T^{GPU}(\epsilon, m^\epsilon)$, where $\epsilon = m^\epsilon$, as the difference is 0 percent.



(a) *SW1*, $m^\epsilon = 1.5$ (b) *SW4*, $m^\epsilon = 0.5$ (c) *SDSS1*, $m^\epsilon = 1.5$ (d) *SDSS2*, $m^\epsilon = 0.4$ (e) *SDSS3*, $m^\epsilon = 0.1$

Fig. 7. Response times versus model (T^{GPU}, and T^{Total}) as a function of area for each dataset. We plot a single model using the values of m^ϵ shown above (see Table 6 for additional model parameters).

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

value of m^{ϵ}, the response time will be overestimated by the $\theta(m^{\epsilon})$ parameter. (ii) When comparing two datasets of roughly the same size, *SW1* and *SDSS1* (Figs. 7a and 7c), we observe that *SW1* requires more computation (and a larger neighbor table) than *SDSS1*, due to larger point overdensities in the dataset. Therefore, the overheads associated with the GPU are largely amortized in *SW1* due to the larger total number of batches required in comparison to *SDSS1*. Since the performance model assumes that the total work scales with area and does not account for the GPU overheads, performance models of datasets with smaller sized neighbor tables are less accurate, as shown on *SDSS1*.

In the performance model, the aim was to empirically measure the model parameters from a *single execution* of HYBRID-DBSCAN to compute the algorithm response time for other ϵ values. We note parenthetically that to improve the accuracy of modeling small values of ϵ, it would be straightforward to take two measurements of the model parameters for small and large values of ϵ (two m^{ϵ} values and associated parameters). However, this would incur the expense of executing HYBRID-DBSCAN twice to measure these parameters.

8 DISCUSSION

Due to the PCIe interconnect, this work relied on pinned memory allocation to improve the speed of data transfers between host and GPU. These memory buffers incur a significant amount of allocation overhead that is amortized when a large number of batches are required to transfer data from the GPU to the host. For small datasets and small values of ϵ (small workloads), we have shown that the pinned memory allocation is a significant overhead (Section 7.7.3). While the overhead is not prohibitive, there is a tradeoff between: (i) faster data transfers using pinned memory; and (ii) slower data transfers, but without the pinned memory allocation overhead. We have not explored this tradeoff, but such an optimization would improve the response time of HYBRID-DBSCAN, particularly on small workloads.

Future interconnects, such as NVLink [33], may reduce the need to use pinned memory in some applications, as NVLink may diminish the current data transfer bottleneck. Determining memory management optimizations will depend on the characteristics of the workload.

This work has focused on 2-D datasets as they are the most common spatial dataset used for clustering and other spatial data analytic applications. However, HYBRID-DBSCAN at higher dimensions is an open question. There is a well-known problem that as data dimensionality increases, the search of an index becomes increasingly exhaustive, due to the curse of dimensionality [34]. Depending on the index used, this can degrade index performance at high dimensionality to the extent that a brute force comparison between all data points is as efficient as using an index. However, this has been shown to only be the case for much higher dimensional data than is common for lower dimensional spatial datasets. For instance, [17] showed that a GPU R-tree outperforms other indexing schemes when there are fewer than 16 dimensions.

In high dimensionality, the memory footprint of the grid index would be intractable when indexing all cells. To obviate this problem in high dimensionality, we can instead store only those cells that are non-empty. That is, given an ϵ-neighborhood query in higher dimensions, a list of "logical cells" can be generated to determine which adjacent cells may contain points that may be within ϵ. However, this set of logical cells can be compared to the non-empty cells, by taking the intersection of the two sets of cells. The result will be the list of non-empty cells that contain data points that may be within ϵ of the point. This strategy ensures that the space required for the index is proportional to the data distribution of the dataset, and limits memory usage, which can be prohibitive to GPU indexing methods. Indeed, this method was shown in Gowanlock & Casanova [22] for indexing 4-D trajectories. Thus, while we we limit our analysis to 2-D datasets in this work, a grid-based index has been shown to be applicable for data of greater dimensionality.

9 CONCLUSIONS

This work departs from related GPU-accelerated DBSCAN research by performing the ϵ-neighborhood searches on the GPU and point assignment on the CPU. This approach has the added benefit of allowing clustering throughput optimization for a range of clustering parameters. Despite the host-GPU data transfer bottleneck, our efficient batching scheme can overcome this bottleneck, and also yields concurrent computation on the CPU and GPU, and overlaps this computation with host-GPU data transfers. The performance of HYBRID-DBSCAN and other hybrid data analysis algorithms are likely to improve over CPU algorithms through increases in global memory bandwidth, and host-GPU bandwidth (such as NVLink [33]).

Future work directions include reexamining the size of ϵ; a smaller ϵ may allow skipping distance calculations where there are very dense regions of points (as in the related work [6]), and integrating the work into a distributed memory implementation.

ACKNOWLEDGMENTS

We acknowledge support from US National Science Foundation ACI-1442997. A preliminary version of this work [2] was completed primarily while all of the authors were at MIT Haystack Observatory. We thank the University of Hawai'i at Manoa for the use of UHHPC.

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. 2nd Int. Conf. Knowl. Discovery Data Mining*, 1996, pp. 226–231.

[2] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius, "Clustering throughput optimization on the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2017, pp. 832–841.

[3] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984, pp. 47–57.

[4] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, 2009, pp. 661–670.

[5] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU accelerated algorithm for density-based clustering," *Procedia Comput. Sci.*, vol. 18, pp. 369–378, 2013.

[6] 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. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 84:1–84:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503262

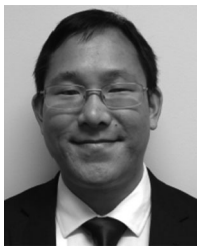
Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.

[7] M. Chen, X. Gao, and H. Li, "Parallel DBSCAN with priority R-tree," in *Proc. 2nd IEEE Int. Conf. Inf. Manage. Eng.*, 2010, pp. 508–511.
[8] 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. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 62:1–62:11.
[9] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: A scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers Comput. Sci.*, vol. 8, no. 1, pp. 83–99, 2014.
[10] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel approximate density-based clustering," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 560–571.
[11] M. Gowanlock, D. M. Blair, and V. Pankratius, "Exploiting variant-based parallelism for data mining of space weather phenomena," in *Proc. 30th IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 760–769.
[12] P. Cal and M. Woźniak, "Data preprocessing with GPU for DBSCAN algorithm," in *Proc. 8th Int. Conf. Comput. Recog. Syst.*, 2013, pp. 793–801.
[13] S. Li and N. Amenta, "Brute-Force k-nearest neighbors search on the GPU," in *Proc. 8th Int. Conf. Similarity Search Appl.*, 2015, pp. 259–270.
[14] O. Green, R. McColl, and D. A. Bader, "GPU merge path: A GPU merging algorithm," in *Proc. 26th ACM Int. Conf. Supercomput.*, 2012, pp. 331–340.
[15] R. J. Durrant and A. Kabán, "When is 'nearest neighbour' meaningful: A converse theorem and implications," *J. Complexity*, vol. 25, no. 4, pp. 385–397, 2009.
[16] I. Volnyansky and V. Pestov, "Curse of dimensionality in pivot based indexes," in *Proc. 2nd Int. Workshop Similarity Search Appl.*, 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 Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2258–2271, Aug. 2015.
[18] M. Gowanlock, D. M. Blair, and V. Pankratius, "Optimizing parallel clustering throughput in shared memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2595–2607, Sep. 2017.
[19] C. Böhm, R. Noll, C. Plant, and A. Zherdin, "Index-supported similarity join on graphics processors," in *Proc. Conf. Database Syst. Bus. Technol. Web*, 2009, pp. 57–66.
[20] J. Zhang, S. You, and L. Gruenwald, "U^{2}STRA: High-performance data management of ubiquitous urban sensing trajectories on GPGPUs," in *Proc. ACM Workshop City Data Manage.*, 2012, pp. 5–12.
[21] M. Gowanlock and H. Casanova, "Indexing of spatiotemporal trajectories for efficient distance threshold similarity searches on the GPU," in *Proc. 29th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 387–396.
[22] M. Gowanlock and H. Casanova, "Distance threshold similarity searches: Efficient trajectory indexing on the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2533–2545, Sep. 2016.
[23] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proc. 4th Workshop Gen. Purpose Process. Graph. Process. Units*, 2011, pp. 3:1–3:8.
[24] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems, Jade ed.* San Mateo, CA, USA: Morgan Kaufmann, 2011.
[25] C. Böhm, R. Noll, C. Plant, B. Wackersreuther, and A. Zherdin, "Data mining using graphics processing units," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems I.* Berlin, Germany: Springer, 2009, pp. 63–90.
[26] C. Böhm, R. Noll, C. Plant, and A. Zherdin, "Index-supported similarity join on graphics processors," in *Proc. Conf. Database Syst. Bus. Technol. Web*, 2009, pp. 57–66.
[27] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 511–524.
[28] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2009, pp. 1–12.
[29] 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*. Berlin, Germany: Springer, 2015, pp. 1–12.
[30] S. Alam, et al., "The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III," *Astrophysical J. Suppl. Series*, vol. 219, 2015. Art. no. 12.
[31] [Online]. Available: ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip. Accessed on: Oct. 20, 2016.
[32] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: Ordering points to identify the clustering structure," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1999, pp. 49–60.
[33] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, Mar./Apr. 2017.
[34] R. E. Bellman, *Adaptive Control Processes: A Guided Tour*. Princeton, NJ, USA: Princeton Univ. Press, 1961.

Michael Gowanlock received the PhD degree in computer science from the University of Hawai'i. He is an assistant professor with the School of Informatics, Computing & Cyber Systems, Northern Arizona University. He was a postdoctoral associate with MIT Haystack Observatory. His research interests include parallel data-intensive computing and astronomy. He is a member of the IEEE.

Cody M. Rude received the PhD degree in physics from the University of North Dakota. He is currently a postdoctoral associate with the MIT Department of Earth, Atmospheric and Planetary Sciences. His research interests include galaxy clusters, informatics, and geodesy.

David M. Blair received the PhD degree in earth, atmospheric, and planetary sciences from Purdue University. He is a scientific computing coordinator with Brown University. He was a postdoctoral associate with the MIT Haystack Observatory. His research interests include planetary geophysics, space exploration, and geoscience informatics.



Justin D. Li received the PhD degree in electrical engineering from Stanford University, completed postdoctoral research with MIT Haystack Observatory, and is now an AAAS Science & Technology Policy Fellow in Washington, D.C. His research interests cover the intersections between signal processing, machine learning, and the geosciences.

Victor Pankratius received the doctorate degree, in 2007, and the habilitation degree in computer science from the Karlsruhe Institute of Technology, Germany, in 2012. His research interests include parallel computing in data science and machine learning. Currently, he is a principal research scientist with the Massachusetts Institute of Technology, Kavli Institute for Astrophysics and Space Research, where he leads the Data Science in Astro-Geoinformatics Group.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on June 16,2021 at 09:36:06 UTC from IEEE Xplore. Restrictions apply.